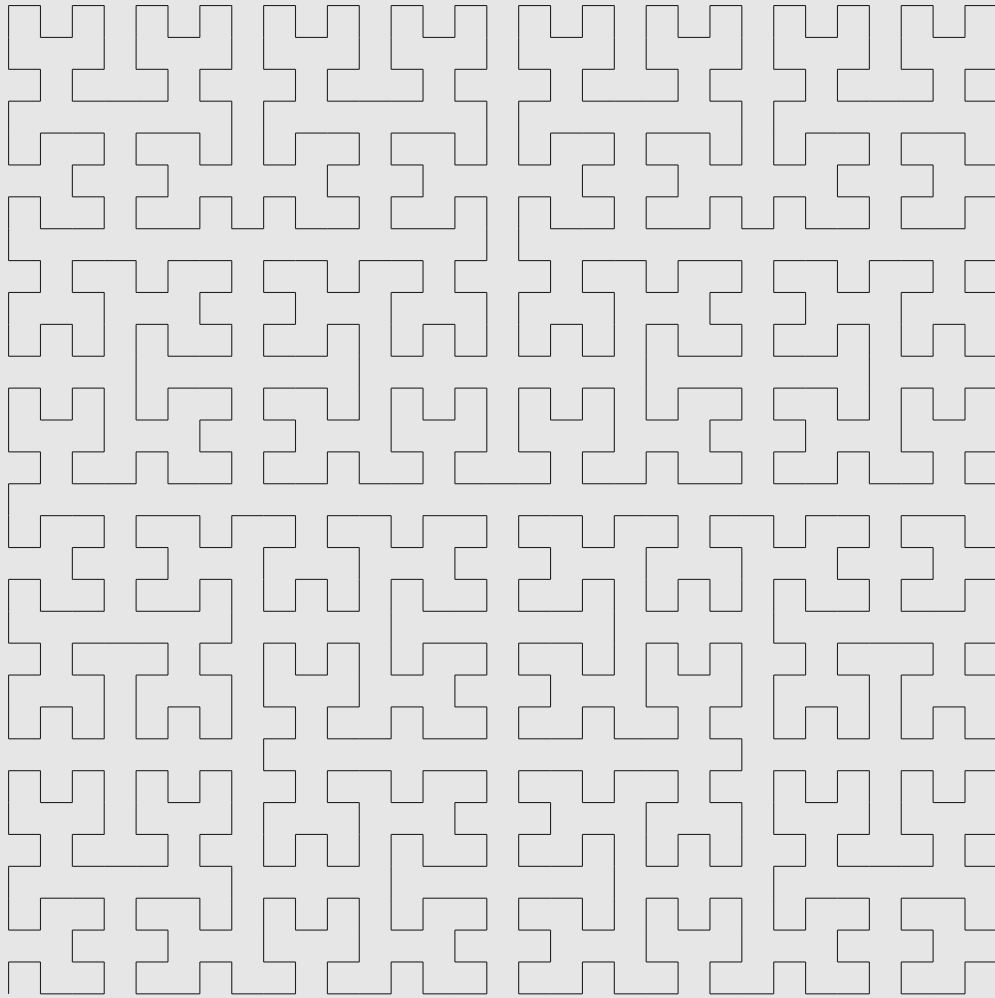


# LEARNING ALGOL 68 GENIE

---



---

Algol 68 Genie 2.0.0 (September 2010)

Edited by Marcel van der Veer

---

*Learning Algol 68 Genie* copyright © Marcel van der Veer, 2008-2010.

*Algol 68 Genie* copyright © Marcel van der Veer, 2001-2010.

*Learning Algol 68 Genie* is a compilation of separate and independent documents or works, consisting of the following parts:

- I. Informal introduction to Algol 68,
- II. Programming with Algol 68 Genie,
- III. Example a68g programs,
- IV. Algol 68 Revised Report,
- V. Appendices

Part I, II, III and V are distributed under the conditions of the GNU Free Documentation License: Permission is granted to copy, distribute and / or modify the text under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled *GNU Free Documentation License*.

See <http://www.gnu.org>.

Part IV is a translation of the Algol 68 Revised Report into  $\LaTeX$  and is therefore subject to IFIP's condition contained in that Report: Reproduction of the Report, for any purpose, but only of the whole text, is explicitly permitted without formality. Chapter 20, "Specification of partial parametrization proposal", is not a part of the Algol 68 Revised Report, and is distributed with kind permission of the author of this proposal, C.H. Lindsey.

---

IBM is a trademark of IBM corporation.

Linux is a trademark registered to Linus Torvalds.

Mac OS X is a trademark of Apple Computer.

Pentium is a trademark of Intel Corporation.

$\TeX$  is a trademark of the American Mathematical Society.

UNIX is a registered trademark of The Open Group.

Wikipedia is a trademark of the Wikimedia Foundation, Inc.

---

Typeset in  $\LaTeX$ .

# Table of contents

|   |           |
|---|-----------|
| <b>Preface</b>                                    | <b>xi</b> |
| <b>I Informal introduction to Algol 68</b>        | <b>1</b>  |
| <b>1 Preliminaries</b>                            | <b>3</b>  |
| 1.1 A brief history of Algol 68 . . . . .         | 3         |
| 1.2 Terminology . . . . .                         | 6         |
| 1.3 Notation of production-rules . . . . .        | 7         |
| <b>2 Basic concepts</b>                           | <b>11</b> |
| 2.1 Modes and values . . . . .                    | 11        |
| 2.2 Integers . . . . .                            | 12        |
| 2.3 Identity-declarations . . . . .               | 13        |
| 2.4 Reals . . . . .                               | 15        |
| 2.5 Formulas . . . . .                            | 18        |
| 2.6 Elementary monadic-formulas . . . . .         | 19        |
| 2.7 Real to integer conversion . . . . .          | 19        |
| 2.8 Elementary dyadic-formulas . . . . .          | 20        |
| 2.9 Real functions . . . . .                      | 22        |
| 2.10 Booleans . . . . .                           | 23        |
| 2.11 Boolean operators . . . . .                  | 24        |
| 2.12 Characters and text . . . . .                | 24        |
| 2.13 Character operators . . . . .                | 25        |
| 2.14 Comparison operators . . . . .               | 25        |
| 2.15 Program structure . . . . .                  | 27        |
| 2.16 Variables . . . . .                          | 28        |
| 2.17 Assignment . . . . .                         | 30        |
| 2.18 Operators combined with assignment . . . . . | 32        |
| 2.19 Order of evaluation . . . . .                | 32        |
| 2.20 Abbreviated variable-declarations . . . . .  | 33        |
| 2.21 Reading variables . . . . .                  | 34        |
| 2.22 References to names . . . . .                | 35        |
| 2.23 The value <code>NIL</code> . . . . .         | 36        |

## TABLE OF CONTENTS

|          |  |            |
|----------|--|------------|
| 2.24     | Comments . . . . .                             | 36         |
| 2.25     | Pragmats . . . . .                             | 37         |
| <b>3</b> | <b>Control structures</b>                      | <b>39</b>  |
| 3.1      | Introduction . . . . .                         | 39         |
| 3.2      | The conditional-clause . . . . .               | 39         |
| 3.3      | Pseudo-operators . . . . .                     | 43         |
| 3.4      | Identity relations . . . . .                   | 44         |
| 3.5      | The integer-case-clause . . . . .              | 45         |
| 3.6      | The loop-clause . . . . .                      | 47         |
| <b>4</b> | <b>Stowed modes</b>                            | <b>51</b>  |
| 4.1      | Introduction . . . . .                         | 51         |
| 4.2      | Rows . . . . .                                 | 51         |
| 4.3      | Flexible names . . . . .                       | 61         |
| 4.4      | Vectors, matrices and tensors . . . . .        | 65         |
| 4.5      | Torrax extensions . . . . .                    | 65         |
| 4.6      | A note on brackets . . . . .                   | 67         |
| 4.7      | Structured modes . . . . .                     | 67         |
| 4.8      | Field-selections . . . . .                     | 73         |
| 4.9      | Mode-declarations . . . . .                    | 74         |
| 4.10     | Complex numbers . . . . .                      | 77         |
| 4.11     | BYTES — a compact [] CHAR . . . . .            | 79         |
| 4.12     | BITS — a compact [] BOOL . . . . .             | 79         |
| <b>5</b> | <b>Routines</b>                                | <b>83</b>  |
| 5.1      | Routines . . . . .                             | 83         |
| 5.2      | Routine modes . . . . .                        | 86         |
| 5.3      | Routines without parameters . . . . .          | 87         |
| 5.4      | Routines with parameters . . . . .             | 88         |
| 5.5      | Routines and scope . . . . .                   | 91         |
| 5.6      | Declaring new operators . . . . .              | 92         |
| 5.7      | Identification of operators . . . . .          | 94         |
| 5.8      | Recursion . . . . .                            | 95         |
| 5.9      | Recursion and data structures . . . . .        | 99         |
| 5.10     | Partial parametrisation and currying . . . . . | 103        |
| <b>6</b> | <b>United modes</b>                            | <b>105</b> |
| 6.1      | United-mode-declarations . . . . .             | 105        |
| 6.2      | United modes as arguments . . . . .            | 107        |
| 6.3      | United-case-clauses . . . . .                  | 108        |
| 6.4      | Well-formed modes . . . . .                    | 110        |
| 6.5      | Equivalence of modes . . . . .                 | 113        |

|           |  |            |
|-----------|--|------------|
| <b>7</b>  | <b>Transput</b>                                    | <b>115</b> |
| 7.1       | Transput . . . . .                                 | 115        |
| 7.2       | Channels and files . . . . .                       | 116        |
| 7.3       | Transput and scope . . . . .                       | 117        |
| 7.4       | Reading files . . . . .                            | 118        |
| 7.5       | Writing to files . . . . .                         | 119        |
| 7.6       | String terminators . . . . .                       | 121        |
| 7.7       | Events . . . . .                                   | 121        |
| 7.8       | Formatting routines . . . . .                      | 124        |
| 7.9       | Straightening . . . . .                            | 126        |
| 7.10      | Default-format transput . . . . .                  | 126        |
| 7.11      | Formatted transput . . . . .                       | 128        |
| 7.12      | Binary files . . . . .                             | 138        |
| 7.13      | Using a <code>STRING</code> as a file . . . . .    | 138        |
| 7.14      | Other transput procedures . . . . .                | 139        |
| <b>8</b>  | <b>Algol 68 Genie context-free syntax</b>          | <b>141</b> |
| 8.1       | Introduction . . . . .                             | 141        |
| 8.2       | Reserved symbols . . . . .                         | 141        |
| 8.3       | Common production-rules . . . . .                  | 144        |
| 8.4       | Tags . . . . .                                     | 145        |
| 8.5       | Contexts . . . . .                                 | 146        |
| 8.6       | Coercions . . . . .                                | 147        |
| 8.7       | Particular program . . . . .                       | 151        |
| 8.8       | Clauses . . . . .                                  | 152        |
| 8.9       | Units . . . . .                                    | 152        |
| 8.10      | Enclosed-clauses . . . . .                         | 153        |
| 8.11      | Balancing . . . . .                                | 156        |
| 8.12      | Parallel processing in <code>a68g</code> . . . . . | 157        |
| 8.13      | Primaries . . . . .                                | 158        |
| 8.14      | Secondaries . . . . .                              | 164        |
| 8.15      | Tertiaries . . . . .                               | 164        |
| 8.16      | Units . . . . .                                    | 165        |
| 8.17      | Jumps . . . . .                                    | 167        |
| 8.18      | Assertions as pre- or postconditions . . . . .     | 169        |
| 8.19      | Declarations . . . . .                             | 169        |
| 8.20      | Declarers . . . . .                                | 171        |
| 8.21      | Pragments . . . . .                                | 171        |
| 8.22      | Refinements . . . . .                              | 172        |
| <b>II</b> | <b>Programming with Algol 68 Genie</b>             | <b>173</b> |
| <b>9</b>  | <b>Installing and using Algol 68 Genie</b>         | <b>175</b> |

## TABLE OF CONTENTS

|            |  |            |
|------------|--|------------|
| 9.1        | Algol 68 Genie . . . . .                             | 175        |
| 9.2        | Algol 68 Genie transput . . . . .                    | 179        |
| 9.3        | Installing Algol 68 Genie on GNU/Linux . . . . .     | 181        |
| 9.4        | Synopsis . . . . .                                   | 186        |
| 9.5        | a68g diagnostics . . . . .                           | 188        |
| 9.6        | Options . . . . .                                    | 191        |
| 9.7        | The monitor . . . . .                                | 197        |
| 9.8        | The Preprocessor . . . . .                           | 201        |
| 9.9        | A multi-pass scheme to parse Algol 68 . . . . .      | 203        |
| 9.10       | a68g source code checks . . . . .                    | 204        |
| 9.11       | Limitations and bugs . . . . .                       | 205        |
| <b>10</b>  | <b>Standard-prelude and library-prelude</b>          | <b>207</b> |
| 10.1       | The standard-environ . . . . .                       | 207        |
| 10.2       | The standard-prelude . . . . .                       | 208        |
| 10.3       | Standard modes . . . . .                             | 208        |
| 10.4       | Environment enquiries . . . . .                      | 210        |
| 10.5       | Standard operators . . . . .                         | 213        |
| 10.6       | Standard procedures . . . . .                        | 227        |
| 10.7       | Transput . . . . .                                   | 233        |
| 10.8       | The library-prelude . . . . .                        | 242        |
| 10.9       | <i>ALGOL68C</i> transput procedures . . . . .        | 242        |
| 10.10      | Drawing and plotting . . . . .                       | 244        |
| 10.11      | Mathematical functions . . . . .                     | 249        |
| 10.12      | Linear algebra . . . . .                             | 253        |
| 10.13      | Solution of linear algebraic equations . . . . .     | 257        |
| 10.14      | Fourier transforms . . . . .                         | 260        |
| 10.15      | Laplace transform . . . . .                          | 262        |
| 10.16      | Constants . . . . .                                  | 263        |
| 10.17      | Linux extensions . . . . .                           | 271        |
| 10.18      | Regular expressions in string manipulation . . . . . | 276        |
| 10.19      | Curses support . . . . .                             | 279        |
| 10.20      | PostgreSQL support . . . . .                         | 280        |
| 10.21      | PCM Sounds . . . . .                                 | 289        |
| <b>III</b> | <b>Example a68g programs</b>                         | <b>293</b> |
| <b>11</b>  | <b>Example a68g programs</b>                         | <b>295</b> |
| 11.1       | Lucas numbers . . . . .                              | 295        |
| 11.2       | Roman numbers . . . . .                              | 296        |
| 11.3       | Holidays related to Easter . . . . .                 | 297        |
| 11.4       | The Sierpinski triangle . . . . .                    | 298        |
| 11.5       | Whetstone benchmark . . . . .                        | 299        |

|           |  |            |
|-----------|--|------------|
| 11.6      | Prime numbers . . . . .                        | 301        |
| 11.7      | Hilbert matrix using fractions . . . . .       | 305        |
| 11.8      | Mastermind code breaker . . . . .              | 307        |
| 11.9      | Mandelbrot set . . . . .                       | 309        |
| 11.10     | Knight traversing a chess board . . . . .      | 310        |
| 11.11     | Decision tree . . . . .                        | 311        |
| 11.12     | Peano curve . . . . .                          | 312        |
| 11.13     | Formula manipulation . . . . .                 | 314        |
| 11.14     | Fibonacci grammar . . . . .                    | 320        |
| 11.15     | Formatting routines . . . . .                  | 321        |
| 11.16     | Anti-aliasing in drawing . . . . .             | 324        |
| 11.17     | Accessing a PostgreSQL database . . . . .      | 324        |
| 11.18     | Filter for SOUND values . . . . .              | 326        |
| 11.19     | Monte Carlo simulation . . . . .               | 327        |
| 11.20     | Hamming numbers . . . . .                      | 333        |
| <b>IV</b> | <b>Algol 68 Revised Report</b>                 | <b>335</b> |
| <b>12</b> | <b>Foreword to the translation</b>             | <b>339</b> |
| <b>13</b> | <b>Acknowledgments</b>                         | <b>341</b> |
| <b>14</b> | <b>Introduction</b>                            | <b>345</b> |
| 14.1      | Aims and principles of design . . . . .        | 345        |
| 14.2      | Comparison with Algol 60 . . . . .             | 347        |
| 14.3      | Changes in the method of description . . . . . | 352        |
| <b>15</b> | <b>Language and metalanguage</b>               | <b>357</b> |
| 15.1      | The method of description . . . . .            | 357        |
| 15.2      | Introduction . . . . .                         | 357        |
| 15.3      | Pragmatics . . . . .                           | 358        |
| 15.4      | Translations and variants . . . . .            | 372        |
| 15.5      | General metaproduction rules . . . . .         | 373        |
| 15.6      | General hyper-rules . . . . .                  | 375        |
| <b>16</b> | <b>The computer and the program</b>            | <b>379</b> |
| 16.1      | Terminology . . . . .                          | 379        |
| 16.2      | The program . . . . .                          | 395        |
| <b>17</b> | <b>Clauses</b>                                 | <b>401</b> |
| 17.1      | Closed clauses . . . . .                       | 402        |
| 17.2      | Serial clauses . . . . .                       | 403        |
| 17.3      | Collateral and parallel clauses . . . . .      | 406        |

## TABLE OF CONTENTS

|           |  |            |
|-----------|--|------------|
| 17.4      | Choice clauses . . . . .                                 | 410        |
| 17.5      | Loop clauses . . . . .                                   | 415        |
| <b>18</b> | <b>Declarations, declarers and indicators</b>            | <b>419</b> |
| 18.1      | Declarations . . . . .                                   | 419        |
| 18.2      | Mode declarations . . . . .                              | 420        |
| 18.3      | Priority declarations . . . . .                          | 421        |
| 18.4      | Identifier declarations . . . . .                        | 422        |
| 18.5      | Operation declarations . . . . .                         | 425        |
| 18.6      | Declarers . . . . .                                      | 426        |
| 18.7      | Relationships between modes . . . . .                    | 431        |
| 18.8      | Indicators and field selectors . . . . .                 | 432        |
| <b>19</b> | <b>Units</b>   | <b>435</b> |
| 19.1      | Syntax . . . . .   | 435        |
| 19.2      | Units associated with names . . . . .                    | 436        |
| 19.3      | Units associated with stowed values . . . . .            | 442        |
| 19.4      | Units associated with routines . . . . .                 | 447        |
| 19.5      | Units associated with values of any mode . . . . .       | 453        |
| <b>20</b> | <b>Specification of partial parametrization proposal</b> | <b>455</b> |
| <b>21</b> | <b>Coercion</b>  | <b>463</b> |
| 21.1      | Coercees . . . . .                                       | 463        |
| 21.2      | Dereferencing . . . . .                                  | 466        |
| 21.3      | Deproceduring . . . . .                                  | 466        |
| 21.4      | Uniting . . . . .  | 467        |
| 21.5      | Widening . . . . .                                       | 468        |
| 21.6      | Rowing . . . . .   | 470        |
| 21.7      | Voiding . . . . .  | 471        |
| <b>22</b> | <b>Modes and nests</b>                                   | <b>473</b> |
| 22.1      | Independence of properties . . . . .                     | 473        |
| 22.2      | Identification in nests . . . . .                        | 476        |
| 22.3      | Equivalence of modes . . . . .                           | 478        |
| 22.4      | Well-formedness . . . . .                                | 482        |
| <b>23</b> | <b>Denotations</b>                                       | <b>487</b> |
| 23.1      | Plain denotations . . . . .                              | 487        |
| 23.2      | Bits denotations . . . . .                               | 492        |
| 23.3      | String denotations . . . . .                             | 494        |
| <b>24</b> | <b>Tokens and symbols</b>                                | <b>497</b> |
| 24.1      | Tokens . . . . .   | 497        |
| 24.2      | Comments and pragmat . . . . .                           | 498        |



## TABLE OF CONTENTS

|           |  |            |
|-----------|--|------------|
| 24.3      | Representations . . . . .                            | 500        |
| 24.4      | The reference language . . . . .                     | 501        |
| <b>25</b> | <b>Standard environment</b>                          | <b>513</b> |
| 25.1      | Program texts . . . . .                              | 513        |
| 25.2      | The standard prelude . . . . .                       | 518        |
| 25.3      | Transput declarations . . . . .                      | 530        |
| 25.4      | The system prelude and task list . . . . .           | 603        |
| 25.5      | The particular preludes and postludes . . . . .      | 603        |
| <b>26</b> | <b>Examples</b>                                      | <b>605</b> |
| 26.1      | Complex square root . . . . .                        | 605        |
| 26.2      | Innerproduct 1 . . . . .                             | 605        |
| 26.3      | Innerproduct 2 . . . . .                             | 606        |
| 26.4      | Largest element . . . . .                            | 606        |
| 26.5      | Euler summation . . . . .                            | 607        |
| 26.6      | The norm of a vector . . . . .                       | 607        |
| 26.7      | Determinant of a matrix . . . . .                    | 608        |
| 26.8      | Greatest common divisor . . . . .                    | 609        |
| 26.9      | Continued fraction . . . . .                         | 609        |
| 26.10     | Formula manipulation . . . . .                       | 609        |
| 26.11     | Information retrieval . . . . .                      | 611        |
| 26.12     | Cooperating sequential processes . . . . .           | 613        |
| 26.13     | Towers of Hanoi . . . . .                            | 614        |
| <b>27</b> | <b>Glossaries</b>                                    | <b>615</b> |
| 27.1      | Technical terms . . . . .                            | 615        |
| 27.2      | Paranotions . . . . .                                | 622        |
| 27.3      | Predicates . . . . .                                 | 628        |
| 27.4      | Index to the standard prelude . . . . .              | 630        |
| 27.5      | Alphabetic listing of metaproduction rules . . . . . | 638        |
| <b>V</b>  | <b>Appendices</b>                                    | <b>643</b> |
| <b>A</b>  | <b>Bibliography</b>                                  | <b>645</b> |
| <b>B</b>  | <b>Reporting bugs</b>                                | <b>649</b> |
| B.1       | Have you found a bug? . . . . .                      | 649        |
| B.2       | How and where to report bugs . . . . .               | 650        |
| <b>C</b>  | <b>Release history</b>                               | <b>651</b> |
| <b>D</b>  | <b>Algol 68 Genie manual page</b>                    | <b>659</b> |

TABLE OF CONTENTS

|   |            |
|---|------------|
| <b>E GNU General Public License</b>     | <b>663</b> |
| <b>F GNU Free Documentation License</b> | <b>677</b> |

# Preface

{Les inventions qui ne sont pas connues ont toujours plus de  
censeurs que d'approbateurs.  
Lettres dédicatoires à Monsieur le Chancelier.                      Blaise Pascal. }

*Learning Algol 68 Genie* is distributed with Algol 68 Genie, an efficient open source Algol 68 hybrid compiler-interpreter that can be used for executing Algol 68 programs or scripts. Algol 68 Genie is a new implementation written from scratch, it is not a port of a vintage implementation. This book corresponds to Algol 68 Genie Version 2.0.0. Algol 68 Genie implements practically full Algol 68 as defined by the Revised Report, and extends that language to make it particularly suited to scientific computations. This book describes how to use Algol 68 Genie, as well as its features and incompatibilities, and how to report bugs. This book provides an informal introduction to Algol 68, a manual for Algol 68 Genie, and a  $\text{\LaTeX}$  translation of the Revised Report on Algol 68. The internals of Algol 68 Genie are not documented here. Algol 68 Genie is free software; *free* in the sense that it respects your freedom to run it, to study the source code and to redistribute copies of it or to publish a modified version. The license for Algol 68 Genie is the GNU GPL {see appendix E}.

Algol (short for **A**lgorithmic language) is a family of imperative computer programming languages originally developed around 1960 that became the *de facto* way algorithms were described in textbooks and academic works for essentially three decades. It was designed by a committee of research scientists to avoid some of the perceived problems with Fortran or Cobol. Algol eventually gave rise to many other programming languages, for instance BCPL, B, C, Simula and Pascal. Algol introduced a block structure (*compound statements*), nested procedures and lexical scope. The development of Algol was an international platform for discussing programming languages, compiler - and program construction, et cetera, and stimulated computer science as an academic discipline in its own right. The preservation of Algol 68 is important from both an educational as well as a scientific-historical point of view. Algol 68 has been around for four decades, but some who rediscovered it in recent years, well aware of how the world has moved on, had a feeling best described as *plus ça change, plus c'est la même chose*; presumably also because much of today's programming is still done using older third-generation languages. Though it did not spread widely in its day, Algol 68 introduced innovations that are relevant up to the

## PREFACE

present. Its expressive power, and its being oriented towards the needs of programmers instead of towards the needs of compiler writers, may explain why, since Algol 68 Genie became available under GNU GPL, many appeared interested in an Algol 68 implementation, the majority of them being mathematicians or computer scientists. Some still run proprietary implementations. Due to this continuing interest in Algol 68 it is expected that people are interested in having access to documentation on Algol 68, but nowadays most books on Algol 68 are out of print. Even if one can get hold of a book on Algol 68, it will certainly not describe Algol 68 Genie since this implementation is considerably younger than those books. The formal defining document, the Algol 68 Revised Report, is also out of print but a  $\LaTeX$  version comes with this book<sup>1</sup>. The Revised Report is known to rank among the difficult publications of computer science and is therefore not suited as an *informal* introduction. In fact, it has been said that at the time Algol 68 was presented, the complexity of the Revised Report made some people who allegedly did not use Algol 68, believe that the language itself would be complex as well. After reading this book you will likely agree that Algol 68 is a rather lean language.

This book is in the first place documentation for Algol 68 Genie; it is neither an introduction to programming nor a textbook for a course in computer science. However, Algol 68 is not commonly known, and it is necessary to include an informal introduction to the language in this documentation. I am aware that this creates some unevenness in the set-up and level of this book. If you succeed in programming in Algol 68, the objective of this book is met. I assume that you are already comfortable with computers and programming and you will not find explicit questions or problems since I assume that you are sufficiently creative to think of many *what is the outcome of this expression* or *write some program that applies what you just read* type exercises. The informal introduction precedes the  $\LaTeX$  translation of the Revised Report since the Revised Report is more accessible when one already knows the language. If one is not interested in reading the Revised Report, it can be skipped since Parts I through III are a comprehensive introduction into programming with Algol 68 Genie. The approach adopted in the informal introduction is that of other texts on programming languages that explain a minimal set of features early on with which one can program simple exercises; in due course this basis is elaborated thus dispersing the elaboration of ideas over a number of chapters. Therefore the chapters should be read in order, although it is not possible to explain Algol 68 in a strict left-to-right manner. Hence I sometimes make forward references and leave it up to you to first read the forwardly referenced material. I have consciously included mathematical content since it will either exactly specify the operation of some procedure or operator, or demonstrate some Algol 68 construction. I think that those not versed in mathematics will understand the demonstration nonetheless. This book contains, apart from original Algol 68 Genie documentation, smaller or larger parts from different open sources {see *Bibliography*} that have been edited and blended to form a consistent, new publication. Sian Leitch generously made the  $\TeX$  source of *Programming Algol 68 Made Easy* available under the GNU General Public License through which a comprehensive informal introduction to Algol 68 was at hand. The

---

<sup>1</sup>Next to this translation, other versions of the Algol 68 Revised Report are available on the internet, for instance from [Karl Kleine's collection].

material has been extensively edited and revised over a prolonged period of time as to let it describe Algol 68 Genie.

## Why use or study Algol 68?

{Lisp and Algol, are built around a kernel that  
seems as natural as a branch of mathematics.  
Metamagical Themas.                    Douglas Hofstadter. }

Algol 68 has been around for over forty years, and many are interested in the language to this day. A more or less comprehensive list of reasons for this continuing interest, and preservation of the language, would be:

- *Importance to the history of science.* As already indicated, the development of Algol played a role in establishing computer science as an academic discipline in its own right. Algol 68 was designed by a learned committee whose meeting accounts show that there was, at times vigorous, debate before Algol 68 was presented. Some members left the committee to write a minority report or to work on other approaches to the committee's targets, which led to Algol W and subsequently Pascal. The influence of Algol is still tangible since it is to this day referred to in teaching material, discussions and publications. How would you form your own opinion without knowing about Algol?
- *Academic interest.* People interested in the design and formal specification of programming languages, such as students of computer science, should at an appropriate moment study Algol 68 to understand the influence it had, irrespective of whether the language did or did not spread widely at the time it was presented. Algol 68 lives on not only in the minds of people formed by it, but also in other programming languages, even though the orthogonality in the syntax, elegance and security has been mostly lost.
- *Practical interest.* For programmers, the world has of course moved on, but the reactions to Algol 68 Genie suggest that many people who have seriously programmed in Algol 68 in the past, only moved to other programming languages because the Algol 68 implementations they were using were phased out. Some are still using proprietary implementations and one application of Algol 68 Genie is to run vintage code. But even today, Algol 68 is a beautiful means to denote algorithms and it still has its niche in programming small to medium sized applications for instance in the field of mathematics, or numerical applications in physics, chemistry et cetera, for which an imperative, procedural language is perfectly adequate and more efficient than an object oriented language. Algol 68 has high expressive power that relieves you from having to consider all kind of irrelevant technical detail inherent to programming in many other languages. You can concisely write the solution to the problem you have in mind. Besides, Algol 68 Genie offers many runtime checks, indicating in the source

## PREFACE

code where for instance a variable was used uninitialised, an expression overflowed or an index was out of bounds. Many implementations of other languages will not do that, requiring you to spend a lot of productive time on diagnosing faulty programs using a debugger and `print` statements.

## Algol 68 Genie

The language described in Parts I through III of this book is that implemented by Algol 68 Genie available from:

<http://www.xs4all.nl/~jmvdveer/algol.html>

Prebuilt binaries, for instance WIN32 binaries for Microsoft Windows XP, Vista or Windows 7, are available from:

<http://sourceforge.net/projects/algol68/algol68g>

Marcel van der Veer is the author, maintainer and copyright holder of Algol 68 Genie and its documentation. Algol 68 Genie implements almost all of Algol 68, and extends that language. To run the programs described in this book you will need a computer with Linux or compatible operating system. Chapter 9 describes how you can install Algol 68 Genie on your system, and how you can use it. Algol 68 Genie is open source software distributed under GNU GPL. This software is distributed in the hope that it will be useful, but **without any warranty**. Consult the GNU General Public License<sup>2</sup> for details. A copy of the license is in this book.

Algol 68 Genie version 1 was an interpreter. It constructed a syntax tree for an Algol 68 program and the interpreter executed this syntax tree. As of version 2 and on Linux or compatible operating systems, Algol 68 Genie can run in optimising mode, in which it employs a **unit** compiler that emits C code for many **units** involving operations on primitive modes `INT`, `REAL`, `BOOL`, `CHAR` and `BITS` and simple structures thereof such as `COMPLEX`. Execution time of such **units** by interpretation is dominated by interpreter overhead, which makes compilation of these units worthwhile. Generated C code is compiled and dynamically linked before it is executed by Algol 68 Genie. Technically, the compiler synthesizes per selected **unit** efficient routines compounding elemental interpreter routines needed to execute terminals in the syntax tree; compounding allows for instance common sub-expression elimination. Generated C code is compatible with the virtual stack-heap machine implemented by the interpreter proper, hence generated code has full access to a68g's runtime library and the interpreter's debugger. Many runtime checks are disabled in optimising mode for the sake of efficiency. Therefore, it is recommended to only specify optimisation for programs that work correctly. Due to overhead, optimisation is not efficient for programs with short execution times, or *run-once* programs typical for programming course exercises.

---

<sup>2</sup>See <http://www.gnu.org/licenses/gpl.html>.

Both Algol 68 Genie (a68g) and its documentation are updated frequently. New releases are announced on next usegroups:

`comp.compilers` • `comp.os.linux.announce`

and also on the Algol 68 user group at LinkedIn:

<http://www.linkedin.com/groups?gid=2333923>

## Conventions in this book

Algol 68 source code is typeset in fixed-space font like this:

```
#
Takeuchi's Tarai (or Tak) function. Moore proved its termination.
See http://mathworld.wolfram.com/TAKFunction.html
#
PROC tak = (INT i, j, k) INT:
  IF i <= j
  THEN j
  ELSE tak (tak (i - 1, j, k), tak (j - 1, k, i), tak (k - 1, i, j))
  FI;
```

Sometimes code is substituted with ... when it would not be relevant to the explanation at hand, as in for instance:

```
PROC tak = (INT i, j, k) INT: ...
```

In this book, a68g output is typeset as:

```
$ a68g hello.a68
Hello, world!
```

This book contains references that are listed in the *Bibliography*. A format as [Mailloux 1978] means the entry referring to work of Mailloux, published in the year 1978. An indication AB39.3.1 means ALGOL BULLETIN, volume 39, article 3.1. The ALGOL BULLETIN is still available on the internet. On various places in Parts I through III you will see a reference to the Revised Report in Part IV formatted as for example {25<sub>10</sub>.1.1.A} referring you to chapter 25, section 1.1(.1), mark A, where the chapter number refers to Part IV, while its suffix refers to the chapter number in the original Revised Report.

## Organisation of this book

### Part I. Informal introduction to Algol 68

- Chapter 1 *Preliminaries* gives a brief history of Algol 68 and introduces a notation for production rules.
- Chapter 2 *Basic concepts* introduces standard modes representing plain values (integers, reals, booleans and characters), as well as variables. Basic language constructs are introduced such as **closed-clauses**. This chapter also explores **formulas** involving **operands** of standard modes.
- Chapter 3 *Program structure* describes conditional and case constructs that let you control program flow depending on the value of boolean or integer conditions. It also describes loops.
- Chapter 4 *Stowed modes* describes ordered sets of values: rows and structures. It explains how to extract sub rows from a row, how to select a diagonal in a matrix, et cetera. This chapter also shows how to group objects into structures. `STRING` and `COMPLEX` are introduced.
- Chapter 5 *Routines* explains how to declare procedures and operators, both having a **routine-text** as "**denotation**". This chapter brings together recursion and data structures and is a demonstration of Algol 68's expressive power. This chapter also describes recursion and partial parametrisation. `a68g` is the only Algol 68 implementation to date to implement partial parametrisation.
- Chapter 6 *United modes* completes the description of constructing modes by introducing the united mode. It also gives the rules for a well formed mode.
- Chapter 7 *Transput* is about transput which is an Algol 68 term for input output. Formatted transput is described in this chapter.
- Chapter 8 *Algol 68 Genie context free syntax* provides a reference for Algol 68 Genie syntax. It also describes some constructions not yet treated in earlier chapters.

### Part II. Programming with Algol 68 Genie

- Chapter 9 *Installing and using Algol 68 Genie* describes the Algol 68 Genie (`a68g`), how to install it on your computer system and how to use the program.
- Chapter 10 *Standard-prelude and library-prelude* is an extensive description of the standard-prelude and library-prelude. Standard Algol 68 predefines a plethora of operators and procedures. Algol 68 Genie predefines many operators and procedures in addition to those required by the standard-prelude, that form the library-prelude. This chapter documents these extensions.



**Part III. Example a68g programs**

- Chapter 11 *Example programs* lists a number of a68g **programs** to demonstrate the material covered in this book.

**Part IV. Algol 68 Revised Report**

- Chapters 12—27 constitute a L<sup>A</sup>T<sub>E</sub>X translation of the Revised Report on Algol 68. This report ranks among the difficult publications in computer science.

**Part V. Appendices**

- Appendix A *Bibliography* has references and suggestions for further reading.
- Appendix B *Reporting bugs* gives information on how and where to report bugs in Algol 68 Genie or in this book.
- Appendix C *Release history* describes the history of Algol 68 releases.
- Appendix D *Algol 68 Genie manual page* is the *BSD* a68g manual page.
- Appendix E *GNU General Public License* is a copy of a68g's license.
- Appendix F *GNU Free Documentation License* is a copy of the license for parts I, II and IV of this book.

## Acknowledgements

Thanks go to the following people, in alphabetical order, who were kind enough to report bugs and obscurities, to propose improvements, provide documentation, or to make a contribution in another way: Mark Alford, Bruce Axtens, Ewan Bennett, Lennart Benschop, Jaap Boenders, Bart Botma, Paul Cockshott, Huw Davies, Neville Dempsey, Koos Der- ing, Jón Fairbairn, Scott Gallaher, Jeremy Gibbons, Mayer Goldberg, Dick Grune, Nor- man Hardy, Chap Harrison, Jim Heifetz, Lex Herrendorf, Patrik Jansson, Helmut Ja- rausch, Trevor Jenkins, James Jones, Rob Jongschaap, Henk Keller, Wilhelm Kloke, Has van der Krieken, Sian Leitch, Karolina Lindqvist, Charles Lindsey, Paul McJones, Ray- mond Nijssen, Richard O’Keefe, Lawrence D’Oliveiro, France Pahlplatz, Steven Pember- ton, Richard Pinch, Henk Robbers, Pedro Rodrigues de Almeida, Rubin Simons, Robert Uzgalis, Bruno Verlyck, Nacho Vidal García, Merijn Vogel, Eric Voss, Theo Vosse, Andy Walker, Jim Watt, Glyn Webster, Sam Wilmott and Tom Zahm.

Algol 68 Genie would not be what it is without their help.

Marcel van der Veer  
Uithoorn, 6 September 2010

## Biography

Marcel van der Veer is the author, maintainer and copyright holder of Algol 68 Genie and its documentation. He holds a MSc in Chemistry from the University of Nijmegen and a PhD in Applied Physics from the University of Twente. During his academic years, he worked with *ALGOL68C* and *FLACC* on IBM and compatible mainframes, and also with *ALGOL68RS* on large VAXen. Probably due to the fact that in chemistry - and physics departments people tend to take a pragmatic approach towards computer science, Marcel was undeterred to write his own Algol 68 implementation when Algol 68 implementations were phased out as mainframes were replaced by UNIX workstations which offered C, Fortran or Pascal, but not Algol 68. Marcel started development of Algol 68 Genie in 1992, and after encouragement by some people — especially Henk Keller — he decided to release Algol 68 Genie under GNU GPL in 2001.

# **Part I**

## **Informal introduction to Algol 68**



# Chapter 1

## Preliminaries

{Languages take such a time, and so do  
all the things one wants to know about.  
The Lost Road. John Tolkien. }

### 1.1 A brief history of Algol 68

High level languages are expected to serve two purposes. They should provide concepts and statements allowing a precise formal description of computing processes and facilitate communication between programmers, and they should provide a tool to solve small to medium-sized problems without specialist help. Early high level procedural languages succeeded in this respect, though the evolving need to build complex interactive systems lead to decomposition of a problem into "natural" components, resulting in object oriented programming starting with Simula in the 1960's. The object oriented and procedural paradigms each have strengths and weaknesses and sometimes it is not clear which paradigm is best suited to certain tasks, even large ones — for instance in numerical and scientific computing the benefit of object oriented languages over procedural languages as Algol 68 is controversial. Ideally, a programming language supports systematic expression of algorithms by offering appropriate control structures and data structures, and a precise, consistent formal definition to avoid surprises and portability issues resulting from obscure details that are left to the discretion of an implementation; C for example, is notorious in this respect. Members of the Algol family (Algol 60 and Algol 68, Simula, Pascal and also Ada, et cetera) are considered reasonable approximations of such "ideal" languages, although all of them have strong points as well as disadvantages. Algol 68 offers appropriate means of abstraction and exemplary control structures that leads to a good understanding of programming. Its orthogonality results in an economic use of language constructs making it a beautiful means to write algorithms.

## INFORMAL INTRODUCTION TO ALGOL 68

Algol, **algorithmic language**, is a family of imperative computer programming languages which greatly influenced many other languages and became the *de facto* way algorithms were described in textbooks and academic works for almost three decades. The two specifications relevant to this book are:

*Algol 60*, revised in 1963 • *Algol 68*, revised in 1976.

Algol 58, originally known as IAL (*International Algebraic Language*), was an early member of the Algol family soon superseded by Algol 60. Algol 58 introduced the compound statement, but it was restricted to control flow only, and not tied to identifier scope in the way that Algol 60's blocks are. Algol 60 is said to be a product of transatlantic scientific cooperation around 1960 and was designed by and for numerical mathematicians. In its day Algol 60 was the *Lingua Franca* of computer science. The development of Algol became an international platform for discussions about programming languages, compiler construction, program construction, et cetera, and acted as a catalyst in erecting computer science as an academic discipline in its own right. Algol 60 had systematic block structure and a concise definition that were appreciated by people with a background in mathematics, but it lacked compilers and industrial support which gave the advantage to languages as Fortran and Cobol. To promote Algol, its application range had to be extended. IFIP<sup>1</sup> Working Group 2.1 *Algorithmic Languages and Calculi* (WG 2.1), that to this day has continuing responsibility for Algol 60 and Algol 68, assumed the task of developing a successor to Algol 60. During the project, WG2.1 split into two camps — an ambitious camp that wanted another milestone in language design, leading to Algol 68, and another camp that wanted to save time by adequately extending Algol 60, which eventually produced Pascal. Hence the design of Algol 68 was firmly rooted in the *computing community*, a contemporary term for the small but growing international community of computer professionals and scientists shaping computer science.

In the early 1960's WG 2.1 discussed a successor to Algol 60 and in 1965 descriptions of a language Algol X based on these discussions were invited. This resulted in various papers by Wirth, Seegmüller and Van Wijngaarden<sup>2</sup> that described more or less complete languages and other significant contributions by Hoare and Naur. Van Wijngaarden's paper *Orthogonal design and description of a formal language*<sup>3</sup> featured a new technique for language design and definition and formed the basis for what in the course of further meetings would develop into Algol 68. Many features found in Algol 68 were first proposed in ALGOL BULLETIN by the original authors of Algol 60 like Peter Naur, by new members of WG

---

<sup>1</sup>IFIP, the International Federation for Information Processing is an umbrella organisation for national information processing organisations. It was established in 1960 under the auspices of UNESCO.

<sup>2</sup>Adriaan van Wijngaarden (1916 - 1987) is considered by many to be the founding father of computer science in The Netherlands. He was co-founder of IFIP and one of the designers of Algol 60 and later Algol 68. As leader of the Algol 68 committee, he made a profound contribution to the field of programming language design, definition and description.

<sup>3</sup>Available from [Karl Kleine's collection].

2.1 like Tony Hoare and Niklaus Wirth<sup>4</sup>, and by many others from the world-wide computing community. [Koster 1996] gives a first hand account of the events leading to Algol 68. Algol 68 has had a large influence on the development of programming languages. It introduced new ideas, for example orthogonality, a strong type system, procedures as types, memory allocation, treatment of arrays, a rigorous description of syntax, and parallel processing, but also ideas that have caused debate over the years such as context dependent coercions and quite complicated input-output formatting.

Algol 68's syntax is described by a two-level W-grammar ('W' for Van Wijngaarden) that can define a potentially infinite grammar in a finite number of rules. It can rigorously define some syntactic restrictions that otherwise have to be formulated in natural language, despite their essentially syntactical content; for example, demanding that **applied-identifiers** or operators be declared, or demanding that modes result in finite objects that require finite coercion, et cetera. In that respect, a context-free syntax needs extra rules formulated in natural language to reject incorrect **programs**. This is less elegant, but defining documents for programming languages with a context-free grammar indeed look less complex than the Algol 68 (Revised) Report — see for instance the context-free grammar of Algol 68 Genie in chapter 8. Probably because of the formal character of the Revised Report, which requires study to comprehend, the misconception got hold that Algol 68 is a complex language, while in fact it is rather lean. [Koster 1996] states that the alleged obscurity of description is denied by virtually anyone who has studied it. Algol 68 was defined in a formal document, first published in January 1969, and later published in Acta Informatica and also printed in Sigplan Notices. A Revised Report was issued in 1976. This book includes a L<sup>A</sup>T<sub>E</sub>X translation of the Revised Report. Algol 68 was the first major language for which a full formal definition was made before it was implemented. Though known to be terse, the Revised Report does contain humour *solis sacerdotibus* — to quote from [Koster 1996]: *The strict and sober syntax permits itself small puns, as well as a liberal use of portmanteau words. Transput is input or output. 'Stowed' is the word for structured or rowed. Hipping is the coercion for the hop, skip and jump. MOID is MODE or void. All metanotions ending on ETY have an empty production. Just reading aloud certain lines of the syntax, slightly raising the voice for capitalized words, conveys a feeling of heroic and pagan fun (. . .) Such lines cannot be read or written with a straight face.*

Algol 68 was designed for programmers, not for compiler writers, in a time when the field of compiler construction was not as advanced as it is today. Implementation efforts based on formal methods generally failed; Algol 68's context-dependent grammar required some invention to parse<sup>5</sup>; consider for instance  $\times (y, z)$  that can be either a **call** or a **slice** depending on the mode of  $\times$ , while  $\times$  does not need to be declared before being applied. At the time of Algol 68's presentation compilers usually were made available on mainframes by computing centres, which may explain why Algol 68 was popular in locations rather than areas, for instance Amsterdam, Berlin or Cambridge. It appears that Algol 68 was relatively popular in the United Kingdom, where the *ALGOL68R*, *ALGOL68RS* and *AL-*

---

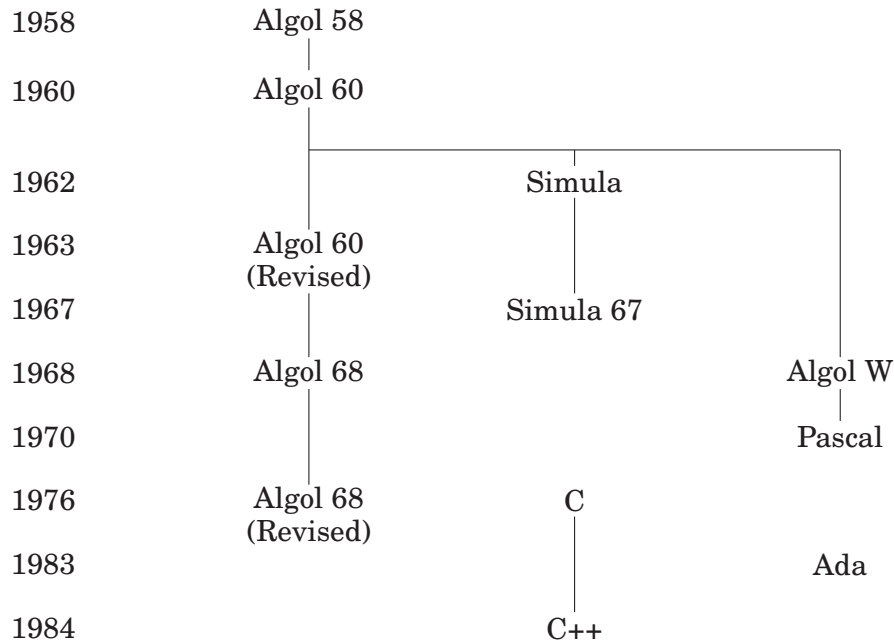
<sup>4</sup>Wirth left WG 2.1 to work with Hoare on Algol W which influenced the later language Pascal.

<sup>5</sup>Algol 68 Genie employs a multi-pass scheme to parse Algol 68 [Lindsey 1993] {see 9.9}.

## INFORMAL INTRODUCTION TO ALGOL 68

*GOL68C* compilers were developed. Hence Algol 68 compilers were few and initiatives to commercialise them were relatively unsuccessful; for instance the *FLACC* compiler sold just twenty-two copies<sup>6</sup>. In the end industry did not pick it up — the market for new universal programming languages evidently did not develop as hoped for during the decade in which the language was developed and implemented. Algol 68 was not widely used, though the influence it had on the development of computer science is noticeable to this day. Interestingly, two other members of the Algol family, Pascal and Ada, still have their niches but also did not spread as widely as some may have hoped.

Algol 68 can be placed in the history of programming languages more or less as in this diagram:



Note that some languages like Euler are not mentioned in this diagram. Some claim that Ada is Algol 68's successor but many dispute that. Therefore Ada is mentioned in above diagram, but there is no line drawn from Algol 68 to Ada. An overview of the development of Algol, and implementations, can be found at Paul McJones's page:

<http://www.softwarepreservation.org/projects/ALGOL/>

## 1.2 Terminology

---

<sup>6</sup>Source: Chris Thomson, formerly with Chion Corporation, on `comp.lang.misc` [1988].



Algol 68 terminology tends to be pedantic. This is a consequence of the rigorous and precise description of Algol 68. Some Algol 68 terms for still existing concepts have not come into common usage:

- *mode* for *type*
- *multiple value* for *row*, *array*  
The term *multiple value* is not used in Part I, the term *row* is used instead.
- *name* for *variable*
- *scope* for *extent* (a dynamic concept)
- **phrase** for *statement*
- *range*, *reach* for *scope* (a static concept)
- *routine* for *procedure*
- *range* for *block*
- **identity-declaration** for *constant declaration*

### 1.3 Notation of production-rules

In Part I, a method from Van Wijngaarden is used to write production-rules that describe Algol 68 Genie syntax. A similar notation for production-rules is used in the Revised Report and in [Koster 1978, 1981]. A production-rule consists of the following items, in order:

- an optional asterisk ;
- a non-empty notion  $N$  ;
- a **colon-symbol** ;
- a non-empty sequence of alternatives for  $N$  separated by **semicolon-symbols**;  
within an alternative, a **comma-symbol** means "is followed by".
- a **point-symbol**.

If a notion is preceded by an optional asterisk, the notion is not used in other production-rules and is used as an abstraction for its alternatives, for example:

- **\*operand: monadic-operand; dyadic-operand.**

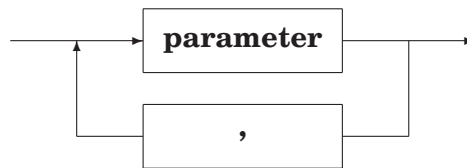
Production-rules can be parametrised: an upper-case word is substituted throughout the production-rule. For example, considering the production-rules:

- **NOTION-option: NOTION; EMPTY.**
- **NOTION-list: NOTION; NOTION, comma-symbol, NOTION-list.**

we can write **parameter-list-option**, where we will substitute **NOTION** for **parameter-list** and **parameter** respectively to obtain:

- **parameter-list-option: parameter-list; EMPTY.**
- **parameter-list: parameter; parameter, comma-symbol, parameter-list.**

From which we see that a **parameter-list-option** is possibly empty, or possibly contains one **parameter** or multiple **parameters** separated by **comma-symbols**. Typically, in C and Pascal documentation, graphic syntax-diagrams are used to describe syntactic constructs; for instance a **parameter-list** would be depicted as:



Common production-rules used in Part I are:

- **EMPTY: .**  
This is an empty production.
- **NOTION-list: NOTION; NOTION, comma-symbol, NOTION-list.**
- **NOTION-list-proper: NOTION, comma-symbol, NOTION-list.**  
A **list-proper** contains at least two **NOTIONs**.
- **NOTION-option: NOTION; EMPTY.**
- **NOTION-sequence: NOTION; NOTION, NOTION-sequence.**
- **NOTION-series: NOTION; NOTION, semicolon symbol, NOTION-sequence.**
- **length: long-symbol-sequence; short-symbol-sequence.**
- **qualifier: heap-symbol; new-symbol; loc-symbol.**
- **sign: plus-symbol, minus-symbol.**

This way of specifying syntax follows the notation used in Part IV, the Algol 68 Revised Report {15<sub>1</sub>.3.2.2}. However, the syntax rules in Part I are simple context free rules, while the Revised Report describes a two level grammar. Throughout part I, concepts for which a production-rule is listed in chapter 8, are typeset in bold face, for instance **integer-denotation** or **call**. To write a plural form of a production rule, the letter **s** is appended to its singular form, for instance **identifiers**. At the beginning of a sentence, the initial letter of a lower-case production rule is capitalised, for instance **Formulas**.

{Die Grenzen der Sprache  
sind die Grenzen der Welt.  
Ludwig Wittgenstein. }

# INFORMAL INTRODUCTION TO ALGOL 68

# Chapter 2

## Basic concepts

{Ihr behandelt die Weltgeschichte wie ein  
Mathematiker die Mathematik, wo es nur Gesetze und  
Formeln gibt, aber keine Wirklichkeit, kein Gut und  
Böse, keine Zeit, kein Gestern, kein Morgen,  
nur eine ewige, flache, mathematische Gegenwart  
Hermann Hesse, Das Glasperlenspiel. }

### 2.1 Modes and values

Two basic concepts in Algol 68 are *value* and *mode*. In other programming languages a *mode* might be called a *type*. Plain values are numbers, characters and truth-values, but you will see that values can be compounded, for instance into text or a matrix of complex numbers. In Algol 68, the representation of a plain value is called a **denotation**. In other programming languages a **denotation** might be called a *constant*. For example, the number with **denotation** 1 has mode INT (obviously meaning integer), the number with **denotation** 1.0 has mode REAL, the truth-value with **denotation** TRUE has mode BOOL and the character with **denotation** "a" has mode CHAR. Text, which is a row of characters, is so common that this is the only compounded mode with a **denotation**, for instance "denotation". Algol 68 provides a list-like construct, the **collateral-clause**, to write values for other compounded modes. For example, the value of a complex number might be written as (0.5, -0.5) which represents the value  $\frac{1}{2} - \frac{i}{2}$ .

I immediately introduce two notable **denotations**, which will not be used until later chapters. In Algol 68 there is only one **denotation** for a *name*, a notion that in another programming language might be called a *pointer*, which is NIL. Compare this to NULL in C or

NIL in Pascal. The mode of NIL depends on the context. In Algol 68 VOID has different status than a mode, though it has a single **denotation** EMPTY.

The symbols INT, REAL, BOOL and CHAR are examples of **mode-indicants** in Algol 68. A **mode-indicant** might be called a *type identifier* in other programming languages. A **mode-indicant** is an example of a **declarer** that specifies a mode. **Mode-indicants** are written in capital letters and can be as long as you like; no intervening spaces are allowed, though a68g allows intervening underscores that are part of the **mode-indicant**.

## 2.2 Integers

In Algol 68 whole numbers, or *integers*, are elements of  $\mathbb{Z}$ , but not the other way round: not all elements of  $\mathbb{Z}$  are integers since a computer is a finite object. One pre-defined **identifier**, in what is called the standard-prelude {see 10.4}, is `max int` that represents the largest representable integer on the platform on which your program runs. Compare `max int` to the pre-defined constant `INT_MAX` in C. Sometimes you need to use integer values larger than `max int`. To that end Algol 68 Genie supports modes `LONG INT` and `LONG LONG INT`. In a68g, the range of `LONG LONG INT` is default twice the length of `LONG INT` but can be made arbitrarily large through the option `precision` {see 9.6.4}. These are the respective maximum values for the three integer lengths available in a68g:

| Identifier                     | Value         | Remarks           |
|--------------------------------|---------------|-------------------|
| <code>max int</code>           | $2^{31} - 1$  | 32-bit Pentium    |
| <code>long max int</code>      | $10^{35} - 1$ |                   |
| <code>long long max int</code> | $10^{70} - 1$ | Default precision |

As in any programming language, you write the **denotation** of an integer in Algol 68 as a sequence of digits 0 to 9. In Algol 68, an **integer-denotation** is unsigned; it is in  $\mathbb{W}$ , not in  $\mathbb{Z}$ . A **sign** is a **monadic-operator**, so if you write `-1` or `+1` you will have written a **formula**, {see 2.5}. In standard Algol 68, a **denotation** for `LONG INT` must be preceded by the reserved word `LONG` and a **denotation** for `LONG LONG INT` must be preceded by the reserved words `LONG LONG`. The production-rule for an **integer-denotation** is:

- **integer-denotation:**  
**length-option, digit-sequence.**

For instance:

- a) `6048000 • 6 048 000`
- b) `LONG 266716800000 • LONG 266 716 800 000`

c) LONG LONG

```
3930061525912861057173624287137506221892737197425280369698987
```

with value (c) being  $3^{127}$ . Note that white space has no meaning in an **integer-denotation**. `a68g` relaxes the use of LONG prefixes when the context imposes a mode for a **denotation**, in which case a **denotation** of a lesser precision is automatically promoted to a **denotation** of the imposed mode.

The procedure `print` will print among others an integer argument, per default to standard output, as in:

a) `print (max int)`

b) `print (LONG 281 474 976 710 656)`

Of course `print` can print multiple arguments in a single **call**. When printing (or reading) multiple arguments, the arguments have to be enclosed in parentheses, as in:

```
print ((max int, LONG 281474976710656))
```

The reason for this peculiarity is that `print` takes as argument a row of printable values. In chapter 4 you will see that a row of more than one value is denoted by writing the values as a list enclosed by either `( ... )` or `BEGIN ... END`. The Algol 68 term for an enclosed list of **units** is **collateral-clause** but when we use it in this book to denote a row, we will call it a **row-display**. Finally, if you want to continue printing on a new line, you specify `new line`, as in:

```
print ((max int, new line, LONG 281474976710656))
```

## 2.3 Identity-declarations

Suppose you want to use the integer 10000 in various parts of your program. Algol 68 provides a construct, an **identity-declaration**, that declares a synonym for a value (in this case, an **integer-denotation**). Similar constructions in other languages are `CONST` declarations in Pascal, `PARAMETER` statements in Fortran, or `#define` directives in the C preprocessor. This is an **identity-declaration** for the above mentioned integer:

```
INT measurements done = 10 000
```

In Algol 68, no white space is required when no ambiguity exists in the meaning of concatenated terms, so you could write:

```
INTmeasurementsdone=10000
```

but it is of course good practice to use white space to improve clarity.

An **identity-declaration** is defined as:

- **identity-declaration:**  
**formal-declarer, identity-definition-list.**
- **identity-definition:**  
**identifier, equals-symbol, strong-unit.**

A **mode-indicant** can be used as a **formal-declarer**. The **formal-declarer** cannot be `VOID`. The difference between a **formal-declarer** and an **actual-declarer** will be explained in chapter 4. An **identifier** is a sequence of one or more characters which starts with a lower-case letter and continues with lower-case letters or digits:

- **identifier:**  
**letter;**  
**identifier, letter;**  
**identifier, digit;**  
**identifier, underscore-symbol-sequence, identifier.**

An **identifier** can be interrupted by spaces or tab characters, but those are ignored. Hence `maxint` is the same **identifier** as `max int`. `a68g` allows underscores in **identifiers**, but an underscore is part of the **identifier** unlike white space. Examples of valid **identifiers** are:

`i • rate 2 pay • eigen value 3`

The following are not **identifiers**:

- a) `3d vector`
- b) `particle-energy`
- c) `rootSymbolPointer`

Example (a) starts with a digit, (b) contains a character which is neither a letter nor a digit nor an underscore, and (c) contains capital letters. An **identifier** looks like a name, in the ordinary sense of that word, but we do not use the term "name" because it has a special meaning in Algol 68 which will be explained later on.

The right hand side of the **equals-symbol** in an **identity-declaration** is a **unit** yielding a value. You will see that the **unit** can be any program text which yields a value of the mode specified by the **mode-indicant**. A **denotation** is an example of a **unit**. Other **units** yielding integers are the routines<sup>1</sup> `read int`, `read long int` and `read long long`

---

<sup>1</sup>These **identifiers** come from *ALGOL68C*.



`int` that yield an integer value read from standard input; if you did not redirect input, this would be your keyboard. In chapter 5 you will see that above mentioned **identifiers** `print`, `read int` et cetera actually are procedures. Note that you cannot write:

```
INT i = INT j = 1
```

since an **identity-declaration** is not a **unit** and cannot yield a value.

There are two ways of declaring **identifiers** for two integers. The first way is to simply write them both in full:

```
INT i = 1; INT j = read int
```

The **semicolon-symbol** `;` is called the **go-on-symbol**. We can abbreviate the above **declarations** as follows:

```
INT i = 1, j = read int
```

The **comma-symbol** separates the two **declarations**, but this does not mean that `i` is declared first, followed by `j`. It is up to `a68g` to determine which **declaration** is elaborated first. They could even be done in parallel. This is known as collateral elaboration, as opposed to sequential elaboration determined by the **go-on-symbol** (the **semicolon-symbol**). Therefore you cannot always combine two **identity-declarations** into one with a **comma-symbol** as in:

```
INT one = 1, start = one
```

because you do not know which **identity-declaration** gets elaborated first as the **comma-symbol** implies collateral elaboration. If `a68g` executes a **declaration** as the one above, it may or may not end in a runtime error since an uninitialised **identifier**, in casu `one` before it is associated with `1`, is being used.

## 2.4 Reals

In Algol 68 real numbers are elements of  $\mathbb{R}$ , but not the other way round: not all elements of  $\mathbb{R}$  are reals since a computer is a finite object; real values have to be represented with a finite number of bits and therefore have finite precision and finite magnitude. The smallest `REAL` which `a68g` can handle is declared in the standard-prelude as **identifier** `min real`. The largest `REAL` which `a68g` can handle is declared as **identifier** `max real` in the standard-prelude. Compare these **identifiers** to `DBL_MIN` and `DBL_MAX` in C. Also, there is an **identifier** `small real` that gives the smallest value that when added to `1.0`, yields a value larger than `1.0`, and thus is a measure of precision. As with integers, sometimes you need to use real values with higher precision than offered by `REAL`. Algol 68 Genie supports modes `LONG REAL` and `LONG LONG REAL`. In `a68g` the precision of `LONG LONG REAL` is default circa twice that of `LONG REAL` but can be made arbitrarily large through

## INFORMAL INTRODUCTION TO ALGOL 68

the option `precision` {see 9.6.4}. These are the respective limiting values for the three real lengths available in `a68g`, which were chosen under the observation that most multi-precision applications require 20-60 significant digits:

| Identifier           | Value                               | Remarks           |
|----------------------|-------------------------------------|-------------------|
| max real             | $1.79769313486235 \times 10^{308}$  | Pentium           |
| long max real        | $1 \times 10^{999999}$              |                   |
| long long max real   | $1 \times 10^{999999}$              |                   |
| min real             | $2.22507385850720 \times 10^{-308}$ | Pentium           |
| long min real        | $1 \times 10^{-999999}$             |                   |
| long long min real   | $1 \times 10^{-999999}$             |                   |
| small real           | $2.22044604925031 \times 10^{-16}$  | Pentium           |
| long small real      | $1 \times 10^{-28}$                 |                   |
| long long small real | $1 \times 10^{-63}$                 | Default precision |

The production-rules for a **real-denotation** read:

- **real-denotation:**  
**length-option, digit-sequence, exponent-part;**  
**length-option, digit-sequence-option, point-symbol, digit-sequence,**  
**exponent-part-option.**
- **exponent-part:**  
**letter-e-symbol, sign-option, digit-sequence.**

A **real-denotation** consists of **digits** followed by at least either a fractional part **point-symbol, digit-sequence** or an **exponent-part**. As usual, `e` is the times ten to the power symbol; for example `9e-9` means  $9 \times 10^{-9}$ . **Real-denotations** are unsigned, as are **integer-denotations**, but the exponent can be preceded by a **sign**<sup>2</sup>. In standard Algol 68, a **denotation** for `LONG REAL` must be preceded by the reserved word `LONG` and a **denotation** for `LONG LONG REAL` must be preceded by the reserved words `LONG LONG`. Example **real-denotations** are:

- a) `.5` • `0.5` • `5.0e-1` • `5e-1`
- b) `LONG 2.718281828459045235360287471`
- c) `LONG LONG`  
`0.707106781186547524400844362104849039284835937688474036588339869`

with value (b) representing `e` and value (c) representing  $\frac{1}{2}\sqrt{2}$ . As with **integer-denotations**, `a68g` relaxes the use of `LONG` prefixes when the context imposes a mode for a **denotation**,

<sup>2</sup>One of the minor difficulties with Algol 68 is that in `INT i = -9`, the `-` means the **monadic-operator**, which could have been user-defined, whereas in `REAL x = 1e-9`, the `-` is the mathematical minus-sign, even if the **monadic-operator** `-` has been re-defined.

in which case a **denotation** of a lesser precision is automatically promoted to a **denotation** of the imposed mode.

Example **identity-declarations** for values of mode REAL are:

```
REAL e = 2.718 281 828,
      electron charge = 1.6021e-19,
      mils added = 1.0
```

Since a68g admits the indicant DOUBLE for LONG REAL, you could also write:

```
DOUBLE pi times 2 = 6.283185307179586476925286767
```

The value of  $\pi$  is declared in the standard-prelude with the **identifier** pi with the three precisions provided:

- REAL pi = 3.14159265358979
- LONG REAL long pi = 3.141592653589793238462643383
- LONG LONG REAL long long pi =  
3.14159265358979323846264338327950288419716939937510582097494459

The length of LONG LONG modes can be made arbitrarily large through the option precision {see 9.6.4}. So you can easily print a thousand digits of  $\pi$  through:

```
$ a68g -p "long long pi" --precision=1000
+3.141592653589793238462643383279502884197169399375105820974944592307816406
286208998628034825342117067982148086513282306647093844609550582231725359408
128481117450284102701938521105559644622948954930381964428810975665933446128
475648233786783165271201909145648566923460348610454326648213393607260249141
273724587006606315588174881520920962829254091715364367892590360011330530548
820466521384146951941511609433057270365759591953092186117381932611793105118
548074462379962749567351885752724891227938183011949129833673362440656643086
021394946395224737190702179860943702770539217176293176752384674818467669405
132000568127145263560827785771342757789609173637178721468440901224953430146
549585371050792279689258923542019956112129021960864034418159813629774771309
960518707211349999998372978049951059731732816096318595024459455346908302642
522308253344685035261931188171010003137838752886587533208381420617177669147
303598253490428755468731159562863882353787593751957781857780532171226806613
00192787661119590921642019893809526e +0
```

It was mentioned above that in an **identity-declaration**, the **unit** must yield a value of the mode required by the **declarer**. Now consider this example **identity-declaration** where the **unit** yields a value of mode INT:

```
REAL z = read int
```

However, the mode required by the **declarer** is REAL. Depending on the context, in Algol 68 a value can change mode through a small set of implicit coercions. There are five

contexts in Algol 68: *strong*, *firm*, *meek*, *weak* and *soft*. The right-hand side of an **identity-declaration** is a strong context. In a strong context, the mode of a **unit** is always imposed (in this case by the **formal-declarer** on the left-hand side). One of the strong coercions is widening that can for instance widen a value of mode `INT` to a value of mode `REAL`.

The procedure `print` will print a real argument, per default to standard output, as in:

```
print (pi) • print (LONG 1.732050807568877293527446342)
```

`a68g` implements the routines `read real`, `read long real` and `read long long real` that yield a real value read from standard input, so you may write:

```
REAL z = read real;
```

On a right-hand side of an **identity-declaration**, the strong context forces the routine `read real` to yield a real value by a coercion called *deproceduring*. *Deproceduring* will be discussed at length in section 8.6.1.

## 2.5 Formulas

**Formulas**, often called *expressions* in other programming languages, consist of operators working on **operands**. Operators are encapsulated algorithms that compute a value from their **operands**. In chapter 5, we will look at operators in more detail, as well as how to define new ones. Algol 68 provides a rich set of pre-defined operators in the standard-prelude, described in chapter 10, and you can define as many more as you need. In this chapter, we will examine the operators in the standard-prelude which can take **operands** of mode `INT`, `REAL`, `BOOL` or `CHAR`. The syntax for **formulas** reads:

- **formula:**  
     **monadic-operator-sequence, monadic-operand;**  
     **dyadic-operand, dyadic-operator, dyadic-operand.**
- **monadic-operand:**  
     **secondary;**
- **dyadic-operand:**  
     **monadic-operator-sequence-option, monadic-operand;**  
     **formula.**
- **\*operand: monadic-operand; dyadic-operand.**

**Secondaries** {see 8.14} are **operands** in **formulas**. Operators come in two forms: **monadic-operators** and **dyadic-operators**. **Operator-symbols** are written as a combination of one or more special symbols, or in upper-case letters like a **mode-indicant**. A **formula**

can be the **unit** of an **identity-declaration**. Thus the following **identity-declarations** are both valid:

```
REAL x = read real + 1.0, y = ABS sin (2 * pi * x)
```

White space is not significant in a **formula** as long as it has a unique meaning. However, an operator cannot contain white space, in contrast to an **identifier**. The reason for this is that in Algol 68, adjacent **identifiers** have no meaning but adjacent **operators** do.

## 2.6 Elementary monadic-formulas

A **monadic-operator** has only one **operand**, while a **dyadic-operator** has two **operands**. A **monadic-operator** precedes its **operand**. For example, the monadic minus  $-$  reverses the **sign** of its **operand**:  $-k$ . There is, likewise, a monadic  $+$  operator which just returns its **operand**:  $+k$ . We say that **monadic-operators**  $-$  or  $+$  take an **operand** of mode `INT` and yield a value of mode `INT`. They can also take an **operand** of mode `REAL` in which case they will yield a value of mode `REAL`. The operator `ABS` takes an **operand** of mode `INT` and yields the absolute value of mode `INT`. For example, `ABS -1` yields 1. In the standard-prelude is another definition of `ABS` that takes an **operand** of mode `REAL` yielding a value of mode `REAL`. When **monadic-operators** are combined, they are of course elaborated in right-to-left order. That is, in `ABS -1`,  $-$  acts on 1 to yield  $-1$ , and then `ABS` acts on  $-1$  to yield 1. Another **monadic-operator** which takes an `INT` or `REAL` **operand** is `SIGN` which yields  $-1$  if the **operand** is negative, 0 if it is zero, and  $+1$  if it is positive. For modes that have multiple precisions, Algol 68 defines the **monadic-operator** `LENG` that will increase precision by one `LONG`, and the **monadic-operator** `SHORTEN` that will decrease precision by one `LONG`, for the **operand** value. Note that a runtime error may result in case the value of a longer precision cannot be represented in a shorter precision, though `REAL` values will be rounded.

## 2.7 Real to integer conversion

We have seen that in a strong context, a value of mode `INT` can be coerced by widening to a value of mode `REAL`. But how do we convert a value of mode `REAL` to a value of mode `INT`? In Algol 68 this is impossible by implicit coercion. The reason behind this design choice in Algol 68 is that the fractional part cannot be implicitly discarded. You must explicitly state how the conversion should take place. Algol 68 offers monadic operators to convert a value of mode `REAL` to a value of mode `INT`. If you want to convert a `REAL` value to an `INT`, you must use one of the operators `ROUND` or `ENTIER`. The operator `ROUND` takes a single **operand** of mode `REAL` and yields an `INT` whose value is the **operand** rounded to the nearest integer. Thus `ROUND 2.7` yields 3, and `ROUND 2.2` yields 2. The same rule applies

with negative numbers, thus `ROUND -3.6` yields `-4`. Essentially, you get an integer result that differs not more than 0.5 from the real value. The operator `ENTIER` takes a `REAL` **operand** and likewise yields an `INT` result, but the yield is the largest integer that is not larger than the real **operand**. Thus `ENTIER 2.2` yields `2`, `ENTIER -2.2` yields `-3`.

## 2.8 Elementary dyadic-formulas

A basic **dyadic-operator** is addition, `+`; for instance:

```
print (read int + 1)
```

The plus operator `+` takes two **operands** of mode `INT` and yields a sum of mode `INT`. It is also defined for two **operands** of mode `REAL` yielding a sum of mode `REAL` :

```
REAL x = read real + offset
```

As we have seen, the maximum integer which `a68g` can represent is `max int` and the maximum real is `max real`. Addition could give a sum which exceeds those two values, which is called overflow. Algol 68 leaves such case undefined, meaning that an implementation can choose what to do. `a68g` will give a runtime error in case of arithmetic overflow, for example:

```
1      (print ((1 + max int)
           1
```

```
a68g: runtime error: 1: INT math error (numerical result out of
range) (detected in [] "SIMPLOUT" closed-clause starting at "("
in this line).
```

The dyadic minus operator for subtraction `-` also takes two **operands** of mode `INT` or two **operands** of mode `REAL` and will yield an `INT` or `REAL` difference respectively:

```
INT difference = a - b, REAL distance = end - begin
```

Since a **formula** yields a value of a particular mode, you can use it as an **operand** for another operator. For example:

```
INT sum = a + b + c
```

the order of elaboration being that **operands** are elaborated collaterally, and then the operators are applied from left-to-right in this particular example, since the two operators have the same priority. The times operator `*` performs arithmetic multiplication and takes `INT` **operands** yielding an `INT` product. For example:

```
INT product = 45 * 36
```

Likewise, `*` is also defined for multiplication of two values of mode `REAL`:

```
REAL pi 2 = 2.0 * pi
```

We already saw with + and – that a **formula** can be an **operand** in another **formula**:

```
INT factorial 6 = 2 * 3 * 4 * 5 * 6;
REAL interpolation = slope * x + intercept
```

In Algol 68, the common precedence of brackets over division, then multiplication, then addition and subtraction, applies and it is implemented by giving a priority to operators. The priority of multiplication is higher than the priority for addition or subtraction. The priority of the **dyadic-operators** + and – is 6, and the priority of the \* operator is 7. For example, the value of the **formula**  $2 + 3 * 4$  is 14. It is possible to change the priority of standard operators, but that does not make sense — **priority-declarations** are meant to define the priority of *new* **dyadic-operators** you introduce. Every dyadic operator has a priority of between 1 and 9 inclusive, and **monadic-operators** bind more tightly than any **dyadic-operator**. You can of course force priorities by writing sub expressions in parentheses:

```
INT m = 1 + (2 * 3), # 7 #, n = (1 + 2) * 3 # 9 #
```

The parentheses in Algol 68 are short-hand for BEGIN ... END and indeed, you could write a clause in parentheses:

```
INT one ahead = 1 + (INT k; read (k); k)
```

Hence there is no special construct for *sub expressions in parenthesis* that you find in many other programming languages. This is a consequence of Algol 68's famed orthogonality. We will see many more of those examples throughout this book. Parentheses can be nested to any depth as long as a68g does not run out of stack space.

On the right-hand side of an **identity-declaration**, widening is allowed, so the following **declaration** is valid:

```
REAL a = 24 * -36
```

The **formula** is elaborated first, and the final INT result is widened<sup>3</sup> to REAL.

Division of integers can have two different meanings. For example,  $3 \div 2$  can yield either 1 or 1.5. Therefore, Algol 68 uses two different **operator-symbols** for division of integers. The operator % takes **operands** of mode INT and yields a value of mode INT. It has the alternative representation OVER. The **formula**  $7 \% 3$  yields the value 2, and the **formula**  $-7 \% 3$  yields the value -2. The priority of % is 7, the same as multiplication. The modulo operator MOD gives the remainder after integer division. It requires two **operands** of mode INT and yields a value also of mode INT. Algol 68 defines  $a \text{ MOD } b$  as this: let  $q \in \mathbb{Z}$  be the

---

<sup>3</sup>An **operand** is in a firm context. In a firm context no widening is allowed, otherwise we could not decide whether to use integer addition or real addition when we add integer **operands**.

## INFORMAL INTRODUCTION TO ALGOL 68

quotient of  $a \in \mathbb{Z}$  and  $b \in \mathbb{Z}, b \neq 0$  and  $r \in \mathbb{W}$  the remainder, such that this relation holds:

$$a = q \times b + r; 0 \leq r < |b|$$

then  $a \text{ MOD } b$  yields  $r$ . Thus  $7 \text{ MOD } 3$  yields  $1$  ( $q = 2$ ), but  $-7 \text{ MOD } 3$  yields  $2$  ( $q = -3$ ). The result of  $\text{MOD}$  is always a non-negative number. Note that the definition of  $\text{OVER}$  is not consistent with the quotient  $q$  in above definition of  $\text{MOD}$ .  $\text{MOD}$  can also be written as  $\%*$  and its priority is 7, the same as division. Division of real numbers is performed by the operator  $/$ . It takes two **operands** of mode  $\text{REAL}$  and yields a  $\text{REAL}$  result. It has a priority of 7. Thus the **formula**  $3.0 / 2.0$  yields  $1.5$ . As indicated above, the operator  $/$  is also defined for integer **operands**; for example  $3 / 2$  yields  $1.5$ . There is no real version of  $\text{MOD}$ .

Algol 68 defines an exponentiation operator  $**$  or its equivalents  $^$  or  $\text{UP}$ . Its priority is 8. The mode of its left **operand** can be either  $\text{REAL}$  or  $\text{INT}$  but its right **operand**, the exponent, must have mode  $\text{INT}$ . If both its **operands** have the mode  $\text{INT}$ , the yield will have mode  $\text{INT}$  and in this case the right **operand** must not be negative; if the left **operand** is real the yield will have mode  $\text{REAL}$  and the right operand can be negative. Thus the **formula**  $3 ** 4$  yields  $81$  and  $3.0 ** 4$  yields  $81.0$ . Exponentiation takes priority over division, multiplication and addition or subtraction. For example, the **formula**  $3 * 2 ** 4$  yields  $48$ , not  $1296$ . A common pitfall is the **formula**  $-x ** 2$  which yields  $x^2$  in stead of  $-(x^2)$ . The monadic minus is elaborated first, followed by the exponentiation. This looks straightforward, but even experienced programmers tend to make this mistake every now and then. This particular example is not specific to Algol 68, for instance Fortran has the same peculiarity.

In the discussion above the arithmetic operators  $+$ ,  $-$  and  $*$  have **operands** of identical modes:

$$\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \bullet \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

In practice, you will frequently use **operands** of mixed modes. The **dyadic-operators**  $+$ ,  $-$ ,  $*$  and  $/$  (but not  $\%$ ) are also defined for mixed modes. That is, any combination of  $\text{REAL}$  and  $\text{INT}$  **operands** can be used:

$$\mathbb{Z} \times \mathbb{R} \rightarrow \mathbb{R} \bullet \mathbb{R} \times \mathbb{Z} \rightarrow \mathbb{R}$$

With mixed modes the yield is always  $\text{REAL}$ . Thus the following **formulas** are all valid:

```
small real + 1 • 2 * pi
```

The priority of the mixed-mode operators is unchanged since as we will see later, the priority relates to the operator symbol rather than modes of **operands** or result.

## 2.9 Real functions



Routines are the subject of chapter 5, however we have already seen the routines `print`, `read` and others like `read int`, `read real`, `read bool` and `read char`. Like other languages, Algol 68 defines various mathematical functions that take a real argument and yield a real result. Note that a runtime error occurs if either argument or result is out of range. Multi-precision versions of these routines are also pre-defined and are preceded by either `long` or `long long`, for instance `long sqrt` or `long long ln`. The following list summarises the routines that are defined in the Algol 68 Genie standard-prelude:

| Function                                       | Description  |
|--|--|
| <code>sqrt (x)</code>                          | Square root of $x$ , $\sqrt{x}$  |
| <code>curt (x)</code> , <code>cbirt (x)</code> | Cube root of $x$ , $\sqrt[3]{x}$   |
| <code>exp (x)</code>                           | Yields $e^x$ . Note that <code>exp</code> yields 0 when the function value is too small to be represented in a REAL value. |
| <code>ln (x)</code>                            | Natural logarithm of $x$ , ${}^e \log(x)$  |
| <code>log (x)</code>                           | Logarithm of $x$ to base 10, ${}^{10} \log(x)$   |
| <code>sin (x)</code>                           | Sine of $x$ , $\sin(x)$ , with $x$ in radians  |
| <code>arcsin (x)</code>                        | Inverse sine of $x$ , $\sin^{-1}(x)$   |
| <code>cos (x)</code>                           | Cosine of $x$ , $\cos(x)$ , with $x$ in radians  |
| <code>arccos (x)</code>                        | Inverse cosine of $x$ , $\cos^{-1}(x)$   |
| <code>tan (x)</code>                           | Tangent of $x$ , $\tan(x)$ , with $x$ in radians   |
| <code>arctan (x)</code>                        | Inverse tangent of $x$ , $\tan^{-1}(x)$  |
| <code>arctan2(x, y)</code>                     | Angle $\theta$ in range $[-\pi, \pi]$ whose tangent is $\tan(\theta) = y/x$  |
| <code>sinh (x)</code>                          | Hyperbolic sine of $x$ , $\sinh(x)$  |
| <code>arcsinh (x)</code>                       | Inverse hyperbolic sine of $x$ , $\sinh^{-1}(x)$   |
| <code>cosh (x)</code>                          | Hyperbolic cosine of $x$ , $\cosh(x)$  |
| <code>arccosh (x)</code>                       | Inverse hyperbolic cosine of $x$ , $\cosh^{-1}(x)$   |
| <code>tanh (x)</code>                          | Hyperbolic tangent of $x$ , $\tanh(x)$   |
| <code>arctanh (x)</code>                       | Inverse hyperbolic tangent of $x$ , $\tanh^{-1}(x)$  |

## 2.10 Booleans

There are only two values of mode `BOOL`, and their **denotations** are `TRUE` and `FALSE`. The procedure `print` prints `T` for `TRUE`, and `F` for `FALSE`. Thus:

```

BOOL t = TRUE, f = FALSE;
print ((t, f));

```

produces `TF` on standard output. Boolean values are also read as `T` and `F` respectively. `a68g` implements procedure `read bool` that yields a boolean value read from standard input, so you can write:

```

BOOL answer = read bool;

```

## 2.11 Boolean operators

The predominant **monadic-operator** for a `BOOL` **operand** is `NOT`, with alternative representations `^` or `~`. Obviously, if its **operand** is `TRUE`, it yields `FALSE`, and if its **operand** is `FALSE`, it yields `TRUE`. The operator `ODD`, a relict from a distant past, yields `TRUE` if its integer **operand** is an odd number and `FALSE` if it is even. `ABS` converts its **operand** of mode `BOOL` and yields an integer result: `ABS TRUE` yields 1 and `ABS FALSE` yields 0. **Dyadic-operators** with boolean result come in two kinds: those that take **operands** of mode `BOOL`, yielding `TRUE` or `FALSE`, and those that operate on **operands** of other modes, which are comparison operators described in section 2.14. Three **dyadic-operators** are declared in `a68g`'s standard-prelude which take **operands** of mode `BOOL`. The operator `AND`, with alternative representation `&`, yields `TRUE` only if both its **operands** yield `TRUE`. The priority of `AND` is 3. The operator `OR` yields `TRUE` if at least one of its **operands** yields `TRUE`. The priority of `OR` is 2. The operator `XOR` yields `TRUE` if exactly one of its **operands** yields `TRUE`. It has no alternative representation. The priority of `XOR` is 3.

## 2.12 Characters and text

The mode of a single character is `CHAR`. Algol 68 Genie recognises  $1 + \max \text{abs char}$  distinct values of mode `CHAR`, some of which cannot be written in **denotations**, for example control characters. The space is declared as the **identifier** `blank` in the standard-prelude. A character is denoted by placing it between quote characters, for instance the **denotation** of lower-case `a` is `"a"`. The quote character `"` is doubled in its **denotation**. This convention works because adjacent **denotations** have no meaning in Algol 68 syntax. These are example character denotations:

```
"M" •"m" •"0" •";" •"\ " •"/ " •" " " •" " "
```

The procedure `print` will print a character value, and `a68g` implements procedure `read char` that yields a `char` value read from standard input. Example **identity-declarations** for values of mode `CHAR` are:

```
CHAR a = "A", z = read char, tilde = " "
```

Values of modes `INT`, `REAL`, `BOOL` and `CHAR` are known as plain values in Algol 68. In chapter 4 you will see that plain values can for instance be organised in rows. In this way text is represented as a row of characters, and vectors are rows of reals, et cetera. Texts are so common that every programming language has a text **denotation**: a quoted string literal. In chapter 4 you will see that in Algol 68 the row of character mode reads `[] CHAR`. We discuss `[] CHAR` briefly here just to introduce the **row-of-character-denotation** so we can print texts from our basic Algol 68 **programs**. The relevant production-rules read:

- **row-of-character-denotation:**

**quote-symbol, string-item-sequence, quote-symbol.**

- **string-item:**  
**character;**  
**quote-symbol, quote-symbol.**

A **row-of-character-denotation** is conventionally delimited by quote characters:

```
"This is a row-of-character-denotation"
```

You can of course print **row-of-character-denotations**, so you can now let your program print descriptive texts:

```
print ("Oops! Too few experiments performed");
```

In general, a68g will concatenate a source line ending in a backslash \ with the next line; this feature can be used in case your **denotation** must be continued on a next line:

```
print ("The first stage in the development of a new program\  
consists of analysing the problem that the program must solve.");
```

If you want to place a quote character " in the **denotation**, you must double it and write "":

```
print ("The minority report stated that ""in their opinion""")
```

which produces

```
The minority report stated that "in their opinion"
```

## 2.13 Character operators

Algol 68 provides operators for character **operands**. There are two **monadic-operators** which involve the mode CHAR. The operator ABS takes a CHAR **operand** and yields the integer corresponding to that character. For example, ABS "A" yields 65 (if your platform uses ASCII encoding). The **identifier** max abs char is declared in the standard-prelude and will give you the maximum number in your encoding. Conversely, we can convert an integer to a character using the **monadic-operator** REPR ; for instance REPR 65 yields the value "A". REPR accepts an integer in the range 0 to max abs char. REPR is of particular value in allowing access to control characters. **Dyadic-operators** + and \* involve (repeated) concatenation of characters and are discussed in chapter 4.

## 2.14 Comparison operators

## INFORMAL INTRODUCTION TO ALGOL 68

Values of modes `INT`, `REAL`, `BOOL` and `CHAR` can be compared to determine their relative ordering. Because the widening coercion is not allowed for **operands**, many extra comparison operators are declared in the standard-prelude that compare values of mixed modes such as `REAL` and `INT`. For example, the **boolean-formula**:

```
3 = 1.0 + 2
```

yields `TRUE`. Similarly:

```
1 + 1 = 1
```

yields `FALSE`. Note that the **equals-symbol** `=` can also be written as `EQ`. Likewise, the **formula**:

```
35.0 EQ 3.5e1
```

should<sup>4</sup> yields `TRUE`. The negation of `=` is `/=` (not equal):

```
3 /= 2
```

yields `TRUE`, and:

```
TRUE /= TRUE
```

trivially yields `FALSE`. Alternative representations of `/=` are `~=` and `NE`. The priority of both `=` and `/=` is 4. The comparison operators `<`, `>`, `<=` and `>=` can be used to compare values of modes `INT`, `REAL` and `CHAR` in the same way as `=` and `/=`. Alternative representations for these operators are `LT` and `GT` for `<` and `>` and `LE` and `GE` for `<=` and `>=` respectively. The priority of all these four comparison operators is 5. Characters are ordered by their absolute (integer) value. Hence if **identifiers** `a` and `b` are declared as having mode `CHAR`, then the **formula**:

```
a < b
```

will yield the same value as the comparison of integers:

```
ABS a < ABS b
```

`BOOL` **formulas** yielding `TRUE` or `FALSE` can of course be combined. For example, to test whether `x` lies between 0 and 1 one writes:

```
x > 0 AND x < 1
```

The priorities of `<`, `>` and `AND` are defined such that parentheses are unnecessary in this case, but using parenthesis can improve legibility of your code. More complicated `BOOL` **formulas** can be written:

---

<sup>4</sup>You should be cautious when comparing two `REAL` values for equality or inequality because of subtle rounding errors and finite precision of an object of mode `REAL`. For instance, `1.0 + small real / 2 = 1.0` will yield `TRUE`.

```
cycle < 10 AND cmd /= "s" OR read bool
```

Because the priority of **AND** is higher than the priority of **OR**, **AND** in the above **formula** is elaborated first. Code clarity can always be improved using parentheses:

```
cycle < 10 AND (cmd /= "s" OR read bool)
```

## 2.15 Program structure

Algol 68 **programs** are free format: the meaning of a program is independent of its layout. This seems a triviality but Algol 68 was presented at a time when languages as Fortran designed special meaning to characters at certain positions in a line. For example, you could write a complete though trivial Algol 68 program like this:

```
(print ("Hello world!"))
```

Note that the pair **BEGIN** and **END** can always be replaced by the pair ( and ). This is another simple example program:

```
BEGIN INT m = read int;
      INT n = read int;
      print ("sum is", m + n)
END
```

Note that there is no **semicolon-symbol** before **END**. Unlike C where a **semicolon-symbol** is a statement terminator, in Algol 68 it is a phrase separator<sup>5</sup>.

The `print` procedure, **applied-identifiers** and **denotations** are examples of **units**. Chapter 8 will explain **units** and coercions in detail. **Units** and **declarations** are separated by the **go-on-symbol**, which is a **semicolon-symbol**. A sequence of at least one **unit** and if needed **declarations**, separated by **semicolon-symbols**, is called a **serial-clause**. A **serial-clause** yields the value of its last **unit**. Since a **serial-clause** yields a value but a **declaration** never does, a **serial-clause** cannot end in a **declaration**. In Algol 68, as in other block oriented languages as C and Pascal, the validity of **declarations** is limited to the "block" that contains them. Algol 68 uses the terms *range* and *reach* to explain the validity of **declarations**:

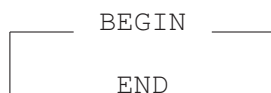
- a range is a validity area for **declarations**, for instance a **serial-clause**. A **declaration** in a range is not valid in its parent ranges, but it is valid in embedded ranges.
- a reach is a range excluding its embedded ranges. In Algol 68, looking for the **dec-**

---

<sup>5</sup>a68g is tolerant with respect to using the **semicolon-symbol** as phrase terminator. You will get a warning, but superfluous **semicolon-symbols** are ignored.

**laration** of a symbol (**identifier**, **mode-indicant**, operator et cetera) amounts to a search out through successive nested reaches. This is the same in languages as C or Pascal.

**Declarations** in a **serial-clause** have validity only in that **serial-clause** and in its embedded **serial-clauses**. The construct `BEGIN serial-clause END` contains a **serial-clause** and thus holds a range which is illustrated by this diagram:



Similar diagrams will be used in this book to illustrate the hierarchy of ranges of more complicated constructs. The construct `BEGIN serial-clause END` is called a **closed-clause** in Algol 68.

## 2.16 Variables

Previous sections dealt with static values: **denotations** or values associated with an **identifier** through an **identity-declaration**. Once associated, the association between the **identifier** and a value does not change as long as the **identifier** is in existence. For most **programs** it is required that such association can change, specifically, that the value can change. In order to change values, we need some way of referring to them. Algol 68 can generate values that refer to other values. A value that refers to another value is called a name. Algol 68 names are often called *variables*, which is also the common term in other programming languages<sup>6</sup>. In programming a variable is a special value (also often called a reference) that has the property of being able to be associated with a value (or no value, this is the name `NIL` {see 2.23}). What is actually "variable" is the association. Obtaining the value associated with a variable is called dereferencing, and the creation or the change of the association is called an **assignment**. The mode of a name is called a "reference mode", and "reference" has the Algol 68 reserved word `REF`. A name which can refer to a value of mode `INT` is said to have the mode `REF INT`. Likewise, we can create names with modes:

`REF INT • REF REAL • REF COMPLEX • REF BOOL`

`REF` can precede any mode except `VOID`. Since names are values, `REF` can also precede a

---

<sup>6</sup>Variables in computer programming are very different from variables in mathematics and the apparent similarity is a source of much confusion. Variables in most of mathematics (those that are extensional and referentially transparent) are unknown values, while in programming a variable can associate with different values during the time that a program is executed.

mode already containing REF. Thus it is possible to construct modes such as<sup>7</sup>:

```
REF INT • REF REF REAL • REF REF REF COMPLEX
```

Names are created in Algol 68 using **generators**. There are two kinds of **generator**: local and global. Local **generators** generate space in the stack, while global **generators** generate space in the heap. The two differ in the dynamic lifetime of the name they generate. We have encountered the concepts of *range* and *reach*. Those are static concepts, as the program text defines the ranges and reaches. However, a value at runtime has a *dynamic* lifetime that in Algol 68 cannot always be correlated to a *static* range since **serial-clauses** yield a value, thereby exporting them out of a range. In Algol 68, the dynamic lifetime of a value is called its scope which amounts to the largest **serial-clause** throughout which that value exists. The scope of a plain value, like 1 or TRUE, and the scope of a global name, is the whole program. The scope of a local name is the smallest enclosing clause which contains its **generator** (which may be hidden by an abbreviated **variable-declaration**, vide infra). In general, values have *scope* and **identifiers** have *range*. In Algol 68 it is often, but not always, possible to check at compile time that names are not applied outside of their scopes. Since this check is not always possible at compile time, a68g applies dynamic scope checking at runtime.

The **generator** LOC INT generates in the stack a name of mode REF INT which can refer to a value of mode INT. The LOC stands for local. It is not wrong to write:

```
read (LOC INT)
```

but the created name is anonymous since it is not associated with an **identifier**; when read terminates, the value is marooned since the name is no longer accessible. We can of course associate an **identifier** with a generated name by means of an **identity-declaration**:

```
REF INT a = LOC INT
```

The **generator** LOC INT generates a name of mode REF INT although any **unit** yielding a value of mode REF INT would do. After this **identity-declaration**, a can be made to refer to a value of mode INT. We can for instance write:

```
read (a)
```

When read finishes, a identifies a name which now refers to an integer. A brief term for a is INT *variable*. Names can also be declared using a previously declared name on the right-hand side of the **identity-declaration** using a:

```
REF INT b = a
```

In this **declaration**, b has mode REF INT so it identifies a name. a also has mode REF INT and therefore also identifies a name. The **identity-declaration** makes b the same

---

<sup>7</sup>Although you can write as many REFs as you like, in practice you will not encounter more than four of them, which are REF-REF-REF variables.

name as `a`. This means that if the name identified by `a` refers to a value, then the name identified by `b` will always refer to the same value — both are an alias. Later on in this chapter you will read that there is an abbreviated **variable-declaration** which will let you write:

```
INT a;
```

with exact same meaning as:

```
REF INT a = LOC INT
```

but we will use the unabbreviated form for the time being to emphasize what a variable actually is. Whereas `LOC` will generate local names, `HEAP` will generate global names. The **generator** `HEAP INT` generates a global name of mode `REF INT` which can refer to a value of mode `INT`, so we could write:

```
REF INT a = HEAP INT
```

`a68g` allows the symbol `NEW` as alternative for `HEAP`, hence we may also write:

```
REF INT a = NEW INT
```

Here we arrive at an interesting question. An **identity-declaration** associates an **identifier** with a value, in this case a name. This association is permanent. Local names are allocated in the stack, and global names are allocated in the heap. You should not assume that a global name is a *constant address* in memory. During execution of an Algol 68 program the heap fills with data, but after some time it will contain much data that is no longer accessible (temporary data, data from ranges that have ceased to exist, et cetera). Algol 68 employs a garbage collector that restores heap space by removing inaccessible data from the heap and compacting it. Garbage collection requires names in the heap to be handles instead of pointers. Heap compaction means that *addresses* are not constant, though of course the association between name and value remains unbroken when data is moved around. This association is permanent, but the physical address need not be constant. Note that even in the stack, data may be moved, although the current implementation of `a68g` only performs garbage collection and compaction of the heap.

## 2.17 Assignment

The process of causing a name to refer to a value is called **assignment**. Using the **identifier** `a` declared above, we can write the **assignment**:

```
a := 3
```

We say *a becomes 3*. Note that the mode of the name identified by `a` is `REF INT`, and the mode of the **denotation** `3` is `INT`. After the **assignment**, the name identified by `a`



refers to the value denoted by 3. An **assignment** consists of three parts: the left-hand-side **unit** yields a name, in the middle is the **becomes-symbol**, and the right-hand-side **unit** yields a value suitable to that name. The right-hand side of an **assignment** can be any **unit** which yields a value whose mode has one less REF than the mode of the name on the left-hand side. You should note that **becomes-symbol** := is not an operator. When an **identifier** for a name is declared, the name can be made to refer to a value immediately. This is commonly called *initialisation*. For example:

```
REF REAL x = LOC REAL := pi
```

where pi is a value declared in the standard-prelude. LOC REAL generates a name of mode REF REAL. If you recall the syntax of the **identity-declaration**:

- **identity-declaration:**  
formal-declarer, identity-definition-list.
- **identity-definition:**  
identifier, equals-symbol, strong-unit.

you will recognise that the **unit** in the **variable-declaration** above reads:

```
LOC REAL := pi
```

which is a valid **assignment**.

The right-hand side of an **assignment** is a strong context so widening is allowed. Thus we can write:

```
x := 3
```

where the 3 is widened to 3.0 before being assigned to x. This is another **identity-declaration** with an initial **assignment**:

```
REF INT c = LOC INT := 5
```

Using the **identifier** a declared earlier, we can write:

```
a := c
```

and say *a becomes c*. The name on the left-hand side of the **assignment** has mode REF INT, so a value which has mode INT is required on the right-hand side, but what has been provided is a name with mode REF INT. Algol 68 has a coercion which replaces a name with the value to which it refers: this is called dereferencing and is allowed in a strong context. In the above **assignment**, the name identified by c is dereferenced yielding the value five. The left-hand side of an **assignment**, a name, is in a soft context. A soft context is the only context in which dereferencing is not allowed.

Every construct in Algol 68 yields a value except a **declaration**, that yields no value. We said earlier that the value of the left-hand side of an **assignment** is a name. In fact, the

value of the whole of the **assignment** is the value of the left-hand side. Because this is a name, it can be used on the right-hand side of another **assignment**. For example:

```
a := b := c
```

## 2.18 Operators combined with assignment

Consider the common **assignment**:

```
a := a + 1
```

The right-hand side of the **assignment** is a **formula**. **Assignments** of this kind are so common that a special operator has been devised to perform them. The above **assignment** can be written:

```
a += 1 or a PLUSAB 1
```

which is read as *a plus-and-becomes one*. The left **operand** must be a name, and the right **operand** may be any **unit** yielding a value that can be assigned to that name. The yield of += is the value of the left **operand**, that is, the name. The operator += is defined for among others a left **operand** of mode REF INT or REF REAL. The REF INT version of the operator expects an integer right **operand**. There are two REF REAL versions for this operator, expecting a right **operand** of mode INT or REAL respectively. There are other operators like +=. They are -=, \*=, /=, %:= and %\*:= with obvious meaning. Their alternative representations are respectively MINUSAB, TIMESAB, DIVAB, OVERAB and MODAB. The operators OVERAB and MODAB are only declared for **operands** with modes REF INT and INT. The priority of all the operators combined with **assignment** is 1. Note that **operators** that perform **assignments** constitute a **formula**, not an **assignment**.

## 2.19 Order of evaluation

The elaboration of left-hand-side destinations and right-hand-side sources is performed collaterally. After evaluating destinations and sources, **assignment** takes places from right-to-left. In the above example, a may be evaluated at the same time as b and the dereferencing of c. Then the value resulting from dereferencing c is assigned to b. Then b is dereferenced and the resulting value is assigned to a. Therefore, take care in **assignments** that the effect of your code does not depend on assumptions on the order of elaboration.

Since an **assignment** yields a value (specifically, a name), it can be used as an **operand** in a **formula**. However, an **assignment** is a **unit**, and a **unit** cannot be a direct **operand** (see chapter 8). The **assignment** must be packed in an **enclosed-clause** using parentheses, or BEGIN and END; for example:

```
2 * (a := a + 1)
```

In Algol 68, the **becomes-symbol** `:=` is not an operator, and an **assignment** cannot be a direct **operand**. Note that the following rephrasing applies the operator `+=`:

```
2 * (a += 1)
```

Parentheses are still needed as multiplication takes priority over `+=`. Now let us return to order of evaluation. In:

```
INT side sq = a ** 2 + b ** 2
```

the order of elaboration is that both terms are elaborated collaterally. Hence do not write code that yields a result that depends on the order of evaluation, for instance:

```
INT poly = a ** 2 + (a += 1)
```

which would yield either  $a^2 + a + 1$  or  $a^2 + 2a + 1$  or  $a^2 + 3a + 2$  at the compiler's discretion<sup>8</sup>. Write this strictly as:

```
a += 1; INT poly = a * (a + 1)
```

## 2.20 Abbreviated variable-declarations

When declaring names, the involved mode is often repeated on both sides. For example:

```
REF REAL x = LOC REAL;
```

Such **declarations** of names are very common. In general Algol 68 allows abbreviation when information is repeated in a same construct. In this case, abbreviation leads to the **variable-declaration**. The relevant production-rules are:

- **variable-declaration:**  
    **qualifier-option, actual-declarer, variable-definition-list.**
- **variable-definition-list:**  
    **identifier, initialisation-option.**
- **initialisation:** **becomes-symbol, strong-unit.**

Thus the above **declaration** can be written:

```
LOC REAL x
```

---

<sup>8</sup>This **formula** could fail completely if e.g. the **assignment** involved several separate byte storages and these got mixed up with the multiplication coming while the value of  $a$  was indeterminate. In C terms, the result is *undefined* rather than *unspecified*.

## INFORMAL INTRODUCTION TO ALGOL 68

or, most commonly, since `LOC` is the default:

```
REAL x
```

Equivalently:

```
REF REAL y = HEAP REAL := 0;
```

can be abbreviated to:

```
HEAP REAL y := 0;
```

or

```
NEW REAL y := 0;
```

but the `HEAP` symbol or `NEW` symbol could not be omitted since `LOC` is the default.

Since `REF` cannot precede `VOID` you cannot use `VOID` as **actual-declarer** in an abbreviated **variable-declaration**. An abbreviated **declaration** uses the **actual-declarer** (the right-hand side of an **identity-declaration**) followed by the **identifier**; and if the **actual-declarer** contains the **generator** `LOC`, you can omit `LOC`. Generally speaking, **actual-declarers** are required when space is generated for an object, for instance in a **variable-declaration** or a **generator**. For `INT`, `REAL`, `BOOL` and `CHAR` the **formal-declarer** is the same as the **actual-declarer**. It is important to note that **identity-declarations** cannot be mixed with abbreviated name **declarations** because the modes are quite different. For example, in:

```
REAL a := 0, b = 1
```

the mode of `a` is `REF REAL`, but the mode of `b` is `REAL`. In the abbreviated **declaration** of a name, the mode given is that of the value to which the name will refer.

## 2.21 Reading variables

We have encountered the procedure `print` that writes data. Wherever we have printed a value of mode `INT` with `print`, we can also print a name of mode `REF INT`, and similarly with all the other printable modes (such as `[, ] REAL`). The **parameters** for `print` are in a firm context and so can be dereferenced before being used. Of course, there is also a counter-part procedure `read` that will read data. We have already seen routines that read data for standard modes, for instance `read int`, but these read one item of the mode they expect at the time. The procedure `read` can be used to read multiple values, not necessarily of a same mode, in one **call**, and also allows for taking new lines et cetera. Now suppose `read` is presented with the character sequence `123` on the input, and is expecting an integer. `read` will convert the character sequence into an `INT` value, and store that value using a name passed as argument to `read`. Generally speaking, values displayed with `print` can

be read with `read`. The **parameters** for `read` must be names, or formatting routines as `newline`. Just like `print`, `read` can take more than one **parameter** by enclosing them in a **row-display**. For example, we may write:

```
REAL r;
read ((r, new line))
```

and the program will skip white space until it meets an optional **sign** followed by optional spaces and at least one digit, when it will expect to read a number, and then will take a new line. If an integer number is present on the input, it will be read and converted to a real. Likewise, `read` may be used to read integers. The plus and minus signs (+ and -) can precede integers and reals, and in this sense values in transput differ from **denotations**. Absence of a **sign** means that a number is positive. Any non-digit will terminate the reading of an integer except for a possible **sign** at the start. Reals can contain `e` as in `31.4e-1`. For a name of mode `REF CHAR`, a single character will be read. Boolean values are read as `T` and `F` respectively, with obvious meaning.

## 2.22 References to names

Any mode which starts with `REF` is the mode of a name. The value to which a `REF AMODE` name refers has mode `AMODE`. Since names are values in their own right, there is no reason why a name should not refer to a name. For example, suppose we declare:

```
INT j, k
```

then the mode of both `j` and `k` is `REF INT`. We could also declare:

```
REF INT jj, kk
```

so that `jj` and `kk` both have the mode `REF REF INT`. Now, according to the definition of an **assignment** {see 8.16}, it is allowed to write:

```
INT j, k
REF INT jj, kk
jj := j
```

because the **identifier** on the left has mode `REF REF INT` and the **identifier** on the right has mode `REF INT`. The potential pitfall in **assignments** to `REF` variables will be clear after coercions are discussed in a later chapter, but the idea can already be explained here: Algol 68 can adapt the number of `REF`s of the source of an **assignment** to the number of `REF`s of the destination, but not vice versa. Hence the **assignment** in:

```
INT j;
REF INT ref j = j;
ref j := 1
```

## INFORMAL INTRODUCTION TO ALGOL 68

fails since the `INT` value 1 is not a value for a name of mode `REF REF INT`. A way around this will be discussed later - the **cast**, that can force coercions where you need them. The above **assignment** should be written forcing the coercion to `REF INT` for the destination:

```
REF INT (ref j) := 1
```

with the effect that also `j` will be associated with 1!

### 2.23 The value `NIL`

In some circumstances you will want that a name refers to no value. This can be arranged by equating the name to `NIL`. `NIL` is the only **denotation** of a name in Algol 68. The mode of `NIL` is that of the name required by the context. For example, consider:

```
REF INT k = NIL
```

then `NIL` has mode `REF INT`. Although `NIL` is a name, you cannot assign to it. An **assignment** to `k` as declared above would cause a runtime error:

```
2      k := 0
      1
```

```
a68g: runtime error: 1: attempt to access NIL name of mode REF INT
(detected in particular program).
```

The principal application of `NIL` is letting a name refer to no value. We will see in the sections on lists and trees that this is a vital function.

### 2.24 Comments

Of course you will want to write **comments** in the source code. **Comments** can be put almost anywhere, but not in the middle of symbols. A **comment** is ignored by `a68g`. A **comment** is delimited by one of the following pairs:

```
COMMENT ... COMMENT
CO ... CO
# ... #
```

where the `...` represent the actual **comment**. If you start a **comment** with `COMMENT` then you must also finish it with `COMMENT`, and likewise for the other **comment-symbols**. This is an example **comment** describing the purpose of a **program**:

```
BEGIN
  COMMENT
```

```

The determinant of the square matrix 'a' of order 'n' by the
method of Crout with row interchanges: 'a' is replaced by its
triangular decomposition, l * u, with all u[k, k] = 1.
The vector 'p' gives as output the pivotal row indices; the k-th
pivot is chosen in the k-th column of T such that
ABS l[i, k] / row norm is maximal
COMMENT
...
END

```

It is of course recommended to indicate limitations of the **program**:

```

BEGIN
  CO This only works for powers of two!  CO
  ...
END

```

It is common practice to comment out pieces of source-code of a **program**, as in:

```

BEGIN INT i = read int;
  COMMENT
    print (i); # trivial check #
  COMMENT
  REAL x := i;
  print (x)
END

```

This is an example of nested **comments**. Since in Algol 68 the symbol starting a **comment** is equal to the symbol ending a **comment**, it is not possible to have proper nested **comments** in Algol 68. To have proper nested **comments**, Algol 68 Genie should be able to count how many **comments** are open, which is not possible if the embedding symbols are equal. Therefore, if the part of your **program** that you want to **comment** out already contains **comments**, you should ensure that the inner **comment** symbols are different from those of the outermost **comment**, because a68g only scans the outermost **comment** and ignores all text until the matching **comment** symbol<sup>9</sup>.

## 2.25 Pragmats

A **pragmat** is a pragmatic remark that you can write in your source code. The Algol 68 definition {24<sub>9</sub>.2} leaves it up to the implementation what such pragmatic remark should

---

<sup>9</sup>Compare this to Pascal where **comment** symbols are { ... } or C where **comment** symbols are /\* ... \*/ making it possible to count how many **comments** are open; however not all implementations of those languages allow nested **comments**.

## INFORMAL INTRODUCTION TO ALGOL 68

do, and in a68g pragmat let you insert command line options in the source code.

A pragmat is surrounded by one of the following pairs:

```
PRAGMAT ... PRAGMAT
PR ... PR
```

For instance, if you have a script that needs a lot of heap space you do not want to have to specify an option at the command line. Instead you write in your source code:

```
#!/usr/local/a68g
```

```
BEGIN PR heap=256M PR
    ...
END
```

The pragmat will execute the option at compile time as if you would have specified `--heap=256M` from the command line.



# Chapter 3

## Control structures

### 3.1 Introduction

In this chapter, we will describe control structures which allow an Algol 68 **program** to choose between alternatives or to perform a repetitive task. The **conditional-clause** elaborates (*executes*, in other languages) code depending on a boolean value. The **integer-case-clause** elaborates code depending on an integer value. The **loop-clause** lets you execute code iteratively.

In chapter 2, it was mentioned that an Algol 68 **program** is a **closed-clause**:

```
BEGIN ... END
```

A **program** can also consist solely of a **conditional-clause**, **integer-case-clause** or **loop-clause** that are introduced in this chapter. Further on you will see that a **program** is an **enclosed-clause**, which comprises the **closed-clause**, **collateral-clause**, **parallel-clause**, **conditional-clause**, **integer-case-clause**, **united-case-clause** and **loop-clause**.

### 3.2 The conditional-clause

The **conditional-clause** lets a **program** elaborate code depending on a `BOOL` value. This value must be yielded by a special **serial-clause** that cannot have **labels**: an **enquiry-clause**. This is a simple example of a **conditional-clause**:

```
IF REAL x = read real;  
  x > 0  
THEN print ((ln (x), new line))  
ELSE stop
```

FI

If the `BOOL` **enquiry-clause** yields `TRUE`, the **serial-clause** following `THEN` is elaborated, otherwise the **serial-clause** following `ELSE` is elaborated. The symbol `FI` following the `ELSE` **serial-clause** is a closing parenthesis to `IF`. The `ELSE` part of a **conditional-clause** can be omitted. When the `ELSE` part is omitted, and the **conditional-clause** is expected to yield a value, an undefined value of the required mode will be yielded if the **enquiry-clause** yields `FALSE`. Actually, if the `ELSE` part is omitted then its **serial-clause** is regarded as consisting of the single **unit** `SKIP`. The use of `IF` with matching `FI` eliminates the dangling-else problem: to what `IF` does a nested `ELSE` belong in nested **conditional-clauses** when `ELSE` is optional? In for example Pascal and C such matters are decided by writing extra rules next to the syntax stating that any `ELSE` is related to the *closest* preceding `IF`. In Algol 68 such extra rules are not necessary. The production-rules for the **conditional-clause** read:

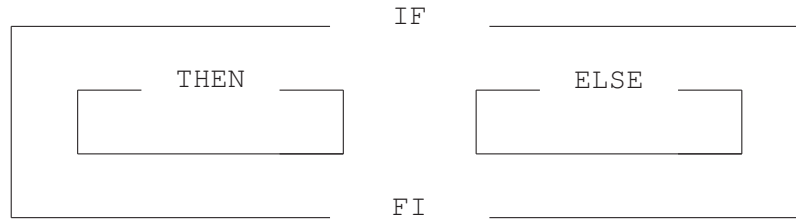
- **conditional-clause:**  
**if-symbol, meek-boolean-enquiry-clause,**  
**then-symbol, serial-clause,**  
**elif-part-option,**  
**else-part-option,**  
**fi-symbol.**
- **elif-part:**  
**elif-symbol, meek-boolean-enquiry-clause,**  
**then-symbol, serial-clause,**  
**elif-part-option.**
- **else-part:**  
**else-symbol, serial-clause.**

The **enquiry-clause** is in a meek context, meaning that only deproceduring and dereferencing (see 8.6) will be applied to the terminal **unit** of the **enquiry-clause**. For instance, if the terminal **unit** would be `read bool`, deproceduring will take place to make the routine yield a boolean value. The **enquiry-clause** on line 1 in above-mentioned example reads:

```
REAL x = read real;
x > 0
```

which obviously ends in a **unit** yielding a value of mode `BOOL`.

An **enquiry-clause** and a **serial-clause** may consist of at least one **unit** and possibly **declarations**. However, in a **conditional-clause** an **enquiry-clause** must end with a **unit** yielding `BOOL`. The hierarchy of ranges in **conditional-clauses** is illustrated by:



The range of any **declaration** in an **enquiry-clause** extends to the **serial-clauses** following THEN and ELSE. All **declarations** in the **conditional-clause** cease to exist when FI is encountered. This diagram also explains why an **enquiry-clause** cannot have **labels**: these **labels** would be visible from within constituent parts of the **conditional-clause**, and would let you jump back into the **if-part**. Note that this design of the **conditional-clause** allows you to write any **declaration** in the smallest range required, thus shielding them from other parts of the **program** that have no need for these **declarations**. This allows for a safe and clean programming style.

The **conditional-clause** can be written wherever a **unit** is permitted, so:

```
IF INT a = read int;
  a < 0
THEN print ((a, " is negative"))
ELSE print ((a, " is not negative"))
FI
```

can also be written as:

```
INT a = read int;
print ((a, " is", IF a >= 0 THEN " not" ELSE "" FI, " negative"))
```

The value of each of the **serial-clauses** following THEN and ELSE in this case is [] CHAR, a **row-of-character**. The **conditional-clause** can appear as an **operand** in a **formula**. A short form for the **conditional-clause** is often used for this: IF and FI are replaced by ( and ) respectively, and THEN and ELSE are both replaced by a bar |. For example, assuming a **declaration** for x:

```
REAL abs = (x < 0.0 | - x | x)
```

which is equivalent to:

```
REAL abs = IF x < 0.0 THEN - x ELSE x FI
```

If you omit the **else-part**, Algol 68 Genie assumes that you wrote ELSE SKIP. For instance, in:

```
REAL quotient = (y /= 0.0 | x / y)
```

the quotient will have an undefined value in case  $y = 0$ . In such case, SKIP will yield

## INFORMAL INTRODUCTION TO ALGOL 68

an undefined value of the mode yielded by the `THEN` **serial-clause**, which is forced by the **formal-declarer** `REAL`. This is an example of balancing {see 8}. Formally, `SKIP` is an algorithm that performs no action, completes in finite time, and yields some value of the mode required by the context. You will see later on in this book that `SKIP` can be quite useful, for instance when you would want to yield some undefined value.

Since the **enquiry-clause** is a special form of **serial-clause**, it can have any number of phrases before the `THEN` symbol. For example:

```
IF INT measurements;  
  read (measurements);  
  measurements < 10  
THEN ...  
FI
```

**Conditional-clauses** can be nested:

```
BOOL leap year =  
  IF year MOD 400 = 0  
  THEN TRUE  
  ELSE IF year MOD 4 = 0  
    THEN year MOD 100 /= 0  
    ELSE FALSE  
  FI  
FI
```

A construction `ELSE IF ... FI` can be contracted to `ELIF ...` which saves indentation:

```
BOOL leap year =  
  IF year MOD 400 = 0  
  THEN TRUE  
  ELIF year MOD 4 = 0  
  THEN year MOD 100 /= 0  
  ELSE FALSE  
  FI
```

Note that there is no contraction of `THEN IF` since that would leave undefined whether an eventual else part would be related to the first condition or to the second. In the abbreviated form `| :` can be used instead of `ELIF`. For example, the above **identity-declaration** for `leap year` could be written:

```
BOOL leap year =  
  (year MOD 400 = 0 | TRUE | : year MOD 4 = 0  
  | year MOD 100 /= 0 | FALSE)
```

but this is generally not very legible. For better legibility it is recommended to alternate the long and brief form of the **conditional-clause**, for instance:

```

BOOL leap year =
  IF year MOD 400 = 0
  THEN TRUE
  ELSE (year MOD 4 = 0 | year MOD 100 /= 0 | FALSE)
  FI

```

### 3.3 Pseudo-operators

Sometimes it is useful to include a **conditional-clause** in the `IF` part of a **conditional-clause**. In other words, a `BOOL` **enquiry-clause** can be a **conditional-clause** yielding a value of mode `BOOL`. This is an example with `a` and `b` declared with mode `BOOL` :

```

IF (a | b | TRUE)
THEN ...
ELSE ...
FI

```

As was mentioned in chapter 2.5, the **operands** of an operator are all elaborated before the operator is elaborated. Sometimes it is useful to refrain from further elaboration when the result of a **formula** can be determined from the value of a single **operand**. To that end a68g implements the pseudo-operator `THEF` (with synonyms `ANDF` and `ANDTH` ) which although it looks like an operator, elaborates its right **operand** only if its left **operand** yields `TRUE`. Compare them with the operator `AND` . The **unit** `p THEF q` is equivalent to:

```

IF p THEN q ELSE FALSE FI

```

An example is the earlier used definition of `leap year`:

```

BOOL leap year =
  IF year MOD 400 = 0
  THEN TRUE
  ELSE year MOD 4 = 0
  THEF year MOD 100 /= 0
  FI

```

There is another pseudo-operator `ELSF` (with synonyms `ORF` and `OREL` ) which is similar to the operator `OR` except that its right **operand** is only elaborated if its left **operand** yields `FALSE`. The **unit** `p ELSF q` is equivalent to:

```

IF p THEN TRUE ELSE q FI

```

These pseudo-operators are an a68g extension. Neither `THEF` nor `ELSF` are part of Algol 68. Compare them with `&&` and `||` in C.

### 3.4 Identity relations

In the chapter on **formulas** we did not address any operation on names. The reason is that there are no operations on names, except for comparison on equality. You cannot add addresses as you can in C. Algol 68 does not prescribe the actual implementation of how a name must refer to a value. In a typical implementation names will of course involve addresses in memory, but you have already seen that a name generated by `LOC` is different in nature, from a name generated by `HEAP` since the latter can be moved through memory by the garbage collector. Since Algol 68 leaves the technical details of the implementation of names up to the implementation, and there is no operation on names except comparison for identity: you can check whether two names are the same, and if they are the same, they refer to the same value if the names are not `NIL`. In Algol 68 names are compared in a special construct, the **identity-relation**. The **identity-relation**:

```
u :=: v
```

yields `TRUE` if  $u$  is the same name as  $v$ . The alternative representation of `:=:` is `IS`. The **identity-relation**:

```
u :/=: v
```

yields `TRUE` if  $u$  is not the same name as  $v$ . The alternative representation of `:/=:` is `ISNT`. There is a potential pitfall in the **identity-relation**: you must make sure that you compare names at the proper level of dereferencing! The following artificial **program** demonstrates the potential difficulty:

```
BEGIN REF INT i := LOC INT, j := LOC INT, k := LOC INT;
  i := (read bool | j | k);
  print(i IS j)
END
```

Here the **identity-relation** `i IS j` always yields `FALSE` because  $i$ ,  $j$  and  $k$  are different variables and thus are different names. What you actually want is to compare the *values* of these pointer variables, which in this case are names themselves. The point is that no automatic dereferencing takes place in an identity relation. To compare the names that both  $i$  and  $j$  refer to, you should place at least one side in a **cast** (see chapter 8):

```
REF INT (i) IS j
```

This will ensure that the right-hand side (in this case) is dereferenced to yield a name of the same mode as the left-hand side. The **identity-relation** is subject to balancing (a subject treated at length in chapter 8): the compiler places one side of the relation in a soft context and the other side in a strong context in such way that the modes on both sides are unique and matching. Balancing is also needed to use the most common application of the **identity-relation**: comparison to `NIL` since by balancing `NIL` takes the mode of the other

name in the **identity-relation**. You should also understand that the **identity-relation** `NIL IS NIL` does not yield `TRUE`, but gives a compile-time error since no unique mode can be established for the names, as the next test using `a68g` demonstrates:

```
$ a68g -p "NIL IS NIL"
1      (print ((NIL IS NIL)))
          1
```

```
a68g: error: 1: construct has no unique mode (detected in
closed-clause starting at "(" in this line).
```

## 3.5 The integer-case-clause

Often choices can be enumerated. Such situation can for example be handled by the following **conditional-clause**:

```
IF n = 1
THEN unit 1
ELIF n = 2
THEN unit 2
ELIF n = 3
THEN unit 3
ELSE unit 4
FI
```

This type of enumerated choice can be expressed more concisely using the **integer-case-clause** in which the `BOOL enquiry-clause` is replaced by an `INT enquiry-clause`, for example:

```
CASE n
IN unit 1, unit 2, unit 3
OUT unit 4
ESAC
```

which could be abbreviated as:

```
(n | unit 1, unit 2, unit 3 | unit 4)
```

**Integer-case-clause** can also be used to code tables. For instance, calendar computations give examples of **integer-case-clauses**:

```
INT days =
  CASE month
  IN 31,
    IF IF year MOD 400 = 0
      THEN TRUE
```

## INFORMAL INTRODUCTION TO ALGOL 68

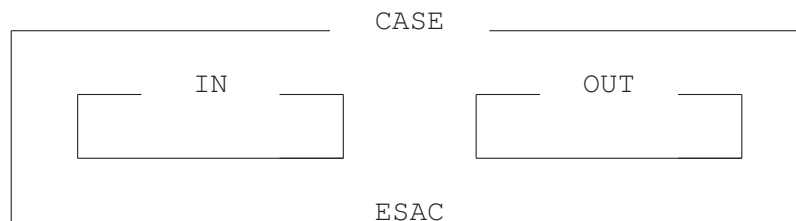
```
    ELSE (year MOD 4 = 0 | year MOD 100 /= 0 | FALSE)
    FI
  THEN 29
  ELSE 28
  FI,
  31, 30, 31, 30, 31, 31, 30, 31, 30, 31
ESAC
```

{Another example is 11.3.} Note that **units** in the **IN** part are separated by **comma-symbols**. If you want more than one phrase for each **unit**, you must make that **unit** an **enclosed-clause**. If the **INT enquiry-clause** yields 1, **unit 1** is elaborated; if it yields 2, **unit 2** is elaborated and so on. If the value yielded is negative or zero, or exceeds the number of **units** in the **IN** part, the **OUT** part is elaborated. The **OUT** part is a **serial-clause**. Like the **conditional-clause**, if you omit the **out-part**, Algol 68 Genie assumes that you wrote **OUT SKIP**. The production-rules for the **integer-case-clause** read:

- **integer-case-clause:**  
**case-symbol, meek-integer-enquiry-clause,**  
**in-symbol, unit-list-proper,**  
**integer-ouse-part-option,**  
**out-part-option**  
**esac-symbol.**
- **integer-ouse-part:**  
**ouse-symbol, meek-integer-enquiry-clause,**  
**in-symbol, unit-list-proper,**  
**integer-ouse-part-option.**
- **out-part:**  
**out-symbol, serial-clause.**

The **integer-case-clause** allows only for simple enumeration. Other languages offer constructs, like the **switch** statement in C, that allow for more complex enumeration schemes. In Algol 68, such complex schemes must be handled by a nested **conditional-clause**. The designers of Algol 68 apparently disliked syntactic sugar.

The hierarchy of ranges in **case-clauses** is illustrated by:





Sometimes the **out-part** consists of another **integer-case-clause**. Just as with `ELIF` in a **conditional-clause**, `OUT CASE ... ESAC ESAC` can be replaced by `OUSE ... ESAC`.

## 3.6 The loop-clause

Often you need to iterate a group of actions a number of times. One mechanism for iteration in Algol 68 is the **loop-clause**:

```
TO n # We iterate n times #
DO # serial-clause to be iterated #
...
OD
```

Above construct is just one form of the **loop-clause**. Unlike a **conditional-clause** or **integer-case-clause**, a **loop-clause** yields no value. The production-rules for the **loop-clause** are:

- **loop-clause:**  
**for-part-option,**  
**from-part-option,**  
**by-part-option,**  
**to-part-option,**  
**while-part-option,**  
**do-part.**
- **for-part:**  
**for-symbol, identifier.**
- **from-part:**  
**from-symbol, meek-integer-unit.**
- **by-part:**  
**by-symbol, meek-integer-unit.**
- **to-part:**  
**to-symbol, meek-integer-unit;**  
**downto-symbol, meek-integer-unit.**
- **while-part:**  
**while-symbol, meek-boolean-enquiry-clause.**
- **do-part:**  
**do-symbol, serial-clause, od-symbol;**  
**do-symbol, serial-clause-option, until-part, od-symbol.**

- **until-part:**  
**until-symbol, meek-boolean-enquiry-clause.**

Due to the many optional parts, the **loop-clause** has been compared by some to a Swiss army knife. The **loop-clause** can have a loop **identifier** counting the iterations, as is shown in the following example:

```
FOR i TO 10
DO print ((i, new line))
OD
```

Here  $i$  is a new `INT` **identifier**, a counting constant, that can be considered as implicitly declared by the **for-part**. Note that the mode of a loop **identifier** is not `REF INT` so you cannot perform an **assignment** to it. Above example will print the numbers 1 to 10. The range of a loop **identifier** is the **loop-clause** that defines it, except the **from-part**, **by-part** and **to-part**; since the **units** of the latter constructs are determined before iteration is started, they cannot depend on the value of the loop **identifier**. In a **to-part**, the **unit** following `TO` can be any **unit** yielding an integer. It is possible to modify the initial value of the loop **identifier** using a **from-part**, for example:

```
FOR n FROM -10 TO 10
DO print ((n, blank))
OD
```

This prints the numbers from  $-10$  to  $+10$  on standard output. The **unit** after `FROM` can be any **unit** which yields a value of mode `INT`. When the **from-part** is omitted, the default initial value of the loop **identifier** is 1. The value of the loop **identifier** is by default incremented by 1. The increment can be changed using a **by-part**. The **unit** after `BY` can be any **unit** which yields a value of mode `INT`. For example, to print the even numbers up to and including 10, you could write:

```
FOR n FROM 0 BY 2 TO 10
DO print ((n, new line))
OD
```

Compare this to its C equivalent:

```
{int n; for (n = 0; n <= 10; n += 2) {printf ("%d\n", n);}}
```

The increment specified by a **by-part** can be negative, in which case the loop will count backwards. `a68g` offers the reserved word `DOWNTO` as an alternative to `TO`. `DOWNTO` originally is an *ALGOL68C* extension. `DOWNTO` will decrement, whereas `TO` will increment, the loop **identifier** by the amount stated by the (implicit) **by-part**; `BY n DOWNTO m` is equivalent to `BY -n TO m`. For example:

```
FOR k FROM 10 DOWNTO 1
DO print (k)
```

OD

If you omit a **to-part**, the loop will iterate indefinitely. There are of course applications for loops that iterate until a condition is met that does not depend on the loop **identifier** alone. Such condition can be programmed using the **while-part** of a loop. For example:

```
WHILE INT int;
  read (int);
  int > 0
DO print (int)
OD
```

In this example, no loop counter is needed and so the **FOR** part is omitted. **WHILE** is followed by an **enquiry-clause** yielding a **BOOL** value. An integer is read each time the loop is elaborated until a non-positive integer is read. The range of any **declarations** in the **enquiry-clause** extends to the **DO ... OD** loop. The **while-part** provides for a *pre-checked* loop. There are those who regret that Algol 68's definition did not include a *post-checked* loop. Apparently it was believed that a post-checked loop was not really necessary since it can be programmed as:

```
WHILE do what has to be done;
  condition
DO SKIP
OD
```

and as has been remarked before, the designers of Algol 68 apparently disliked syntactic sugar. Many think that the **DO SKIP OD** solution is not elegant. To accomodate this, a68g extends Algol 68 by offering a post-checked loop by allowing an optional **until-part** as final part of the **DO ... OD** construction:

```
DO serial-clause-option
  UNTIL boolean-enquiry-clause
OD
```

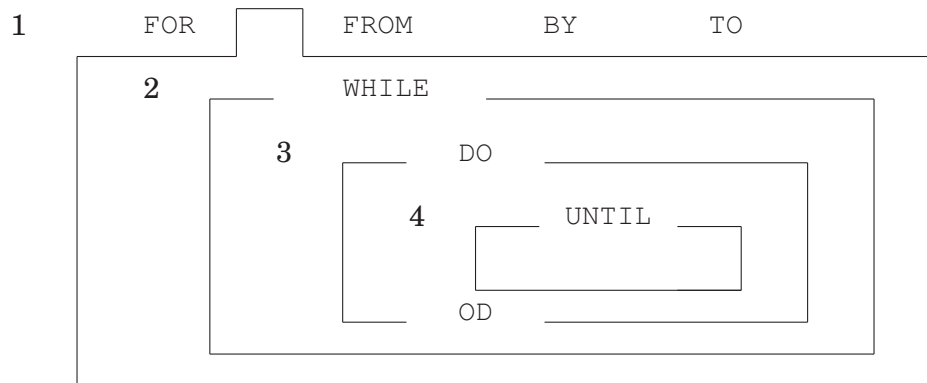
A trivial example of a post-checked loop is:

```
DO UNTIL read char = "."
OD
```

which will skip characters from standard input until a "." is encountered which will disappear from the input as well.

The hierarchy of ranges in the **loop-clause** is illustrated by:

## INFORMAL INTRODUCTION TO ALGOL 68



Again, note that the loop **identifier** is unknown in the **from-part**, **by-part** and **to-part**. Since the **units** of the latter constructs are determined before iteration is started, they cannot depend on the value of the loop **identifier**.

# Chapter 4

## Stowed modes

### 4.1 Introduction

*Stowed* is a portmanteau word for **structured** or **rowed**. Up to now, we have dealt with plain values, that is, values with modes `INT`, `REAL`, `BOOL` or `CHAR`. We will now start using compounded modes, starting with ordered sets of elements: rows and structures. The first compounded mode introduced in this chapter is a row, which is an ordered set of elements of a same mode, like in any other programming language. For example, text is a string of characters and many of us use vectors and matrices which are one - and two dimensional rows respectively. The other compounded mode introduced in this chapter is a structure, which is an ordered set of objects not necessarily of a same mode. This chapter will introduce the basic modes `STRING` and `COMPLEX`, together with the operations defined for them, and also modes `BYTES` and `BITS` will be introduced.

### 4.2 Rows

A row consists of a number of elements with a common mode which cannot be `VOID`. The mode of a row is written as the mode for each element preceded by square brackets, and is called *row of* followed by the name of the mode of each element, such as *row of int* or *row of bool*. As an example we write an **identity-declaration** for a row:

```
[ ] CHAR a = "tabula materna combusta est"
```

The mode on the left-hand-side is read *row of char*, which we will write throughout this book as `[ ] CHAR`. The **unit** on the right-hand-side of the **equals-symbol** is in this case the **denotation** of a `[ ] CHAR` value. Note that we can use a **formal-declarer** in the **identity-relation** since `a` will just be an alias of `"tabula materna combusta est"`,

and the **denotation** has implicit **bounds** 1..27. If you want to declare a row variable, you need to supply **bounds**. The **declarer** must be an **actual-declarer**. For instance, you *must* write:

```
[1 : 27] CHAR a := "tabula materna combusta est"
```

even though the **bounds** are implicit in this specific example. The difference with an **identity-declaration** is the general situation that when a row is the source of an **assignment**, also when the source appears in a **variable-declaration**, the source is copied into the destination and the **bounds** of source and destination must match. The **bounds** must match since Algol 68 does not regenerate the destination row unless the row is flexible {see 4.3}. In an **identity-declaration** you only make an alias for a row descriptor, which does not involve copying a row. The maximum number of elements in a row is equal to `max int`.

### 4.2.1 Names of rows

Following is an **identity-declaration** for a name referring to a row:

```
REF [] INT i = LOC [1 : read int] INT
```

which can be abbreviated to a **variable-declaration**:

```
[1 : read int] INT i
```

There are two things to notice about the first **declaration**. First, the mode on the left-hand side is a **formal-declarer**. It says what the mode of the **identifier** is, but not what its **bounds** are. Second, the **generator** on the right-hand side is an **actual-declarer**. It specifies the **bounds** of the row to be generated. If the **lower-bound** is 1 it may be omitted, so the above **declaration** could also have been written:

```
REF [] INT i = LOC [read int] INT
```

This **declaration** can be abbreviated to a **variable-declaration**:

```
[read int] INT i
```

A dynamic name is one which can refer to a row whose **bounds** are determined at the time the **program** is elaborated. This means that you can declare names referring to rows of the size you actually require, rather than the maximum size that you might need. The **bounds** of a row do not have to start from 1. You are free to choose the value of the **lower-bound**. In this **identity-declaration**:

```
REF [] INT i at 0 = HEAP [0 : 6] INT
```

or its equivalent **variable-declaration**:

```
HEAP [0 : 6] INT i
```

the **bounds** of the row will be `[0 : 6]`. The minimum value for a **lower-bound** is `-max int`. The maximum value for an **upper-bound** is `max int`. But remember that the maximum number of elements in a row is also `max int`, hence the following condition must be satisfied:

$$-max\ int \leq upper\ bound - lower\ bound + 1 \leq max\ int$$

The condition

$$upper\ bound - lower\ bound + 1 \leq 0$$

that occurs when you specify a **lower-bound** that exceeds the **upper-bound**, means that a row is empty; it has no elements. Such row is called a flat row in Algol 68 jargon.

## 4.2.2 Reading and printing rows

We have already used `print` to print plain values. Actually, `print` takes a row of values to be output, so it is valid to write:

```
[ ] INT i;
...
print ((i, new line))
```

which will print all elements of `i`. In case of multi-dimensional rows, the elements are printed in row-order, that is, the rightmost **subscript** varies most frequently. This is the same order as C, but not as Fortran that stores in column-order. You can call `new line` and `new page` explicitly to ensure that the next value to be input will start at the beginning of the next line or page. With respect to rows, `read` behaves just like `print` in that a whole row can be read in one **call**. The only difference between the way `read` is used and the way `print` is used is that the values for `read` must be names whereas `print` accepts values which are not names. Note that if `read` is used to read a `[ ] CHAR` with fixed **bounds** as in:

```
REF [ ] CHAR sf = LOC [80] CHAR;
read (sf)
```

the number of characters specified by the **bounds** will be read, `new line` and `new page` being called as needed.

## 4.2.3 Row-displays

Of all rows, only `[ ] CHAR` has a **denotation**. Values for other rows are denoted by a construct called a **row-display**. The production-rule for a **row-display** reads:

- **\*row-display:**  
**begin-symbol, unit-list-proper-option, end-symbol.**

## INFORMAL INTRODUCTION TO ALGOL 68

A **row-display** consists of none or two or more **units** separated by **comma-symbols** and enclosed by parentheses (or `BEGIN` and `END`). Also `[] CHAR` has a **row-display**:

```
[] CHAR a = ("a", "b", "c", "d")
```

which means the same as:

```
[] CHAR a = "abcd"
```

It is important to note that the **units** in the **row-display** could be quite complicated. Any **unit** yielding the a value of the mode of an element of the row is permitted. For example, here is another **declaration** for a row with mode `[] CHAR`:

```
[] CHAR a = (blank, (CHAR z; read (z); z), "")
```

In this **declaration** the number of elements is 3. The **lower-bound** of a **row-display** is always 1, so the **upper-bound** equals the number of elements in the **row-display**. Since a **row-display** is only allowed in a strong context, such as the right-hand side of an **identity-declaration** or the source of an **assignment**, its constituent **units** are also in a strong context. Thus, the **units** in a **row-display** can for instance be widened as in:

```
[] REAL zero vector = (0, 0, 0)
```

An empty **row-display** can be used to yield a flat row, which is a row with no elements. We could initialise a line like this:

```
[] CHAR empty line = ()
```

In this particular case, the **denotation** for a flat `[] CHAR` can also be used:

```
[] CHAR empty line = ""
```

A row can have a single element but a **row-display** cannot have a single **unit** because such construct would coincide with a **closed-clause**. This causes ambiguity in the **uniting coercion** {8.6.5}. In this case and in a strong context, we write a single **unit** yielding a value of correct mode for the element, which is coerced to a row with a single element by the rowing coercion {see 8.6.7}. For example:

```
[] INT z = 0
```

yields a row with one element. A **closed-clause** could be used instead:

```
[] INT z = (0)
```

since a **closed-clause** is also a **unit** {see 8.10} but note that coercions move inside clauses — coercions are not applied to **enclosed-clauses** but to the terminal **units** contained therein. Again, a **row-display** can only be used in a strong context. We have seen that the context of an **operand** is firm, so a **row-display** cannot appear as **operand** in a **formula**. There is a way around this using a **cast**, for example `[] INT (1, 2, 3)` {see 8.13}. The **denotation** for a `[] CHAR` is not a **row-display** and does need a **cast** to be used as an



**operand** in a **formula**.

## 4.2.4 Dimensions

One of the properties of a row is its number of dimensions. All the rows declared so far have one dimension. The number of dimensions affects the mode. A two-dimensional row of integers has mode

```
[ , ] INT
```

read *row row of int*, while a 3-dimensional row of reals has mode

```
[ ,, ] REAL
```

which reads *row row row of real*. Note that the number of **comma-symbols** is always one less than the number of dimensions. In Algol 68, rows with any number of dimensions can be declared. To cater for more than one dimension, each of the **units** of a **row-display** can also be a **row-display**. The **row-display** for a row with mode [ , ] INT could be:

```
((1, 2, 3),
 (4, 5, 6))
```

For two dimensions, it is convenient to talk of rows and columns. This is an **identity-declaration** using the previous row display:

```
[ , ] INT e =
 ((1, 2, 3),
 (4, 5, 6))
```

The first row of *e* is yielded by the **row-display** (1, 2, 3) and the second row is yielded by (4, 5, 6). The first column of *e* is yielded by the **row-display** (1, 4), the second column by (2, 5), and the third column by (3, 6). Note that the number of elements in each row is the same, and the number of elements in each column is also the same, but that the number of rows and columns do not need to be the same.

The mode of a row element can be any mode, including another row mode. For example:

```
[ ] [ ] CHAR days = ("Monday", "Tuesday", "Wednesday",
 "Thursday", "Friday", "Saturday",
 "Sunday")
```

The mode here is read *row of row of CHAR*. This is another example using integers:

```
[ ] [ ] INT trapezium = (1, (2, 3), (3, 4, 5))
```

Note that all rows in a [ , ] MODE object are of a same length while the rows in a [ ] [ ] MODE can be of different length.

## 4.2.5 Subscripts, bounds and slices

Since a row is an ordered set of elements, each element of a row has one index, an integer value, associated with it for each dimension. These integers increase by 1 from the first to the last element in each dimension. For example, after the **declaration**:

```
[ ] INT odds = (1, 3, 5)
```

elements of `odds` can be accessed as `odds[1]` yielding 1, `odds[2]` yielding 3 and `odds[3]` yielding 5. The integers 1,2,3 are called **subscripts** or **indexers**. Selecting elements from a row is called slicing. A construction as `odds[1]` is called a **slice**. Slicing binds more tightly than any operator, so a **slice** can be an **operand** in a **formula**. The related production-rules read:

- **slice:**  
weak-primary, sub-symbol, indexer-list, bus-symbol.
- **indexer:**  
subscript;  
trimmer.
- **trimmer:**  
lower-index-option,  
colon-symbol,  
upper-index-option,  
revised-lower-bound-option.
- **subscript: meek-integer-unit.**
- **lower-index: meek-integer-unit.**
- **upper-index: meek-integer-unit.**
- **revised-lower-bound:**  
at-symbol, meek-integer-unit.

In the two-dimensional row:

```
[, ] INT v =
  ((1, 2, 3),
   (4, 5, 6),
   (7, 8, 9))
```

the **subscripts** for 1 are [1, 1], those for 3 are [1, 3] and the **subscripts** for 9 are [3, 3]. A **slice** can also select a sub-row from a row. For example, after declaring `v` as above, we can write:

```
v[1, ]
```

which yields the row denoted by the **row-display** (1,2,3). Note that the absence of an **indexer** implicitly selects all elements for that dimension. Vertical slicing is also possible, for instance:

```
v[,3]
```

yields (3,6,9). The ability to make an alias for both rows and columns in a two-dimensional row is a notable property of Algol 68.

The **bounds** of a row can be interrogated using the operators LWB for the **lower-bound**, and UPB for the **upper-bound**. The **bounds** of the first, or only, dimension can be interrogated using the monadic form of these operators. When the row is multi-dimensional, the **bounds** are interrogated using the dyadic form of LWB and UPB: the left **operand** is the dimension being interrogated while the right **operand** is a **unit** yielding a row. The priority of the **dyadic-operators** is 8. For example:

```
[1 : 10, -5 : 5] INT r;
print (1 LWB r); # prints 1 #
print (1 UPB r); # prints 10 #
print (2 LWB r); # prints -5 #
print (2 UPB r); # prints 5 #
```

An extension of Algol 68 provided by a68g are the **monadic-operator** and **dyadic-operator** ELEMS that operate on any row. The dyadic version gives the number of elements in the specified dimension of a row, and the monadic version yields the total number of elements of a row. For example:

```
[1 : 10, -5 : 5] INT r;
print (1 ELEMS r); # prints +10 #
print (2 ELEMS r); # prints +11 #
print (ELEMS r) # prints +110 #
```

The **monadic-operator** returns the total number of elements while the **dyadic-operator** returns the number of elements in the specified dimension, if this is a valid dimension.

The **loop-clause** will be used frequently to operate on elements of rows:

```
REAL abs := 0;
FOR k FROM LWB vector TO UPB vector
DO abs += vector[k] ^ 2
OD
```

Note the use of the LWB and UPB operators. If you try to access an element whose **subscript** is greater than the **upper-bound** or less than the **lower-bound**, the **program** will fail at run time with an appropriate error message. Using the **bounds** interrogation operators LWB, UPB and ELEMS is useful because:

## INFORMAL INTRODUCTION TO ALGOL 68

1. it ensures that your **program** does not try to use a **subscript** outside the **bounds** of the row.
2. if the **bounds** are changed when the **program** is being maintained, the **loop-clause** can remain unchanged. This simplifies the maintenance of Algol 68 **programs**.
3. a compiler could forego **bounds** checking. For large rows, this can speed up processing considerably.

For one-dimensional rows, a68g offers the operator `ELEM` which provides compatibility with the vintage *ALGOL68C* compiler. This is an example of a **subscript** using `ELEM`:

```
[ ] INT first primes = (1, 3, 5, 7, 11);
print (first primes[1]);
print (1 ELEM first primes); # prints two times +1 #
```

In a 3-dimensional row, both 2-dimensional and 1-dimensional **slices** can be produced. Given the **declaration**:

```
[, ] INT r =
  ((1, 2, 3, 4),
   (5, 6, 7, 8),
   (9, 10, 11, 12),
   (13, 14, 15, 16))
```

these are the yields of different **slices**:

1. `r[2, 2]` yields 6
2. `r[3, ]` yields (9,10,11,12)
3. `r[, 2 UPB r]` yields (4,8,12,16)
4. `r[3, 2]` yields 10
5. `r[2, ]` yields (5,6,7,8)
6. `r[, 3]` yields (3,7,11,15)

A **slice** can be used to change the **bounds** of a row using the `@` construction. For example, after the **declaration**:

```
[ ] CHAR digits = "0123456789"[@0]
```

the **bounds** of `digits` are `[0 : 9]`. Note that `@` can also be written `AT`.

If you slice a name, you want the sliced element to be a name as well, otherwise you could not assign to an element of the row. The important rule in Algol 68 is that if you slice an object of mode `REF [...] MODE`, the yield will be of mode `REF MODE`. So if you first write:

```
[1 : products] REAL price;
```

you can later write:

```
price[1] := 0;
```

since slicing a `[] REAL` variable yields a `REAL` variable. Again, if you slice an object of mode `[...] MODE`, the yield will be of mode `MODE`. If you slice an object of mode `REF [...] MODE`, the yield will be of mode `REF MODE`. But if you slice an object of mode `REF REF [...] MODE`, the yield will still be of mode `REF MODE`. This coercion is called weak dereferencing and is explained in section 8.6.3.

## 4.2.6 Trimming

A **trimmer** makes a **slice** select a sub-row from a row. A **trimmer** reads `first element : last element`. The positions `first element` and `last element` are **meek-integral-units**. If `first element` is omitted, the **lower-bound** for that dimension is taken, and if `last element` is omitted, the **upper-bound** for that dimension is taken. Omission of both **subscripts** yields all elements in the specific dimension. Trimming is particularly useful with values of mode `[] CHAR`. Given the **declaration**:

```
[] CHAR quote = "Habent sua fata libelli"
```

the **trimmers**:

```
quote[: 6]
quote[8 : 10]
quote[12 : 15]
```

yield the first three words. The **lower-bound** of a **trimmer** is 1 by default, but may be changed by the use of `AT` or `@`. The `AT` construction is called a **revised-lower-bound**. For example, `quote[: 6]` has **bounds** `[1 : 6]`, but `quote[: 6 AT 2]` `quote[: 6@2]` has **bounds** `[2 : 7]`. `a68g` allows you to replace the **colon-symbol** `:` by `..` in **bounds** and **trimmers**, which is the Pascal style. Hence in `a68g` next **trimmers** are identical:

```
quote[: 6] and quote[.. 6]
quote[8 : 10] and quote[8 .. 10]
quote[12 : 15] and quote[12 .. 15]
```

## 4.2.7 Operators with rows

Rows of `CHAR` are so common that **dyadic-operators** are available implementing concatenation and comparison of text. The concatenation operator `+` is defined for all combinations

## INFORMAL INTRODUCTION TO ALGOL 68

of CHAR and [] CHAR. Thus, the **formula**:

```
"abc" + "d"
```

yields the value denoted by "abcd". The operator has a priority of 6, the same as addition; remember that all operators with the same symbol have the same priority in Algol 68. Multiplication, meaning repeated concatenation, of values of mode CHAR or [] CHAR is defined using the operator \* . The operator takes a [] CHAR **operand** and a INT **operand**, and the yield has mode [] CHAR. For example, in the **declaration**:

```
[] CHAR word = 3 * "ab"
```

the **formula** yields "ababab". The **formula** may also be written with the integer as the right-hand **operand** "ab" \* 3. In both cases, the operator only makes sense with a positive integer.

The operators = and /= are also defined for **operands** of mode [] CHAR. Corresponding elements must be equal if the = operator is to yield TRUE. Thus:

```
"a" = "abc"
```

yields FALSE. Note that the **bounds** do not have to be the same:

```
([] CHAR a = "Rose"[@0], b = "Rose"; a = b)
```

yields TRUE. The negation of = is /= which test for inequality. So the **formula**:

```
"Algol" /= "Algol"
```

yields FALSE. Alternative representations of /= are ~= , ^= and NE . The ordering operators < , > , <= and >= can be used to compare values of mode [] CHAR in the same way as = and /=. For values of mode [] CHAR, ordering is alphabetic. The **formula**:

```
"abcd" > "abcc"
```

yields TRUE. Two values of mode [] CHAR of different length can be compared. For example, both:

```
"aaa" <= "aaab"
```

```
"aaa" <= "aaaa"
```

yield TRUE. Alternative representations for these operators are LT and GT for < and > and LE and GE for <= and >= respectively. Because the rowing coercion is not allowed for firm **operands in formulas**, the comparison operators are declared in the standard-prelude for mixed modes CHAR and [] CHAR.

Note that apart from values of mode [] CHAR and the interrogation operators LWB, UPB, no operators are defined in the Revised Report for rows. a68g, further to ELEMS, defines pseudo-operators for vectors and matrices (section 4.5) and provides operators to support

linear algebra (section 10.12).

## 4.2.8 Assigning to row names

We can assign values to the elements of a row either individually or collectively.

### 4.2.8.1 Individual assignment

We can access an individual element of a row by subscripting that element. The rules of the language state that a subscripted element of a row name is itself a name. In fact, the elaboration of a **slice** of a row name creates a new name. Unless you store the new name by means of an **identity-declaration**, the new name will cease to exist after the above **assignment** has been elaborated.

### 4.2.8.2 Collective assignment

There are two ways of assigning values collectively. First, this can be done with a **row-display** or a `[] CHAR` **denotation**. For example:

```
[1 : 5] INT first primes := (1, 3, 5, 7, 11);
```

Note that the **bounds** of both `first primes` and the **row-display** are `[1 : 5]`. In the **assignment** of a row, the **bounds** of the row on the right-hand side must match the **bounds** of the row name on the left-hand side. If they differ, a run time error is generated. The second way of assigning to the elements of a row collectively is to use as the source of the **assignment** any **unit** yielding a row of the correct mode with the required **bounds**.

## 4.3 Flexible names

In the previous section, we declared row names. The **bounds** of the row to which the name can refer are included in the **generator**. In subsequent **assignments**, the **bounds** of the new row to be assigned must be the same as the **bounds** given in the **generator**. In Algol 68, it is possible to declare names which can refer to a row of any number of elements (including none) and, at a later time, can refer to row with a different number of elements. These are called flexible names. Consider this **identity-declaration** for a flexible name:

```
REF FLEX [] INT fn = LOC FLEX [1 : 0] INT
```

or, abbreviated:

## INFORMAL INTRODUCTION TO ALGOL 68

```
FLEX [1 : 0] INT fn
```

There are several things to note about this **declaration**:

1. the mode of the name is not REF [] INT, but REF FLEX [] INT. FLEX means that the bounds of the row to which the name can refer can differ from one **assignment** to the next.
2. the **bounds** of the name generated at the time of the **declaration** are [1 : 0]. Since the **upper-bound** is less than the **lower-bound**, the row is said to be flat; it has no elements at the time of its **declaration**.

We can now assign rows of integers to fn:

```
fn := (1, 2, 3, 4)
```

The **bounds** of the row to which fn now refers are [1 : 4]. Again, we can write:

```
fn := (2, 3, 4)
```

Now the **bounds** of the row to which fn refers are [1 : 3]. We can even write:

```
fn := 7
```

in which the right-hand side will be rowed to yield a one-dimensional row with **bounds** [1 : 1], and:

```
fn := ()
```

giving **bounds** of [1 : 0].

If a flexible name is sliced, the resulting name is called a transient name because it can only exist so long as the flexible name stays the same size. Such names have a restricted use to avoid that of names in time could refer to deallocated memory. For example, consider the **declaration** and **assignment**:

```
REF FLEX [] CHAR c1 = LOC FLEX [1 : 0] INT := "abcdef";
```

Suppose now we could have the **declaration**:

```
REF [] CHAR lc1 = c1[2 : 4]; # Wrong! #
```

followed by this **assignment**:

```
c1 := "z";
```

Now lc1 no longer refers to anything meaningful. Thus transient names cannot be stored. They cannot be bound to **identifiers**, nor used as **parameters** for a routine (whether operator or procedure). However they can be the destination in an **assignment**, as in:



```
STRING s := "abcdefghijklmnopqrstuvwxy";
s[2 : 7] := s[9 : 14]
```

where the name yielded by `s[9 : 14]` is immediately dereferenced. So the **assignment**:

```
s[2 : 7] := "abc"
```

would produce a run time error.

What has not been made apparent up to now is a problem arising from the ability that we can have rows of any mode except `VOID`, so also rows of rows, et cetera. Now consider the **declaration**:

```
FLEX [1 : 0][1 : 3] INT flexfix
```

Because the mode of `flexfix` is `REF FLEX [][] INT`, when it is subscripted, the mode of each element is `REF [] INT` with **bounds** `[1 : 3]`. Clearly, after the **declaration**, `flexfix` has no elements, so how would we know about dimensions of the `REF [] INT` sub-row of `flexfix`? According to the Revised Report a row must have a *ghost element*, inaccessible by subscripting, to preserve information on **bounds** in case no elements are present. This ghost element prohibits writing:

```
flexfix := LOC [1 : 4][1 : 4] REAL
```

although `a68g` will allow this last **assignment** since it does not implement ghost elements.

Algol 68 has been criticised for offering flexible rows, but not a simple way to extend an existing row with a number of elements while leaving the present elements untouched. This has to be achieved by declaring a new (larger) flexible row, assigning existing elements to it, and then copying back:

```
INT n = read int;
FLEX [n] INT u;
...
# Extend u by one element #
[UPB u + 1] INT v;
v[: UPB u] := u;
u := v
```

after which the row is extended with one yet uninitialised element.

### 4.3.1 The mode `STRING`

The mode `STRING` is defined in the standard-prelude as `FLEX [1 : 0] CHAR`. That is, the **identity-declaration**:

```
REF STRING s = LOC STRING
```

## INFORMAL INTRODUCTION TO ALGOL 68

has exactly the same effect as the **declaration**:

```
REF FLEX [] CHAR s = LOC FLEX [1 : 0] CHAR
```

You will notice that although the **mode-indicant** `STRING` appears on both sides of the **identity-declaration** for `s`, in the second **declaration** the **bounds** are omitted on the left-hand side (the mode is a formal-declarer) and kept on the right-hand side (the **actual-declarer**). Without getting into grammatical explanations, just accept that if you define a mode like `STRING`, whenever it is used on the left-hand side of an **identity-declaration** the compiler will ignore the **bounds** inherent in its definition. We can now write:

```
s := "String"
```

which gives **bounds** of `[1 : 6]` to `s`. We can **slice** that row to get a value with mode `REF CHAR` which can be used in a **formula**. Often, where `[] CHAR` appears, it may be safely replaced by `STRING`. This is because it is only names which are flexible so the flexibility of `STRING` is only available in `REF STRING` **declarations**.

When reading `STRING` values, reading will not go past the end of the current line<sup>1</sup>. If the reading position is already at the end of the line, the row will have no elements. When reading a `STRING`, `new line` must be called explicitly for transport to continue.

Two operators are defined in the standard-prelude which take an **operand** of mode `REF STRING`: `PLUSAB`, whose left **operand** has mode `REF STRING` and whose right **operand** has mode `STRING` or `CHAR`, and `PLUSTO`, whose left **operand** has mode `STRING` or `CHAR` and whose right **operand** has mode `REF STRING`. Using the concatenation operator `+`, their actions can be summarised as follows:

1. `a PLUSAB b` means `a := a + b`
2. `a PLUSTO b` means `b := b + a`

Thus `PLUSAB` concatenates `b` onto the end of `a`, and `PLUSTO` concatenates `a` to the beginning of `b`. Their alternative representations are `+=` and `+=` respectively. For example, if `a` refers to `"abc"` and `b` refers to `"def"`, after `a PLUSAB b`, `a` refers to `"abcdef"`, and after `a PLUSTO b`, `b` refers to `"abcdefdef"` (assuming the `PLUSAB` was elaborated first).

At the end of this section we demonstrate how row variables and trimming can be used in a practical situation which involves circular assignment of a string:

```
BEGIN STRING u := "Barbershop";
  TO UPB u + 1
  DO print ((u, new line));
  CHAR v = u[UPB u]; STRING w = u[1 : UPB u - 1];
  u[2 : UPB u] := w;
```

---

<sup>1</sup>For details of string terminators see {7.6}.

```

    u[1] := v
  OD
END

```

which produces:

```

$ a68g barbershop.a68
Barbershop
pBarbersho
opBarbersh
hopBarbers
shopBarber
rshopBarbe
ershopBarb
bershopBar
rbershopBa
arbershopB
Barbershop

```

## 4.4 Vectors, matrices and tensors

Algol 68 Genie has extensions to support vectors, matrices and tensors provided they are represented as rows. Next section shows how to extract a transpose, or a diagonal from a matrix. The library-prelude {see 10.12} provides a simple vector and matrix interface to Algol 68 rows of mode:

```

[] REAL # vector #
[, ] REAL # matrix #
[] COMPLEX # complex vector #
[, ] COMPLEX # complex matrix #

```

These routines require the GNU Scientific Library.

## 4.5 Torrix extensions

a68g implements pseudo-operators as described by [Torrix 1977]. These are of particular interest to vector - and matrix algebra. Original Torrix code implements these symbols as operators on one- and two-dimensional rows of real and complex numbers. The pseudo-operator implementation offered by a68g is more general as it works on one- and two-dimensional rows of any mode. The syntactic position of these pseudo-operator expressions is at the same level as a **formula**, which is a **tertiary** as we will see in a forthcoming chapter. Next list compiles the definitions of these pseudo-operators. Note that all yield a

## INFORMAL INTRODUCTION TO ALGOL 68

descriptor. This means that the yield of the pseudo-operator refers to the same elements as the row operated on, only the indices are mapped. In the list below  $a$  is a two-dimensional row:

$$a = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{n2} & \dots & a_{mn} \end{pmatrix}$$

$u$  is a one-dimensional row:

$$u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}$$

and  $i$ ,  $j$  and  $k$  are integers. Next pseudo-operators are available:

1. TRNSP constructs for a matrix, without copying, a descriptor such that:

$$(\text{TRNSP } a)[j, i] = a[i, j]$$

for valid  $i$  and  $j$ .

2. DIAG constructs for a square matrix, without copying, a descriptor such that:

$$(k \text{ DIAG } a)[i] = a[i, i + k]$$

for valid  $i$  and  $k$ . The monadic form of DIAG is equivalent to  $0 \text{ DIAG } \dots$ .

3. COL constructs, without copying, a descriptor such that:

$$(k \text{ COL } u)[i, k] = u[i]$$

for valid  $i$  and  $k$ . The monadic form of COL is equivalent to  $1 \text{ COL } \dots$ .

4. ROW constructs, without copying, a descriptor such that:

$$(k \text{ ROW } u)[k, i] = u[i]$$

for valid  $i$  and  $k$ . The monadic form of ROW is equivalent to  $1 \text{ ROW } \dots$ .

These pseudo-operators yield a new descriptor, but do no copy data. They give a new way to address the elements of an already existing one - or two dimensional row. For example, next code sets the diagonal elements of a matrix:

```
[3, 3] REAL matrix;
DIAG matrix := (1, 1, 1);
print (DIAG matrix)
```

This will print three ones as the result of the **assignment**. These pseudo-operators delivering new descriptors to existing data cannot be coded in standard Algol 68. There is an analogy with slicing a name. If you apply an above pseudo-operator to an object of mode

[...] MODE, the yield will be of mode [...] MODE, but the number of dimensions will of course change. If you operate on an object of mode REF [...] MODE, the yield will be of mode REF [...] MODE. But if you operate on an object of mode REF REF [...] MODE, the yield will still be of mode REF [...] MODE. This coercion is called weak dereferencing and is explained in section 8.6.3.

## 4.6 A note on brackets

Algol 68 allows square brackets, used when working with rows, to be replaced with parenthesis. a68g supports this feature so in stead of:

```
[3, 3] REAL matrix;
DIAG matrix := (1, 1, 1);
print ((DIAG matrix)[1])
```

you may write:

```
(3, 3) REAL matrix;
DIAG matrix := (1, 1, 1);
print ((DIAG matrix)(1))
```

as long as you close [ with ] and ( with ).

## 4.7 Structured modes

We have seen how a number of individual values can be collected together to form a row whose mode was expressed as *row of mode*. The principal characteristic of rows is that all the elements have the same mode. Often the value of an object cannot be represented by a value of a single object of a standard mode. Think for instance of a book, that has an author, a title, year of publication, ISBN, et cetera. A structure is another way of grouping data elements where the individual parts may be of different modes. In general, accessing the elements of a row is determined at run time by the elaboration of a **slice**. In a structure, access to the individual parts, called fields, is determined at compile time. In some programming languages, structured modes are called *records*. The mode constructor STRUCT is used to create structured modes. This is a **identity-declaration** involving a structure:

```
STRUCT (INT index, STRING title) s = (1, "De bello gallico")
```

The mode of the structure is:

```
STRUCT (INT index, STRING title)
```

## INFORMAL INTRODUCTION TO ALGOL 68

The terms `index` and `title` are called field selectors and are part of the mode. They are not actually **identifiers**, even though the production rule for **identifiers** applies to them. Their only use is to provide access to fields in a structured value. The expression to the right of the **equals-symbol** is called a **structure-display** that has this production-rule:

- **\*structure-display: strong-collateral-clause.**

Like **row-displays**, **structure-displays** can only appear in a strong context. A **structure-display** has two or more fields; allowing a single-field **structure-display** would introduce ambiguity in Algol 68 syntax. In a strong context, `a68g` can determine which mode is required and so it can tell whether a **row-display** or a **structure-display** has been provided. We could now declare another such structure:

```
STRUCT (INT index, STRING title) t = s
```

and `t` would have the same value as `s`.

We can write a structure **declaration** with different field selectors:

```
STRUCT (INT count, STRING title) ss =  
  (1, "Reflexions sur la puissance motrice du feu")
```

which looks almost exactly like the first structure **declaration** above, except that the field selector `index` has been replaced with `count`. The structure `ss` has a different mode from `s` because not only must the constituent modes be the same, but the field selectors must also be identical.

Structure names can be declared:

```
REF STRUCT (INT index, STRING title) sn =  
  LOC STRUCT (INT index, STRING title)
```

Because the field selectors are part of the mode, they appear on both sides of the **declaration**. The abbreviated form is:

```
STRUCT (INT index, STRING title) sn
```

We could then write:

```
sn := s
```

in the usual way, but not:

```
sn := ss
```

The mode of a field can be any mode except `VOID`. For example, we can declare:

```
STRUCT (REAL x, REAL y, REAL z) vector
```

which can be abbreviated to:

```
STRUCT (REAL x, y, z) vector
```

and later on write an **assignment**:

```
vector := (0, 0, 0)
```

where the value 0 would be widened to 0.0 since the right hand side is in a strong context. A structure can also contain another structure:

```
STRUCT (STRING c, STRUCT (REAL x, y) point) ori = ("O", (0, 0))
```

In this case, the inner structure has the field selector `point` with field selectors `x` and `y`. If size of rows is relevant, as in **generators** and **variable-declarations**, the mode of a field selector is an **actual-declarer**. Otherwise, as in an **identity-declaration**, the mode is a **formal-declarer**.

### 4.7.1 Selection

The field selectors of a structured mode are used to extract the individual fields of a structure by means of a **selection** that has following production-rule:

- **selection: identifier, of-symbol, secondary.**

For example, given this **declaration** for the structure `s`:

```
STRUCT (INT index, STRING title) s = (1, "De bello gallico")
```

we can select the first field of `s` using the **selection**:

```
index OF s
```

The mode of the **selection** is `INT` and its value is 1. Note that the construct `OF` is not an operator, as is its equivalent in C. The second field of `s` can be selected using the **selection**:

```
title OF s
```

whose mode is `STRING` with value "De bello gallico". The field selectors cannot be used on their own: they can only be used in a **selection**. A **selection** binds more tightly than any operator, so a **selection** can be used as an **operand**. However, a **slice** or a **call** {see 5} binds more tightly than a **selection**. Consider the **formula**:

```
[inventory size] STRUCT (INT index, STRING title) inventory;
```

```
...
```

```
index OF inventory[1] + 1
```

then first `inventory[1]` is elaborated, then the **selection**, and finally the addition. Sometimes you need to write parenthesis — actually, an **enclosed-clause** — to ensure correct

## INFORMAL INTRODUCTION TO ALGOL 68

elaboration of a construct; consider for instance:

```
STRUCT ([inventory size] INT indices, STRING title) inventory;  
...  
(indices OF inventory)[1] + 1
```

Would you have written the last **unit** as

```
indices OF inventory[1] + 1
```

a runtime error would occur since this strictly means

```
(indices OF (inventory[1])) + 1
```

which involves slicing of the value `inventory` that is not a row in this case, and a68g will protest:

```
32   indices OF inventory[1] + 1  
      1
```

```
a68g: error: 1: REF STRUCT ([] INT indices, STRING title) identifier does  
not yield a row or procedure (detected in particular-program).
```

The two fields of the structure:

```
STRUCT (STRING c, STRUCT (REAL x, y) point) ori
```

can be selected by writing:

```
c OF ori  
point OF ori
```

and their modes are `STRING` and `STRUCT (REAL x, y)` respectively. Now the fields of the inner structure `point` of `ori` can be selected by writing:

```
x OF point OF ori  
y OF point OF ori
```

and both **selections** have mode `REAL`. Note that nested **selection** proceeds from right-to-left.

If you want to assign to a field, the **selection** of that field must somehow yield a name. As with rows, Algol 68 applies the rule that a field selected from a name is itself a name. Consider for instance the structure name `sn` declared by:

```
STRUCT (INT index, STRING title) sn;
```

The mode of `sn` is:

```
REF STRUCT (INT index, STRING title)
```

This means that the mode of the **selection**:



```
index OF sn
```

must be REF INT, and the mode of the **selection**:

```
title OF sn
```

must be REF STRING. That is, the modes of the fields of a structure name get preceded by REF. Otherwise you would not be able to assign to a single field in a structured object. The important general rule is that if you select a field with mode MODE from an object with mode STRUCT (...), then the yield will be of mode MODE as well. If you select a field with mode MODE from an object with mode REF STRUCT (...), then the yield will be of mode REF MODE. But if you select a field with mode MODE from an object with mode REF REF STRUCT (...), then the yield will still be of mode REF MODE. This coercion is called weak dereferencing and is explained in section 8.6.3. Thus, instead of assigning a complete structure using a **structure-display**, you can assign values to individual fields. That is, the **assignment**:

```
sn := (2, "The republic")
```

is equivalent to the **assignments**:

```
index OF sn := 2;
title OF sn := "The republic"
```

except that the two **units** in the **structure-display** are separated by a **comma-symbol** and hence are elaborated collaterally.

Given the **declaration**:

```
STRUCT (CHAR mark, STRUCT (REAL x, y) point) ori;
```

the **selection**:

```
point OF ori
```

has mode REF STRUCT (REAL x, y), and so you could assign directly to it:

```
point OF ori := (0, 0)
```

as well as to its fields:

```
x OF point OF ori := 0;
y OF point OF ori := 0
```

## 4.7.2 Reading and printing structures

Structures are read or printed field-by-field from left to right if the modes of every field can be transput. For example, the following **program** fragment will print a complex number:

## INFORMAL INTRODUCTION TO ALGOL 68

```
STRUCT (CHAR mark, STRUCT (REAL x, y) point) ori := ("O", (0, 0));
print ((ori, new line))
```

For details of how this works, see the remarks on straightening in section 7.9.

### 4.7.3 Rows in structures

If a structure contains rows, the structure **declaration** should only include required **bounds** if it is an **actual-declarer**. For example, we could declare:

```
STRUCT ([] CHAR forename, surname, title)
  lecturer = ("Albert", "Einstein", "Dr")
```

where the mode on the left is a **formal-declarer** (remember that the mode on the left-hand side of an **identity-declaration** is always a **formal-declarer**). When declaring a name, an **actual-declarer** precedes the **identifier**, and **bounds** must be included. A suitable **declaration** for a name which could refer to `lecturer` would be:

```
STRUCT ([7] CHAR forename, [6] CHAR surname, [3] CHAR title)
  new lecturer;
```

but we cannot assign `lecturer` to it. A better **declaration** would use `STRING` :

```
STRUCT (STRING forename, surname, title) person
```

in which case we could now write:

```
person := lecturer
```

Using field **selection**, we can write:

```
title OF person
```

which would have mode `REF STRING`. Thus, using field **selection**, we can assign to the individual fields of `person`:

```
surname OF person := "Schweitzer"
```

When slicing a field which is a row, it is necessary to remember that slicing binds more tightly than selecting {see 8}. Thus the first character of the surname of `person` would be accessed by writing:

```
(surname OF person)[1]
```

which would have mode `REF CHAR`. The parentheses ensure that the **selection** is elaborated before the slicing. Similarly, the first five characters of the forename of `person` would be accessed as:

```
(forename OF person)[ : 5]
with mode REF [] CHAR.
```

#### 4.7.4 Multiple selection

In the last section, we considered rows in structures. What happens if we have a row each of whose elements is a structure? If we had declared:

```
[10] STRUCT (REAL re, im) z
```

then the **selection** `re OF z` would yield a name with mode `REF [] REAL` and **bounds** `[1 : 10]`. Because `z` is a name, you can assign to it:

```
re OF z := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

This extraction of a row of fields from a row of a structured value is called **multiple selection**. Multiple **selection** yields a new descriptor, so one could write aliases for the fields:

```
[10] COMPL z;
[] REAL x = re OF z, y = im OF z
```

but note that both `x` and `y` are aliases, so any **assignment** to either one would also affect `z`. To avoid this side effect, the rows should be copied by a **variable-declaration**:

```
[10] STRUCT (REAL re, im) z
[10] REAL x := re OF z, y := im OF z
```

after which both `x` and `y` contain the actual values of `z` but have no connection to `z` anymore. Selecting the field of a sliced row of structured elements is straightforward. Since the row is sliced before the field is selected, no parentheses are necessary. Thus the real part of the third `STRUCT (REAL re, im)` of `z` above is given by the expression:

```
re OF z[3]
```

## 4.8 Field-selections

Algol 68 was one of the first languages to introduce structured values, and the production-rule for a **selection**:

- **selection: identifier, of-symbol, secondary.**

nowadays appears a bit odd since this syntax evaluates from right-to-left while most other languages evaluate **selections** from left-to-right. Compare for instance the Algol 68 construction:

```
x OF point OF z
```

which in C or Pascal would read<sup>2</sup>:

```
z.point.x
```

a68g offers alternative syntax to the classic Algol 68 **selection**, called a **field-selection**, which is described by this production-rule:

- **field-selection:**  
**weak-primary, sub-symbol, identifier-list, bus-symbol.**

by which you can write above example in a68g as:

```
z[point, x] or z(point, x)
```

which is elaborated as if you would have written:

```
x OF point OF z
```

The **field-selection** is a **primary** and is very similar to a **call** (see 5.4) or a **slice**. Unlike a **selection**, a **field-selection** cannot perform multiple selection; if it could, then for example in a range containing the declaration `INT im, [3] COMPL z`, the **primary** `z[im]` would be ambiguous.

## 4.9 Mode-declarations

Structure **declarations** are very common in Algol 68 **programs** because they are a convenient way of grouping disparate data elements, but writing out modes in every **declarer** is clumsy and error-prone. Using the **mode-declaration**, a new **mode-indicant** can be declared to indicate a mode. Relevant production-rules are:

- **mode-declaration:**  
**mode-symbol, mode-definition-list.**
- **mode-definition:**  
**mode-indicant, equals-symbol, actual-declarer.**

An indicant can be an upper-case tag as `RECORD` or `VECTOR`. a68g accepts a tag that starts with an upper-case letter, optionally followed by upper-case letters or underscores. Since

---

<sup>2</sup>For convenience we ignore the distinction between the `.` and `->` operators.

spaces are not allowed in an upper-case tag to avoid ambiguity, underscores can be used to improve legibility. The use of underscores in tags is not allowed in standard Algol 68. For example, the **mode-declaration**:

```
MODE VECTOR = STRUCT (REAL x, y, z)
```

or

```
MODE VECTOR = [1 : 3] REAL
```

makes VECTOR a synonym for the mode specification on the right-hand side of the **equals-symbol**, and new objects using VECTOR can be declared in the ordinary way:

```
VECTOR vec = (1, 2, 3);
VECTOR vn := vec;
[10] VECTOR va;
MODE TENSOR = STRUCT (VECTOR x, y, z)
```

The mode STRING is declared in the standard-prelude as:

```
MODE STRING = FLEX [1 : 0] CHAR
```

Note that **bounds** are conveniently ignored when a newly declared **indicant** is used as a **formal-declarer**, for instance in an **identity-declaration**. Now consider this small **program** using matrix objects:

```
BEGIN INT n;
  MODE MATRIX = [n, n] REAL;
  WHILE print ((new line, "Give the dimension (n): "));
    read (n);
    n > 0
  DO MATRIX m;
    FOR i TO 1 UPB m
      DO FOR j TO 2 UPB m
        DO m[i, j] := 1 / (i + j - 1)
          OD
        OD;
      print (("Hilbert matrix:", new line, m))
    OD
  END
```

In the above **declaration** involving MATRIX, the **bounds** will be elaborated at the **declaration** of any name of mode MATRIX. When MATRIX is used as a **formal-declarer**, the **bounds** are ignored and not evaluated.

Suppose you want a mode which refers to another mode which has not yet been declared before, and a second mode that refers to the first mode, for example:

## INFORMAL INTRODUCTION TO ALGOL 68

```
MODE A = STRUCT (STRING title, REF B next),  
  B = STRUCT (STRING name, REF A next)
```

This can for instance not be written in Pascal or C without using some sort of *forward declaration* in Pascal or *incomplete type* in C. In Algol 68, tags like **identifiers**, **indicants** or **operators** do not have to be declared before they are applied, so you can straightforwardly declare the two modes as listed above. The syntax of Algol 68 forces that a mode cannot give rise to (1) an infinitely large object or (2) endless coercion. Using a **mode-declaration**, you might be tempted to declare a mode such as:

```
MODE CIRCULAR = STRUCT (INT i, CIRCULAR c) CO wrong! CO
```

but this is not allowed since a **declaration**:

```
CIRCULAR z;
```

would quickly consume all memory on your system and then complain that memory is exhausted. Next **declaration** will also give an infinitely large object:

```
MODE BINARY = [1 : 2] BINARY
```

and is therefore not allowed. However, there is nothing wrong with modes as:

```
MODE LIST = STRUCT (STRING s, REF LIST next)
```

because only a reference to itself is declared within the structure. We will say that `REF` shields a mode definition from its application.

### 4.9.1 Recursive mode-declarations

A curiosity of Algol 68 that does not receive a lot of attention is that since a tag can be applied before it is declared, application in actual **bounds** of a rowed mode while it is being declared can make a **mode-declaration** recursive. This may lead to fuzzy code like this example:

```
MODE FAC = [1 : (n > 0 | m *:= n; n -= 1; LOC FAC; n | 0)] INT;  
INT m := 1, n := 10;  
LOC FAC;  
print ((m, n))
```

Such code actually runs under a68g:

```
$ a68g rec_mode.a68  
+3628800 +0
```

Using this type of recursion will be regarded as amusing by some, but most will agree that this is not a recommended programming style.

## 4.10 Complex numbers

The standard-prelude contains the **mode-declaration**:

```
MODE COMPL = STRUCT (REAL re, im)
```

a68g considers the symbols COMPL and COMPLEX as equivalent.

Multi-precision **declarations** exist in a68g:

```
MODE LONG COMPL = STRUCT (LONG REAL re, im);
MODE LONG LONG COMPL = STRUCT (LONG LONG REAL re, im)
```

As with modes INT and REAL, the length of LONG LONG modes can be made arbitrarily large through the option `precision` {see 9.6.4}. As described in the previous section, the indicant COMPL can be used wherever a mode is required. From the section on field **selection**, it is clear that in the **declarations**:

```
COMPL z = read complex;
COMPL w := z
```

the **selection**:

```
re OF z
```

yields a value of mode REAL, while the **selection**:

```
re OF w
```

yields a value of mode REF REAL. The predefined **monadic-operator** RE takes a COMPL **operand** and yields its re field with mode REAL. Likewise, the **monadic-operator** IM takes an **operand** of mode COMPL and yields its im field with mode REAL. Note that the **formula** RE w yields a value of mode REAL, not REF REAL, because RE is an operator whose single **operand** has mode COMPL. In the above expression, w will be dereferenced before RE is elaborated. Thus it is not valid to write:

```
RE w := 0
```

which should be written as:

```
re OF w := 0
```

In a strong strong context, a real number will be widened to a complex number. So, for example, in the following **identity-declaration**:

```
COMPL z = pi
```

z will have the same value as if it had been declared by:

```
COMPL z = (pi, 0)
```

Next to using a **row-display** to denote a complex number, the predefined **dyadic-operator** `I` can be used taking left- and right-**operands** of any combination of `REAL` and `INT` yielding a `COMPL` value. It has a priority of 9. For example `-1 I 1` yields  $i - 1$ . Of course, there is a **declaration** for `I` that takes left- and right-**operands** of any combination of `LONG REAL` and `LONG INT` and yields a `LONG COMPL` value, and a **declaration** that takes left- and right-**operands** of any combination of `LONG LONG REAL` and `LONG LONG INT` and yields a `LONG LONG COMPL` value.

Many operators you need to manipulate complex numbers have been declared in the standard-prelude. You can use the **monadic-operators** `+` and `-` which have also been declared for values of mode `COMPL`. For a complex number `z`, `CONJ z` yields  $\text{RE } z \text{ I} - \text{IM } z$ . The operator `ARG` gives the argument of its **operand** in the interval  $[-\pi, \pi]$ . The **monadic-operator** `ABS` for a complex number is defined as:

```
OP ABS = (COMPL z) REAL: sqrt (RE z ** 2 + IM z ** 2)
```

Note that in the **formula** `RE z ** 2`, the operator `RE` is monadic and so is elaborated first. The **dyadic-operators** `+`, `-`, `*` and `/` are declared for all combinations of complex numbers, real numbers and integers, as are the comparison operators `=` and `/=`. The **dyadic-operator** `**` is declared for a left hand **operand** of mode `COMPL` and a right hand **operand** of mode `INT`. The assignation operators `TIMESAB`, `DIVAB`, `PLUSAB`, and `MINUSAB` all take a left **operand** of mode `REF COMPL` and a right **operand** of modes `INT`, `REAL` or `COMPL`. In fact, `a68g` supplies **operator-declarations** for all combinations of **operand** precision, always resulting in a value with longest precision of either **operands**. `a68g` implements routines for complex arithmetic that circumvent unnecessary overflow when large real or imaginary values are used. Naive implementation of for example division or `ABS` can overflow while the result is perfectly representable.

### 4.10.1 Complex functions

Routines are the subject of a later chapter, however we have already met the mathematical functions for real values. Algol 68 Genie extends the Revised Report requirements for the standard-prelude by also defining mathematical functions for mode `COMPLEX`. Note that a runtime error occurs if either argument or result are out of range. Multi-precision versions of these routines are declared and are preceded by either `long` or `long long`, for instance `long complex sqrt` or `long long complex ln`. The predefined routines are:



| Function           | Description                          |
|--------------------|--------------------------------------|
| complex sqrt (z)   | Square root of z, $\sqrt{z}$         |
| complex exp (z)    | Yields $e^z$                         |
| complex ln (z)     | Natural logarithm of z, $e \log(z)$  |
| complex sin (z)    | Sine of z, $\sin(z)$                 |
| complex arcsin (z) | Inverse sine of z, $\sin^{-1}(z)$    |
| complex cos (z)    | Cosine of z, $\cos(z)$               |
| complex arccos (z) | Inverse cosine of z, $\cos^{-1}(z)$  |
| complex tan (z)    | Tangent of z, $\tan(z)$              |
| complex arctan (z) | Inverse tangent of z, $\tan^{-1}(z)$ |

## 4.11 BYTES — a compact [] CHAR

The mode `BYTES` is a compact representation of `[] CHAR`. It is a structure with an inaccessible field that stores a row of characters of fixed length in a compact way. A fixed-size object of mode `BYTES` may serve particular purposes (such as file names on a particular file system) but the mode appears of limited use - it was useful in a time when memories were small. On modern hardware there is no reason to use an object of mode `BYTES` in stead of a `[] CHAR`. For reasons of compatibility with old **programs**, a68 implements modes `BYTES` and `LONG BYTES`. Since these modes are of little practical use they are not extensively treated here and you are referred to chapter 10 that lists available operators and procedures for `BYTES` and `LONG BYTES`.

## 4.12 BITS — a compact [] BOOL

The mode `BITS` is a compact representation for a `[] BOOL`. The `BOOL` values are represented as single bits. A `BITS` value is traditionally stored in a machine word, compare it to C type `unsigned`. The typical application is for masks, or to represent small sets where each bit is associated with a set member. The advantage of `BITS` is efficiency: a value uses little storage but also parallel operation on bits for a number of operators. The number of bits in one machine word is given by the environment enquiry `bits width`; on modern hardware this value on a68g will be either 32 or 64. It is important to note that in Algol 68 the most significant bit in a `BITS` value is bit 1 and the least significant bit is bit 32 (or bit 64). Nowadays this seems counter-intuitive, but when Algol 68 was designed this was the way mainframes as the IBM 360 or PDP 10 stored data - the sign bit was bit 0, the most significant bit was bit 1, and the least significant bit was either bit 31 (32-bit machines) or bit 35 (36 bit machines). Sometimes you want to use bits values with more bits than offered by `BITS`. Algol 68 Genie supports modes `LONG BITS` and `LONG LONG BITS`. The range of `LONG LONG BITS` is default circa twice the length of `LONG BITS` but can be

## INFORMAL INTRODUCTION TO ALGOL 68

made arbitrary large through the option `precision` {see 9.6.4}. These are the respective bits widths for the three lengths available in `a68g`:

| Identifier           | Value |
|----------------------|-------|
| bits width           | 32    |
| long bits width      | 116   |
| long long bits width | 232   |

Algol 68 implements strong widening from a `BITS` value to a `[] BOOL` value. Default formatting in transput of a value of mode `BITS` is as a row of `bool`. A `BITS` value can also be denoted in four different ways using **denotations** written with radices of 2 (binary), 4, 8 (octal) or 16 (hexadecimal). The **denotations** in next **declaration**:

```
BITS a = 2r001011101101,  
      b = 4r23231,  
      c = 8r1355,  
      d = 16r2ed
```

are all equivalent because they all denote the same value. Note that the radix precedes the letter `r` and is written in decimal. Recall that numbers can be written with spaces, or new lines, in the middle of the number; but you cannot put a **comment** in the middle of the number. In standard Algol 68, a **denotation** for `LONG BITS` must be preceded by the reserved word `LONG` and a **denotation** for `LONG LONG BITS` must be preceded by the reserved words `LONG LONG`. As with **integer-denotations** and **real-denotations**, the `a68g` interpreter relaxes the use of prefixes when the context imposes a mode for a **denotation**, in which case a **denotation** of a lesser precision is automatically promoted to a **denotation** of the imposed mode.

### 4.12.1 Operators for `BITS`

There are many operators for `BITS` values. These are called bit-wise operators because their action on each bit is independent of the value of other bits. The **monadic-operator** `BIN` takes an `INT` **operand** and yields the equivalent value with mode `BITS`. The operator `ABS` converts a `BITS` value to its equivalent with mode `INT`. The `NOT` operator which you first met in chapter 2.5 for a Boolean value (section 2.11), takes a `BITS` **operand** and yields a `BITS` value where every bit in the **operand** is reversed, corresponding to forming a set of all members not present in the original set.

Three dyadic-operators taking two `BITS` **operands** are `AND`, `OR` (both of which you also met in chapter 2.5 for values of mode `BOOL`) and `XOR`. All three take two `BITS` **operands**

and yield a `BITS` value. They are bit-wise operators with following action:

| Left bit | Right bit | AND | OR | XOR |
|----------|-----------|-----|----|-----|
| F        | F         | F   | F  | F   |
| F        | T         | F   | T  | T   |
| T        | F         | F   | T  | T   |
| T        | T         | T   | T  | F   |

The priority of `AND` and `XOR` is 3 and the priority of `OR` is 2. The `AND` operator is particularly useful for either forming a set of common members from two **operand** sets. `OR` joins the two **operand** sets while `XOR` forms a set of members present in either **operand** set, but not both.

Other dyadic operators involve setting and extracting single bits in a `BITS` value, which corresponds to accessing individual set members. It is possible to extract a single bit as a Boolean value using the operator `ELEM`. For example, given the **declaration** `BITS set` and supposing we want the third bit (recall that the leftmost bit is bit-1), we could write the following **declaration**:

```
BOOL member 3 = 3 ELEM set
```

Thus, if the third bit is a binary 1, the **declaration** will give the value `TRUE` for bit 3. The priority of `ELEM` is 7. Operators `SET` and `CLEAR` yield a `BITS` value with the bit indicated by the left **operand** set or cleared in the `BITS` value yielded by the right **operand**. The priority of `SET` and `CLEAR` is 7.

Lastly, the **dyadic-operators** `SHL` and `SHR` shift a left hand `BITS` **operand** to the left, or to the right respectively, by the number of bits specified by their right hand `INT` **operand**. When shifting left (`SHL`), bits shifted beyond the most significant part of the word are lost. New bits shifted in from the right are always zero. When shifting right (`SHR`), the reverse happens. For `SHL`, if the number of bits to be shifted is negative, the `BITS` value is shifted to the right and likewise for `SHR`. `UP` and `DOWN` are synonyms for `SHL` and `SHR` respectively. The priorities of `SHL` and `SHR` are both 8.

The operators `=` and `/=` are defined for mode `BITS` with obvious meaning. Noteworthy are the operators `<=` and `>=` that for mode `BITS` are defined as follows:

```
s <= t is equivalent to (s OR t) = t
```

```
s >= t is equivalent to t <= s
```

This means that if objects of mode `BITS` are used to represent sets, that the **formula** `s <= t` tests whether `s` is a subset of `t`.



# Chapter 5

## Routines

### 5.1 Routines

A routine is a set of encapsulated actions which can be elaborated in other parts of the **program**. A traditional use for routines is to avoid duplicate code or to segment large programs, but there are of course many valid reasons to create a routine:

1. Reduce a program's complexity by hiding or abstracting information. For instance, hide the internal workings of your data structure. Hide how a set of equations is solved if the actual way in which it is solved is not relevant to your problem — use routines from a scientific library to do your linear equations; the persons who wrote them are experts in linear algebra, while you probably are just interested in obtaining accurate results.
2. Avoid duplicate code, facilitating maintenance of that code, and also making central points to control the operation of code.
3. Use a routine as a **refinement**, a tool in top-down program construction introduced by N. Wirth. As an alternative to using routines as **refinements**, a68g offers a **refinement** preprocessor (see 8.22 and 9.8.3).

In Algol 68, a routine is used in two ways:

1. As a procedure. The routine is invoked by parameter-less deproceduring {5.3} or by a **call** where an **actual-parameter-list** {5.4} is supplied to a routine.
2. As an operator. The routine is invoked by writing a **formula** {2.5}.

An Algol 68 routine is a value with a well-defined mode. The value of a routine is expressed as a **routine-text** which has following production-rules:

## INFORMAL INTRODUCTION TO ALGOL 68

- **routine-text:**  
  **routine-specification, colon-symbol, strong-unit.**
- **routine-specification:**  
  **parameter-pack-option, formal-declarer.**  
  **parameter-pack-option, void-symbol.**
- **parameter-pack:**  
  **open-symbol, formal-parameter-list, close-symbol.**
- **formal-parameter: formal-declarer, identifier.**

A **routine-text** is a **unit**.

Consider next example **routine-text** that sums the element of a row:

```
([] INT a) INT:
  BEGIN INT sum := 0;
    FOR i FROM LWB a TO UPB a
      DO sum += a[i] OD;
    sum
  END
```

In this example, the **routine-specification** is:

```
([] INT a) INT
```

which could be read as *with row of integer parameter yielding integer*. The mode of the routine `sum` is implicitly given by the **routine-specification** :

```
PROC ([] INT) INT
```

We say that the routine takes one **parameter** of mode `[] INT` and yields a value of mode `INT`. Often **parameters** can be written in a concise manner. Now consider the example **routine-specification**:

```
(REAL x, REAL y, REAL z) BOOL
```

A contraction of identifiers is possible in this case:

```
(REAL x, y, z) BOOL
```

A **routine-text** without its **routine-specification** is sometimes called a *body*. The body of the above **routine-text** is:

```
BEGIN INT sum := 0;
  FOR i FROM LWB a TO UPB a
    DO sum += a[i] OD;
```

```

sum
END

```

The **routine-text** yields the sum of the individual elements of the **parameter** *a*. The body of a **routine-text** is a **unit**. In this case, the body is an **closed-clause**. Since a **routine-text** is a value it can be associated with an **identifier** by means of an **identity-relation**:

```

PROC ([ ] INT) INT sum = ([ ] INT a) INT:
  BEGIN INT sum := 0;
    FOR i FROM LWB a TO UPB a
      DO sum += a[i] OD;
    sum
  END

```

Algol 68 allows not only for procedure constants, but also for procedure variables, so the next **declaration** is also valid:

```

PROC ([ ] INT) INT sum := ([ ] INT a) INT:
  BEGIN INT sum := 0;
    FOR i FROM LWB a TO UPB a
      DO sum += a[i] OD;
    sum
  END

```

Since in **procedure-declarations** the mode is stated on both sides of the **equals-symbol** or **becomes-symbol**, Algol 68 allows an abbreviation according these production-rules:

- **procedure-declaration:**  
**proc-symbol, procedure-definition-list.**
- **procedure-definition:** **identifier, equals-symbol, routine-text.**
- **procedure-variable-declaration:**  
**qualifier-option, proc-symbol, procedure-variable-definition-list.**
- **procedure-variable-definition:** **identifier, becomes-symbol, routine-text.**

For example:

```

PROC sum = ([ ] INT a) INT:
  BEGIN INT sum := 0;
    FOR i FROM LWB a TO UPB a
      DO sum += a[i] OD;
    sum
  END

```

In the **routine-specification** of above routine, `[] INT a` is a **formal-parameter**. At the time the routine is declared, `a` does not identify a value and therefore the size of a row **parameter** is irrelevant. That is why it is called a **formal-parameter**. It is only when the procedure is called that `a` will identify a value. According to the **routine-specification**, the routine must yield a value of mode `INT`. The context of the body of a routine is strong and the terminal **unit** of the **serial-clause** will be coerced. In this case, we have a value of mode `REF INT`, `sum`, which in a strong context will be coerced to a value of mode `INT` by dereferencing. Since **procedure-declarations** are **declarations** like any other, they can appear within other **procedure-declarations**. This is called nesting:

```
PROC print days = (DATE first, last) VOID:
  BEGIN
    PROC day of week = (DATE d) STRING:
      ...
      print ((week day (first), week day (last), new line))
    END
```

In this way you can hide details on the operation of the outer procedure if the inner procedure is not relevant outside the outer procedure.

## 5.2 Routine modes

The mode of a routine starts with the reserved word `PROC`, irrespective whether the routine is a procedure or an operator. A routine has zero or more **parameters**. The mode of the **parameters** may be any mode except `VOID` and the value yielded may be any mode including `VOID`. In the mode of a routine, **identifiers** are omitted. The modes written for the **parameters** and the yield are always **formal-declarers**, so no **bounds** are specified if the modes of the **parameters** or yield involve rows. Consider the **routine-specification**:

```
(REAL x, y, z) BOOL
```

The mode following from this **routine-specification** is:

```
PROC (REAL, REAL, REAL) BOOL
```

In section 4.9 it was remarked that a **mode-declaration** cannot lead to infinitely large objects or endless coercion. It was indicated that `REF` shields a **mode-indicant** from its **declaration** through a `STRUCT`. `PROC` also shields a **mode-indicant** from its **declaration** through a `STRUCT`. A **parameter-pack** shields a **mode-indicant** from its **declaration**. It is therefore possible to declare, for example:

```
MODE NODE = (STRING info, PROC NODE process)
```

or



```
MODE P = PROC (P) P # rather academic, but ok #
```

A routine must yield a value of some mode, but it is possible to discard that value using the voiding coercion. The mode `VOID` has a single value denoted by `EMPTY`. `VOID` cannot be a **formal-declarer** of a **parameter** to a routine, so it is not allowed to write:

```
PROC (VOID) VOID # which is allowed in C #
```

though the effect of above (invalid) procedure can be obtained in Algol 68 by writing:

```
PROC VOID # a parameter-less procedure yielding VOID #
```

A routine, with or without **parameters**, can have a result mode `VOID` in case the routine performs actions but yields no value. In some languages like Fortran this is the distinction between a `SUBROUTINE` and a `FUNCTION` — the former only performs actions, the latter (also) yields a value.

In programming practice, because the context of the yield of a routine is strong and any resulting value is voided {see 8.6.8}, explicitly using `EMPTY` is almost always unnecessary. A `FOR` loop always yields `EMPTY`. A **semicolon-symbol** voids the **unit** that it terminates. **Declarations** yield no value, not even `EMPTY`. Therefore clauses cannot end in a **declaration**, as demonstrated by this a68g example:

```
$ a68g -e "PROC init = VOID: (INT i := 0)"
1      (PROC init = VOID: (INT i := 0))
      2                                1
a68g: syntax error: 1: clause does not yield a value.
a68g: syntax error: 2: clause does not yield a value.
```

## 5.3 Routines without parameters

Procedures can have no **parameters** at all; for example consider:

```
PROC report = VOID: print ("Now at point ", counter += 1);
```

A parameter-less procedure can be invoked by writing its **identifier**. A parameter-less procedure is invoked by the deproceduring coercion that is available in every context, even in a soft context such as the destination of an **assignment**. Deproceduring is a forced **call** of a parameter-less procedure. If the context requires the mode of the **identifier** itself, a routine without parameters is not invoked, as in:

```
PROC INT get int = read int # read int will not be invoked #
```

In case the context requires a value with mode that is the result mode of the routine, the routine without parameters is invoked, as in:

## INFORMAL INTRODUCTION TO ALGOL 68

```
LONG INT k := read long int # read long int will be invoked #
```

where a `PROC LONG INT` is deprocedured to yield a `LONG INT` value. In case the context imposes voiding, a parameter-less routine is deprocedured and if it is not a `PROC VOID` the value after deproceduring is discarded. To avoid unexpected behaviour, deproceduring is not used to coerce an **assignment**, a **generator** or a **cast** to `VOID`. Hence if we consider:

```
PROC p = VOID: ...;
PROC VOID pp;
pp := p;
```

then voiding the **assignment** `pp := p` does not involve deproceduring `p` after the **assignment** is completed.

## 5.4 Routines with parameters

**Parameters** of procedures can have any mode, including procedures, except `VOID`. Unlike operators, procedures can have any number of **parameters**. A procedure with **parameters** is not implicitly called by deproceduring, but by writing a **call**. A **call** has these production-rules:

- **call:**  
**meek-primary, open-symbol, actual-parameter-list, close-symbol.**
- **actual-parameter:**  
**strong-unit-option.**

In strict Algol 68, an **actual-parameter** is a **strong-unit** while in `a68g` it is a **strong-unit-option** because `a68g` allows for partial-parametrisation {see 5.10}. The most common example of a **call** is writing the **identifier** of a procedure followed by an argument list. The argument list consists of arguments separated by **comma-symbols**; the list is enclosed in parenthesis. For example, the procedure `sum` declared in section 5.1 can be invoked by a **call**:

```
print ((sum ((1, 2, 3, 4, 5, 6, 7, 8, 9, 10)), new line))
```

which will produce 55 on standard output. A **call** binds as tightly as a **slice**, that is, more tightly than a **selection** or a **formula**. This will be treated at length in chapter 8. Algol 68's allows you to write, instead of a procedure **identifier**, any **primary** (see chapter 8) that yields a procedure; for example:

```
r * (on x axis | cos | sin)(angle)
```

calls either `sin (angle)` or `cos (angle)` depending on the `BOOL` value of `on x axis`.

Algol 68's orthogonality allows you to write a **mode-declaration** involving a routine-mode, or declare a structure with a procedure field:

```
MODE METHOD = STRUCT (FUN f, CHAR name),
  FUN = PROC (REAL) REAL;
METHOD sinus := (sin, "sin")
```

In the structure `sinus`, the procedure can be selected by:

```
f OF sinus • sinus[f]
```

which is a **selection** with mode `REF PROC (REAL) REAL`, which is a **procedure-variable**. A **procedure-variable** will be dereferenced before being called. Recall that a **call** binds as tightly as a **slice**, that is, more tightly than a **selection** or a **formula**. Hence if you want to call the procedure `f OF sinus`, you must enclose the **selection** in parentheses:

```
(f OF sinus)(pi) • sinus[f](pi)
```

Since a routine is a value, it is possible to declare values whose modes include a procedure mode, for example:

```
[ ] PROC (REAL) REAL trig = (sin, cos, tan)
```

after which you could write the **call** :

```
trig[read int](read real)
```

In standard Algol 68, a **call** of a routine must supply the same number of actual **parameters**, and in the same order, as there are formal **parameters** in the **procedure-declaration**. However, a68g offers an extension called partial parametrisation, which is treated in section 5.10.

Recall that in Algol 68, **bounds** are not part of a row mode. Since a **formal-parameter** which is a row has no **bounds** written in it, any row having that mode could be used as the **actual-parameter**. This means that if you need to know the **bounds** of the actual row, you will need to use operators interrogating **bounds** as `LWB`, `UPB` or `ELEMS`. An example is a **routine-text** which finds the smallest element in its row **parameter** `a`:

```
([ ] INT a) INT:
  (INT min := a[LWB a];
  FOR i FROM LWB a + 1 TO UPB a
  DO (a[i] < min | min := a[i])
  OD;
  min
  )
```

When a **parameter** is a name, the body of the routine can have an **assignment** which makes the name refer to a new value. For example:

## INFORMAL INTRODUCTION TO ALGOL 68

```
(REF INT a) INT: a := 0
```

Note that the **unit** in this case is a single **unit** and so does not need to be enclosed.

If a flexible name is used as an **actual-parameter**, then the mode of the **formal-parameter** must include the mode constructor `FLEX`. For example:

```
(REF FLEX [] CHAR s) INT:
```

In this example, the mode of `s` could equivalently have been written as `REF STRING`.

You will have to be particularly careful when a **formal-parameter** of a procedure is a flexible name. A mechanism is in place in `a68g` to ensure that one cannot alter the **bounds** of a non-flexible row by aliasing it to a flexible row. This is particularly the case when passing names as **parameters** to procedures. For example, in a range that holds next **declarations**:

```
PROC x = (REF STRING s) VOID: ...,  
PROC y = (REF [] CHAR c) VOID: ...;
```

these problems could occur:

```
x (LOC STRING); # OK #  
x (LOC [10] CHAR); # Not OK, suppose x changes the bounds of s! #  
y (LOC STRING); # OK #  
y (LOC [10] CHAR); # OK #
```

`a68g` issues an error if it encounters a construct that might alter the **bounds** of a non-flexible row.

**Parameters** can be of any mode but `VOID`, so it is possible to pass procedures as **parameters**. Many procedures take a procedure as a **parameter**, for instance:

```
PROC series sum = (INT n, PROC (INT) REAL func) REAL:  
  (LONG REAL s := 0;  
   FOR i TO n  
   DO s += LENG func (i)  
   OD;  
   SHORTEN s  
  )
```

Note that the mode of the procedure **parameter** is a formal mode so no **identifier** is required for its `INT` **parameter** in the **routine-specification** of `sum`. In the **loop-clause**, the procedure is called with an **actual-parameter**. When a **parameter** must be a procedure, any **unit** yielding a routine-text can be supplied. For instance, a predeclared procedure **identifier** can be supplied, as in:

```
PROC rec = (INT a) REAL: 1 / a;
```

```
series sum (1000, rec)
```

or a **routine-text**:

```
series sum (1000, (INT a) REAL: 1 / a)
```

In this case, the routine text has mode `PROC (INT) REAL`, so it can be used in the **call** of `series sum`. Note also that, because the routine text is an **actual-parameter**, its **routine-specification** includes the **identifier** `a`. In fact, **routine-texts** can be used wherever a procedure is required, as long as the **routine-text** has the required mode. The **routine-text** given in the **call** is on the right-hand side of the implied **identity-declaration** of the elaboration of the **parameter**.

## 5.5 Routines and scope

Since the yield of a routine can be a value of any mode, a routine can yield a name, but there is a restriction: the name yielded must have a scope larger than the body of the routine. This means that any names declared to be *local*, cannot be yielded by the routine. The reason for this is simple: when the routine terminates, it discards the local objects it generated, so any reference to those local object would point at something that no longer exists. We have seen that a new name can be generated using the **generator** `LOC`. Now look at this routine which should yield a name generated within its body:

```
(INT a) REF INT: LOC INT := a # wrong #
```

This routine is wrong because the scope of the name generated by `LOC INT` is limited to the body of the routine. `a68g` provides both compile-time and run-time scope checking, and will flag this error. There is a way of yielding a name declared in a routine. This is achieved using a global **generator**:

```
(INT a) REF INT: HEAP INT := a
```

In section 2.16 we saw that **identifiers** have range, but values have scope. The dynamic lifetime of a value is called its scope in Algol 68. A **routine-text** also is a value, but a potential scope problem arises when you write a routine that yields another routine. There is a danger that a routine gets exported to some place where the symbols that the exported routine applies, no longer exist. For example, next **declaration** is wrong:

```
MODE FUN = PROC (REAL) REAL;
MODE OPERATOR = PROC (FUN) FUN;
OPERATOR deriv = (FUN f) FUN: (REAL x) REAL: f (x) - f (x - 1);
```

the danger is that `f` may no longer exist when the routine yielded by `deriv` gets called, and `a68g` will warn you for the potential danger:

```
$ a68g examples/scope.a68 {-}{-}warnings
3     OPERATOR deriv = (FUN f) FUN: (REAL x) REAL: f (x) - f (x - 1);
                                1
a68g: warning: 1: PROC (REAL) REAL value from routine-text could
be exported out of its scope (detected in particular-program).
```

For this reason you cannot export a routine out of the ranges that hold all **declarations** (**identifiers**, operators, modes) that the exported routine applies. A routine is said to have thus a necessary environment outside of which the routine is meaningless.

## 5.6 Declaring new operators

An operator is called monadic when it takes one **operand**, or dyadic when it takes two **operands** in which case it also needs a priority. **Monadic-operators** have priority over any **dyadic-operator**. Operators are declared according these production-rules:

- **brief-operator-declaration:**  
operator-symbol, brief-operator-definition-list.
- **brief-operator-definition-list:**  
operator, equals-symbol, routine-text.
- **operator-declaration:**  
operator-plan, operator-definition-list.
- **operator-plan:**  
operator-symbol, routine-specification.
- **operator-definition-list:**  
operator, equals-symbol, strong-unit.

For example:

```
OP (INT) INT ABS = (INT a) INT: (a >= 0 | a | -a)
```

There are several points to note. The mode of the operator is PROC (INT) INT. That is, it takes a single **operand** of mode INT and yields a value of mode INT. The right-hand side of the **identity-declaration** is a routine text. Since the **routine-text** forces a mode for the operator, an abbreviated **brief-operator-declaration** can be used:

```
OP ABS = (INT a) INT: (a >= 0 | a | -a)
```

An operator symbol can be a tag as ABS or REPR. a68g accepts a tag that starts with an upper-case letter, optionally followed by upper-case letters or underscores. Since spaces are not allowed in an upper-case tag to avoid ambiguity, underscores can be used to improve legibility. The use of underscores is not allowed in standard Algol 68. Many of the operators described up to now are not words but composed of mathematical symbols as + or \*\*. Since spaces have no meaning in between **operator-symbols** composed of mathematical symbols, rules apply to guarantee that any sequence of symbols has one unique meaning. To avoid ambiguity in parsing operator symbol sequences, operator characters are divided into **monads** and **nomads**, the latter group being inherently dyadic. Next rules force that for instance  $1++-1$  can only mean  $(1) + (+(-2))$ , and  $1+>2$  can only mean  $(1) +> (2)$  and not  $(1) + (>2)$ .

1. a **monadic-operator** symbol is a **monad**, optionally followed by a **nomad**.
2. a **dyadic-operator** symbol is either a **monad** or a **nomad**, optionally followed by a **nomad**.
3. an operator symbol consisting of **monads** or **nomads** may be followed by either := or =:.
4. **nomads** are <, >, /, = and \*. A **monadic-operator** cannot start with a **nomad**.

Note that Algol 68 forbids **operator-symbols** that start with a double **monad**, such as ++, - or &&, although **dyadic-operator** symbols starting with a double **nomad** (\*\*, >>, et cetera) are allowed. **Declarations** of operators using above rules are:

```
OP % = (BOOL b) BOOL: b AND read int > 0;
OP -:= = (REF CHAR c, d) INT: c := REPR (ABS c - ABS d)
```

The difference between **monadic-operators** and **dyadic-operators** is that the latter have a priority and take two **operands** in stead of one. Therefore the **routine-text** used for a **dyadic-operator** has two **formal-parameters**. The priority of a **dyadic-operator** is declared using the reserved word `PRIO` :

- **priority-declaration:**  
prio-symbol, priority-definition-list.
- **priority-definition-list:**  
operator, equals-symbol, priority-digit

For example:

```
PRIO ** = 9, +=: = 1
```

The **declaration** of the priority of the operator uses a **priority-digit** in the range 1 to 9 on the right-hand side. Consecutive **priority-declarations** do not need to repeat the

reserved word `PRIOR`, but can be abbreviated in the usual way. The **priority-declaration** relates to the operator symbol, not to the modes of **operands** or result. Hence the same operator cannot have two different priorities in the same range, but there is no reason why an operator cannot have different priorities in different ranges.

## 5.7 Identification of operators

An obvious consequence of being able to declare operators with common **operator-symbols** as `+` or `ELEM` is that there can be more than one **declaration** of the same operator symbol. Think for instance of a `+` for integers, for reals, for complex values, et cetera. This is called operator overloading. How does Algol 68 Genie identify the operator to use when various definitions exist? As in identifying any **declaration**, Algol 68 Genie will first search the range in which it finds the application of that **declaration** (in this case, an operator symbol). If no such **declaration** is in that range, it will search the embedding range, and so on outwards until finally the standard-prelude is searched. Suppose Algol 68 Genie finds a **declaration** with matching operator symbol and matching number of **operands**. How is it identified as the operator to be actually used? To answer that question we must know the coercions that apply to an **operand**. The syntactic position of an **operand** is a firm context. In a firm context we can have (repeated) dereferencing and deproceduring, followed by uniting. Uniting will be discussed in chapter 8. In determining which operator to use, Algol 68 Genie finds, in the smallest range enclosing the **formula**, that **operator-declaration** with correct number of **operands**, whose modes can be obtained from the **operands** in question using coercions allowed in a firm context. This way of identifying operators has an important consequence: if in a same range there could exist two **operator-declarations** taking the same number of **operands** of modes that can be coerced to each other by firm coercions (in which case we say that these modes are firmly related), Algol 68 Genie would have no way to choose between the two. This condition is therefore forbidden, it is not valid Algol 68. You cannot in the same range declare multiple operators whose **operands** are firmly related. For instance you could not declare in one range:

```
OP ? = (INT k) INT: k OVER 2;
OP ? = (REF INT k) BOOL: k IS NIL
```

since this would lead to ambiguous identification:

```
? 9
```

may identify the first **declaration** but:

```
INT k := read int;
? k
```

is ambiguous. The identification of **dyadic-operators** proceeds exactly as for **monadic-**



**operators** except that the most recently declared priority in the same range is used to determine the order of elaboration of operators in a **formula**.

## 5.8 Recursion

An elegant property of Algol 68 procedures and operators is that they can call themselves. This is known as recursion. A paradigm of recursion is the Ackermann function which is a well-known recursive definition in mathematics. This is the Rózsa Péter version of 1935:

```
PROC ack = (INT m, n) INT:
  IF m = 0
  THEN n + 1
  ELSE (n = 0 | ack (m - 1, 1) | ack (m - 1, ack (m, n - 1)))
  FI;
```

The Ackermann function, conceived in 1928, is Turing computable, but not primitive recursive (id est, requires indefinite iteration) and was a counterexample to the belief in the early 1900's that every computable function would also be primitive recursive. This is a function only of interest in number theory, but it does find practical application in testing compilers for their ability to perform deep recursion well since it grows faster than an exponential or even multiple exponential function. For  $m \geq 4$  results get spectacular, for example

$$A(4, n) = 2 \uparrow\uparrow (n + 3) - 3$$

where  $2 \uparrow\uparrow k; k \geq 1$  means 2 squared  $k - 1$  times. Next example is another illustration of a recursive procedure that calculates the number of ways to split an amount of money in coins of 2 Euro, 1 Euro, 50 ct, 20 ct, 10 ct and 5 ct. It applies back-tracking, which is constructing a tree of all possible combinations and cutting off impossible branches as early as possible:

```
PROC count = (INT rest, max) INT:
  IF rest = 0
  THEN 1 # Just right, valid combination found #
  ELIF rest < 0
  THEN 0 # Invalid combination, subtracted too much #
  ELSE [] INT values = (5, 10, 20, 50, 100, 200);
    INT combinations := 0;
    FOR i TO UPB values
    WHILE values[i] <= max
    DO combinations += count (rest - values[i], values[i])
    OD;
    combinations
  FI;
```

```
INT amount = 500 # cts #;
print (count (amount, amount))
```

### 5.8.1 Fast Fourier Transform

Some more points of interest to recursion will be illustrated by a non-trivial example which is a workhorse from numerical analysis: the Fast Fourier Transform or abbreviated, FFT. It can be easily proven that the  $k^{th}$  element of a discrete Fourier transform

$$F_k = \sum_{j=0}^{N-1} e^{2\pi i k j / N} f_j$$

can be rewritten as the sum of two sub-transforms

$$F_k = F_{k,even} + e^{2\pi i k / N} F_{k,odd}$$

where  $F_{k,even}$  is the transform of the even-numbered elements and  $F_{k,odd}$  is the transform of the odd-numbered elements. This splitting of the transform can be applied recursively, meaning that transform  $F_{k,even}$  and  $F_{k,odd}$  can be obtained by calculating their sub-transforms et cetera, up until the point where the transform becomes trivial when there is just one element:

$$F_1 = f_1; N = 1$$

which is an identity. The advantage of binary splitting is that the complexity of a Fourier transform can be greatly reduced. A Fourier transform of length  $N$  requires  $N^2$  operations. The binary splitting reduces that to  $N \log_2 N$  operations. That is an immense improvement for large  $N$ , and many practical datasets are large. This is a naive but instructive implementation of the FFT<sup>1</sup>:

```
OP FFT = (REF [] COMPLEX f) VOID:
  IF # Unnormalised Fast Fourier Transform in recursive form:
    ELEMS f must be a power of 2 and LWB f must be zero. #
    INT length = ELEMS f;
    length = 1
  THEN # When the length is 1, the transform is an identity #
    SKIP
  ELSE INT middle = length OVER 2;
    # Calculate sub-transforms recursively #
    [0 .. middle - 1] COMPLEX f even, f odd;
```

---

<sup>1</sup>Note that this simple form of FFT involves a binary split in every sub-transform, and that therefore the number of elements in  $F$  must be a power of 2. Algorithms that lift this limitation of FFT are out of the scope of this text.

```

FOR i FROM 0 TO middle - 1
DO f even[i] := f[2 * i];
   f odd[i] := f[2 * i + 1]
OD;
(FFT f even, FFT f odd);
# Calculate transform at this level #
FOR k FROM 0 TO middle - 1
DO REAL phi = 2 * pi * k / length;
   COMPLEX w = cos (phi) I sin (phi);
   f[k] := f even[k] + w * f odd[k];
   f[k + middle] := f even[k] - w * f odd[k]
OD
FI;

```

where `(FFT f even, FFT f odd)` is an example of a VOID **collateral-clause** {see 8.10}. Note that the routine does **assignments** to elements of its arguments that must therefore be of mode `REF [] COMPLEX`. When the **parameter** `f` holds a single element, the transform is an identity and the routine just returns. When the length of the **parameter** is a power of two, the transform is calculated by making a binary split into even and odd elements, which are then transformed recursively. Obviously, when the transformation is in progress, various **calls** to `FFT` are made, and every instant of `FFT` — which is called an incarnation of the routine — must wait for the sub-transform to finish, before it can continue. Hence various incarnations of a routine can exist at the same time; only the deepest (youngest) one being active, and the older ones awaiting completion of the younger ones. Algol 68 elaborates every new incarnation of a routine as if the called routine's body were *textually* and if applicable *recursively* inserted into the source code. Algol 68 is defined such that no incarnation can have access to locally defined tags in another incarnation. If such access is needed, such tag must be passed along as a **parameter** when calling a deeper incarnation. Hence every **declaration** in a routine is private to an incarnation; for instance, every incarnation of `FFT` has its own `f even` and `f odd` but also `length` et cetera.

## 5.8.2 Space-filling curves

A beautiful demonstration of the elegance of recursion is the drawing of so-called space-filling curves. The mathematician Giuseppe Peano studied among many other subjects the projection of a square onto a line<sup>2</sup>. He considered a set of curves, constructed from straight line segments. The  $n^{\text{th}}$  member of the set is a continuous curve that passes all points in the square at a distance of at most  $2^{-n}$ , a so-called Peano curve. There are various such sets and here we will demonstrate an algorithm due to A. van Wijngaarden that draws a particular solution that is known as a Hilbert-curve. Drawing the Hilbert-curve actually was an exercise in Algol 68 programming classes. The  $0^{\text{th}}$  member of the curve is trivial,

---

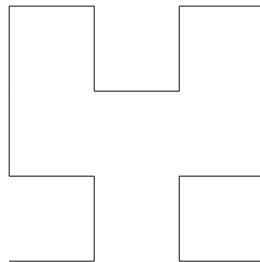
<sup>2</sup>G. Peano. Sur une courbe, qui remplit toute une aire plane. Math. Annln. (36) 157-160 [1890]

## INFORMAL INTRODUCTION TO ALGOL 68

it is a point in the centre of the plane. The first member  $n = 1$  starts at the bottom of the figure and end at its top, we call this orientation "up":



By successive clockwise rotation by  $90^\circ$  we also get orientations "right", "down", and "left". The divide-and-conquer strategy in this problem is to divide the plane into four equal squares, and combine the four orientations to allow drawing a continuous curve by recursively applying our strategy. For example, we construct the orientation "up" of the second member  $n = 2$  in a convenient way:

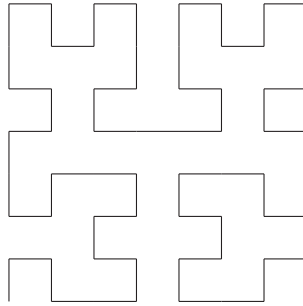


Above figures are drawn using next procedure for orientation "up":

```
PROC draw up = (INT n) VOID:
  IF n > 0
  THEN draw right (n - 1);
    line to ((x OF origin + d, y OF origin));
    draw up (n - 1);
    line to ((x OF origin, y OF origin + d));
    draw up (n - 1);
    line to ((x OF origin - d, y OF origin));
    draw left (n - 1)
  FI;
```

The routine composes orientation "up" of the  $n^{\text{th}}$  member of the curve by combining  $(n-1)^{\text{th}}$  members of the curve until the member  $n = 1$  gets drawn. We can now compose member

$n = 3$  in an analogous way and see how it is composed of  $n = 2$  members:



A complete **program** that was used to make above drawings is in example 11.12. The front page illustration of this book is the  $n = 5$  member of the family drawn with this example **program**. It is left as an exercise to show that the Euclidean length of the Hilbert curve, member  $n$ , reads:

$$2^n - \frac{1}{2^n}$$

that is, grows exponentially.

### 5.8.3 Recursion versus iteration

An iterative routine is likely to be slightly faster in practice than a recursive equivalent because iterative routines do not have the **call** overhead of recursive routines. There are other types of problems whose solutions are inherently recursive, because they need to keep track of prior state. One example is tree traversal; others include the Ackermann function and divide-and-conquer algorithms such as quicksort. All of these algorithms can be implemented iteratively with the help of a stack, but the need for the stack arguably nullifies the advantages of the iterative solution. Another possible reason for choosing an iterative rather than a recursive algorithm is that in today's programming languages, the stack space available to a thread is often much less than the space available in the heap, and recursive algorithms tend to require more stack space than iterative algorithms.

## 5.9 Recursion and data structures

In mathematics and computer science, graph theory is the study of mathematical structures used to model pairwise relations between objects from a certain collection. A graph in this context refers to a collection of vertices or 'nodes' and a collection of edges that connect pairs of vertices. This is not the place to discuss topics in graph theory but it sets

the background for this section. We will demonstrate how to handle data structures where nodes of a same mode refer to each other.

### 5.9.1 Linear lists

A linked list is a fundamental data structure, and can be used to implement other data structures. A linked list is a self-referring data structure because each node contains a pointer to another node of the same type. Practical examples are process queues or lists of free blocks in heap management. Linked lists permit insertion and removal of nodes at any point in the list, but do not allow random access. A linear list is a finite set in which the elements have the relation *is followed by*. We call it linear because each element has one successor. We could of course try to represent it by a row of elements:

```
MODE LIST = FLEX [1 : 0] ELEMENT
```

The principal benefit of a linked list over a row is that the order of the linked items may be different from the order that the data items are stored, allowing the list of items to be traversed in a different order. Also, inserting an element in a row is an expensive operation. An elegant representation is a self-referring data structure:

```
# Data structure and primitive operations: #
MODE NODE = STRUCT (ELEMENT info, LIST successor),
  LIST = REF NODE;
LIST empty list = NIL;
OP ELEM = (LIST list) REF ELEMENT: info OF list,
OP REST = (LIST list) REF LIST: successor OF list
OP EMPTY = (LIST list) BOOL: list IS empty list
OP ELEM = (INT n, LIST list) REF ELEMENT:
  IF EMPTY list OR n < 1
  THEN empty list # thus we mark an error condition #
  ELSE (n = 1 | ELEM list | (n - 1) ELEM list)
  FI;
PRIO ELEM = 9
# A routine to have a denotation for lists: #
PROC make list = ([] ELEMENT row) LIST:
  IF ELEMS row = 0
  THEN empty list
  ELSE HEAP NODE := (row[i], make list (row[LWB row + 1 : UPB row]))
```

```

FI
# Note that REST yields a REF REF NODE, a pointer-variable #
# Operators to add an element to a list: #
# Add at the head #
OP + = (ELEMENT elem, LIST list) VOID:
  HEAP LIST := (elem, list)
# Add at the tail #
OP +:= = (REF LIST list, ELEMENT elem) VOID:
  IF EMPTY list
  THEN list := HEAP LIST := (elem, empty list)
  ELSE REST list +:= elem
FI

```

Several different types of linked list exist:

1. singly-linked lists as discussed above, where each node points at its successor,
2. doubly-linked lists in which each node points to both predecessor and successor,
3. circularly-linked lists in which the last node points back at the first one which is especially useful when implementing buffers.

Linked lists sometimes have a special dummy or sentinel node at the beginning or at the end of the list, which is not used to store data. Its purpose is to simplify some operations, by ensuring that every node always has a previous and next node, and that every list (even an empty one) always has a first and last node. Lisp has such a design — the special value `nil` is used to mark the end of a proper singly-linked list; a list does not have to end in `nil`, in which case it is termed *improper*.

## 5.9.2 Trees

A binary tree is a data structure in which each node refers to two other nodes. Binary trees are commonly used to implement binary search trees and binary heaps. Next is the paradigm quicksort routine in Algol 68:

```

MODE NODE = STRUCT (INT k, TREE smaller, larger),
  TREE = REF NODE;
TREE empty tree = NIL;
PROC sort = (REF TREE root, INT k) VOID:
  IF root IS empty tree

```

## INFORMAL INTRODUCTION TO ALGOL 68

```
    THEN root := HEAP NODE := (k, empty tree, empty tree)
    ELSE sort ((k < k OF root | smaller OF root | larger OF root), k)
    FI;

PROC write = (TREE root) VOID:
  IF root ISNT NIL
  THEN write (smaller OF root);
    print ((whole (k OF root, 0), " "));
    write (larger OF root)
  FI;

TREE root := empty tree;
WHILE INT n = read int;
  n > 0
DO sort (root, n)
OD;
write (root)
```

**Compare the version above with next iterative version, which is the so-called triple-REF trick:**

```
MODE NODE = STRUCT (INT v, REF NODE less, more);
REF NODE empty tree = NIL;

PROC sort = (INT v) VOID:
  BEGIN
    REF REF REF NODE place = LOC REF REF NODE := root;
    WHILE place ISNT empty tree
    DO place := (v < v OF place | less OF place | more OF place)
    OD;
    REF REF NODE (place) := HEAP NODE := (v, empty tree, empty tree)
  END;

PROC write = (TREE root) VOID:
  IF root ISNT NIL
  THEN write (smaller OF root);
    print ((whole (k OF root, 0), " "));
    write (larger OF root)
  FI;

REF REF NODE root = LOC REF NODE := empty tree;
WHILE INT n = read int;
  n > 0
DO sort (n)
OD;
write (root)
```



This is an example of an application of a REF REF REF object, that is, a pointer-to-pointer variable. **Program 11.11** demonstrates building a decision tree in Algol 68. An elaborate example using lists is example **program 11.13** which does **formula** manipulation.

## 5.10 Partial parametrisation and currying

a68g implements C.H. Lindsey's proposal for partial parametrisation [Lindsey AB39.3.1] giving the imperative language Algol 68 a functional sub language [Koster 1996]. A formal description of this proposal can be found in this book as an appendix to the Algol 68 Revised Report {see chapter 20}. With a68g a **call** does not require that all arguments be specified. In case not all of the **actual-parameters** are specified, a **call** yields a routine (with identical body but with already specified arguments stored) that requires the unspecified **actual-parameters**. When specification of all required **actual-parameters** is complete (complete closure) a procedure will actually be evaluated to yield its result. Partial parametrisation is closely related to currying, a transformation named after Haskell Curry. Currying is transforming a function  $f : (x \times y) \rightarrow z$  into  $f : (x) \rightarrow (y \rightarrow z)$ . This transformation is for instance used in  $\lambda$ -calculus.

a68g does not save copies of the stack upon partial parametrisation, as happens for example in Lisp; the yield of a partially parametrised **call** in an environ  $E$ , cannot be newer in scope than  $E$ . Therefore stored **actual-parameters** cannot refer to objects in stack frames that no longer exist, and dynamic scope checking extends to stored **actual-parameters**. A routine may be parametrised in several stages. Upon each stage the yields of the new **actual-parameters** are stored inside the routine's locale and the scope of the routine becomes the newest of its original scope and the scopes of those yields. Following is an example of partial parametrisation:

```
# Raising a routine to a power #
MODE FUN = PROC (REAL) REAL;
PROC pow = (FUN f, INT n, REAL x) REAL: f (x) ** n;
OP ** = (FUN f, INT n) FUN: pow (f, n, );
# Example: sin (3 x) = 3 sin (x) - 4 sin ** 3 (x);
  This follows from DeMoivre's theorem. #
REAL x = read real;
print ((new line, sin (3 * x), 3 * sin (x) - 4 * (sin ** 3) (x)))
```



# Chapter 6

## United modes

### 6.1 United-mode-declarations

In computer science, a union is a data structure that stores one of several types of data at a single location. Algol 68 implements tagged unions. A tagged union, also called a variant, variant record, discriminated union, or disjoint union, is a data structure used to hold a value that could take on several different, but fixed types. Only one of the types can be in use at any one time, and a tag field explicitly indicates which one is in use. `UNION` is used to create a united mode. For example `UNION (INT, STRING)` can either hold an `INT` value or a `STRING` value. Experience shows that united modes are not used very often in actual **programs**. One way of using unions is to save a little memory space when two values can be mapped onto each other, but this argument hardly holds any more considering the specifications of modern hardware. A good way to approach unions is:

1. to use them as an abstraction tool, when you want to explicitly express that an object can be of one of several modes. Transput is a good example of this, for instance:

```
MODE NUMBER = UNION (INT, REAL,  
    LONG INT, LONG REAL,  
    LONG LONG INT, LONG LONG REAL);
```

or think of a Lisp interpreter in Algol 68 where you could declare for example this:

```
MODE VALUE = UNION (ATOM, LIST),  
    ATOM = STRING,  
    LIST = REF NODE,  
    NODE = STRUCT (VALUE car, cdr);
```

Also here, the union serves to provide abstraction.

2. to otherwise use a structure with a field for every mode you need.

## INFORMAL INTRODUCTION TO ALGOL 68

Note that in a union, unlike structures, there are no field selectors. This is because a united mode does not consist of constituent parts. The order of the modes in the union is irrelevant, they are associative; so next unions are equivalent:

```
UNION (INT, STRING) • UNION (STRING, INT)
```

And above unions are equivalent to:

```
UNION (STRING, INT, STRING)
```

since identical modes within a union are absorbed. Like structured modes, united modes are often declared with the mode **declaration**. This is a suitable **declaration** of a united mode containing the constituent modes `STRING` and `INT`:

```
MODE CARDINAL = UNION (STRING, INT)
```

We could create another mode `NUMERAL` in two ways:

```
MODE NUMERAL = UNION (CARDINAL, REAL)
```

or the equivalent phrasing:

```
MODE NUMERAL = UNION (STRING, INT, REAL)
```

Using an above **declaration** for `CARDINAL`, we could declare:

```
CARDINAL u = (roman mood | "MMVIII" | 2008)
```

In this **identity-declaration**, the mode yielded by the right hand side is either `INT` or `STRING`, but the mode required is `UNION (STRING, INT)`. The value on the right-hand side is coerced to the required mode by a coercion called **uniting** which is available in firm and strong contexts. This means that operators which accept **operands** with united modes will also accept **operands** whose modes are any of the constituent modes. We will return to this in a further section. The united mode `CARDINAL` is a mode whose values either have mode `INT` or mode `STRING`. Any value of a united mode actually has a mode which is one of the constituent modes of the union. So there are no new values for a united mode. Because a united mode does not introduce new values, there are no **denotations** for united modes. **identifier** `u` as declared above identifies a value which is either an `INT` or a `STRING`. We will see later how to determine the mode of the value currently contained in a united object. Almost any mode can be a constituent of a united mode. Consider this united mode containing a procedure and `VOID`:

```
MODE OPTIONAL = UNION (PROC (REAL) REAL, VOID)
```

and a **declaration** applying it:

```
OPTIONAL function := EMPTY
```

indicating that `function` is initially undefined as it holds no `PROC (REAL) REAL` value. The only limitation on united modes is that none of the constituent modes may be firmly

related (see the section 5.7) and a united mode cannot appear in its own **declaration**. The following **declaration** is wrong because a value of one of the constituent modes can be deprocedured in a firm context to yield a value of the united mode:

```
MODE WRONG = UNION (PROC WRONG, INT)
```

Names for values with united modes are declared in exactly the same way as before. Look at this **declaration** for a name using a local **generator**:

```
REF UNION (BOOL, INT) un = LOC UNION (BOOL, INT);
```

The abbreviated **declaration** reads:

```
UNION (BOOL, INT) un;
```

Likewise, we could declare a name for the mode **CARDINAL**:

```
CARDINAL sn;
```

In other words, objects of united modes can be declared in the same way as other objects.

## 6.2 United modes as arguments

We can now partly address the problem of the **parameters** for `print` and `read`. It should be possible to construct united modes which will accept all the modes accepted by those two procedures. In fact, these united modes would almost look the same as the two following **declarations**:

```
MODE SIMPLOUT = UNION (
  INT, REAL, BOOL, CHAR,
  [] INT, [] REAL, [] BOOL, [] CHAR,
  [, ] INT, [, ] REAL, [, ] BOOL, [, ] CHAR,
  ...
),
```

```
PROC print = ([] SIMPLOUT) VOID;
```

```
MODE SIMPLIN = UNION (
  REF INT, REF REAL, REF BOOL, REF CHAR,
  REF [] INT, REF [] REAL, REF [] BOOL, REF [] CHAR,
  REF [, ] INT, REF [, ] REAL, REF [, ] BOOL, REF [, ] CHAR,
  ...
),
```

```
PROC read = ([] SIMPLIN) VOID;
```

## INFORMAL INTRODUCTION TO ALGOL 68

As you can see, the mode `SIMPLIN` used for `read` is united from modes of names. The modes `SIMPLOUT` and `SIMPLIN` are more complicated than this (see chapters 9 and 11), but you now get the basic idea. To demonstrate the uniting coercion in a **call** of `print`, consider next example. If `a` has mode `REF INT`, `b` has mode `[] CHAR` and `c` has mode `PROC REAL`, then the **call**:

```
print ((a, b, c))
```

causes the following to happen:

1. `a` is dereferenced to mode `INT` and then united to mode `SIMPLOUT`.
2. `b` is united to mode `SIMPLOUT`.
3. `c` is deprocedured to produce a value of mode `REAL` and then united to mode `SIMPLOUT`.
4. The three elements are regarded as a **row-display** for a `[] SIMPLOUT`.
5. `print` is called with its single **parameter**.

`print` uses a **united-case-clause** (see next section) to extract the actual value from each element in the row. A curiosity is that although `read` and `print` take a united argument, you cannot read a united value, since united modes do not introduce new values. You have to read a value of a constituent mode, which requires that the united value is initialised before `read` is attempted. For example, next code fails:

```
UNION (INT, REAL) number;  
read (number);
```

An attempt at execution generates a runtime error:

```
2   read (number);  
   1  
a68g: runtime error: 1: attempt to use uninitialised  
UNION (REAL, INT) value (detected in particular-program).
```

since at the time of the **call** to `read`, it is not yet known whether `number` holds an `INT` or a `REAL`. This can be arranged for instance by initialising `number`:

```
UNION (INT, REAL) number := 0.0;  
read (number);
```

after which `read` will expect a `REAL` value from `stdin`.

### 6.3 United-case-clauses

We will now discuss how a value can be extracted from a united mode since its constituent mode cannot be determined at compile-time. There is no de-uniting coercion in Algol 68 and the constituent mode of the value must be determined using a variant of the **integer-case-clause** {see 3.5} which is called a **united-case-clause** or **conformity-clause**. The production-rules for the **united-case-clause** read:

- **united-case-clause:**  
**case-symbol, meek-united-enquiry-clause,**  
**in-symbol, specified-unit-list,**  
**united-ouse-part-option,**  
**out-part-option,**  
**esac-symbol.**
- **specified-unit:**  
**open-symbol, formal-declarer, identifier-option, close-symbol, colon-symbol,**  
**unit.**  
**open-symbol, void-symbol, close-symbol, colon-symbol, unit.**
- **united-ouse-part:**  
**ouse-symbol, meek-united-enquiry-clause,**  
**in-symbol, specified-unit-list,**  
**united-ouse-part-option.**
- **out-part:**  
**out-symbol, serial-clause.**

For our discussion, we return to the **declaration**:

```
MODE CARDINAL = UNION (STRING, INT);
CARDINAL u = (roman mood | "MMVIII" | 2008)
```

The constituent mode of `u` can be determined by:

```
CASE u
IN (INT): print ("decimal mood"),
  (STRING): print ("roman mood")
ESAC
```

If the constituent mode of `u` is `INT`, the first case will be selected. Note that the mode selector is enclosed in parentheses and followed by a **colon-symbol**. Above example could also have been written as:

```
CASE u
IN (STRING): print ("roman mood")
OUT print ("decimal mood")
ESAC
```

## INFORMAL INTRODUCTION TO ALGOL 68

Usually, when we determine the mode of an united object with a **united-case-clause**, we also want to extract the value to operate on it. This can be done in this way:

```
CASE u
IN (INT i): print ("decimal mood: ", i),
   (STRING s): print ("roman mood: ", s)
ESAC
```

In this example, the **declarer** and **identifier**, which is called the specifier, act as the left-hand side of an **identity-declaration**. The value can be used, but not changed as it is declared using an identity **declaration**, in the **unit** following the **semicolon-symbol**. A specifier can also select a united subset of a united mode. For example:

```
MODE ICS = UNION (INT, CHAR, STRING);

# Now define multiplication with an INT: #

OP * = (ICS a, INT b) ICS:
CASE a
IN (UNION (STRING, CHAR) ic):
   (ic | (CHAR c): c * b, (STRING s): s * b),
   (INT n): n * b
ESAC
```

Note that **united-case-clauses** do not usually have an **OUT** clause. One usually defines a specific action by a **united-case-clause** on all possible modes at once.

## 6.4 Well-formed modes

It was already pointed out that not all possible **mode-declarations** are allowed in Algol 68. The syntax of Algol 68 forces that a mode cannot give rise to (1) an infinitely large object or (2) endless coercion. Now that the description of mode constructing elements is complete, we can give the rules for determining whether a mode is well-formed. There are two reasons why a mode might not be well-formed:

1. elaboration of a **declaration** using that mode would generate an infinitely large object:

```
MODE FRACTAL = [100] FRACTAL,
LIST = STRUCT (INT index, LIST next)
```

2. coercion of that mode leads to an endless or ambiguous sequence of coercions:



```
MODE POINTER = REF POINTER,
  CELL = UNION (STRING content, REF CELL next)
```

All non-recursive **mode-declarations** are well-formed. It is only in recursive and mutually-recursive modes that we have to apply a test for well-formedness. Let us look at some examples of modes which are not well-formed. Consider the invalid **mode-declaration**:

```
MODE LIST = STRUCT (INT index, LIST next)
```

where field `LIST next` would expand to a further `STRUCT` and so on *ad infinitum*. However, if the mode within the `STRUCT` is shielded by `REF` or `PROC`, then the **mode-declaration** is valid since an object of such mode will be of finite size:

```
MODE LIST = STRUCT (INDEX index, REF LIST a)
```

Likewise, the **declaration**:

```
MODE LIST = STRUCT (INT index, PROC LIST action)
```

is well-formed because in any **declaration**, the second field is a procedure (or a name referring to such a procedure) which is not the original structure and so does not generate objects of infinite size. A `UNION` does not shield the mode as does a `STRUCT`. The **mode-declarations**:

```
MODE A = UNION (INT, REF A);
MODE B = STRUCT (UNION (CHAR, B) u, CHAR c)
```

are not well-formed. If we would have an object of mode `REF A` that would need coercion to `A` in a firm or strong context, the coercion could either be dereferencing or uniting, which is ambiguous. `B` again leads to objects of infinite size since `B` must be expanded to know the size of the union. All the above **declarations** have been recursive, but not mutually recursive. It is not possible to declare:

```
MODE A = STRUCT (B a, INT i),
  B = STRUCT (A a, CHAR i)
```

since the elaboration of **declarations** using either mode would generate an infinite object, so the modes are not well-formed. The following pair of **mode-declarations** are however well-formed since proper shielding is provided:

```
MODE A = STRUCT (REF B a, INT i),
  B = STRUCT (PROC A a, CHAR i)
```

### 6.4.1 Determination of well-formedness

## INFORMAL INTRODUCTION TO ALGOL 68

In any mutually-recursive **mode-declarations**, or any recursive **mode-declaration**, to get from a particular mode on the left hand side of a **mode-declaration** to the same **mode-indicant** written on the right hand side of a **mode-declaration**, it is necessary to traverse various mode constructors. Above each STRUCT or PROC with **parameters**, write *yang*. Above each REF or parameter-less PROC write *yin*. A UNION gets neither *yin* nor *yang*. Now trace the path from the **mode-indicant** in question on the left-hand side of the **mode-declaration** until you arrive at the same **mode-indicant** on the right-hand side. If you have traversed at least one *yin* and at least one *yang*, the mode is well-formed — we say that the definition is *properly shielded* from its application. Let us demonstrate this method using a recursive **mode-declaration**:

```
MODE LIST = STRUCT (INT index, LIST next)
```

Now write *yang* above STRUCT. To get from LIST on the left to LIST on the right, only a *yang* is traversed thus LIST is not well-formed. In the **mode-declaration**:

```
MODE LIST = STRUCT (INDEX index, REF LIST next)
```

we traverse a *yang* (STRUCT) and a *yin* (REF), so LIST is well-formed. More examples are:

1. MODE A = INT  
is well-formed.
2. MODE B = PROC (B) VOID  
is well-formed.
3. MODE B = PROC (B) B  
is well-formed.
4. MODE C = [3, 3] C  
is not well-formed.
5. MODE D = STRUCT (BOOL p, D m)  
is not well-formed.
6. MODE E = STRUCT (STRING s, REF E m)  
is well-formed.
7. MODE A = STRUCT (REF B f),  
B = PROC (INT) A  
is well-formed.
8. MODE A = PROC (B) VOID,  
B = STRUCT (A a)  
is well-formed.
9. MODE A = PROC (B) A,  
MODE B = STRUCT (PROC C c, STRUCT (B b, INT i) d),

```
MODE C = UNION (A, B)
is not well-formed.
```

## 6.5 Equivalence of modes

In Algol 68, modes are equivalent if they have an equivalent structure. This principle is called structural equivalence. Compare this for example with Pascal where two objects are only of the same mode (type, in that language) when they are declared with the same type identifier. The rules for structural equivalence in Algol 68 are intuitive:

1. rows are of equivalent mode when the elements have equivalent modes. **Bounds** are not part of a mode.
2. structures are equivalent when fields in a same position have equivalent modes and identical field names. Field names are part of a mode.
3. procedures are equivalent when **parameters** in a same position have equivalent modes, and the result modes are equivalent,
4. unions are equivalent when for every constituent mode in one there is an equivalent constituent mode in the other.

For instance, these two modes are equivalent:

```
MODE COMPLEX = STRUCT (REAL re, im), PERPLEX = STRUCT (REAL re, im)
```

but these two are not equivalent:

```
MODE COMPLEX = STRUCT (REAL re, im), POLAR = STRUCT (REAL r, theta)
```

and these are equivalent:

```
MODE FUNC = UNION (COMPLEX, PROC (COMPLEX) PERPLEX),
  GUNC = UNION (PERPLEX, PROC (PERPLEX) COMPLEX)
```

As you are well aware by now, **mode-declarations** can get quite entangled. The algorithm that a68g uses to determine the equivalence of two modes, is to prove equivalence assuming that the two are equivalent. The latter assumption is necessary since in many cases a proof of equivalence eventually comes down to cyclic sub-proofs as  $A = B$  if and only if  $A = B$ , and mentioned assumption resolves these situations. Structural equivalence has an important consequence when defining modes for unrelated quantities. For instance, one could write:

```
MODE WEIGHT = REAL, DISTANCE = REAL;
WEIGHT w = read real; DISTANCE r = read real;
print (w + r) # ? #
```

## INFORMAL INTRODUCTION TO ALGOL 68

It makes of course no sense to add a `WEIGHT` to a `DISTANCE`, but since both are structurally equivalent, a `REAL`, it is impossible to set them apart in Algol 68. We could pack them in a structure:

```
MODE WEIGHT = STRUCT (REAL u), DISTANCE = STRUCT (REAL v);
```

but this is no more than a trick, for example used in **program** 11.13. In Algol 68 there is no simple way to abstract from a mode's representation, which is considered a weakness of the language.

# Chapter 7

## Transput

### 7.1 Transput

Transput is a portmanteau word for *input and output*. At various points in this book we have been reading data from standard input and writing data to standard output. These normally are your keyboard and screen, respectively. This chapter addresses the means whereby an Algol 68 **program** can obtain data from other devices and send data to devices other than the screen. Data transfer is called input-output in many other languages, but in Algol 68 it is called *transput*. We will use this term as a verb: we will use *to transput* meaning *to perform input or output*. This chapter describes transput as implemented by a68g. Though a68g transput preserves many characteristics of Algol 68 transput, there are differences. According to the Revised Report, Algol 68 discriminates *files* and *books*. A *file* described how to handle a *book* that represented a dataset having pages and lines, which in the 1970's was a convenient way to view data. Consider for instance a deck of punch cards as a dataset of as many lines as there are cards or a line printer as a dataset with pages and lines, or disk files organised as records of fixed or variable width. This way of viewing files has changed with Unix/Linux: here an important concept is a generalised mechanism for accessing a wide range of resources that are called *files*: documents, directories, devices, processes, network communications et cetera. This fundamental concept actually is twofold:

1. Every file is a stream of bytes.
2. The file system is a universal name-space.

Since a68g is developed for Unix/Linux we will not talk about books with pages and lines which is a deliberate deviation of the Revised Report. In a68g, we have files that are

sequences of bytes<sup>1</sup>. An object of mode `FILE` is a file descriptor as in many other contemporary programming languages. As in Algol 68, `a68g` transput is event-driven. You can at forehand specify actions to be taken when certain events occur during transput. This relieves you of contineously checking all kind of conditions when reading or writing (such as: did you encounter end of file, can you actually write to this file, did a conversion error take place, et cetera). We shall be examining later the kinds of event that can occur during transput.

## 7.2 Channels and files

Files have various properties. They usually have an identification string, the simplest example of which is a file name. Some files are read-only, some files are write-only and others permit both reading and writing. Some files allow you to browse, that is, they allow you to start anywhere in the file and read (or write) from that point on. `a68g` keeps track of the status of a file by means of a complicated structure of mode `FILE` which is declared in the standard-prelude. There are four names of mode `REF FILE` declared in the standard prelude:

- 1) `stand in` is the standard input (normally the keyboard),
- 2) `stand out` is the standard output (normally a screen),
- 3) `stand error` is the standard error channel (normally a screen), and
- 4) `stand back` is a file that can both be read and written.

You cannot read from `stand out` or `stand error`, nor write to `stand in`. `a68g` opens above files at the start of the **program**. You normally have no need to close one of this standard files. A file is associated with a channel. In the early days of Algol 68 channels could be considered as a description of a device-driver, but nowadays such details are abstracted in the kernel. They are maintained in `a68g` since they specify access rights to a file and specify whether a file is used for drawing. The mode of a channel is `CHANNEL` which is declared in the standard-prelude. In `a68g`, five principal channels are provided in the standard-prelude:

- 1) `stand in channel`,
- 2) `stand out channel`,
- 3) `stand error channel`,
- 4) `stand back channel` and

---

<sup>1</sup>A file is not necessarily finite, think for example of `/dev/null`.

5) stand draw channel.

The first is used for read-only files, the second and third are used for write-only files, and the fourth for files which permit both reading and writing. The fifth does not allow for reading and writing, but specifies that a file is used as a plotting device; it comes close to the device-driver specification mentioned earlier. The first four channels mentioned above are buffered.

## 7.3 Transput and scope

Before we treat at length the transput capabilities of Algol 68, you should note that it is rather easy to provoke a scope error in transput routines. As we have seen in section 2.16, the scope of a local name is the smallest enclosing clause which contains its **generator** (which may be hidden by an abbreviated **variable-declaration**) and the scope of a global name extends to the whole **program**. Some transput routines take a name, routine or format (vide infra) as argument, storing them in objects of mode `FILE` or `CHANNEL`. You must take care that these values still exist when you address such stored value in a later part of the **program**. For example, when you run next **program**:

```
FILE z;
STRING name = "test.data";

IF file is regular (name)
THEN VOID (open (z, name, standin channel))
ELSE associate (z, LOC STRING := "1")
FI;

INT k;
get (z, k);
print (k)
```

you will get next diagnostic in case file `test.data` does not exist:

```
6      ELSE associate (z, LOC STRING := "1")
      1
a68g: runtime error: 1: REF STRING value is exported out of its scope
no such file or directory) (detected in VOID conditional-clause starting
at "IF" in line 4).
```

The reason for this error is that `a68g` detects that `LOC STRING` may cease to exist before `z` ceases to exist. You can solve this for instance by writing `HEAP STRING`.

## 7.4 Reading files

Before you can read the contents of an existing file, you need to open the file. The **declaration** of the procedure `open` starts with:

```
PROC open = (REF FILE f, STRING idf, CHANNEL chan) INT
```

On Linux, the routine yields zero if the file is a regular file and non-zero otherwise. Example:

```
FILE inf;
STRING name = "results";

IF open (inf, name, stand in channel) /= 0
THEN print (("Cannot open ", results, new line))
FI
```

After a file has been opened, data can be read using the procedure `get`, the **declaration** of which starts with:

```
PROC get = (REF FILE f,
  [] UNION (INTYPE, PROC (REF FILE) VOID) items) VOID
```

The mode `INTYPE` is a united mode defined by the Revised Report {25<sub>10</sub>.3.2.2} and encompasses all modes that can be read. `INTYPE` is processed by a process called straightening that converts any structure or row into a row of values of the constituent fields or elements. Allowed modes for latter elements in `a68g` are specified by the mode `SIMPLIN`:

```
MODE SIMPLIN = UNION (
  REF INT, REF REAL, REF COMPLEX,
  REF LONG INT, REF LONG REAL, REF LONG COMPLEX,
  REF LONG LONG INT, REF LONG LONG REAL, REF LONG LONG COMPLEX,
  REF BOOL,
  REF BITS, REF LONG BITS, REF LONG LONG BITS,
  REF CHAR, REF [] CHAR
)
```

This means that `get` can read directly any structure or row (or even rows of structures or rows) as long as its elements are in `SIMPLIN`. The `PROC (REF FILE) VOID` mode in `[] UNION (INTYPE, PROC (REF FILE) VOID)` lets you use routines like `new page` and `new line` as **parameters**. The **declaration** of `new line` starts with:

```
PROC new line = (REF FILE f) VOID
```

and you can call it from `get` if you want. On input, the rest of the current line is skipped and a new line is taken. The position in the file is at the start of the new line, just before the



first character of that line. Consider next **program** fragment which opens a file and then reads the first line and makes a name of mode REF STRING to refer to it. After reading the string, new line is called explicitly:

```
FILE inf;
open (inf, "file", stand in channel);
STRING line;
get (inf, line);
new line (inf)
```

This could equally well have been written:

```
FILE inf;
open (inf, "file", stand in channel);
STRING line;
get (inf, (line, new line))
```

Of course you can declare your own procedures with mode PROC (REF FILE) VOID. The procedure read which you have already encountered in this book is declared as:

```
PROC read = ([] UNION (INTYPE, PROC (REF FILE) VOID) items) VOID
  get (stand in, items)
```

in the standard-prelude. As you can see, it gets data from stand in.

a68g offers two routines familiar from C and Pascal that determine whether we are at the end of a line, or at the end of the file, while reading. Note that these routines are complementary to the event routines discussed in 7.7.

```
PROC end of line = (REF FILE file) BOOL
```

This routine yields TRUE if the file pointer of file is at the end of a line. You can advance the file pointer with get (file, new line) or new line (file).

```
PROC end of file = (REF FILE file) BOOL
```

This routine yields TRUE if the file pointer of file is at the end of the file. When you have finished reading data from a file, you should close the file by calling the procedure close. Its **declaration** starts with:

```
PROC close = (REF FILE f) VOID
```

## 7.5 Writing to files

## INFORMAL INTRODUCTION TO ALGOL 68

You can use the `establish` procedure to create a new file. The **declaration** of `establish` starts with:

```
PROC establish = (REF FILE f, STRING idf, CHANNEL chann) INT
```

This **program** fragment creates a new file called `results`:

```
FILE outf;  
  
IF establish (outf, "results", stand out channel) /= 0  
THEN print (("Cannot establish file", new line));  
    stop  
FI
```

The procedure `establish` can fail if the disk you are writing to is full or you do not have write access (in a network, for example) in which case it will return a non-zero value. The procedure used to write data to a file is `put`. Its **declaration** starts with:

```
PROC put =  
    (REF FILE f, [] UNION (OUTTYPE, PROC (REF FILE) VOID) items) VOID
```

The mode `OUTTYPE` is a united mode defined by the Revised Report {25<sub>10</sub>.3.2.2} and encompasses all modes that can be printed. Like `INTYPE`, `OUTTYPE` is processed by straightening that converts any structure or row into a row of values of the constituent fields or elements. Allowed modes for latter elements in `a68g` are specified by the mode `SIMPLOUT`:

```
MODE SIMPLOUT = UNION (  
    INT, REAL, COMPLEX,  
    LONG INT, LONG REAL, LONG COMPLEX,  
    LONG LONG INT, LONG LONG REAL, LONG LONG COMPLEX,  
    BOOL,  
    BITS, LONG BITS, LONG LONG BITS,  
    CHAR, [] CHAR  
)
```

Again, a `PROC (REF FILE) VOID` routine as `new line` and `new page` can be used as argument to `put` as in the following example:

```
FILE outf;  
IF establish (outf, "newfile", stand out channel=) /= 0  
THEN put (stand error, ("Cannot establish newfile"));  
    stop  
ELSE put (outf, ("Data for newfile", new line));  
    close (outf)  
FI
```

On output, the new line character is written to the file. `new page` behaves just like `new`

line except that a form feed character is searched for on input, and written on output. When you have completed writing data to a file, you must close it with the procedure `close`. This is particularly important when writing files because the channel is buffered as explained above. Using `close` ensures that any remaining data in the buffer is flushed to the file. The procedure `print` directs output to `stand out`:

```
print ("Your name", new line)
```

which is equivalent to:

```
put (stand out, ("Your name", new line))
```

The procedure `write` is synonymous with `print`.

## 7.6 String terminators

A useful feature available for reading data is being able to specify when the reading of a string should terminate. Usually, this is set as the end of the line only. However, using the procedure `make term`, the string terminator can be a single character or any one of a set of characters. The **declaration** of `make term` starts with:

```
PROC make term = (REF FILE f, STRING term) VOID
```

so if you want to read a line word by word, defining a word as any sequence of non-space characters, you can make the string terminator a space by writing:

```
make term (inf, blank)
```

because `blank` (synonymous with " ") is rowed in the strong context of a **parameter** to `[] CHAR`. This will not remove the end-of-line as a terminator because the character `lf` is always added whatever characters you specify. You should remember that when a string is read, the string terminator is available for the next read — it has *not* been read by the previous read. Note that in `a68g` it is possible to inquire about the terminator string of a file by means of the routine:

```
PROC term = (REF FILE f) STRING
```

## 7.7 Events

The Revised Report implementation of Algol 68 transput is characterised by the use of events. Events are for instance end of file, or a conversion error. For each event, a file has a routine that is activated when its associated event occurs. `a68g` detects these events:

## INFORMAL INTRODUCTION TO ALGOL 68

1. Reaching end of file.
2. Reaching end of line.
3. Reaching end of page.
4. Reaching end of format.
5. A file open error.
6. A value error.
7. A format error.
8. A transput error not caught by other events.

The default action when an event occurs depends on the event. For instance, end-of-line will per default provoke that `newline` is called after which transput continues, while end-of-file will provoke a runtime error. The default action can be replaced by a programmer-defined action.

### 7.7.1 File end

When the end of a file is detected, the default action is to generate a runtime error. A programmer-supplied action must be a procedure with mode `PROC (REF FILE) BOOL`. The yield of this procedure should be `TRUE` if some action has been taken to remedy the end-of-file condition, in which case transput is resumed, or `FALSE` otherwise in which case the default action will be taken.

The **declaration** of the procedure on logical file end starts with:

```
PROC on logical file end =  
  (REF FILE f, PROC (REF FILE) BOOL p) VOID
```

Algol 68 discerns {25<sub>10</sub>.3.1.3} a *logical file end* and a *physical file end*. In a68g this difference is not implemented and `on file end`, `on logical file end` and `on physical file end` are the same procedures. Next **program** will display the contents of its text input file and print a message at its end:

```
IF FILE inf; [] CHAR infn = "file";  
  open (inf, infn, stand in channel) /= 0  
THEN put (stand error, ("Cannot open ", infn, new line));  
  stop  
ELSE on logical file end  
  (inf, (REF FILE f) BOOL:  
    (write ("End of ", idf (f), new line));
```

```

    close (f);
    stop
  )
);
STRING line;
DO get (inf, (line, new line));
    write ((line, new line))
OD
FI

```

The anonymous procedure provided as the second **parameter** to `on logical file end` prints an informative message and closes the file. Note also that the `DO` loop simply repeats the reading of a line until the logical file end procedure is called by `get`. The procedure `idf` is described in section 7.14. You should be careful if you do `transput` on the **parameter** `REF FILE f` since you could get endless recursion in case such `transput` provokes the same event that the routine tried to handle. Also, because the `on logical file end` procedure stores its procedure **parameter** in its `REF FILE` **parameter**, you should be cautious when using `on logical file end` in limited ranges since a scope error may result. Any **unit** yielding an object of mode `PROC (REF FILE) BOOL` in a strong context is suitable as the second **parameter** of `on logical file end`. If you want to reset the action to the default action, use the **call** :

```
on logical file end (f, (REF FILE) BOOL: FALSE)
```

## 7.7.2 Line end and page end

These events are caused when while reading, the end of line or end of page is encountered. These conditions are events so you can provide a routine that for instance automatically counts the number of lines or pages read. The procedures `on line end` and `on page end` let you provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE`, `transput` continues, otherwise a new line in case of end of line, or new page in case of end of page, is executed and then `transput` resumes. Be careful when reading strings, since end of line and end of page characters are string terminators. If you provide a routine that mends the line - or page end, be sure to call `new line` or `new page` before returning `TRUE`. In case of a default action, `new line` or `new page` must be called explicitly, for instance in the `read` procedure, otherwise you will read nothing but empty strings as you do not eliminate the terminator.

## 7.7.3 Format end

This event is caused when in formatted `transput`, the format gets exhausted. The procedure `on format end` lets you provide a procedure of mode `PROC (REF FILE) BOOL`. If

the programmer-supplied routine yields `TRUE`, `transput` continues, otherwise the format that just ended is restarted and then `transput` resumes.

### 7.7.4 Open error

This event is caused when a file cannot be opened as required. For instance, you want to read a file that does not exist, or write to a read-only file. The procedure `on open error` lets you provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE`, the **program** continues, otherwise a runtime error occurs.

### 7.7.5 Value error

This event is caused when transputting a value that is not a valid representation of the mode of the object being transput. The procedure `on value error` lets the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If you do `transput` on the file within the procedure ensure that a value error will not occur again. If the programmer-supplied routine yields `TRUE`, `transput` continues, otherwise a runtime error occurs.

### 7.7.6 Format error

This event is caused when an error occurs in a format, typically when patterns are provided without objects to transput. The procedure `on format error` lets the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE`, the **program** continues, otherwise a runtime error occurs.

### 7.7.7 Transput error

This event is caused when an error occurs in transput that is not covered by the other events, typically conversion errors (value out of range et cetera). The procedure `on transput error` lets the programmer provide a procedure whose mode must be `PROC (REF FILE) BOOL`. If the programmer-supplied routine yields `TRUE`, the **program** continues, otherwise a runtime error occurs.

## 7.8 Formatting routines

One of the problems of using the rather primitive facilities given so far for the output of real and integer numbers is that although they allow numbers to be printed in columns, the widths and precisions are fixed. It is necessary to have some means of controlling width and precision of printed values. The procedures `whole`, `fixed`, `float` and `real` provide these means. A specification of the routines in this section, as Algol 68 source code, is in 11.15.

The **declaration** of the procedure `whole` starts with:

```
PROC whole = (NUMBER v, INT width) STRING
```

and takes two **parameters**. The first is a real or integer value and the second is an integer which tells `whole` the width of the output number. If you pass a real number to `whole`, it calls the procedure `fixed` with **parameters** `width` and 0.

If `width = 0`, then the number is printed with the minimum possible width. A positive value for `width` will give numbers preceded by a "+" if positive and a "-" if negative. A negative value for `width` will provide a minus-sign for negative numbers but no plus-sign for positive numbers and the width will be `ABS width`. Note that where the integer is wider than the available space, the output field is filled with the character denoted by `error char` (which is declared in the standard-prelude as the asterisk (\*)).

The **declaration** of the procedure `fixed` starts with:

```
PROC fixed = (NUMBER v, INT width, after) STRING
```

and takes three **parameters**. The first two are the same as those for `whole` and the third specifies the number of decimal places required. The rules for `width` are the same as the rules for `width` for `whole`.

When you want to print numbers in scientific format, you should use `float` which takes four **parameters**. Its **declaration** starts with:

```
PROC float = (NUMBER v, INT width, after, exp) STRING
```

The first three are the same as the **parameters** for `fixed`, while the fourth is the width of the exponent field. Thus the **call** :

```
print (float (pi * 10, 8, 2, -2)
```

produces the output `+3.14e 1`. The **parameter** `exp` obeys the same rules, applied to the exponent, as `width`.

a68g implements a further routine `real` for formatting reals in a way closely related to `float`, declared as:

```
PROC real =  
(NUMBER x, INT width, after, exp width, modifier) STRING
```

If `modifier` is a positive number, the resulting string will present `x` with its exponent a multiple of `modifier`. If `modifier = 1`, the returned string is identical to that returned by `float`. A common choice for `modifier` is 3 which returns the so-called engineers notation of `x`. If `modifier` is zero or negative, the resulting string will present `x` with `ABS modifier` digits before the **point-symbol** and its exponent adjusted accordingly; compare this to Fortran `nP` syntax.

## 7.9 Straightening

The term straightening is the process whereby a compounded mode is separated into its constituent modes, which are themselves straightened if required. For example, the mode:

```
STRUCT (INT a, CHAR b, UNION (REAL, STRING) u)
```

would be straightened into values of the following modes:

1. INT
2. CHAR
3. UNION (REAL, STRING)

The mode:

```
REF STRUCT (INT a, CHAR b, UNION (REAL, STRING) u)
```

would be straightened into a number of names having the modes:

1. REF INT
2. REF CHAR
3. REF UNION (REAL, STRING)

However, a value of mode `COMPLEX` is not straightened into two values both of mode `REAL`. Also, any row is separated into its constituent elements. For example:

```
[1 : 3] COMPLEX
```

will be straightened into three values of mode `COMPLEX`.

## 7.10 Default-format transput

In default-format transput, each primitive mode is written as follows:



1. CHAR  
Output a character to the current position in the file.
2. [ ] CHAR  
Output all the constituent characters to the file.
3. BOOL  
Output `flip` for TRUE or `flop` for FALSE to the file.
4. L BITS  
Output `flip` for each set bit and `flop` for each zero bit.
5. L INT  
Output the integer using the **call** :  

```
whole (i, L int width + 1)
```

which will output the integer value in:  

```
L int width + 1
```
6. L REAL  
Output the real using the **call** :  

```
float (r, L real width + L exp width + 4,  
      L real width - 1, L exp width + 1)
```

which will output the real value in:  

```
L real width + L exp width + 4
```

positions preceded by a **sign**.
7. L COMPLEX  
The complex value is output as two real numbers.
8. PROC (REF FILE) VOID  
An `lf` character is output if the routine is `newline` and an `ff` character if the routine is `new page`. User-defined routines with this mode can be used.

In default-format transput, each primitive mode is read as follows:

1. REF CHAR  
Characters `c`, `c < blank` are skipped and the first character `c ≥ blank` is assigned to the name. If a REF [ ] CHAR is given, then the above action occurs for each of the required characters of the row.
2. REF STRING  
All characters, including any control characters, are assigned to the name until any character in the character set specified by the `string term` field of `f` is encountered (but this is not read).

## INFORMAL INTRODUCTION TO ALGOL 68

### 3. REF BOOL

The next non-white-space character is read and, if it is `flip`, `TRUE` is assigned, or if it is `flop`, `FALSE` is assigned.

### 4. REF L BITS

The action for `REF BOOL` is repeated for each bit in the name.

### 5. REF L INT

White space is skipped until a **sign** or a digit is encountered. Reading of decimal digits continues until a character which is not a decimal digit is encountered.

### 6. REF L REAL

A real number consists of 3 parts:

- (a) an optional **sign** possibly followed by spaces
- (b) an optional integral part
- (c) a "." followed by a fractional part
- (d) an optional exponent preceded by a character in the set "Ee". The exponent may have a **sign**. Absence of a **sign** is taken to mean a positive exponent

### 7. REF L COMPLEX

Two real numbers are read from the file. The first number is regarded as the real part and the second the imaginary part.

## 7.11 Formatted transput

Formatted transput gives the programmer a high degree of control over the transput of values. If you program in Fortran or in C then the concept of formatted transput is not strange to you. Algol 68 implements a similar mechanism but in its own orthogonal way.

A format is a description of how values should be transput. A format in Algol 68 is a value of mode `FORMAT`. The standard-prelude defines the mode `FORMAT` as a structure the fields of which are inaccessible to you. You can have format identities, format variables, procedures yielding formats et cetera. There are however no pre-defined operators for objects of mode `FORMAT`. You can define them yourself but since the fields are inaccessible to you there is little sense in passing a format as **operand**. In a68g, formats behave like anonymous routines and follow the scope rules of routine-texts. Formats are associated with files by including them in the **parameter-list** of routines `putf` and `getf`. There are (of course) related routines `printf` and `readf` that perform `putf` on `standout`, or `getf` on `standin`, respectively. These routines have respective **routine-specifications**:

```
PROC putf = (REF FILE f, [] UNION (OUTTYPE,  
    PROC (REF FILE) VOID, FORMAT) items) VOID
```

```

PROC getf = (REF FILE f, [] UNION (INTYPE,
  PROC (REF FILE) VOID, FORMAT) items) VOID

PROC printf =
  ([] UNION (OUTTYPE, PROC (REF FILE) VOID, FORMAT) items) VOID

PROC readf =
  ([] UNION (INTYPE, PROC (REF FILE) VOID, FORMAT) items) VOID

```

### 7.11.1 Format-texts

In this section we will introduce formatted transput by describing the constituting constructs. In Fortran, a format is defined by a format statement like:

```

WRITE (6, 10) N, RESULT
10 FORMAT (/1X, 'RESULT', 1X, I3, '=', 1X, F9.5)

```

In Algol 68, a format has a pseudo **denotation** that is called a **format-text**, for instance:

```
$1, 1x, "Result", 1x, 2zd, "=", 1x, g (9.5)$
```

and you could write

```

FORMAT my format = $1, 1x, "Result", 1x, 2zd, "=", 1x, g (9.5)$;
FORMAT variable format := my format

```

but typically a **format-text** is used as a **parameter** to transput routines:

```
printf (($1, 1x, "Result", 1x, 2zd, "=", 1x, g (9.5)$, n, result))
```

A **format-text** has these general syntax elements:

- **format-text:**  
**formatter-symbol, picture-list-formatter-symbol.**
- **MARKER-frame:**  
**insertion-option, replicator-option, letter-s option, MARKER;**

A **format-text** is a list of pictures; each picture representing an individual item to be transput:

- **picture:**  
**insertion;**  
**pattern;**  
**collection;**  
**replicator-collection.**

## INFORMAL INTRODUCTION TO ALGOL 68

Before describing other elements, a **replicator** is a static or dynamic specification of how often a subsequent construct should be repeated:

- **replicator:**  
**integer-denotation;**  
**letter-n-symbol, meek-integer-enclosed-clause.**

Examples of replicators are:

- `6 # Repeat six times #`
- `n (k + 1) # Repeat k + 1 times #`
- `n (read int) # Repeat as many times as you type #`

A **replicator** should not be less than zero. The Revised Report {25<sub>10</sub>.3.4.1.dd} states that a negative **replicator** should be treated as if it were zero, but a68g assumes that a negative **replicator** means that something went wrong and will give a runtime error. A collection is simply a way to group a list of pictures, typically used when a **replicator** must be applied to it:

- **collection:**  
**open-symbol, picture-list, close-symbol.**

Insertions give you control over new lines, new pages, spaces forward, or lets you insert string literals:

- **insertion: insertion-item-list.**
- **insertion-item:**  
**replicator-option, letter-k-symbol;**  
performs as many spaces forward as its **replicator** indicates.  
**replicator-option, letter-l-symbol;**  
performs a new line; it calls `new line .`  
**replicator-option, letter-p-symbol;**  
performs a new page, it calls `new page .`  
**replicator-option, letter-x-symbol;**  
performs a space; it calls `space .`  
**replicator-option, letter-q-symbol;**  
performs a space; it calls `space .`  
**replicator-option, row-of-character-denotation.**  
Specifies a sequence of strings to be taken literally.

Examples:

- 20k
- 31"Answer"
- $n (k + 1)x$
- "Algol 68"

Using insertions, we can for instance draw a sine-curve to standard output:

```
INT n = 24;
FOR i TO n
DO REAL x = 2 * pi * (i - 0.5) / n;
  printf ($1, n (40 - ROUND (30 * sin (x)))k, "*" $)
OD
```

Patterns specify how to:

1. transput a value of a standard mode.
2. direct transput depending on a boolean - or integer expression.
3. execute an embedded format.

a68g implements these patterns:

- **pattern:**
  - general-pattern;**
  - integer-pattern;**
  - real-pattern;**
  - complex-pattern;**
  - bits-pattern;**
  - string-pattern;**
  - boolean-pattern;**
  - choice-pattern;**
  - c-style-pattern;**
  - format-pattern.**

These are standard Algol 68 patterns, except for the **c-style-pattern**, which is an a68g extension.

## 7.11.2 C-style-patterns

## INFORMAL INTRODUCTION TO ALGOL 68

If you are a C or C++ programmer you will be familiar with C format string placeholders. a68g offers a notation closely resembling C format string placeholders called **c-style-patterns**. These are an extension to standard Algol 68 **patterns**. Applicable production-rules are:

- **c-style-pattern:**
  - percent-symbol,**
  - minus-symbol-option,**
  - plus-symbol-option,**
  - width-option,**
  - precision-option,**
  - c-type.**
- **width: replicator.**
- **precision: point-symbol, replicator.**
- **c-type:**
  - letter-b-symbol,**
  - letter-c-symbol,**
  - letter-d-symbol,**
  - letter-e-symbol,**
  - letter-f-symbol,**
  - letter-g-symbol,**
  - letter-i-symbol,**
  - letter-o-symbol,**
  - letter-s-symbol,**
  - letter-x-symbol.**

Example **c-style-patterns** are:

- `%-80s`
- `%+23.15f`
- `%d`
- `%n(bits width)b`

The **minus-symbol** specifies that the resulting string representation of the transput value will be left-justified instead of right-justified. The **plus-symbol** specifies that a sign must be printed for positive integer or real values. **Width** specifies the width of the string representation of the transput value. If you omit **width** then a default value is chosen. **Precision** applies to real values; if you omit this field then a default value is chosen. Note that fields which are not applicable to the mode of the value being transput are simply ignored; no diagnostic will be issued. The letter **c-type** determines the mode of the transput value:

- b binary transput of a value of mode L BITS.
- c transput of a value of mode CHAR.
- d transput of a value of mode L INT.
- e transput of a value of mode L REAL in scientific format.
- f transput of a value of mode L REAL in fixed format.
- g transput of a value of mode L REAL in e or f format, whichever is more appropriate.
- i transput of a value of mode L INT.
- o octal transput of a value of mode L BITS.
- s transput of a value of mode [] CHAR or STRING.
- x hexadecimal transput of a value of mode L BITS.

On reading a STRING or [] CHAR value, with omitted **width** field, string terminators as set by `make term` are obeyed. If you specify **width**, then exactly that number of characters is read, irrespective of terminators set by `make term`.

### 7.11.3 General-patterns

The **general-pattern** performs transput following default formatting:

- **general-pattern:**  
**letter-g-symbol, strong-row-of-integer-enclosed-clause;**  
**letter-h-symbol, strong-row-of-integer-enclosed-clause;**

Any standard mode can be transput with "g" or "h" when the **strong-row-of-integer-enclosed-clause** is absent; default formatting is used which is the formatting that default-format routines `read`, `get`, `out` or `print` would apply. In case of an integer or real value to transput, the length of the strong-row-of-integer determines the formatting to be applied. In case of letter "g" we get:

- `element (i).whole (... , i)` is called.
- `elements (i, j).fixed (... , i, j)` is called.
- `elements (i, j, k).float (... , i, j, k)` is called.

In case of letter "h" we always get scientific notation. Let `K` be `1 + the value exp with` for the length of the integer or real value to transput, then we have this behaviour for "h":

- `element (i). real (... , i, i + K + 4, K, 3)` is called. Since this gives  $i$  decimals before the exponent is adjusted to be a multiple of 3, we get  $i + 1$  digits in engineers notation and the width is automatically adjusted.
- `elements (i, j). real (... , i, i + K + 4, K, j)` is called. If you want to print a number with exactly  $i$  decimals, you specify `h (i, 1)` and the width is automatically adjusted.
- `elements (i, j, k). real (... , i, j, K, k)` is called. This prints as much digits as will fit in a width  $i$ , including  $j$  decimals, and the exponent is adjusted to be a multiple of  $k$ .
- `elements (i, j, k, l). real (... , i, j, k, l)` is called. This gives you full control of the routine `real` as no defaults are applied.

### 7.11.4 Integer-patterns

An **integer-pattern** transputs integer values and consists of an optional **sign-mould** that lets you decide how to transput a **sign**, followed by an integer-mould that specifies how to transput each individual digit. We will meet elements here that are called frames: "s", "z" and "d". Frame "s" means that the next frame will be suppressed, "z" means to print a non-zero digit or a space otherwise, and "d" means to print a digit. Hence the **sign-mould** determines how to put a **sign**, would you want one.

- **integer-pattern: sign-mould-option, integer-mould.**
- **sign-mould: integer-mould-option, sign.**
- **integer-mould: digit-marker-sequence, insertion-option.**
- **digit-marker:**  
     **letter-z-frame;**  
     **letter-d-frame.**

Frames + and - are sign-frames; + forces a **sign** to be put, but a - only puts a minus if the number is negative. Example **sign-moulds** are:

- +
- -
- zzz-
- 3z-



If you specify **letter-z-frames** before the **sign** "+" or "-" then the **sign** will shift left one digit for every non-zero digit that is begin put. In a68g, when a **sign** is shifted in a **sign-mould**, any character output by literal insertions in that **sign-mould** is replaced with a space as well, starting from the first **letter-z-frame** until the **sign** is put. The insertions let you put all kind of literals in between digits. For example:

- 4d # transputs 9 as "0009", error on negative #
- zzz-d # transputs -9 as " -9" #
- 3z-d # transputs -9 as " -9" #
- 3z", "2z-d # transputs 100000 as " 100,000" #
- 3d"- "3d"- "4d # transputs 5551234567 as 555-123-4567 #

### 7.11.5 Real-patterns

A **real-pattern** follows the same logic as an **integer-pattern**. **Sign-moulds** are optional, and the exponent-part is optional. The **decimal-point-frame** and the **exponent-frame** "e" are suppressible by preceding them with "s"; this can for instance be used to replace the standard exponent character by an insertion of your liking.

- **real-pattern:**  
**sign-mould-option, integer-mould-option, letter-s-symbol-option, point-symbol, insertion-option, integer-mould, exponent-frame-option;**  
**sign-mould-option, integer-mould, letter-s-symbol-option, point-symbol, insertion-option, integer-mould-option, exponent-frame-option;**  
**sign-mould-option, integer-mould, exponent-frame;**
- **exponent-frame:**  
**letter-s-symbol-option, letter-e-symbol, insertion-option, sign-mould-option, integer-mould.**

Examples:

- d.3d # transputs pi as "3.142" #
- d.3dez-d # transputs 0.3333 as "3.333e -1" #
- ds.", "3dse"^^"z-d # transputs 0.3333 as "3,333^-1" #

### 7.11.6 Complex-patterns

A **complex-pattern** consists of a **real-pattern** for the real part and a **real-pattern** for the imaginary part of the value. The **plus-i-times-frame** "i" is suppressible and can thus be replaced by an insertion.

- **complex-pattern:**  
**real-pattern, letter-s-symbol-option, letter-i-symbol, insertion-option, real-pattern.**

Examples:

- `-d.3di-d.3d # transputs f.i. 0.000i-1.000 #`
- `-d.3dsi-d.3d, "j" # transputs f.i. 0.000-1.000j #`

### 7.11.7 Bits patterns

A **bits-pattern** has no **sign-mould**, since a BITS value is unsigned. In a68g the radix is specified by a **replicator** by which it can be dynamic.

- **bits-pattern:**  
**replicator, letter-r-symbol, integer-mould.**

Examples:

- `2r7zd # binary 8-bit byte with trailing zero-bit suppression #`
- `16r8d # hexadecimal 32 bit word #`

### 7.11.8 String-patterns

The **string-pattern** transputs an object of modes STRING, CHAR and [] CHAR. The **letter-a-frames** can be suppressed.

- **string-pattern: letter-a-frame-sequence, insertion-option.**

Examples:

- `printf ($7a$, "Algol68") # prints "Algol68" #`

- `printf ($5a-"2a$, "Algol68") # prints "Algol-68" #`
- `printf ($5a2sa$, "Algol68") # prints "Algol" #`

### 7.11.9 Boolean-patterns

The boolean-expression transputs a `BOOL` value.

- **boolean-pattern:**  
**letter-b-symbol;**  
**letter-b-symbol, open-symbol, row-of-character-denotation,**  
**comma-symbol, row-of-character-denotation, close-symbol.**

If you just want the standard "T" or "F" transput, do not specify the **row-of-character-denotations**. If you do specify those, a **boolean-pattern** is the **format-text** equivalent of a **conditional-clause**. Example:

- `printf ($b ("true", "not true")$, read bool)`

### 7.11.10 Choice-patterns

A **choice-pattern** transputs a row-of-character-denotation depending on an integer value. It is the **format-text** equivalent of an integer-case-clause.

- **choice-pattern:**  
**letter-c-symbol, open-symbol, row-of-character-denotation-list, close-symbol.**

Examples:

- `printf ($c ("one", "two", "three")$, read int)`

The Revised Report specification of getting using an **integral-choice-pattern** has the peculiarity that when two literals start with the same sequence of characters, the longer literal should appear first in the list. `a68g` makes no such demand, and will select the correct literal from the list whatever their order.

### 7.11.11 Format-patterns

The **format-pattern** lets you dynamically choose the format you want to employ.

- **format-pattern:**  
**letter-f-symbol, meek-format-enclosed-clause.**

This is an example where a format may be selected dynamically:

- `f (um | (INT): $g (0)$, (REAL): $g (0, 4)$)`

The effect of a **format-pattern** is that it temporarily supersedes the active format for the file at hand. When this temporary format ends, the original format continues without invoking an format-end event.

## 7.12 Binary files

Files that are not meant to be read by you personally can be in a compact binary form. These files are called binary files or unformatted files. Many large files will be stored in this form. Algol 68 allows you to read and write the same values to binary files as are allowed for text files. The difference between transport on binary files is that instead of using the procedures `put` and `get`, you use the procedures `put bin` and `get bin`. The modes accepted by these procedures are identical with those accepted by `put` and `get` respectively. Values of mode `[] CHAR` or `REF STRING` can be written by `put bin`, and read by `get bin`, but note that the string terminator set using the procedure `make term` is ignored — on binary output, `a68g` first writes the number of characters that will be printed and consequently on binary input, `a68g` first reads the number of characters that will be read. Note that the procedure `read bin` is equivalent to `get bin (stand back, ...)` and the procedure `write bin` is equivalent to `put bin (standback, ...)`.

## 7.13 Using a STRING as a file

Sometimes you want to write a large amount of data to a file, and later read it again in the same **program**. In such case, instead of locating the file on a device such as a hard disk, a file can be associated with an object of mode `STRING` that will hold the data; hence file contents reside in memory. To this end one uses the routine `associate` which is declared as:

```
PROC associate = (REF FILE, REF STRING) VOID
```

On putting, the string is dynamically lengthened and output is added at the end of the string. Attempted getting outside the string provokes an end of file condition. When a file that is associated with a string is reset, getting restarts from the start of the associated string. Next (trivial) code fragment illustrates the use of `associate` in transport:

```
[n] COMPL u, v;
# code to give values to elements in u #
FILE in, out,
STRING z;
associate (in, z);
associate (out, z);
putf (out, u);
getf (in, v);
```

## 7.14 Other transput procedures

The **declaration** of the procedure `idf` starts with:

```
PROC idf = (REF FILE f) STRING
```

and yields the identification of the file handled by the file `f`. There are two other ways of closing a file. One is `scratch` and the other is `lock`:

```
PROC scratch = (REF FILE f) VOID
PROC lock = (REF FILE f) VOID
```

The procedure `scratch` deletes the file once it is closed. It is often used with temporary files. The procedure `lock` closes its file and then locks it so that it cannot be opened without some system action. It is possible to reset, or rewind, a file, to let the next transput operation start from the beginning of a file. The procedure provided for this purpose is `reset`. Its **declaration** starts with:

```
PROC reset = (REF FILE f) VOID
```

One possible use of this procedure is to output data to a file, then use `reset` followed by `get` to read the data from the file. `a68g` allows the alternative spelling `rewind`. Whether a file supports `reset` can be interrogated by using:

```
PROC reset possible = (REF FILE f) BOOL
```

This routine yields `TRUE` when `f` can be reset, or `FALSE` otherwise. `a68g` allows the alternative spelling `rewind possible`. In `a68g` there is also a procedure `set` that attempts to set the file pointer to a position other than the beginning of the file as done by `reset`:

```
PROC set = (REF FILE f, INT n) INT
```

This routine deviates from the standard Algol 68 definition. In `a68g`, `set` attempts to move the file pointer of `f` by `n` character positions with respect to the current position. If the file pointer would as a result of this move get outside the file, it is not changed and the routine `set` by `on file end` is called. If this routine returns `FALSE`, and end-of-file runtime error

## INFORMAL INTRODUCTION TO ALGOL 68

is produced. The routine returns an `INT` value representing system-dependent information on this repositioning. Whether a file allows setting, can be interrogated with `set possible`:

```
PROC set possible = (REF FILE f) BOOL
```

which does what its name suggests — yield `TRUE` when `f` can be set, or `FALSE` otherwise.

Related to `set` are the routines `space` and `backspace` :

```
PROC space = (REF FILE f) VOID
```

The procedure advances the file pointer in file `f` by one character. It does *not* read or write a blank.

```
PROC backspace = (REF FILE f) VOID
```

The procedure attempts to retract the file pointer in file `f` by one character. It actually executes

```
VOID (set (f, -1))
```

and thus is subject to the properties of the routine `set`.

# Chapter 8

## Algol 68 Genie context-free syntax

{There are writings which are lovable although ungrammatical,  
and there are other writings which are extremely grammatical,  
but are disgusting.  
This is something that I cannot explain to superficial persons.  
On Flowers and Women. Chang Ch'ao. }

### 8.1 Introduction

This chapter is a reference for Algol 68 Genie syntax. The syntax described here is context-free. The advantage of presenting a context-free syntax is that the backbone of constructions can be explained briefly. The disadvantage is that a context-free grammar cannot reject **programs** that are semantically incorrect, for instance those that apply undeclared symbols. The two-level Algol 68 syntax is described in the Revised Report in Part IV.

### 8.2 Reserved symbols

Next symbols are reserved in a68g:

## INFORMAL INTRODUCTION TO ALGOL 68

| Symbol                | Representation |
|-----------------------|----------------|
| <b>andf-symbol</b>    | ANDF           |
| <b>andth-symbol</b>   | ANDTH          |
| <b>assert-symbol</b>  | ASSERT         |
| <b>assign-symbol</b>  | ASSIGN         |
| <b>at-symbol</b>      | AT             |
| <b>begin-symbol</b>   | BEGIN          |
| <b>becomes-symbol</b> | :=             |
| <b>bits-symbol</b>    | BITS           |
| <b>bool-symbol</b>    | BOOL           |
| <b>bus-symbol</b>     | ]              |
| <b>by-symbol</b>      | BY             |
| <b>bytes-symbol</b>   | BYTES          |
| <b>case-symbol</b>    | CASE           |
| <b>channel-symbol</b> | CHANNEL        |
| <b>char-symbol</b>    | CHAR           |
| <b>class-symbol</b>   | CLASS          |
| <b>close-symbol</b>   | )              |
| <b>code-symbol</b>    | CODE           |
| <b>col-symbol</b>     | COL            |
| <b>comment-symbol</b> | COMMENT        |
| <b>complex-symbol</b> | COMPLEX        |
| <b>compl-symbol</b>   | COMPL          |
| <b>diag-symbol</b>    | DIAG           |
| <b>do-symbol</b>      | DO             |
| <b>downto-symbol</b>  | DOWNTO         |
| <b>edoc-symbol</b>    | EDOC           |
| <b>elif-symbol</b>    | ELIF           |
| <b>else-symbol</b>    | ELSE           |
| <b>elsf-symbol</b>    | ELSF           |
| <b>empty-symbol</b>   | EMPTY          |
| <b>end-symbol</b>     | END            |
| <b>environ-symbol</b> | ENVIRON        |
| <b>equals-symbol</b>  | =              |
| <b>esac-symbol</b>    | ESAC           |
| <b>exit-symbol</b>    | EXIT           |
| <b>false-symbol</b>   | FALSE          |
| <b>file-symbol</b>    | FILE           |
| <b>fi-symbol</b>      | FI             |
| <b>flex-symbol</b>    | FLEX           |
| <b>format-symbol</b>  | FORMAT         |



## ALGOL 68 GENIE CONTEXT-FREE SYNTAX

|                       |         |
|-----------------------|---------|
| <b>for-symbol</b>     | FOR     |
| <b>from-symbol</b>    | FROM    |
| <b>go-symbol</b>      | GO      |
| <b>goto-symbol</b>    | GOTO    |
| <b>heap-symbol</b>    | HEAP    |
| <b>if-symbol</b>      | IF      |
| <b>in-symbol</b>      | IN      |
| <b>int-symbol</b>     | INT     |
| <b>isnt-symbol</b>    | ISNT    |
| <b>is-symbol</b>      | IS      |
| <br>                  |         |
| <b>loc-symbol</b>     | LOC     |
| <b>long-symbol</b>    | LONG    |
| <b>mode-symbol</b>    | MODE    |
| <b>new-symbol</b>     | NEW     |
| <b>nil-symbol</b>     | NIL     |
| <b>od-symbol</b>      | OD      |
| <b>of-symbol</b>      | OF      |
| <b>op-symbol</b>      | OP      |
| <b>open</b>           | (       |
| <b>orel-symbol</b>    | OREL    |
| <br>                  |         |
| <b>orf-symbol</b>     | ORF     |
| <b>ouse-symbol</b>    | OUSE    |
| <b>out-symbol</b>     | OUT     |
| <b>par-symbol</b>     | PAR     |
| <b>pipe-symbol</b>    | PIPE    |
| <b>pragmat-symbol</b> | PRAGMAT |
| <b>prio-symbol</b>    | PRIO    |
| <b>proc-symbol</b>    | PROC    |
| <b>real-symbol</b>    | REAL    |
| <b>ref-symbol</b>     | REF     |
| <br>                  |         |
| <b>row-symbol</b>     | ROW     |
| <b>sema-symbol</b>    | SEMA    |
| <b>short-symbol</b>   | SHORT   |
| <b>skip-symbol</b>    | SKIP    |
| <b>sound-symbol</b>   | SOUND   |
| <b>string-symbol</b>  | STRING  |
| <b>struct-symbol</b>  | STRUCT  |
| <b>sub-symbol</b>     | [       |
| <b>thef-symbol</b>    | THEF    |
| <b>then-symbol</b>    | THEN    |

|                          |       |
|--------------------------|-------|
| <b>to-symbol</b>         | TO    |
| <b>trnsp-symbol</b>      | TRNSP |
| <b>true-symbol</b>       | TRUE  |
| <b>underscore-symbol</b> | —     |
| <b>union-symbol</b>      | UNION |
| <b>until-symbol</b>      | UNTIL |
| <b>void-symbol</b>       | VOID  |
| <b>while-symbol</b>      | WHILE |

### 8.3 Common production-rules

In Part I, a method from Van Wijngaarden is used to write production-rules. A production-rule consists of the following items, in order:

- a non-empty notion  $N$  ;
- a **colon-symbol** ;
- a non-empty sequence of alternatives for  $N$  separated by **semicolon-symbols**; within an alternative, a **comma-symbol** means "is followed by".
- a **point-symbol**.

Production-rules can be parametrised: an upper-case word is substituted throughout the production-rule. Common production-rules are:

- **EMPTY: .**  
This is an empty production.
- **NOTION-list: NOTION; NOTION, comma-symbol, NOTION-list.**
- **NOTION-list-proper: NOTION, comma-symbol, NOTION-list.**  
A list proper contains at least two NOTIONS.
- **NOTION-option: NOTION; EMPTY.**
- **NOTION-sequence: NOTION; NOTION, NOTION-sequence.**
- **NOTION-series: NOTION; NOTION, semicolon-symbol-symbol, NOTION-sequence.**
- **length:**  
**long-symbol-sequence;**  
**short-symbol-sequence.**
- **qualifier: heap-symbol; new-symbol; loc-symbol.**
- **sign: plus-symbol, minus-symbol.**

## 8.4 Tags

Tags are the symbols for **identifiers**, **mode-indicants** and **operators**.

### 8.4.1 Identifier tags

- **mode-indicant:**  
**bold-letter;**  
**mode-indicant, bold-letter;**  
**mode-indicant, underscore-symbol-sequence, mode-indicant.**
- **identifier:**  
**letter;**  
**identifier, letter;**  
**identifier, digit;**  
**identifier, underscore-symbol-sequence, identifier.**
- **label:**  
**identifier, colon-symbol.**

### 8.4.2 Operator tags

Note that white space cannot be written within an operator symbol.

- **operator:**  
**monadic-operator;**  
**dyadic-operator.**
- **monadic-operator:**  
**bold-operator;**  
**monad, nomad-option, becomesety.**
- **dyadic-operator:**  
**bold-operator;**  
**monad, nomad-option, becomes-option;**  
**nomad, nomad-option, becomes-option.**
- **bold-operator:**  
**bold-letter;**  
**bold-operator, bold-letter;**  
**bold-operator, underscore-symbol-sequence, bold-operator.**

- **becomes:**  
   **becomes-symbol-option;**  
   **assigns-to-symbol-option.**

## 8.5 Contexts

A context determines what coercions are allowed at a specific position in your program. Each context has an intrinsic strength. There are five contexts called strong, firm, meek, weak and soft. A strong context allows all coercions, whereas a soft context allows only one. The places in a **program** which have these contexts are:

1. Strong contexts In a strong context, the mode that is expected from a construct, is imposed.
  - (a) The actual **parameters** of **calls**
  - (b) The **enclosed-clauses** of **casts**
  - (c) The source **unit** of **assignments**
  - (d) The source **unit** of **identity-declarations**
  - (e) The source **unit** of **variable-declarations**
  - (f) The **units** of **routine-texts**
  - (g) VOID **units**
  - (h) Subordinated constituents of a balanced construct
2. Firm contexts
  - (a) **Operands** of **formulas**
3. Meek contexts A meek context allows coercions specifically needed to arrive at INT or BOOL values.
  - (a) **Enquiry-clauses**
  - (b) **Primaries** of **calls**
  - (c) The **units** following FROM, BY and TO or DOWNTO
  - (d) **Units** in **trimmers**, **subscripts** and **bounds**
  - (e) **Enclosed-clause** of a **format-pattern**
  - (f) **Enclosed-clause** of a **replicator**
  - (g) Left-hand side integral **tertiary** of **stowed-functions**
  - (h) The boolean **enclosed-clause** of an **assertion**

## 4. Weak contexts

- (a) **Primaries of slices**
- (b) **Secondaries of selections**
- (c) Right-hand side **tertiary** of a **stowed-function**

## 5. Soft contexts

- (a) The destination **tertiary** of **assignments**
- (b) One **tertiary** in an **identity-relation**, as to adapt it to the other **tertiary**.

## 8.6 Coercions

There are six coercions in Algol 68:

1. dereferencing
2. deproceduring
3. uniting
4. widening
5. rowing
6. voiding

Allowed coercions per context are:

1. Strong context
  - (a) dereferencing or deproceduring
  - (b) uniting
  - (c) widening
  - (d) rowing
  - (e) voiding
2. Firm context
  - (a) dereferencing or deproceduring
  - (b) uniting

3. Meek context

- (a) dereferencing or deproceduring

4. Weak context

- (a) dereferencing or deproceduring yielding a name if possible or a value otherwise.

5. Soft context

- (a) deproceduring

Note that first a sequence of deproceduring or dereferencing — alternately if needed — can take place, then uniting, then a sequence of widenings, after that rowing and finally voiding can take place. For example, you cannot coerce `[] INT` to `[] REAL`. In any context, you can use a **cast** {see 8.13} which will always make a context strong and because all coercions are available in a strong context, you can use the **cast** to specify the mode you require (provided the source mode can be coerced to the required mode in a strong context).

### 8.6.1 Deproceduring

This coercion is available in every context. Deproceduring is the mechanism by which a parameter-less procedure is called. For example, a procedure having mode `PROC REAL`, when called yields a `REAL`. We can represent the coercion by:

$$\text{PROC REAL} \rightarrow \text{REAL}$$

The coercion can be applied repeatedly, for instance:

$$\text{PROC PROC REAL} \rightarrow \text{PROC REAL} \rightarrow \text{REAL}$$

Deproceduring only occurs with parameter-less procedures, and only if the context requires an object of the mode of the yield of the procedure in question.

### 8.6.2 Dereferencing

This is the process of moving from a name to the value to which it refers. That value can again be a name. For example, if we dereference `REF REAL`, the coercion can be represented by:

$$\text{REF REAL} \rightarrow \text{REAL}$$

As deproceduring, dereferencing can be applied repeatedly to produce for instance:

$$\text{REF REF REAL} \rightarrow \text{REF REAL} \rightarrow \text{REAL}$$

### 8.6.3 Dereferencing in a weak context

This is a variant of the dereferencing coercion in which any number of `REF`s can be removed except the final one. This can be represented by:

$$\text{REF REF REAL} \rightarrow \text{REF REAL}$$

This coercion is only available in weak contexts and is needed in the **primary** of **slices** and the **secondary** of a **selection** to ensure that a **slice** or **selection** can yield a name — without this you would not be able to assign a value to an element of a row or to a field of a structure.

### 8.6.4 Dereferencing or deproceduring

In strong, firm, meek and weak contexts dereferencing and deproceduring will be applied alternately if this leads of coercion of the source mode to the mode required by the context. For instance, in strong, firm and meek contexts we could have:

$$\text{REF PROC REF REAL} \rightarrow \text{PROC REF REAL} \rightarrow \text{REF REAL} \rightarrow \text{REAL}$$

though in a weak context we would have:

$$\text{REF PROC REF [] REAL} \rightarrow \text{PROC REF [] REAL} \rightarrow \text{REF [] REAL}$$

### 8.6.5 Uniting

In this coercion, the mode of a value becomes a united mode. For example, if `I` were an operator with both **operands** of mode `UNION(INT, REAL)`, then in the **formula**:

$$0 \text{ I pi}$$

both **operands** will be united to `UNION(INT, REAL)` before the operator is elaborated. These coercions can be represented by:

$$\left. \begin{array}{l} \text{INT} \\ \text{REAL} \end{array} \right\} \rightarrow \text{UNION}(\text{INT}, \text{REAL})$$

Uniting is available in firm contexts and also in strong contexts where it precedes rowing.

### 8.6.6 Widening

In a strong context, an integer can be replaced by a real number and a real number replaced by a complex number, depending on the mode required. `a68g` implements next widenings:

## INFORMAL INTRODUCTION TO ALGOL 68

1. INT  $\rightarrow$  REAL
2. INT  $\rightarrow$  LONG INT
3. LONG INT  $\rightarrow$  LONG REAL
4. LONG INT  $\rightarrow$  LONG LONG INT
5. LONG LONG INT  $\rightarrow$  LONG LONG REAL
6. REAL  $\rightarrow$  COMPLEX
7. REAL  $\rightarrow$  LONG REAL
8. LONG REAL  $\rightarrow$  LONG COMPLEX
9. LONG REAL  $\rightarrow$  LONG LONG REAL
10. LONG LONG REAL  $\rightarrow$  LONG LONG COMPLEX
11. COMPLEX  $\rightarrow$  LONG COMPLEX
12. LONG COMPLEX  $\rightarrow$  LONG LONG COMPLEX
13. BITS  $\rightarrow$  LONG BITS
14. LONG BITS  $\rightarrow$  LONG LONG BITS
15. BITS  $\rightarrow$  [ ] BOOL
16. LONG BITS  $\rightarrow$  [ ] BOOL
17. LONG LONG BITS  $\rightarrow$  [ ] BOOL
18. BYTES  $\rightarrow$  [ ] CHAR
19. LONG BYTES  $\rightarrow$  [ ] CHAR

Widening is not available in **formulas** since **operands** are in a firm context. Widening can be repeated, for example

INT  $\rightarrow$  REAL  $\rightarrow$  LONG REAL  $\rightarrow$  LONG COMPLEX

### 8.6.7 Rowing

If, in a strong context, a row is required and a value is provided whose mode is the mode of the elements of the row, then the value will be rowed. The resulting row (or added dimension) has one element and **bounds** 1 : 1. There are two cases to consider:



1. Suppose we want to row an element of mode `SOME` to a row `[] SOME`. The required rowed value is not a name. For example, if the required mode is `[] INT`, then the mode of an element is `INT`. In the **identity-declaration**:

```
[] INT i = 0
```

the value yielded by the right-hand side (an integer) will be rowed and the coercion can be expressed as:

$$\text{INT} \rightarrow \text{[]} \text{INT}$$

If the value given is a row mode, such as `[] INT`, then there are two possible rowings:

(a) `[] INT`  $\rightarrow$  `[] [] INT`, or

(b) `[] INT`  $\rightarrow$  `[, ] INT`

(an extra dimension is added to the row)

2. If the row required is a name, then a name of an element can be rowed. For example, if the value supplied is a name with mode `REF SOME`, then a name with mode `REF [] SOME` will be created. Likewise, a name of mode `REF [] SOME` can be rowed to a name with mode `REF [, ] SOME` or a non-name with mode `[] REF [] SOME`, depending on the mode required.

## 8.6.8 Voiding

In a strong context, a value can be discarded, either because the mode `VOID` is explicitly stated, as in a procedure yielding `VOID`, or because the context demands it, as in the case of a **semicolon-symbol** (the **go-on-symbol**). Voiding can be applied to any value that is not a (variable whose value is a) parameter-less routine. A parameter-less routine is deprocedured and if it is not a `PROC VOID` the value after deproceduring is discarded. To avoid unexpected behaviour, deproceduring is not used to coerce an **assignment**, a **generator** or a **cast** to `VOID`. Hence if we consider:

```
PROC p = VOID: ...;
PROC VOID pp;
pp := p;
```

then voiding the **assignment** `pp := p` does not involve deproceduring `p` after the **assignment** is completed.

## 8.7 Particular program

A **particular-program** is the actual application, embedded in the standard environ.

- **particular-program:**  
   **label-sequence-option, enclosed-clause.**

## 8.8 Clauses

**Serial-clauses** and **enquiry-clauses** describe how **declarations** and **units** are put in sequence. Algol 68 requires clauses to yield a value. As **declarations** yield no value, **serial-clauses** and **enquiry-clauses** cannot end in a **declaration**. **EXIT** leaves a serial-clause, yielding the value of the preceding **unit**. If the **unit** following **EXIT** would not be labelled, it could never be reached. In a **serial-clause** there cannot be **labelled-units** before **declarations** to prevent re-entering **declarations** once they have been executed.

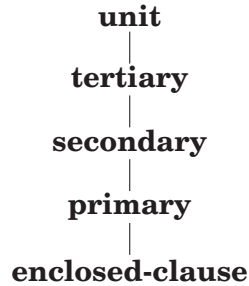
- **serial-clause:**  
   **initialiser-series-option,**  
   **labelled-unit-series.**
- **labelled-unit:**  
   **unit;**  
   **label, unit;**  
   **unit, exit-symbol, label, unit.**
- **initialiser:**  
   **unit;**  
   **declaration-list.**
- **\*phrase: declaration; unit.**

An **enquiry-clause** yields a value to direct the **conditional-clause**, **integer-case-clause**, **united-case-clause**, **while-part** or **until-part** in a **loop-clause**. An **enquiry-clause** contains no **labels**, so you cannot for instance jump back to the **enquiry-clause** at **IF** from the **serial-clause** at **THEN**.

- **enquiry-clause:**  
   **initialiser-series-option,**  
   **unit-series.**

## 8.9 Units

**Units** are orthogonal; for instance an **enclosed-clause** can be an **operand** in a **formula**. The constituent constructs of phrases are **primaries**, **secondaries**, **tertiaries** and **units**. **Units** are arranged in this hierarchy:



where each class includes the lower class. For example, all **primaries** are **secondaries**, but not vice versa. This hierarchy of **units** prevents writing ambiguous **programs**; it specifies for example that:

```
z[k] := z[k] + 1
```

can only mean:

```
((z)[k]) := ((z)[k]) + (1)
```

or that:

```
ref OF ori OF z :=: NIL
```

can only mean:

```
(ref OF (ori OF (z))) :=: (NIL)
```

## 8.10 Enclosed-clauses

**Enclosed-clauses** provide structure for a **particular-program**. There are seven types of **enclosed-clause**.

1. The simplest is the **closed-clause** which consists of a **serial-clause** enclosed in parentheses (or `BEGIN` and `END`).
2. **Collateral-clauses** are generally used as **row-displays** or structure displays: there must be at least two **units**. The **units** are elaborated collaterally. This means that the order is undefined and may well be in parallel.
3. A **parallel-clause** is a **collateral-clause** preceded by `PAR`. The constituent **units** are executed in parallel.
4. The **loop-clause** is discussed in section 3.6.
5. The **conditional-clause** is discussed in section 3.2.

6. The **integer-case-clause** is discussed in section 3.5.
7. The **united-case-clause** is discussed in section 6.3.

It should be noted that the **enquiry-clause** in a **conditional-clause**, **case-clause** or **loop-clause** is in a meek context whatever the context of the clause. Thus, the context of the clause is passed on only to the terminal **unit** in the `THEN`, `ELSE`, `IN` or `OUT` **serial-clauses**.

- **enclosed-clause:**  
  **closed-clause,**  
  **collateral-clause,**  
  **parallel-clause,**  
  **conditional-clause,**  
  **integer-case-clause,**  
  **united-case-clause,**  
  **loop-clause.**
- **closed-clause:**  
  **begin-symbol, serial-clause, end-symbol.**
- **collateral-clause:**  
  **begin-symbol, unit-list-proper, end-symbol.**
- **\*row-display:**  
  **begin-symbol, unit-list-proper-option, end-symbol.**
- **\*structure-display: strong-collateral-clause.**

In a **parallel-clause** the elaboration of **units** can be synchronised using semaphores. Section 8.12 discusses the `a68g` implementation of the **parallel-clause**.

- **parallel-clause:**  
  **par-symbol, begin-symbol, unit-list, end-symbol.**
- **conditional-clause:**  
  **if-symbol, meek-boolean-enquiry-clause,**  
  **then-symbol, serial-clause,**  
  **elif-part-option,**  
  **else-part-option,**  
  **fi-symbol.**
- **elif-part:**  
  **elif-symbol, meek-boolean-enquiry-clause,**

**then-symbol, serial-clause,  
elif-part-option.**

- **else-part:**  
else-symbol, serial-clause.
- **integer-case-clause:**  
case-symbol, meek-integer-enquiry-clause,  
in-symbol, unit-list-proper,  
integer-ouse-part-option,  
out-part-option  
esac-symbol.
- **integer-ouse-part:**  
ouse-symbol, meek-integer-enquiry-clause,  
in-symbol, unit-list-proper,  
integer-ouse-part-option.
- **out-part:**  
out-symbol, serial-clause.
- **united-case-clause:**  
case-symbol, meek-united-enquiry-clause,  
in-symbol, specified-unit-list,  
united-ouse-part-option,  
out-part-option,  
esac-symbol.
- **specified-unit:**  
open-symbol, formal-declarer, identifier-option, close-symbol, colon-symbol,  
unit.  
open-symbol, void-symbol, close-symbol, colon-symbol, unit.
- **united-ouse-part:**  
ouse-symbol, meek-united-enquiry-clause,  
in-symbol, specified-unit-list,  
united-ouse-part-option.
- **case-clause:**  
integer-case-clause;  
united-case-clause.
- **loop-clause:**  
for-part-option,  
from-part-option,

**by-part-option,**  
**to-part-option,**  
**while-part-option,**  
**do-part.**

- **for-part:**  
**for-symbol, identifier.**
- **from-part:**  
**from-symbol, meek-integer-unit.**
- **by-part:**  
**by-symbol, meek-integer-unit.**
- **to-part:**  
**to-symbol, meek-integer-unit;**  
**downto-symbol, meek-integer-unit.**
- **while-part:**  
**while-symbol, meek-boolean-enquiry-clause.**
- **do-part:**  
**do-symbol, serial-clause, od-symbol;**  
**do-symbol, serial-clause-option, until-part, od-symbol.**
- **until-part:**  
**until-symbol, meek-boolean-enquiry-clause.**

## 8.11 Balancing

**Enclosed-clauses** such as **conditional-clauses**, **integer-case-clauses** and **united-case-clauses** yield one of a number of **units**, and it is quite possible for the **units** to yield values of different modes. The principle of balancing allows the context of all these **units**, except one, to be promoted to strong, whatever the context of the **enclosed-clause**. The one that is not promoted to strong remains in the imposed context of the clause, is of course the one mode to which the promoted **units** can be strongly coerced. Balancing is also invoked for **identity-relations** which are discussed in section 3.4. Consider for example the **formula**:

$$1 + (a > 0 \mid 1 \mid 0.0)$$

where the context of the **conditional-clause** is firm hence widening is not allowed. Without balancing, the **conditional-clause** could yield a **REAL** or an **INT**. In this example, the principle of balancing would promote the context of the **INT** to strong and widen it to **REAL**. In a balanced clause, one of the yielded **units** remains in the context of the clause and all the others are in a strong context, irrespective of the actual context of the clause.

## 8.12 Parallel processing in a68g

Algol 68 supports parallel processing on platforms that support POSIX threads, such as Linux. Using the reserved word `PAR`, a **collateral-clause** becomes a **parallel-clause**, in which the synchronisation of actions is controlled using semaphores. In a68g the parallel actions are mapped to POSIX threads when available on the hosting operating system (and a68g must be built to support the **parallel-clause** {see 9.3.2}). Parallel **units** coordinate their actions by means of semaphores. A semaphore constitutes the classic method for restricting access to shared resources, such as shared stack and heap. It was invented by Edsger Dijkstra. Usually (and also in Algol 68) the term refers to a counting semaphore, since a binary semaphore is known as a mutex. A counting semaphore is a counter for a set of available resources, rather than a locked-unlocked flag of a single resource. Semaphores are the classic solution to preventing race conditions in the dining philosophers problem (a generic, abstract problem used for explaining issues with mutual exclusion), although they do not prevent resource deadlocks. Semaphores remain in common use in programming languages that do not support other forms of synchronisation. Semaphores are the primitive synchronisation mechanism in many operating systems. The trend in programming language development is towards more structured forms of synchronisation, such as monitors and channels. In addition to their inadequacies in dealing with (multi-resource) deadlocks, semaphores do not protect the programmer from easy mistakes like taking a semaphore that is already held by the same process, or forgetting to release a semaphore. It has been put forward that it is debatable whether a low-level feature as the semaphore is a proper feature for a high-level language as Algol 68. A semaphore in Algol 68 is an object of mode `SEMA` :

```
MODE SEMA = STRUCT (REF INT F)
```

Note that the field cannot be directly selected. For a semaphore, next operators are defined:

1. `OP LEVEL = (INT a) SEMA`  
Yields a semaphore whose value is `a`.
2. `OP LEVEL = (SEMA a) INT`  
Yields the level of `a`, that is, field `F OF a`.
3. `OP DOWN = (SEMA a) VOID`  
The level of `a` is decremented. If it reaches 0, then the parallel **unit** that called this operator is hibernated until another parallel **unit** increments the level of `a` again.
4. `OP UP = (SEMA a) VOID`  
The level of `a` is incremented and all parallel **units** that were hibernated due to this semaphore being down are awakened.

Next classical example demonstrates the use of semaphores in Algol 68: producer-consumer-type parallel processes. In this simple example, production is incrementing an integer and

consumption is decrementing it. Synchronisation is necessary to prevent both "production" and "consumption" from accessing the integer (the "resource") at the same time; hence the "consuming" action has to wait for the "producing" action to finish, and vice versa. This is done in line 2 by giving both actions a semaphore, the "producing" semaphore at level 1 so production can start, and the "consuming" semaphore at level 0 so it must initially wait for something being produced. Each parallel action starts by performing `DOWN` on its semaphore, then do its job, and then perform an `UP` on the semaphore of the other action.

```
BEGIN
  INT n := 0, SEMA consume = LEVEL 0, produce = LEVEL 1;
  PAR BEGIN # produce one #
    DO DOWN produce;
      print (n += 1);
    UP consume
  OD,
  # consume one #
  DO DOWN consume;
    print (n -= 1);
  UP produce
  OD
END
END
```

The a68g **parallel-clause** deviates from the standard Algol 68 **parallel-clause** when **parallel-clauses** are nested. a68g parallel **units** behave like threads with private stacks. Hence if parallel **units** modify a shared variable then this variable must be declared outside the outermost **parallel-clause**, and a **jump** out of a parallel **unit** can only be targeted at a **label** outside the outermost **parallel-clause**.

## 8.13 Primaries

**Primaries** are **denotations**, **applied-identifiers**, **casts**, **calls**, **format-texts**, **enclosed-clauses** and **slices**. **Applied-identifiers** are **identifiers** being used in a context, rather than in their **declarations** where they are defining **identifiers**. **Routine-texts** are not **primaries**, though **format-texts** are.

### 8.13.1 Primaries

- **primary:**  
   **enclosed-clause,**  
   **identifier,**



**specification,  
cast,  
format-text,  
denotation.**

- **specification:**  
  **call,  
  slice,  
  field-selection.**

**Calls** were discussed in sections 5.3 and 5.4. A **call** invokes a procedure, but can be partially parametrised: partial parametrisation adds arguments to a procedure's locale; when the locale is complete the procedure is called, otherwise currying takes place. In section 8.5, it was mentioned that the **primary** of a **call** is in a meek context. This applies even if the **call** itself is in a strong context.

- **call:**  
  **meek-primary, open-symbol, actual-parameter-list, close-symbol.**
- **actual-parameter:**  
  **strong-unit-option.**

A **slice** selects an element or a sub-row from a rowed value and were discussed **slices** in section 4.2.5. Whatever the context of the **slice**, the context of the **primary** of the **slice** is weak. This means that if the primary yields a name, then the slice will yield a name. **Collateral-clauses** used as **row-displays** can only be used in a strong context. So if you want to slice a **row-display**, the **row-display** must be enclosed in a **cast**. The context of **units** in **subscripts** and **trimmers** is meek.

- **slice:**  
  **weak-primary, sub-symbol, indexer-list, bus-symbol.**
- **indexer:**  
  **subscript;  
  trimmer.**
- **trimmer:**  
  **lower-index-option,  
  colon-symbol,  
  upper-index-option,  
  revised-lower-bound-option.**

- **subscript: meek-integer-unit.**
- **lower-index: meek-integer-unit.**
- **upper-index: meek-integer-unit.**
- **revised-lower-bound:**  
     **at-symbol, meek-integer-unit.**

**Field-selections** are an a68g extension described in section 4.8.

- **field-selection:**  
     **weak-primary, sub-symbol, identifier-list, bus-symbol.**

A **cast** consists of a **mode-indicant** followed by an **enclosed-clause**. A **cast** puts its **enclosed-clause** in a strong context. It can be used to dereference exactly to the name required, as seen in paragraph 2.22. It can also be used to force the mode of a display when required:

`z + COMPLEX (1, 0)`

- **cast:**  
     **formal-declarer, strong-enclosed-clause.**
- **denotation:**  
     **integer-denotation;**  
     **real-denotation;**  
     **boolean-denotation;**  
     **bits-denotation;**  
     **character-denotation;**  
     **row-of-character-denotation;**  
     **void-denotation.**
- **integer-denotation:**  
     **length-option, digit-sequence.**
- **real-denotation:**  
     **length-option, digit-sequence, exponent-part;**  
     **length-option, digit-sequence-option, point-symbol, digit-sequence,**  
     **exponent-part-option.**
- **exponent-part:**  
     **letter-e-symbol, sign-option, digit-sequence.**

- **boolean-denotation:**  
true-symbol;  
false-symbol.
- **bits-denotation:**  
length-option, bits-digit-sequence;  
bits-digit-sequence
- **character-denotation:**  
quote-symbol, string-item, quote-symbol.
- **string-item:**  
character;  
quote-symbol, quote-symbol.
- **row-of-character-denotation:**  
quote-symbol, string-item-sequence, quote-symbol.
- **void-denotation:**  
empty-symbol.

### 8.13.1.1 Formats

**Format-texts** are discussed in section 7.11.

- **format-text:**  
formatter-symbol, picture-list-formatter-symbol.
- **replicator:**  
integer-denotation;  
letter-n-symbol, meek-integer-enclosed-clause.
- **MARKER-frame:**  
insertion-option, replicator-option, letter-s option, **MARKER**;
- **picture:**  
insertion;  
pattern;  
collection;  
replicator-collection.
- **collection:**  
open-symbol, picture-list, close-symbol.
- **insertion:** insertion-item-list.

- **insertion-item:**
  - replicator-option, letter-k-symbol;
  - replicator-option, letter-l-symbol;
  - replicator-option, letter-p-symbol;
  - replicator-option, letter-x-symbol;
  - replicator-option, letter-q-symbol;
  - replicator-option, row-of-character-denotation.
  
- **pattern:**
  - general-pattern;
  - integer-pattern;
  - real-pattern;
  - complex-pattern;
  - bits-pattern;
  - string-pattern;
  - boolean-pattern;
  - choice-pattern;
  - format-pattern.
  - c-style-pattern.
  
- **general-pattern:**
  - letter-g-symbol, strong-row-of-integer-enclosed-clause;
  - letter-h-symbol, strong-row-of-integer-enclosed-clause;
  
- **integer-pattern:** sign-mould-option, integer-mould.
  
- **sign-mould:** integer-mould-option, sign.
  
- **integer-mould:** digit-marker-sequence, insertion-option.
  
- **digit-marker:**
  - letter-z-frame;
  - letter-d-frame.
  
- **real-pattern:**
  - sign-mould-option, integer-mould-option, letter-s-symbol-option, point-symbol, insertion-option, integer-mould, exponent-frame-option;
  - sign-mould-option, integer-mould, letter-s-symbol-option, point-symbol, insertion-option, integer-mould-option, exponent-frame-option;
  - sign-mould-option, integer-mould, exponent-frame;
  
- **exponent-frame:**
  - letter-s-symbol-option, letter-e-symbol, insertion-option, sign-mould-option, integer-mould.

- **complex-pattern:**  
real-pattern, letter-s-symbol-option, letter-i-symbol, insertion-option, real-pattern.
- **complex-pattern:**  
real-pattern, letter-s-symbol-option, letter-i-symbol, insertion-option, real-pattern.
- **bits-pattern:**  
replicator, letter-r-symbol, integer-mould.
- **string-pattern:** letter-a-frame-sequence, insertion-option.
- **boolean-pattern:**  
letter-b-symbol;  
letter-b-symbol, open-symbol, row-of-character-denotation,  
comma-symbol, row-of-character-denotation, close-symbol.
- **choice-pattern:**  
letter-c-symbol, open-symbol, row-of-character-denotation-list, close-symbol.
- **format-pattern:**  
letter-f-symbol, meek-format-enclosed-clause.
- **c-style-pattern:**  
percent-symbol,  
minus-symbol-option,  
plus-symbol-option,  
width-option,  
precision-option,  
c-type.
- **width:** replicator.
- **precision:** point-symbol, replicator.
- **c-type:**  
letter-b-symbol,  
letter-c-symbol,  
letter-d-symbol,  
letter-e-symbol,  
letter-f-symbol,  
letter-g-symbol,

**letter-i-symbol,  
letter-o-symbol,  
letter-s-symbol,  
letter-x-symbol.**

## 8.14 Secondaries

- **secondary:**  
**primary;**  
**selection;**  
**generator.**

**Selections** are described in section 4.7.1.

- **selection: identifier, of-symbol, secondary.**

**Generators** are introduced in section 2.16.

- **generator: qualifier, actual-declarer.**

## 8.15 Tertiaries

- **tertiary:**  
**secondary;**  
**nihil;**  
**formula;**  
**stowed-function.**

**NIL** is introduced in section 2.23.

- **nihil: nil-symbol.**

**Formulas** are covered in section 2.5.

- **formula:**  
**monadic-operator-sequence, monadic-operand;**  
**dyadic-operand, dyadic-operator, dyadic-operand.**

- **monadic-operand:**  
secondary;
- **dyadic-operand:**  
monadic-operator-sequence-option, monadic-operand;  
formula.
- **\*operand:** monadic-operand; dyadic-operand.

**Stowed-functions** are described in section 4.5.

- **stowed-function:**  
monadic-stowed-symbol, weak-tertiary,  
meek-integral-tertiary, dyadic-stowed-symbol, weak-tertiary.
- **monadic-stowed-symbol:**  
diag-symbol, trnsp-symbol, row-symbol, col-symbol.
- **dyadic-stowed-symbol:**  
diag-symbol, row-symbol, col-symbol.

## 8.16 Units

**Units** are **assignments**, **routine-texts**, **identity-relations** {see 3.4},

### 8.16.1 Units

- **unit:**  
tertiary;  
assignment;  
routine-text;  
identity-relation;  
jump;  
skip;  
assertion;  
conditional-function.

**Assignations** are discussed in section 2.17.

- **assignment:** soft-tertiary, becomes-symbol, strong-unit.

The **identity-relation** is introduced in section 3.4.

- **identity-relation:**  
  **soft-tertiary-is-symbol, soft-tertiary;**  
  **soft-tertiary-isnt-symbol, soft-tertiary;**

For a description of **jumps** {see 8.17}.

- **jump: goto-symbol-option, identifier.**

A **skip** is indicated by the reserved word `SKIP`. This construct terminates and yields an undefined value of the mode required by the context, and can therefore only occur in a strong context. It is particularly useful in cases where the result is undefined:

```
OP INVERSE = (MATRIX m) MATRIX:  
  IF DET m = 0 THEN SKIP ELSE ... FI
```

If the determinant of a matrix is zero, no unique inverse exists and the operator yields an undefined matrix.

- **skip: skip-symbol.**

**Assertions** are described in section 8.18.

- **assertion: assert-symbol, meek-boolean-enclosed-clause.**

**Conditional-functions** are discussed in {see 3.3}.

- **conditional-function:**  
  **meek-boolean-tertiary, conditional-symbol, meek-boolean-tertiary.**
- **conditional-symbol:**  
  **thef-symbol; andf-symbol; andth-symbol;**  
  **elsf-symbol; orf-symbol; orel-symbol.**

**Routine-texts** were discussed in chapter 5.

- **routine-text:**  
  **routine-specification, colon-symbol, strong-unit.**
- **routine-specification:**  
  **parameter-pack-option, formal-declarer.**  
  **parameter-pack-option, void-symbol.**



- **parameter-pack:**  
open-symbol, formal-parameter-list, close-symbol.
- **formal-parameter:** formal-declarer, identifier.

## 8.17 Jumps

Sometimes it is desirable to have more than one possible end-point of a **serial-clause**. This typically happens when a piece of code needs to abort. Like most other programming languages, Algol 68 allows to label **units** in **serial-clauses**. After a **label** definition, you cannot write **declarations**. A **label** is an **identifier**, unique in its reach, followed by a **colon-symbol**. **Declarations** cannot be labelled. A **jump** to a **label** is invoked by writing the name of a **label** as a single **primary**. We have seen one **jump** already in this book:

```
stop
```

which effectively terminates execution since `stop` is a **label** defined at the last phrases of the standard environment. Another example is the use of a **jump** in a transput event-routine:

```
on file end (standin, PROC (REF FILE f) BOOL: GO TO file empty)
```

To invoke a **jump**, you may write `GO TO` or `GOTO` in front of the **identifier**:

```
stop
GOTO stop
GO TO stop
```

**Labels** are not allowed in **enquiry-clauses** since you would be able to jump for instance from a **then** part back into the condition. **Declarations** not allowed after a **label** definition since you would for instance be able to jump back to before a **declaration** and execute it again, facilitating dangling names and scope errors. The rule is that any **declaration** in a **serial-clause** must be seen at most once before that **serial-clause** ends. This means that after a **labelled-unit**, you are free to use **enclosed-clauses** containing their private **declarations** since a **jump** out of a **serial-clause** effectively ends that **serial-clause**, and above rule is satisfied. **Jumps** are like **skips**; they also yield an undefined object of the mode required by the context. But if the context expects a parameter-less procedure of mode `PROC VOID`, then a `PROC VOID` routine whose **unit** is that **jump** is yielded, instead of making the **jump**:

```
#!/usr/local/bin/a68g
```

```
# Commented text taken from:
C. H. Lindsey, A History of Algol 68,
ACM Sigplan Notices, Volume 28, No. 3 March 1993 #
```

## INFORMAL INTRODUCTION TO ALGOL 68

```
# ... But worse! Van Wijngaarden was now able to exhibit his
pride and joy - his pseudo-switch [R8.2.7.2]. #
[] PROC VOID switch = (e1, e2, e3);

# or even #
LOC [1 : 3] PROC VOID switch var := (e1, e2, e3);
switch var[2] := e3;

# To my shame, I must admit that this still works, although implementations
tend not to support it. #

switch var[2];
print ("should not be here");
stop;
e3: e2: e1: print ("jumped correctly")
```

Put above code in a file *jump.a68*, make it executable with `chmod` and execute the file to find:

```
$ ./jump.a68
jumped correctly
```

which demonstrates that a68g implements proceduring to PROC VOID.

A completer, with reserved word `EXIT`, provides a completion point for a **serial-clause** but cannot occur in an **enquiry-clause**. A completer can be placed wherever a **semicolon-symbol** (the **go-on-symbol** `;`) can appear. A completer must be followed by a **label** definition or the **unit** following the completer could not be reached. This is an example of a completer in line 5:

```
BEGIN REAL x = read real;
  IF i < 0
  THEN GOTO negative
  FI;
  sqrt (i) EXIT
  negative:
  0
END
```

The example also illustrates why this is not a recommendable programming style. In fact, **jumps** have their use when it is required to jump out of nested clauses when something unexpected occurs from which recovery is not practical. Use of **jumps** should be confined to exceptions since they can make your **programs** illegible.

## 8.18 Assertions as pre- or postconditions

Algol 68 Genie supports an extension called **assertions**. **Assertions** can be viewed in two ways. First, they provide a notation for invariants that can be used to code a proof of correctness together with an algorithm. Hardly anyone does this, but **assertions** make for debugging statements. **Assertion** syntax reads:

- **assertion: assert-symbol, meek-boolean-enclosed-clause.**

Under control of the pragmat items 'assertions' and 'noassertions', the **BOOL enclosed-clause** of an **assertion** is elaborated at runtime. If the **enclosed-clause** yields **TRUE** execution continues but if it yields **FALSE**, a runtime error is produced. For example:

```
OP FACULTY = (INT n) INT:
  IF ASSERT (n >= 0);
    n > 0
  THEN n * FACULTY (n - 1)
  ELSE 1
  FI
```

will produce a runtime error when **FACULTY** is called with a negative argument.

## 8.19 Declarations

**Declarations** introduce new tags, or specify the priority of an operator.

- **declaration:**
  - mode-declaration;**
  - identity-declaration;**
  - variable-declaration;**
  - procedure-declaration;**
  - procedure-variable-declaration;**
  - operator-declaration;**
  - priority-declaration.**
- **mode-declaration:**
  - mode-symbol, mode-definition-list.**
- **mode-definition:**
  - mode-indicant, equals-symbol, actual-declarer.**

- **identity-declaration:**  
formal-declarer, identity-definition-list.
- **identity-definition:**  
identifier, equals-symbol, strong-unit.
- **variable-declaration:**  
qualifier-option, actual-declarer, variable-definition-list.
- **variable-definition-list:**  
identifier, initialisation-option.
- **initialisation:** becomes-symbol, strong-unit.
- **procedure-declaration:**  
proc-symbol, procedure-definition-list.
- **procedure-definition:** identifier, equals-symbol, routine-text.
- **procedure-variable-declaration:**  
qualifier-option, proc-symbol, procedure-variable-definition-list.
- **procedure-variable-definition:** identifier, becomes-symbol, routine-text.
- **brief-operator-declaration:**  
operator-symbol, brief-operator-definition-list.
- **brief-operator-definition-list:**  
operator, equals-symbol, routine-text.
- **operator-declaration:**  
operator-plan, operator-definition-list.
- **operator-plan:**  
operator-symbol, parameter-pack, formal-declarer.
- **operator-definition-list:**  
operator, equals-symbol, strong-unit.
- **priority-declaration:**  
prio-symbol, priority-definition-list.
- **priority-definition-list:**  
operator, equals-symbol, priority-digit

## 8.20 Declarers

**Declarers** specify modes. The context determines whether a mode is formal, virtual, or actual. Formal or virtual **declarers** are needed where the size of rows is irrelevant. **Actual-declarers** are needed where the size of rows must be known, for instance when allocating memory for rows.

- **victal: virtual; actual; formal.**
- **victal-declarer:**  
   **length-option, primitive-declarer;**  
   **mode-indicant;**  
   **ref-symbol, virtual-declarer;**  
   **sub-symbol, victal-bounds-list, bus-symbol, victal-declarer,**  
   **flex-symbol, sub-symbol, victal-bounds-list, bus-symbol, victal-declarer,**  
   **struct-symbol, open-symbol, victal-declarer-identifier-list, close-symbol,**  
   **union-symbol, open-symbol, united-declarer-list, close-symbol;**  
   **proc-symbol, formal-declarer-pack-option, formal-declarer;**  
   **proc-symbol, formal-declarer-pack-option, void-symbol.**
- **formal-declarer-pack:**  
   **open-symbol, formal-declarer-list, close-symbol.**
- **united-declarer: formal-declarer; void-symbol.**
- **formal-bounds: colon-symbol-option.**
- **virtual-bounds: colon-symbol-option.**
- **actual-bounds: lower-bound-option, upper-bound.**
- **lower-bound: meek-integer-unit, colon-symbol.**
- **upper-bound: meek-integer-unit.**

## 8.21 Pragments

**Pragments** are either **pragmats** {2.25} or **comments** {2.24}. In a68g, **pragmats** contain preprocessor directives, or set options from within a **program**.

- **pragment: pragmat; comment.**
- **pragmat:**  
   **pragmat-symbol, pragmat-item-sequence, pragmat-symbol.**

- **comment:**  
**comment-symbol-character-sequence-comment-symbol.**

## 8.22 Refinements

A low-level tool for top-down program construction is the **refinement** preprocessor. **Refinements** are not a part of Algol 68 syntax, but are superimposed on top of it. Note that **refinements** interfere somewhat with **labels**.

- **refined-program:** **paragraph, point-symbol, refinement-definition-sequence.**
- **refinement-definition:** **identifier, colon-symbol, paragraph, point-symbol.**
- **paragraph:** **character-sequence.**

## **Part II**

# **Programming with Algol 68 Genie**





# Chapter 9

## Installing and using Algol 68 Genie

{En cuestiones de cultura y de saber,  
sólo se pierde lo que se guarda;  
sólo se gana lo que se da.  
Antonio Machado. }

### 9.1 Algol 68 Genie

This chapter describes Algol 68 Genie, a hybrid compiler-interpreter; how to install it on your computer system and how to use the program. Algol 68 Genie implements almost all of Algol 68, and extends<sup>1</sup> that language. To run the **programs** described in this book you will need a computer with a Linux system or work-alike. Algol 68 Genie (`a68g`) is open source software distributed under GNU GPL. This software is distributed in the hope that it will be useful, but **without any warranty**. Consult the GNU General Public License<sup>2</sup> for details. A copy of the license is in this book.

An interpreter is a program that executes code written in a programming language. While interpretation and compilation are the two principal means for implementing programming languages, these are not fully distinct categories. An interpreter may be a program that either (1) executes source code directly, (2) translates source code into some intermediate representation which is executed or (3) executes stored code from a compiler which is part of the interpreter system. Perl, Python, MATLAB, and Ruby are examples of type (2), while UCSD Pascal and Java and `a68g`<sup>3</sup> are type (3). After successful parsing of an entire source **program**, `a68g` will interpret the syntax tree that serves as an intermediate **program** representation. The **unit** compiler may construct routines by which Algol 68

---

<sup>1</sup>You can disable most extensions using the `strict` option, see section 9.6.4.

<sup>2</sup>See <http://www.gnu.org/licenses/gpl.html>.

<sup>3</sup>If you specify `optimise` `a68g` is type (3), otherwise it is type (2).

Genie can efficiently execute selected **units**. Algol 68 Genie employs a multi-pass scheme to parse Algol 68 [Lindsey 1993] {see 9.9}. The interpreter proper performs many runtime checks<sup>4</sup>:

1. Assigning to, or dereferencing of, `NIL` .
2. Using uninitialised values.
3. Invalid **operands** to standard-*prelude* operators and procedures.
4. **Bounds** check when manipulating rows.
5. Overflow of arithmetic modes.
6. *Dangling references*, that are names that refer to deallocated memory.

### 9.1.1 The Algol 68 Genie unit compiler

On Linux or compatible operating systems, Algol 68 Genie can run in optimising mode, in which it employs a **unit** compiler that emits C code for many **units** involving operations on primitive modes `INT`, `REAL`, `BOOL`, `CHAR` and `BITS` and simple structures thereof such as `COMPLEX`. Execution time of such **units** by interpretation is dominated by interpreter overhead, which makes compilation of these units worthwhile. Generated C code is compiled and dynamically linked before it is executed by Algol 68 Genie. Technically, the compiler synthesizes per selected **unit** efficient routines compounding the elemental routines needed to execute terminals in the syntax tree which allows for instance common sub-expression elimination. Generated C code is compatible with the virtual stack-heap machine implemented by the interpreter proper, hence generated code has full access to `a68g`'s runtime library and the interpreter's debugger. Note that this scheme involves *ahead of time* compilation, the **unit** compiler is not a *just in time* compiler as used in for example Java implementations. The **unit** compiler omits many runtime checks for the sake of efficiency. Therefore, it is recommended to not specify option `optimise` or `compile` while your program is still in development, and to use it only for programs that work correctly. Due to overhead, optimisation is not efficient for programs with short execution times, or *run-once* programs typical for programming course exercises. Your mileage will vary depending on your source code; expect a speed increase ranging from *hardly noticeable* to *five times as fast*. The **unit** compiler requires `gcc` as back-end for code generation and the `libdl` library, as dynamic linker loader. If you use the option `compile` to generate shell scripts from Algol 68 programs, also `GNU tar` and `sed` are required. On platforms where the **unit** compiler cannot run, Algol 68 Genie simply executes the intermediate syntax-tree.

---

<sup>4</sup>In this respect `a68g` resembles *FLACC* [Mailloux 1978].

## 9.1.2 Features of Algol 68 Genie

1. Precision of numeric modes {see 2.2, 2.4, 4.10 and 4.12}:
  - (a) Implementation of LONG INT, LONG REAL, LONG COMPLEX and LONG BITS with roughly doubled precision with respect to INT, REAL, COMPLEX and BITS.
  - (b) Implementation of multi-precision arithmetic through LONG LONG INT, LONG LONG REAL, LONG LONG COMPLEX and LONG LONG BITS which are modes with user defined precision which is set by an option.
2. On systems that support them, Linux extensions that allow e.g. for executing child processes that communicate through pipes, matching regular expressions or fetching web page contents, {see 10.17}.
3. Procedures for drawing using the GNU Plotting Utilities {see 10.17}.
4. Various extra numerical procedures, many of which from the GNU Scientific Library {see 10.11}.
5. Basic linear algebra and Fourier transform procedures from the GNU Scientific Library {see 10.12, 10.13 and 10.14}.
6. Support for WAVE/PCM sound format {see 10.21}.
7. Support for PostgreSQL, an open-source relational database management system, enabling client applications in Algol 68 {see 10.20}.
8. **Format-texts**, straightening and formatted transput. Transput routines work generically on files, (dynamic) strings and UNIX pipes {see 7.11}.
9. **Parallel-clause** on platforms that support POSIX threads {see 8.12}.
10. Implementation of C.H. Lindsey's partial parametrisation proposal, giving Algol 68 a functional sub language [Koster 1996] {see 5.10}.
11. A simple **refinement** preprocessor to facilitate top-down **program** construction {see 9.8.3}.
12. Symbol NEW as alternative for symbol HEAP.
13. **Field-selections** as alternative syntax for **selections**, see 4.8.
14. Implementation of pseudo-operators ANDF and ORF (or their respective alternatives THEF, ANDTH and OREL, ELSF {see 3.3}.
15. Implementation of pseudo-operators TRNSP, DIAG, COL and ROW as described by [Torix 1977] {see 4.5}.

16. Implementation of `DOWNTO` with comparable function as `TO` in **loop-clauses**; `DOWNTO` decreases, whereas `TO` increases, the loop counter by the value of the (implicit) by-part {see 3.6}.
17. Implementation of a post-checked loop. A do-part may enclose a **serial-clause** followed by an optional until-part, or just enclose an until-part. This is an alternative to the paradigm Algol 68 post-check loop `WHILE ... DO SKIP OD`. An until-part consists of the reserved word `UNTIL` followed by a meek-boolean-enquiry-clause. The **loop-clause** terminates when the **enquiry-clause** yields `TRUE` {see 3.6}.
18. Implementation of monadic- and **dyadic-operator** `ELEMS` that operate on any row. The **monadic-operator** returns the total number of elements while the **dyadic-operator** returns the number of elements in the specified dimension, if this is a valid dimension {see 4.2.5}.
19. When option `brackets` is specified, `(...)`, `[...]` and `{...}` are equivalent to the parser and any pair can be used where Algol 68 requires open-symbols and close-symbols. This allows for clearer coding when parenthesis are nested. If `brackets` is not specified, `(...)` is an alternative for `[...]` in **bounds** and **indexers**, which is traditional Algol 68 syntax.
20. Implementation of operators `SET` and `CLEAR` for mode `BITS`.
21. The parser allows for **colon-symbols**, used in **bounds** and **indexers**, to be replaced by `...` which is the Pascal style.
22. Upper stropping is the default, quote stropping is optional.

### 9.1.3 Deviations from the Revised Report language

1. The important difference with the Revised Report transput model is that Algol 68 Genie transput does not operate on `FLEX [] FLEX [] FLEX [] CHAR`, but on `FLEX [] CHAR`. This maps better onto operating systems as `UNIX` or `Linux`.
2. The Algol 68 Genie parallel-clause deviates from the standard Algol 68 **parallel-clause** when **parallel-clauses** are nested. Algol 68 Genie's parallel **units** behave like threads with private stacks. Hence if parallel **units** modify a shared variable then this variable must be declared outside the outermost parallel-clause, and a **jump** out of a parallel **unit** can only be targeted at a **label** outside the outermost **parallel-clause**.
3. Algol 68 Genie does not implement so-called ghost-elements {16<sub>2</sub>.1.3.4}, hence it cannot check **bounds** when assigning to rows of mode starting with **flexible-rows-of-rows-of** when the destination has a flat descriptor (i.e. has zero elements). A mode `flexible-row-of-...-row-of-...` is treated as if it were flexible throughout. The scope

checker recognises the potential introduction of transient references resulting from this peculiarity in `a68g`.

4. Algol 68 Genie does not recognise non-local environs {19<sub>5</sub>.2.3.2}. All environs are local. Hence constructs as:

```
REF INT i = (... | LOC INT | SKIP);
```

or:

```
[n] REF INT a; FOR k TO n DO a[k] := LOC INT OD
```

result in a scope violation error with Algol 68 Genie.

5. It is not possible to declare in a `[WHILE ...] DO ... [UNTIL ...] OD` part an **identifier** with equal spelling as the loop **identifier**.
6. If the context of a **jump** expects a parameterless procedure of mode `PROC VOID`, then a `PROC VOID` routine whose **unit** is that **jump** is yielded instead of making the **jump**. In standard Algol 68, this proceduring will take place if the context expects a parameterless procedure, while Algol 68 Genie limits this to `PROC VOID`.
7. Algol 68 Genie maps a **declarer** whose length is not implemented onto the most appropriate length available {16<sub>2</sub>.1.3.1}. Algol 68 Genie considers mapped modes equivalent to the modes they are mapped onto, while standard Algol 68 would still set them apart. Routines or operators for not-implemented lengths are mapped accordingly.

## 9.2 Algol 68 Genie transput

`a68g` transput deviates from the Revised Report specification, as described below. For an overview of implemented procedures refer to the standard-prelude reference.

### 9.2.1 Features of Algol 68 Genie transput

1. Transput procedures operate generically on files, strings and Linux pipes.
2. Implementation of a procedure `real` that extends the functionality of `float`.
3. `a68g` implements *ALGOL68C* routines as `read int` and `print int` but not routines as `get int` and `put int`.
4. There are two extra procedures to examine a file: `idf` and `term`.
5. If a file does not exist upon calling `open`, the default action will be to create it on the file system.

6. If a file exists upon calling `establish`, the event handler set by `on open error` will be invoked.
7. `a68g` can write to file `stand error` with associated channel `stand error channel`. This file is linked to the standard error stream that is usually directed at the console.
8. Insertions can be any combination of alignments and literals.
9. The argument for a **general-pattern** is an **enclosed-clause** yielding `[] INT` (which is a superset of the Revised Report specification).
10. Extended **general-pattern** `h` for transputting real values.
11. The radix for a **bits-pattern** can be `2...16`. The radix can be dynamic.
12. Implementation of C-style format placeholders.
13. The Revised Report specification of getting using an **integral-choice-pattern** has the peculiarity that when two literals start with the same sequence of characters, the longer literal should appear first in the list. `a68g` makes no such demand, and will select the correct literal from the list whatever their order in the list.

## 9.2.2 Deviations of standard Algol 68

1. The important difference with the Revised Report transput model is that `a68g` transput does not operate on a `FLEX [] FLEX [] FLEX [] CHAR`, but on a `FLEX [] CHAR`. This maps better onto operating systems as `Linux`.
2. `establish` does not take arguments specifying the size of a file.
3. Getting and putting a file is essentially sequential. Only `reset` can intervene with sequential processing.
4. `a68g` does not currently permit switching between read mood and write mood unless the file is `reset` first (and the file was opened with `standback channel`). Whether a file is in read mood or write mood is determined by the first actual transput operation (`put` or `get`) on a file after opening it.
5. Since `a68g` transput operates on a `FLEX [] CHAR`, routine `associate` is associates a file to a `REF STRING` object. On putting, the string is dynamically lengthened and output is added at the end of the string. Attempted getting outside the string provokes an end of file condition, and `on file end` is invoked. When a file that is associated with a string is `reset`, getting restarts from the start of the associated string.

6. Since an end on file event handler cannot move the file pointer to a good position (re-set also resets a file's read mood and write mood), encountering end of file terminates getting of a `STRING` value. Conform the Revised Report, getting of a `STRING` value will resume if the end of line event handler or the end of page handler returns `TRUE`.
7. There is no event routine on `char error mended`. Attempted conversion of an invalid value for a required mode, or attempted transput of a value that cannot be converted by the current format-pattern, evokes the event routine set by `on value error`.
8. When all arguments in a **call** of `readf`, `printf`, `writeln`, `getf` or `putf` are processed, the format associated with the corresponding file is purged - that is, remaining insertions are processed and the format is discarded. If a **pattern** is encountered while purging, then there was no associated argument and the event routine set by `on format error` is called. When this event routine returns `FALSE` (the default routine always returns `FALSE`) a runtime error will be produced.
9. `SIMPLIN` and `SIMPOUT` are generic for both formatted and unformatted transput. Therefore `SIMPLIN` and `SIMPOUT` include both mode `FORMAT` and `PROC (REF FILE) VOID`. If a transput procedure encounters a value whose transput is not defined by that procedure, a runtime error occurs.
10. Insertion `x` has the same effect as insertion `q`; both call space.
11. When a **sign** is shifted in a **sign-mould**, any character output by literal insertions in that **sign-mould** is replaced with a space as well, starting from the first **letter-z-frame** until the **sign** is put.
12. When printing using a **pattern**, a **letter-z-frame** behaves as a **letter-d-frame** once a non-zero digit has been printed, and in a fractional part of a real number a **letter-z-frame** always behaves as a **letter-d-frame**.
13. When getting a literal insertion, `space` is performed for every character in that literal. It is not checked whether read characters actually match the literal.
14. Routine `set` moves with respect to the current file pointer.

## 9.3 Installing Algol 68 Genie on GNU/Linux

First, you will need to download the latest version of Algol 68 Genie from

<http://www.xs4all.nl/~jmvdveer/algol.html>

In this chapter we assume that the latest version is packed in a file called:

`algol68g-2.0.0.tgz`

which is a gzipped tar-archive.

### 9.3.1 Brief installation guide

If you are a reasonably experienced Linux user, the following list of commands may suffice to help you on your way. If you do not feel confident to follow this short list, follow the detailed instructions in next section. Enter:

```
tar -xzf algol68g-2.0.0.tgz
cd algol68g-2.0.0
./configure -O --threads
make
```

and if you want to install a68g when building as user root, type:

```
make install
```

Targets for files a68g and a68g.h are /usr/local/bin and /usr/local/include, and /usr/local/man for a68g.1. If latter directory cannot be accessed, /usr/share/man and then /usr/man are tried. If you have no permission to write to those directories, the target is the present working directory. This is a fail-safe in case you attempt make install when you are not root.

### 9.3.2 Detailed installation guide

This section describes into detail how to install a68g on Linux.

#### 9.3.2.1 Optional libraries

If available on your system, a68g can use next libraries to extend its functionality:

1. GNU plotutils for drawing from Algol 68, see <http://www.gnu.org/software/plotutils/>
2. GNU scientific library to extend the mathematical capabilities of the standard environment, see <http://www.gnu.org/software/gsl/>
3. PostgreSQL for writing PostgreSQL database client applications, see <http://www.postgresql.org/>

The configure script checks directories:



```

/usr/lib
/usr/local/lib
/usr/include
/usr/local/include

```

for shared libraries and necessary includes. If necessary files are not detected then `a68g` builds without support for absent libraries.

### 9.3.2.2 Generic installation

For a generic installation, follow next instructions.

#### Step 1: Unpacking the distribution

Unpack the distribution by typing:

```
tar -xzf algol68g-2.0.0.tgz
```

#### Step 2: Configuring

Make the distribution directory the present working directory by typing:

```
cd algol68g-2.0.0
```

In the distribution directory type:

```
./configure
```

This will generate a makefile, which will be used by `make` to build `a68g`. The configure script accepts various options. For an overview of options, type:

```
./configure --help
```

Note that various options for optimisation, debugging and profiling assume you are using `gcc`, `gdb` and `gprof`.

If you are interested in `a68g`'s parallel-clause, you need to build using POSIX threads. To build with POSIX threads, type:

```
./configure --threads
```

**Step 3: Building** The standard makefile will turn warnings off in case `gcc` is used. If you do not use `gcc`, or `make gcc` issue warnings through the `--warnings` option of `configure`, it is possible that your C compiler will issue spurious warnings, see section {9.10}. Now type:

```
make
```

The executable `a68g` will now be in the present working directory, together with a file

## PROGRAMMING WITH ALGOL 68 GENIE

`a68g.h` which is used in optimising mode. To install `a68g` when building as user `root` type:

```
make install
```

to make available the executable `a68g`, an include file `a68g.h` and a manual page so you can type:

```
man a68g
```

This will produce a manual page as presented in Appendix D.

Targets for files `a68g`, `a68g.h` and the manual page are `/usr/local/bin`, `/usr/local/include` and `/usr/local/man` respectively. If latter directory cannot be accessed, `/usr/share/man` and then `/usr/man` are tried. If you have no permission to write to those directories, the target is the present working directory. This is a fail-safe in case you attempt `make install` when you are not `root`.

Installation can be undone by typing:

```
make uninstall
```

Note that in you can supply `CPPFLAGS`, `CFLAGS` and `LDFLAGS` to `make`; for instance type:

```
make CFLAGS="-O"
```

to optimise the code so it will run faster. To remove binaries used for building, thus recovering some disk space, type:

```
make clean
```

To remove all files used for building, type:

```
make distclean
```

### Step 4: Starting Algol 68 Genie

When you install `a68g` in the present working directory, you may want to include the current directory in your `PATH` variable. Otherwise you will need to indicate the current directory by typing `./a68g` as in the following examples. On Linux you can expect:

```
$ ./a68g --version
Algol 68 Genie Version 2.0.0 (released September 2010),
copyright 2001-2010 J. Marcel van der Veer.
Algol 68 Genie is free software covered by the GNU General Public License.
There is ABSOLUTELY NO WARRANTY for Algol 68 Genie.
See the GNU General Public License for more details.
```

```
Compiled on Linux 2.6.29.6-smp i686 with gcc 4.3.3
Configured on 1 September 2010 01:00 with options "-O --threads"
GNU libplot 4.4
```

```
GNU Scientific Library 1.14
PostgreSQL libpq 8.4.2
Alignment 4 bytes
Default frame stack size: 3072 kB
Default expression stack size: 1024 kB
Default heap size: 24576 kB
Default handle pool size: 4096 kB
Default stack overhead: 512 kB
Effective system stack size: 10240 kB
```

If instead of above version statement you get a message as:

```
a68g: error in loading shared libraries
libgsl.so.0: cannot open shared object file:
No such file or directory
```

or:

```
psql: error in loading shared libraries
libpq.so.5.2: cannot open shared object file:
No such file or directory
```

then your shared library search path was not properly set for the GNU Scientific Library or PostgreSQL's libpq library, respectively. The method to set the shared library search path varies between platforms, but the most widely used method is to set the environment variable LD\_LIBRARY\_PATH. In Bourne shells (sh, ksh, bash, zsh) you would use:

```
LD_LIBRARY_PATH=/usr/local/lib:/usr/local/pgsql/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

while in csh or tcsh you would use:

```
setenv LD_LIBRARY_PATH /usr/local/lib:/usr/local/pgsql/lib:$LD_LIBRARY_PATH
```

You should put these commands into a shell start-up file such as

```
/etc/profile
```

or

```
~/.bash_profile.
```

When in doubt, refer to the manual pages of your system.

## 9.4 Synopsis

To start `a68g` from the command-line you would use:

```
a68g [options] filename
```

If `a68g` cannot open `filename`, it will try opening with extensions `.a68`, `.a68g`, `.algol68` or `.algol68g` respectively. Alternatively, under Linux an Algol 68 source file can be made a script by entering:

```
#! [/usr/local/bin/]a68g [option]
```

as the *first* line in the script file. Note that shells as `bash` only pass a single option to `a68g` in a script; other options can be entered on the command line. After making the script executable by typing:

```
chmod +x filename
```

the script can be started from the command line by typing:

```
filename [options]
```

For programs for which you expect a long run time, and that you consider fit to run without many of the runtime checks that Algol 68 Genie offers, you can enable the **unit** compiler by typing:

```
a68g --optimise [options] filename
```

or alternatively,

```
a68g -O [options] filename
```

After successful compilation you will have a file with name `filename` with its extension changed for `.so`. Consider for example the program as listed in section 11.5 that will give you a Whetstone rating, which is a vintage measure of floating-point performance of your platform and programming language:

```
$ ./a68g whetstone.a68
0.38 26.3
```

which means that we needed 0.38 seconds giving a rating of 26.3 Mwhets on this particular platform. Now we try optimisation:

```
$ ./a68g -O whetstone.a68
0.09 116.3
```

showing that the **unit** compiler significantly improves the runtime of this program; this because the Whetstone rating involves many basic operations on primitive modes, which is the forte of the **unit** compiler. If you use the **unit** compiler, and wish to run a same

program another time without recompiling generated C code, you have two options:

1. You can request to skip the code generation and compilation phase by typing:

```
a68g --rerun [options] filename
```

For example:

```
$ ./a68g --rerun whetstone.a68
0.09 116.3
```

Note that `a68g` uses a time stamp to determine whether previously compiled code can still be used. Even when using option `rerun`, a C file will be emitted, since Algol 68 Genie needs to reconstruct the names of the symbols to resolve.

2. You can request `a68g` to build a shell script for your program by typing:

```
a68g --compile [options] filename
```

or, briefly

```
a68g -c [options] filename
```

The shell script will be named `filename` with its extension exchanged for `.sh`. You can in a later stage run this shell script. For example:

```
$ ./a68g -c whetstone.a68
$ ./whetstone.sh
0.09 116.3
```

If you inspect the shell script you will find little intelligible information. The script consists of a call to `a68g` followed by a compressed source code fork and compressed binary fork. Thus the shell script can place diagnostics in the original source code, also when the original source file is no longer available. The shell script checks whether it executes the same `a68g` version that was used to build it. The shell script is a pseudo-executable; it is independent of the original source file, but still depends on `a68g`.

On Linux, Algol 68 Genie recognises below environment variables. If they are undefined, `a68g` assumes default values for them:

`A68G_STANDIN`

The value will be the name of the file to which `stand in` will be redirected.

`A68G_STANDOUT`

The value will be the name of the file to which `stand out` will be redirected.

`A68G_STANDERROR`

The value will be the name of the file to which `stand error` will be redirected.

## A68G\_OPTIONS

The value will be tokenised and processed as options. These options will supersede options set in file `.a68grc` (vide infra).

## 9.5 a68g diagnostics

This section discusses how `a68g` specifies its diagnostics. Algol 68 Genie checks for many events and since some diagnostics are synthesized it is not possible to provide a finite list of `a68g`'s diagnostics. Diagnostics would typically be presented to you like this:

```
7          REF [] INT q2 = p[3,];
              1      2
a68g: warning: 1: tag "q2" is not used (detected in VOID closed-clause
starting at "BEGIN" in line 1).
a68g: error: 2: attempt at storing a transient name (detected in VOID
closed-clause starting at "BEGIN" in line 1).
```

If possible, `a68g` writes the offending line and indicates by means of single-digit markers the positions where diagnostics have been issued. Note that the source of the error will often only be *near* that marker. Then a list follows of diagnostics issued for that line. As is usual in the GNU/Linux world, the first element of a diagnostic is the name of the program giving the diagnostic, in casu `a68g`. If the error occurs in an included source file, the included source file's name will be the next element. If the source is contained in a single file then the source file name is not stated. Then follows the marker that indicates the position in the line where the diagnostic was issued. Finally a descriptive message is written.

In case a diagnostic is not related to a specific source line, the diagnostic looks simpler since a source line and markers can be omitted, for instance:

```
$ a68g --precision=1k --heap=256G examples/mc.a68
a68g: error in option "--heap=256g" (numerical result out of range).
```

which on a 32-bit platform suggests you probably wanted to specify `--heap=256M`.

Diagnostics come in different levels of severity. A *warning* will not impede execution of a **program**, but it will draw your attention to some issue; for instance:

```
106          ABS diff < 100 ANDF exp (- diff) > random
              1
a68g: warning: 1: construct is an extension.
```

indicates that this phrase contains an extension (here, `ANDF`) so this **program** may not be portable. Another example demonstrates that `a68g` will point out that an **identifier**

or **indicant** can hide a previous **declaration** of the same, which may lead to unexpected results:

```
$ ./a68g hidden --warnings
1      INT term := 0, REAL pi := 355 / 113, k := 0;
      1          2
a68g: warning: 1: declaration hides prelude declaration of
PROC (REF FILE) STRING "term".
a68g: warning: 2: declaration hides prelude declaration of REAL "pi".
2      IF REAL k := pi * term;
      1
a68g: warning: 1: declaration hides a declaration of "k" with larger reach.
```

Since a warning does not impede execution, a68g suppresses many of them unless you supply the `--warnings` option at the command line. Some warnings cannot be suppressed, for instance when a construct is most likely not what you intended, like implicitly voiding a **formula**.

An *error* impedes successful compilation and execution of a **program**. Some condition was encountered that you must fix before execution could take place. For instance:

```
2      IN (UNION (INT, BOOL)): SKIP,
      1
a68g: error: 1: UNION (BOOL, INT) is neither component nor subset of
UNION (CHAR, BOOL) (detected in united-case-clause starting at "CASE"
in line 1).
```

or:

```
38      matvec (1, 10, bool, ca, aa);
      1
a68g: error: 1: REF BOOL cannot be coerced to INT in a
strong-argument (detected in closed-clause starting at "("
in line 3).
```

obviously are errors that need fixing before successful compilation and execution.

A syntax error is given when an error has been made in syntax according to chapter 8. Such diagnostic is typically issued by the parser. In case of a syntax error, a68g tries to give additional information in an attempt to help you localising and diagnosing the exact error. For instance, in:

```
103      CASE v
      1
a68g: syntax error: 1: incorrect parenthesis nesting; encountered
end-symbol in line 152 but expected esac-symbol; check for "END"
without matching "BEGIN" and "CASE" without matching "ESAC".
```

## PROGRAMMING WITH ALGOL 68 GENIE

a **case-clause** was terminated with `END` so `a68g` straightforwardly suggests to check for a missing `BEGIN` or a missing or misspelled `ESAC`. In next example:

```
317   OP * = (VEC, v REAL r) VEC: r * v;  
      1
```

`a68g`: syntax error: 1: construct beginning with "(" in line 317 followed by a declarer and then ",", "v", a parameter-list, ")" is not a valid parameter-pack.

the parser explains that it wanted to parse a parameter-pack but it could not complete that since a **declarer** was followed by a **comma-symbol**. Clearly `VEC, r` should have been written `VEC r, .` These synthetic explanations can get quite verbose, as in:

```
385   proc set pixel = ( ICON i, PT p, COLOR c ) VOID:  
      1
```

`a68g`: syntax error: 1: construct beginning with a serial-clause starting in line 5 followed by "=" in line 385 and then a routine-text, ";" in line 386, "procsetpixelrgb" in line 387, "=", a routine-text, ";" in line 388 et cetera is not a valid serial-clause.

which indicates that things looked all-right up to line 385, but then a **serial-clause** was equated to a routine-text which is odd. You will be quick to note that a missing `PROC` symbol was incorrectly spelled in lower-case, and the same will have happened in line 387.

When your **program** is being executed, a *runtime error* can occur indicating that some condition arose that made continuation of execution either impossible or undesirable. A typical example is:

```
8       CASE CASE n  
      1
```

`a68g`: runtime error: 1: REF [] INT value from united value is exported out of its scope (detected in VOID united-case-clause starting at "CASE" in this line).

but if any information can be obtained from runtime support about *what* went wrong, it will be included in the diagnostic as in:

```
41       get (standin, (x, space, y, space, number, new line));  
      1
```

`a68g`: runtime error: 1: cannot open ".txt" for getting (no such file or directory) (detected in VOID loop-clause starting at "TO" in line 38).

which means that the file name that you specified with `open` does not exist now that you decided to read from it — it indeed seems you only wrote an extension. Note that you can intercept a runtime error by specifying option `--monitor` or `--debug` by which the monitor will be entered when a runtime error occurs, so you can inspect what went wrong {see 9.7}.



a68g limits the number of diagnostics it issues, since it is believed that there is no point in generating many diagnostics that might be related to an earlier one. Since a68g compiles in several passes, each traversing the source from begin to end, it is possible that a diagnostic announcing suppression of further ones appears in the middle of other diagnostics. Also, to make diagnostics more intelligible, a68g attempts to substitute modes for indicants that are declared for them. Therefore you could expect for a single **program**:

```

38      HEAP CALL := (name, parameter);
          1
a68g: warning: 1: REF FUNCTION value from identifier could be
exported out of its scope (detected in CALL collateral-clause
starting at "(" in this line).
41      HEAP FUNCTION := (bound var, body);
          1
a68g: warning: 1: further warning diagnostics suppressed
(detected in FUNCTION collateral-clause starting at "(" in this
line).
199          is const (f)
          1
a68g: warning: 1: construct is an extension.
208          THEN IF is const (f)
          1
a68g: warning: 1: construct is an extension.
217          THEN make dyadic (g, times, f);
          1
a68g: warning: 1: skipped superfluous semi-symbol.

```

## 9.6 Options

Options are passed to a68g either from the file `.a68grc` in the working directory, the environment variable:

```
A68G_OPTIONS
```

or the command-line, or through pragmat. Precedence is as follows: pragmat supersede command-line options, command-line options supersede options from:

```
A68G_OPTIONS
```

that supersede options in:

```
.a68grc
```

Options have syntax `-option [[=] value[suffix]]`

Option names are not case sensitive, but option arguments are. Note that option syntax is a superset of Linux standards since a68g was initially written on other operating systems

than Linux and had to maintain a degree of compatibility with option syntax of all of them. Listing options, tracing options and `--pragmat`, `--no-pragmat`, take their effect when they are encountered in a left-to-right pass of the **program** text, and can thus be used to generate a cross reference for a particular part of the user **program**. In integral arguments, suffix *k*, *M* or *G* is accepted to specify multiplication by  $2^{10}$ ,  $2^{20}$  or  $2^{30}$  respectively. The suffix is case-insensitive.

### 9.6.1 One-liners

- `--print unit | -p unit`

Prints the value yielded by Algol 68 **unit** unit. In this way `a68g` can execute one-liners from the command-line. The one-liner is written in file `.a68g.x` in the working directory.

Example: `a68g -p "sqrt (2 * pi)"`

- `--execute unit | -e unit`

Executes Algol 68 **unit** unit. In this way `a68g` can execute one-liners from the command-line. The one-liner is written in file `.a68g.x` in the working directory.

Example: `a68g -e "printf (($lh$, 4 * atan (-1)))"`

### 9.6.2 Memory size

Algol 68 relieves the programmer from managing memory, hence `a68g` manages allocation and de-allocation of heap space. However, if memory size were not bounded, `a68g` might claim all available memory before the garbage collector is called. Since this could impede overall system performance, memory size is bounded.

- `--heap number`

Sets heap size to `number` bytes. This is the size of the block that will actually hold data, not handles. At runtime `a68g` will use the heap to store temporary arrays, so even a **program** that has no heap **generators** requires heap space, and can invoke the garbage collector.

Example: `PR heap=32M PR`

- `--handles number`

Sets handle space size to `number` bytes. This is the size of the block that holds handles that point to data in the heap. A reference to the heap does not point to the heap, but to a handle as to make it possible for the garbage collector to compact the heap.

Example: `PR handles=2M PR`

- `--frame number`

Sets frame stack size to `number` bytes. A deeply recursive **program** with many local variables may require a large frame stack space.

Example: `PR frame=1M PR`

- `--stack number`

Sets expression stack size to `number` bytes. A deeply recursive **program** may require a large expression stack space.

Example: `PR stack=512k PR`

- `--overhead number`

Sets the overhead, which is a safety margin, for the expression stack and the frame stack to `number` bytes. Since stacks grow by relatively small amounts at a time, Algol 68 Genie checks stack sizes only where recursion may set in. It is checked whether stacks have grown into the overhead. For example, if the frame stack size is 512 kB and the overhead is set to 128 kB, Algol 68 Genie will signal an imminent stack overflow when the frame stack pointer exceeds  $512 - 128 = 384$  kB. When the overhead is set to a too small value, a segment violation may occur.

Example: `PR overhead=64k PR`

### 9.6.3 Listing options

- `--extensive`

Generates an extensive listing, including source listing, syntax tree, cross reference, generated C code et cetera. This listing can be quite bulky.

- `--listing`

Generates concise listing.

- `--moids`

Generates overview of moids in listing file.

- `--prelude-listing`

Generates a listing of preludes.

- `--object, --no-object`

Switches listing of object C code in listing file. This requires `--optimise`.

## PROGRAMMING WITH ALGOL 68 GENIE

- `--source, --no-source`

Switches listing of source lines in listing file.

- `--statistics`

Generates statistics in listing file.

- `--tree, --no-tree`

Generates syntax tree listing in listing file. This option can make the listing file bulky, so use considerately.

- `--unused`

Generates an overview of unused tags in the listing file.

- `--xref, --no-xref`

Switches generating a cross reference in the listing file.

### 9.6.4 Interpreter options

- `--optimise, --no-optimise`

Enables, or disables, the **unit** compiler which emits C code for many **units**, compiles and dynamically links this code before it is executed by Algol 68 Genie. This option omits many runtime checks. This option only works on Linux and compatible systems.

- `--compile, --no-compile`

Enables, or disables, generation of a shell script for your program. The shell script will have the name `filename` stripped from its extension, padded with `.sh`. The shell script stores the source code and the generated shared library, in a compressed manner. You can in a later stage run this shell script. This option invokes `--optimise` and only works on Linux and compatible systems.

- `--rerun`

Run using the shared library generated by a previous run. See also option `--optimise`.

- `--monitor | --debug`

Start the **program** in the monitor. You will have to start **program** execution manually. Also, upon encountering a runtime error, instead of terminating execution, the monitor will be entered.

- `--assertions, --no-assertions`

Switches elaboration of assertions.

- `--backtrace`, `--no-backtrace`

Switches stack back tracing in case of a runtime error.

- `--breakpoint`, `--no-breakpoint`

Switches setting of breakpoints on following **program** lines.

- `--trace`, `--no-trace`

Switches tracing of a **program**.

- `--precision` number

Sets precision for `LONG LONG` modes to `number` significant digits. The precision cannot be less than the precision of `LONG` modes. Algorithms for extended precision in `a68g` are not really suited for precisions larger than about a thousand digits. State of the art in the field offers more efficient algorithms than implemented here.

- `--time-limit` number

On Linux, interrupts Algol 68 Genie after `number` seconds; a *time limit exceeded* runtime error will be given. This can be useful to detect endless loops. Trivial example:

```
$ a68g --time-limit=1 -e "DO SKIP OD"
1      (DO SKIP OD)
      1
a68g: runtime error: 1: time limit exceeded (detected in
VOID loop-clause starting at "DO" in this line).
```

## 9.6.5 Miscellaneous options

- `--apropos`, `--help`, `--info` [ `string` ]

Prints info on options if `string` is omitted, or prints info on `string` otherwise.

- `--upper-stropping`, `--quote-stropping`

Sets the stropping regime. Upper stropping is the default. Quote stropping is implemented to let `a68g` scan vintage source code.

- `--brackets`

Enables non-traditional use of brackets. This option makes `...` and `[ ... ]` equivalent to `( ... )` for the parser.

- `--check` | `--no-run`

## PROGRAMMING WITH ALGOL 68 GENIE

Check syntax only, interpreter does not start.

- `--run`

Override `--norun`.

- `--exit | --`

Ignore further options from command line or current pragmat. This option is usually provided under Linux to allow source file names to have a name beginning with a minus-sign.

- `--file string | -f string`

Accept argument `string` as generic filename.

- `--pedantic`

Equivalent to `--warning --portcheck`.

- `--portcheck, --no-portcheck`

Enable or suppress portability warning messages.

- `--strict`

Ignores most a68g extensions to Algol 68 syntax as described by the Revised Report. This option implies `--portcheck`, and a68g will not reserve words `ANDF`, `ANDTH`, `CLASS`, `CODE`, `COL`, `DIAG`, `DOWNT0`, `EDOC`, `ELSF`, `ENVIRON`, `NEW`, `OREL`, `ORF`, `ROW`, `THEF`, `TRNSP` and `UNTIL`.

- `--warnings, --no-warnings`

Enable warning messages or suppresses suppressible warning messages.

- `--pragmats, --no-pragmats`

Switches elaboration of pragmat items. When disabled, pragmat items are ignored, except for option `PRAGMATS`.

- `--reductions`

Prints reductions made by the parser. This option was originally meant for parser debugging but can be quite instructive (and very verbose).

- `--verbose`

Informs on actions.

- `--version`

States the version of the running copy of a68g.

## 9.7 The monitor

The `a68g` interpreter has a monitor for basic debugging of a running **program**. The monitor is invoked when `a68g` receives signal `SIGINT` (which usually comes from typing `CTRL-C`), when standard procedures `debug` or `break` are called, or when a runtime error occurs while option `monitor` or `debug` is in effect. Breakpoints can be indicated a priori through **pragmats** as well.

Following shows an example where the monitor is invoked by typing `CTRL-C`:

```
$ a68g buggy
^C
121  WHILE n ~= 0 DO
      —
Terminate a68g (yes|no): n
This is the main thread
(a68g)
```

The prompt `(a68g)` tells you that the monitor is awaiting a command. Next section describes the commands that the monitor can process.

### 9.7.1 Monitor commands

As with options, monitor command names are not case sensitive, but arguments are. In the list below, upper-case letters denote letters mandatory for recognition. Currently, the monitor recognises next commands:

- `APropos, Help, INfo [ string ]`

Prints info on monitor commands if `string` is omitted, or prints info on `string` otherwise.

- `Breakpoint n [ IF expression ]`

Set breakpoints on **units** in line `n`. If you supply an expression then interruption will only take place if this expression yields `TRUE`. If the expression is not correct, it will be deleted. Algol 68 Genie only breaks in serial-, collateral- and enquiry-clauses, in unit-lists in in-parts of integer- or united-case-clauses, and sources in **declarations**. Note that a breakpoint expression is a nice debugging tool, but it slows down execution a lot.

- `Breakpoint Watch [ expression ]`

Sets the watchpoint expression. Interruption will take place whenever the expression yields `TRUE`. Algol 68 Genie only breaks in serial-, collateral- and enquiry-clauses, in unit-lists in in-parts of integer- or united-case-clauses, and sources in **declarations**. If the expression is not correct, it will be deleted. Note that a watchpoint expression is a nice debugging tool, but it slows down execution a lot.

## PROGRAMMING WITH ALGOL 68 GENIE

- Breakpoint n Clear

Clears breakpoints on **units** in line n.

- Breakpoint [ List ]

Lists all breakpoints and the watchpoint expression.

- Breakpoint Clear [ ALL ]

Clears all breakpoints and the watchpoint expression.

- Breakpoint Clear Breakpoints

Clears all breakpoints.

- Breakpoint Clear Watchpoint

Clears the watchpoint expression.

- BT n

Print n frames in the stack following the dynamic link (Back Trace) (default n = 3).

- CALLs n

Print n frames in the **call** stack (default n = 3).

- Continue, Resume

Continue execution.

- DO command, EXEC command

Pass command to the shell and print the return code.

- Evaluate expression, X expression

Evaluate expression and show its result. For a description of monitor expressions see the next section.

- EXamine n

Print value of all **identifiers** named n in the **call** stack.

- EXIt, HX, Quit

Terminates a68g.

- Frame n

Select stack frame n as the current stack frame. If n is not specified, print the current stack frame. If 0 is specified, the current stack frame will be the top of the frame stack.



- HEAp *n*

Print contents of the heap with address not greater than *n*.

- HT

Halts typing to standard output.

- LINK *n*

Print *n* frames in the stack, following the static link (default  $n = 3$ ). This will give generally give a shorter back trace than `stack` since the static link links to the frame embedding (a) the actual lexical level or (b) the incarnations of the active procedure, while the dynamic link just points at the previous lexical level (and thus walks through all incarnations of a recursive procedure and all ranges of clauses).

- RERun, REStart

Restarts the Algol 68 **program** without resetting breakpoints.

- RESET

Restarts the Algol 68 **program** and resets breakpoints.

- RT

Resumes typing to standard output.

- List *n m*

If *m* is omitted, show *n* lines around the interrupted line (default  $n = 10$ ). If *n* and *m* are supplied, show lines *n* up to *m*.

- Next

Resume execution until the next **unit** (that can be a breakpoint) is reached; do not enter routine-texts.

- STEp

Resume execution until the next **unit** (that can be a breakpoint) is reached.

- FINish, OUT

Resume execution until the next **unit** (that can be a breakpoint) is reached, *after* the current procedure incarnation will have finished. You typically use this to leave a procedure and to stop in the caller.

- Until *n*

Resume execution until the first **unit** (that can be a breakpoint) is reached, on line number

## PROGRAMMING WITH ALGOL 68 GENIE

n.

- PROMpt n

Set prompt to n. Default prompt is (a68g).

- SIZES

Show sizes of various memory segments.

- STACK n

Print n frames in the stack following the dynamic link (default n = 3).

- WHERE

Show the line where interruption took place.

- XREF n

Give detailed information on source line n.

### 9.7.2 Monitor expressions

Monitor expressions provide basic means to do arithmetic in the monitor, and to change stored values. Monitor expression syntax is similar to Algol 68 syntax. Important points are:

1. Expressions, **assignment** sources, and **assignment** destinations, are strictly evaluated from left to right.
2. Actual assignment is done from right to left:

```
(a68g) x a[1] := 0
(REF INT) refers to heap (892000)
(INT) +0
(a68g) x a[2] := a[1] := 1
(REF INT) refers to heap (892016)
(INT) +1
```

Note that the result of the evaluation of an expression is preceded by the mode of that result.

3. Only procedures and operators from standard environ can be called, and operator priorities are taken from standard environ.
4. **Operands** are **denotations**, **identifiers**, **closed-clauses**, **calls** and **slices**. **Casts** are also **operands** but can only be made to mode [LONG] [LONG] REAL or to a (multiple) reference to an indicant:

```
(a68g) x REAL (1)
(REAL) +1.000000000000000E +0
(a68g) x REF TREE (k) IS NIL
(BOOL) F
(a68g) x REF TREE (k) := NIL
(REF REF STRUCT (INT f, REF SELF next)) refers to heap (600000)
(REF STRUCT (INT f, REF SELF next)) NIL
```

5. Slicing does not support **trimmers**. Only [ and ] are allowed as brackets for **indexers**.

6. **Selections** are allowed, but not multiple **selections**:

```
(a68g) x im OF z
(REF LONG REAL) refers to frame (80)
(LONG REAL) +3.1415926535897932384626433832795029
```

7. Dereferencing is the only coercion in arithmetic. Names can be compared through IS and ISNT, and NIL is supplied, but there is no soft coercion of **operands**.

## 9.8 The Preprocessor

a68g has a basic preprocessor. Currently, the preprocessor supports these features:

1. concatenation of lines,
2. inclusion of files,
3. **refinement** preprocessor,
4. switching the preprocessor on or off.

### 9.8.1 Concatenation of lines

Concatenation of lines is similar to what the C preprocessor does. Any line that ends in a backslash ('\') will be concatenated with the line following it. For example:

```
STRING s := "spanning two \
lines"
```

will become:

```
STRING s := "spanning two lines"
```

Using a backslash as an escape character causes no interference with Algol 68 source text since when using upper stropping, a backslash is an unworthy character; when using quote stropping a backslash is a times-ten-symbol, but a real-denotation cannot span end-of-line hence no interference occurs.

Note that when you make use of line concatenation, diagnostics in a concatenated line may be placed in an earlier source line than its actual line in the original source text. In order to preserve line numbering as much as possible, a line is emptied but not deleted if it is joined with a preceding one. This shows in the listing file as emptied lines.

### 9.8.2 Inclusion of files

a68g supports inclusion of other files in an Algol 68 source text. The inclusion directive reads:

```
PR read "filename" PR
```

or:

```
PR include "filename" PR
```

The file with name filename is inserted textually before the line that holds the file inclusion directive. In this way tokens remain in their original lines, which will give more accurate placement of diagnostics. It is therefore recommended that a file inclusion directive be the only text on the line it is in. A file will only be inserted once, on attempted multiple inclusion the file is ignored. Attempted multiple inclusion may for example result from specifying, in an included file, an inclusion directive for an already included file.

### 9.8.3 The refinement preprocessor

Algol 68 Genie is equipped with a basic **refinement** preprocessor, which allows programming through stepwise **refinement** as taught in [Koster 1978, 1981]. A similar preprocessor was actually used in computer science classes at the University of Nijmegen as a front-end for FLACC. The idea is to facilitate **program** construction by elaborating the description of the solution to a problem as ever more detailed steps until the description of the solution is complete. (See also Wirth's well known lecture).

**Refinement** syntax can be found in the syntax summary. **Refinements** cannot be recursive, nor can their definitions be nested. Also, refinement definitions must be unique, and a **refinement** can only be applied once (**refinements** are not procedures). a68g will check whether a **program** looks like a stepwise refined **program**. The **refinement** preprocessor is transparent to **programs** that are not stepwise refined. An example of using **refinements** is in 11.1.

### 9.8.4 Switching the preprocessor on or off

It is possible to switch the preprocessor on or off. The preprocessor is per default switched on. If switched off, it will no longer process preprocessor items embedded in pragmas, except for switching the preprocessor on again. Concatenation of lines takes place even if the preprocessor is switched off.

```
PR preprocessor PR
```

Switches the preprocessor on if it is switched off.

```
PR nopreprocessor PR
```

Switches the preprocessor off if it is switched on.

## 9.9 A multi-pass scheme to parse Algol 68

This section explains, without going into all detail, how `a68g` executes a **program**. Algol 68 Genie employs a multi-pass scheme to parse Algol 68 [Lindsey 1993]:

1. The *tokeniser*. The source file is tokenised, and if needed a **refinement** preprocessor elaborates a stepwise refined **program**. The result is a linear list of tokens that is input for the parser, that will transform the linear list into a syntax tree. `a68g` tokenises all symbols before the parser is invoked. This means that scanning does not use information from the parser. The scanner does some rudimentary parsing: **format-texts** can have **enclosed-clauses** in them, so information is recorded in a stack as to know what is being scanned. Also, the **refinement** preprocessor implements a (trivial) grammar.
2. The *parser*. First, parentheses are checked to see whether they match. Then a top-down parser determines the basic-block structure of the **program** so symbol tables can be set up that the bottom-up parser will consult as you can define things before they are applied. After that the bottom-up parser parses without knowing about modes while parsing and reducing. It can therefore not exchange `[ ... ]` with `( ... )` as is allowed by the Revised Report. This is solved by treating **calls** and **slices** as equivalent for the moment and letting the mode checker sort it out later. This is a Mailloux-type parser, in the sense that it scans a range for **declarations** — **identifiers**, **operator-symbols** and operator priorities — before it starts parsing, and thus allows for tags to be applied before they are declared.
3. The *mode checker*. The modes in the **program** are collected. Derived modes are calculated. Well-formedness is checked and structural equivalence is resolved. Then the modes of constructs are checked and coercions are inserted.

4. The *static-scope checker*. This pass checks whether you export names out of their scopes. Some cases cannot be detected by a static-scope checker, therefore Algol 68 Genie applies dynamic-scope checking.
5. The *unit compiler*. This is an optional phase. The unit compiler emits C code for many **units** and has this code compiled by `gcc`, after which the dynamic linker loader will make this code available to Algol 68 Genie.
6. The *interpreter*. The interpreter executes the syntax tree that results from the previous passes.

## 9.10 a68g source code checks

During development, each source file for a68g is checked and compiled using next commands:

```
$ sparse
$ cppcheck {-}{-}all
$ splint $f -weak +ret-val-int +posixlib -abstract -unrecog
  -I/usr/include -I/usr/local/include -I/usr/local/pgsql/include
  -DENABLE_GRAPHICS -DENABLE_NUMERICAL -DENABLE_CURSES
  -DENABLE_TERMINFO -DENABLE_DIRENT -DENABLE_HTTP
  -DENABLE_REGEX -DENABLE_POSTGRESOL -DENABLE_IEEE_754
  -DENABLE_PAR_CLAUSE
$ gcc $f -S -pedantic -W -Wall -Wshadow -Wconversion -Wstrict-prototypes
  -fno-common -O2 -I/usr/include -I/usr/local/include
  -I/usr/local/pgsql/include -DENABLE_GRAPHICS -DENABLE_NUMERICAL
  -DENABLE_CURSES -DENABLE_TERMINFO -DENABLE_DIRENT -DENABLE_HTTP
  -DENABLE_REGEX -DENABLE_POSTGRESOL -DENABLE_IEEE_754
  -DENABLE_PAR_CLAUSE -o /dev/null
```

The objective is to have no diagnostics except those that cannot be avoided using these tools, on a 32 bit platform. Some unavoidable warnings come from include files outside the a68g project. An unavoidable `gcc` diagnostic in a68g is:

For some reason `gcc` ignores that indicated variables are declared `volatile`, therefore no risk of clobbering exists, and such warning message is ignored. Another unavoidable `gcc` diagnostic in a68g is:

```
warning: conversion to 'short unsigned int' from 'int' may alter its value
```

which is caused by the option `-Wconversion` from `gcc` that has the quirk to issue a warning in case a function parameter would undergo a conversion in case no function prototype would be present — there is no way to avoid such warning for functions taking a `short unsigned` parameter. Yet another unavoidable warning is:

**warning: dereferencing type-punned pointer will break strict-aliasing rules**

which is caused by the dynamic linker load library `libdl` which insists on using `void *` pointers for function pointers, which is undefined by the C standard. `gcc` produces this warning even though `a68g` uses the POSIX.1-2003 (Technical Corrigendum 1) workaround. An unavoidable diagnostic from `cppcheck` in `a68g` is:

**(error) Memory leak: matches**

where `cppcheck` has missed that such function returns under condition that `matches == NULL`. As you know, nowadays `free` will safely ignore a `NULL` pointer, but it still makes no sense to `free` a `NULL` pointer and therefore such diagnostic is ignored.

## 9.11 Limitations and bugs

Next `a68g` issues are known:

1. Algol 68 Genie offers optional checking of the system stack. When this check is not activated, or on systems where this check would not work, the following may result in a segment violation (and possibly a core dump):
  - (a) Deep recursion or garbage collection of deeply recursive data structures.
  - (b) Using **jumps** to move between incarnations of recursive procedures.
2. When the stack overhead {9.6.2} is set to a too small value, a segment violation may occur.
3. Algorithms for extended precision (`LONG LONG` arithmetic modes) are not really suited for precisions larger than about a thousand digits. State of the art in the field offers more efficient algorithms than implemented here.
4. Overflow- and underflow checks on `REAL` and `COMPLEX` operations require IEEE-754 compatibility. Many processor types, notably `ix86`'s and `PowerPC` processors, are IEEE-754 compatible.
5. A garbage collector cannot solve all memory allocation issues. It is therefore possible to get an unexpected "out of memory" diagnostic. Two options in such case are:
  - (a) Increase heap size.
  - (b) Call standard-prelude routine "sweep heap" or "pre emptive sweep" at strategic positions in the **program**.

## PROGRAMMING WITH ALGOL 68 GENIE

6. There are some `libplot` related issues. Algol 68 Genie cannot work around these problems in `libplot`:
  - (a) In some versions of `libplot`, pseudo-gif plotters produce garbled graphics when more than 256 different colours are specified.
  - (b) Some platforms cannot give proper `libplot` support for all plotter types. Linux with the X window system lets `libplot` implement all plotters but for example the Win32 executable provided for `a68g` will give runtime errors as X plotter missing or postscript plotter missing due to an incomplete `libplot` library for that platform. On other platforms it is possible that a plotter produces garbage or gives a message as output stream jammed.
  - (c) In some versions of `libplot`, X plotters do not flush after every plotting operation. It may happen that a plotting operation does not show until `close` is called for that plotter (after closing, an X plotter window stays on the screen (as a forked process) until you type "q" while it has focus, or click in it).



# Chapter 10

## Standard-prelude and library-prelude

### 10.1 The standard-environ

An Algol 68 **program** run with a68g is embedded in the next environ:

```
BEGIN
  COMMENT
  Here the standard-prelude and library-prelude are included.
  COMMENT
  PR include "prelude" PR
  BEGIN
    MODE DOUBLE = LONG REAL;
    start: commence:
    BEGIN COMMENT
      Here your program is embedded.
      COMMENT
      PR include "program" PR
    END;
    stop: abort: halt: SKIP
  END
END
```

At line 5 the prelude is included. At line 12 your code is included. A consequence of this standard-environ is that an a68g **program** does not need to be an **enclosed-clause**; a **serial-clause** suffices.

## 10.2 The standard-prelude

Next sections up to and including the section on transput describe the facilities in the standard-prelude supplied with a68g.

## 10.3 Standard modes

Many of the modes available in the standard-prelude are built from the standard modes of the language which are all defined in the Revised Report:

1. VOID

This mode has one value: `EMPTY`. It is used as the yield of routines, in **casts** and in unions.

2. INT

In a68g, these precisions are available:

(a) INT

(b) LONG INT

(c) LONG LONG INT

3. REAL

In a68g, these precisions are available:

(a) REAL

(b) LONG REAL

(c) LONG LONG REAL

4. BOOL

This mode has two values, `TRUE` and `FALSE`.

5. CHAR

This mode is used for most character operations.

6. STRING

This mode is defined as:

```
MODE STRING = FLEX [1 : 0] CHAR
```

7. COMPLEX, COMPL

This is not a primitive mode because it is a structure with two fields:

```
MODE COMPLEX = STRUCT (REAL re, im)
```

However, the widening coercion will convert a REAL value into a COMPLEX value, and transput routines will not straighten a COMPLEX - or REF COMPLEX value. Like REALS, the following precisions are available in a68g:

- (a) COMPLEX
- (b) LONG COMPLEX
- (c) LONG LONG COMPLEX

## 8. BITS

This mode is equivalent to a computer word regarded as a group of bits (binary digits) numbered 1 to bits width. These precisions are available in a68g :

- (a) BITS
- (b) LONG BITS
- (c) LONG LONG BITS

## 9. BYTES

This mode stores a row-of-character in a single value. These precisions are available in a68g:

- (a) BYTES
- (b) LONG BYTES

## 10. SEMA

Semaphores are used to synchronise parallel actions. This mode is defined as:

```
MODE SEMA = STRUCT (REF INT F)
```

Note that the field cannot be directly selected.

## 11. CHANNEL

Channels describe the properties of a FILE.

## 12. FILE

A FILE structure holds status of transputting to or from a specific stream of bytes.

## 13. FORMAT

Holds an internal representation of **format-texts** and their elaboration.

## 14. PIPE

Pipes describe software pipelines with which an output stream can be directly connected to an input stream. This is a Unix feature that is available under Linux.

## 15. SOUND

Sounds hold sound values which can be manipulated with a68g.

## 10.4 Environment enquiries

Algol 68 was the first programming language to contain **declarations** which enable a programmer to determine the characteristics of the implementation. The enquiries are divided into a number of different groups.

### 10.4.1 Enquiries about precisions

Any number of `LONG` or `SHORT` can be given in the mode specification of numbers, but only a few such modes are distinguishable in any implementation. The following environment enquiries tell which modes are distinguishable:

1. `INT int lengths`  
1+ the number of extra lengths of integers.
2. `INT int shorths`  
1+ the number of short lengths of integers.
3. `INT real lengths`  
1+ the number of extra lengths of real numbers.
4. `INT real shorths`  
1+ the number of short lengths of real numbers.
5. `INT bits lengths`  
1+ the number of extra lengths of `BITS`.
6. `INT bits shorths`  
1+ the number of short lengths of `BITS`.
7. `INT bytes lengths`  
1+ the number of extra lengths of `BYTES`.
8. `INT bytes shorths`  
1+ the number of short lengths of `BYTES`.

### 10.4.2 Characteristics of modes

1. `INT max int`  
The maximum value of mode `INT`.
2. `LONG INT long max int`  
The maximum value of mode `LONG INT`.

## STANDARD-PRELUDE AND LIBRARY-PRELUDE

3. LONG LONG INT long long max int  
The maximum value of mode LONG INT.
4. REAL max real  
The largest real value.
5. REAL min real  
The smallest real value.
6. REAL small real  
The smallest real which, when added to 1.0, gives a sum larger than 1.0.
7. LONG REAL long max real  
The largest long real value.
8. LONG REAL long min real  
The smallest long real value.
9. LONG REAL long small real  
The smallest long real which, when added to 1.0, gives a sum larger than 1.0.
10. LONG LONG REAL long long max real  
The largest long long real value.
11. LONG LONG REAL long long min real  
The smallest long long real value.
12. LONG LONG REAL long long small real  
The smallest long long real which, when added to 1.0, gives a sum larger than 1.0.
13. INT int width  
The maximum number of decimal digits expressible by an integer.
14. INT long int width  
The maximum number of decimal digits expressible by a long integer.
15. INT long long int width  
The maximum number of decimal digits expressible by a long long integer.
16. INT bits width  
The number of bits required to hold a value of mode BITS.
17. INT long bits width  
The number of bits required to hold a value of mode LONG BITS.
18. INT long long bits width  
The number of bits required to hold a value of mode LONG LONG BITS.
19. INT bytes width  
The number of bytes required to hold a value of mode BYTES.

## PROGRAMMING WITH ALGOL 68 GENIE

20. INT long bytes width  
The number of bytes required to hold a value of mode LONG BYTES.
21. INT real width  
The maximum number of significant decimal digits in a real.
22. INT exp width  
The maximum number of decimal digits in the exponent of a real.
23. INT long real width  
The maximum number of significant decimal digits in a long real.
24. INT long exp width  
The maximum number of decimal digits in the exponent of a long real.
25. INT long long real width  
The maximum number of significant decimal digits in a long real.
26. INT long long exp width  
The maximum number of decimal digits in the exponent of a long long real.

### 10.4.3 Mathematical constants

1. REAL pi  
The value 3.14159265358979.
2. LONG REAL long pi  
The value 3.1415926535897932384626433833.
3. LONG LONG REAL long long pi  
The value 3.14159265358979323846264338327950288419716939937510582097494459  
at default precision.

The long values of  $\pi$  are calculated using an AGM due to Borwein and Borwein<sup>1</sup>:

---

<sup>1</sup>J.M. Borwein and P.B. Borwein. *Pi and the AGM: A study in analytical number theory and computational complexity*. Wiley [1987].

$$\begin{aligned}
 x_0 &= \sqrt{2} & x_{i+1} &= \frac{1}{2} \left( \sqrt{x_i} + \frac{1}{\sqrt{x_i}} \right) \\
 \pi_0 &= 2 + \sqrt{2} & \pi_{i+1} &= \pi_i \left( \frac{1 + x_i}{1 + y_i} \right) \\
 y_0 &= \sqrt{\sqrt{2}} & y_{i+1} &= \frac{y_i \sqrt{x_i} + \frac{1}{\sqrt{x_i}}}{1 + y_i}
 \end{aligned}$$

The number  $\pi$  equals the limit  $\pi_\infty$ .

### 10.4.4 Character set enquiries

The absolute value of Algol 68 characters range from 0 to the value of `max abs char`. Furthermore, the operator `REPR` will convert any `INT` up to `max abs char` to a character. What character is represented by `REPR 225` will depend on the character set used by the displaying device.

1. `INT max abs char`  
The largest positive integer which can be represented as a character.
2. `CHAR null character`  
This is `REPR 0`.
3. `CHAR blank`  
This is the space character.
4. `CHAR error char`  
This character is used by the formatting routines for invalid values.
5. `CHAR flip`  
This character is used to represent `TRUE` in `transput`.
6. `CHAR flop`  
This character is used to represent `FALSE` in `transput`.

## 10.5 Standard operators

The number of distinct operators is vastly increased by the availability of `SHORT` and `LONG` modes. Thus it is imperative that some kind of shorthand be used to describe the operators. Following the subsection on the method of description are sections devoted to operators with classes of **operands**. The end of this section contains tables of all the operators.

### 10.5.1 Method of description

Where an operator has **operands** and yield which may include LONG or SHORT, the mode is written using L. For example:

```
OP + = (L INT, L INT)L INT:
```

is shorthand for the following operators:

```
OP + = (INT, INT) INT:
```

```
OP + = (LONG INT, LONG INT) LONG INT:
```

```
OP + = (LONG LONG INT, LONG LONG INT) LONG LONG INT:
```

Ensure that wherever L is replaced by SHORTS or LONGS, it should be replaced by the same number of SHORTS or LONGS throughout the definition of that operator. This is known as "consistent substitution". Note that any number of SHORTS or LONGS can be given in the mode of any value whose mode accepts such constructs (INT, REAL, COMPLEX and BITS), but the only modes which can be distinguished are those specified by the environment enquiries in section 10.4.1. a68g maps a **declarer** whose length is not implemented onto the most appropriate length available {16<sub>2</sub>.1.3.1}. Routines or operators for unimplemented lengths are mapped accordingly. a68g considers mapped modes equivalent to the modes they are mapped onto, while standard Algol 68 would still set them apart.

### 10.5.2 Operator synonyms

Algol 68 provides a plethora of operators, and in some cases also synonyms for operators. In the listings further on in this chapter, only one synonym will be defined per operator. This is a list of synonyms for **operator-symbols** implemented in a68g:

1. AND for &
2. ^ for \*\*
3. ~ for NOT
4. ~= for /=
5. ^= for /=
6. I for +\*
7. EQ for =
8. NE for /=
9. LE for <=



10. LT for <
11. GE for >=
12. GT for >
13. OVER for %
14. MOD for %\*
15. PLUSAB for +:=
16. MINUSAB for -:=
17. TIMESAB for \*:=
18. DIVAB for /:=
19. OVERAB for %:=
20. MODAB for %\*:=
21. PLUSTO for +=

### 10.5.3 Standard priorities

The priority of an operator is independent of the mode of the **operands** or result. The standard-prelude sets next priorities for operators:

1. +:=, -:=, \*:=, /:=, %:=, %\*:=, +=
2. OR
3. AND, XOR
4. =, /=
5. <, <=, >=, >
6. -, +
7. \*, /, %, %\*, ELEM
8. \*\*, UP, DOWN, SHL, SHR, LWB, UPB
9. +\*, I

These priorities can be changed by `PRIO` in your **program**, but this easily leads to incomprehensible **programs**. Use `PRIO` for your own **dyadic-operators**.

### 10.5.4 Operators with row operands

Both monadic and dyadic forms are available. We will use the mode `ROW` to denote the mode of any row.

1. **Monadic.**

OP LWB = (ROW r) INT

OP UPB = (ROW r) INT

Yield the **lower-bound** or **upper-bound** for the first dimension of r.

OP ELEMS = (ROW r) INT

Yields the number of elements, in all dimensions, of r.

2. **Dyadic.**

OP LWB = (INT n, ROW r) INT

OP UPB = (INT n, ROW r) INT

Yield the **lower-bound** or **upper-bound** of the n-th dimension of r.

OP ELEMS = (ROW) INT

Yields the number of elements in the n-th dimension of r.

### 10.5.5 Operators with `BOOL` operands

1. OP ABS = (BOOL a) INT

ABS TRUE yields a non-zero number and ABS FALSE yields zero.

2. OP AND = (BOOL a, b) BOOL

Logical AND.

3. OP OR = (BOOL a, b) BOOL

Logical inclusive OR.

4. OP XOR = (BOOL a, b) BOOL

Logical exclusive OR, XOR.

5. OP NOT = (BOOL a) BOOL

Logical NOT: yields TRUE if a is FALSE and yields FALSE if a is TRUE.

6. OP == (BOOL a, b) BOOL

TRUE if a equals b and FALSE otherwise.

7. OP /= (BOOL a, b) BOOL

TRUE if a not equal to b and FALSE otherwise.

## 10.5.6 Operators with INT operands

The `L` shorthand is used for operators taking values of any precision.

### Monadic-operators

Consistent substitution applies to all those operators in this section which use the `L` shorthand: apart from `LENG` and `SHORTEN`, the precision of the yield is the same as the precision of the **operand**.

1. `OP + = (L INT a) L INT`  
The identity operator.
2. `OP - = (L INT a) L INT`  
The negation operator.
3. `OP ABS = (L INT a) L INT`  
The absolute value:  $(a < 0 \mid - a \mid a)$ .
4. `OP SIGN = (L INT a) INT`  
Yields  $-1$  for a negative **operand**,  $+1$  for a positive **operand** and  $0$  for a zero **operand**.
5. `OP ODD = (L INT a) BOOL`  
Yields `TRUE` if the **operand** is odd and `FALSE` if it is even. This is a relic of times long past.
6. `OP LENG = (L INT a) LONG L INT`  
Converts its **operand** to the next longer precision.
7. `OP SHORTEN = (LONG L INT a) L INT`  
Converts its **operand** to the next shorter precision. If `a` exceeds `l max int` for the next shorter precision, a runtime error occurs.

### Dyadic-operators

In this section, consistent substitution is used wherever the `L` shorthand is used. For operators with mixed **operands**, see section 10.5.9.

1. `OP + = (L INT a, L INT b) L INT`  
Integer addition:  $a + b$ .
2. `OP - = (L INT a, L INT b) L INT`  
Integer subtraction:  $a - b$ .

## PROGRAMMING WITH ALGOL 68 GENIE

3. OP \* = (L INT a, L INT b) L INT  
Integer multiplication:  $a \times b$ .
4. OP / = (L INT a, L INT b) L REAL  
Integer fractional division. Even if the quotient is a whole number (for example, 6/3), the yield always has mode L REAL.
5. OP % = (L INT a, L INT b) L INT  
Integer division.
6. OP %\* = (L INT a, L INT b) L INT  
Integer modulo, which always yields a positive result {see ??}.
7. OP \*\* = (L INT a, INT b) L INT  
Computes  $a^b$  for  $b \geq 0$ .
8. OP ++ = (L INT a, L INT b) L COMPLEX  
Joins two integers into a complex number  $a + bi$  of the same precision.
9. OP = = (L INT a, L INT b) BOOL  
Integer equality:  $a = b$ .
10. OP /= = (L INT a, L INT b) BOOL  
Integer inequality:  $a \neq b$ .
11. OP < = (L INT a, L INT b) BOOL  
Integer "less than"  $a < b$ .
12. OP <= = (L INT a, L INT b) BOOL  
Integer "not greater than"  $a \leq b$ .
13. OP >= = (L INT a, L INT b) BOOL  
Integer "not less than":  $a \geq b$ .
14. OP > = (L INT a, L INT b) BOOL  
Integer "greater than":  $a > b$ .

### 10.5.7 Operators with REAL operands

The shorthand L is used for these operators can have **operands** of any precision.

#### Monadic-operators

1. OP + = (L REAL a) L REAL  
Real identity.

2. OP - = (L REAL a) L REAL  
Real negation:  $-a$ .
3. OP ABS = (L REAL a) L REAL  
The absolute value.
4. OP SIGN = (L REAL a) INT  
Yields  $-1$  for negative **operands**,  $+1$  for positive **operands** and  $0$  for a zero **operand**.
5. OP ROUND = (L REAL a) L INT  
Rounds its **operand** to the nearest integer. The operator checks for integer overflow.
6. OP ENTIER = (L REAL a) L INT  
Yields the largest integer not larger than the **operand**. The operator checks for integer overflow.
7. OP LENG = (L REAL a) LONG L REAL  
OP LENG = (SHORT L REAL a) L REAL  
Converts its **operand** to the next longer precision.
8. OP SHORTEN = (LONG L REAL a) L REAL  
Converts its **operand** to the next shorter precision. If a value exceeds `l max real` for the next shorter precision, a runtime error occurs. The mantissa will be rounded.

## Dyadic-operators

In this section, consistent substitution is used wherever the `L` shorthand appears. For operators with mixed **operands**, see section 10.5.9.

1. OP + = (L REAL a, L REAL b) L REAL  
Real addition  $a + b$ .
2. OP - = (L REAL a, L REAL b) L REAL  
Real subtraction  $a - b$ .
3. OP \* = (L REAL a, L REAL b) L REAL  
Real multiplication  $a \times b$ .
4. OP / = (L REAL a, L REAL b) L REAL  
Real division  $a/b$ .
5. OP +\* = (L REAL a, L REAL b) L COMPLEX  
Joins two reals into a complex number  $a + bi$  of the same precision.
6. OP = = (L REAL a, L REAL b) BOOL  
Real equality:  $a = b$ .

## PROGRAMMING WITH ALGOL 68 GENIE

7. OP /= = (L REAL a, L REAL b) BOOL  
Real inequality:  $a \neq b$ .
8. OP < = (L REAL a, L REAL b) BOOL  
Real "less than":  $a < b$ .
9. OP <= = (L REAL a, L REAL b) BOOL  
Real "not greater than":  $a \leq b$ .
10. OP >= = (L REAL a, L REAL b) BOOL  
Real "not less than":  $a \geq b$ .
11. OP > = (L REAL a, L REAL b) BOOL  
Real "greater than":  $a > b$ .

### 10.5.8 Operators with COMPLEX operands

Algol 68 Genie offers a rich set of operators and routines for complex numbers. Consistent substitution applies to operators using the L shorthand.

#### Monadic-operators

1. OP RE = (L COMPLEX a) L REAL  
Yields the real component: re OF a.
2. OP IM = (L COMPLEX a) L REAL  
Yields the imaginary component: im OF a.
3. OP ABS = (L COMPLEX a) L REAL  
Yields the absolute value (a magnitude) of its argument.
4. OP ARG = (L COMPLEX a) L REAL  
Yields the argument of the complex number.
5. OP CONJ = (L COMPLEX a) L COMPLEX  
Yields the conjugate complex number.
6. OP + = (L COMPLEX a) L COMPLEX  
Complex identity.
7. OP - = (L COMPLEX a) L COMPLEX  
Complex negation.
8. OP LENG = (L COMPLEX a) LONG L COMPLEX  
Converts its **operand** to the next longer precision.

9. OP SHORTEN = (LONG L COMPLEX a) L COMPLEX  
 Converts its **operand** to the next shorter precision. If either of the components of the complex number exceeds `1 max real` for the next shorter precision, a runtime error occurs.

## Dyadic-operators

In this section, consistent substitution is used wherever the L shorthand appears. For operators with mixed **operands**, see section 10.5.9.

1. OP + = (L COMPLEX a, L COMPLEX b) L COMPLEX  
 Complex addition for both components  $a + b$ .
2. OP - = (L COMPLEX a, L COMPLEX b) L COMPLEX  
 Complex subtraction for both components  $a - b$ .
3. OP \* = (L COMPLEX a, L COMPLEX b) L COMPLEX  
 Complex multiplication  $a * b$ .
4. OP / = (L COMPLEX a, L COMPLEX b) L COMPLEX  
 Complex division  $a/b$ .
5. OP = = (L COMPLEX a, L COMPLEX b) BOOL  
 Complex equality  $a = b$ .
6. OP /= = (L COMPLEX a, L COMPLEX b) BOOL  
 Complex inequality  $a \neq b$ .

## 10.5.9 Operators with mixed operands

Consistent substitution is applicable to all operators using the L shorthand. Additional short hands are used as follows:

1. The shorthand P stands for +, -, \* or /.
2. The shorthand R stands for <, <=, =, /=, >=, >, or LT, LE, EQ, NE, GE, GT.
3. The shorthand E stands for = /=, or EQ or NE.

1. OP P = (L INT a, L REAL b) L REAL
2. OP P = (L REAL a, L INT b) L REAL

## PROGRAMMING WITH ALGOL 68 GENIE

3. OP P = (L INT a, L COMPLEX b) L COMPLEX
4. OP P = (L COMPLEX a, L INT b) L COMPLEX
5. OP P = (L REAL a, L COMPLEX b) L COMPLEX
6. OP P = (L COMPLEX a, L REAL b) L COMPLEX
7. OP R = (L INT a, L REAL b) BOOL
8. OP R = (L REAL a, L INT b) BOOL
9. OP E = (L INT a, L COMPLEX b) BOOL
10. OP E = (L COMPLEX a, L INT b) BOOL
11. OP E = (L REAL a, L COMPLEX b) BOOL
12. OP E = (L COMPLEX a, L REAL b) BOOL
13. OP \*\* = (L REAL a, INT b) L REAL
14. OP \*\* = (L COMPLEX a, INT b) L COMPLEX
15. OP +\* = (L INT a, L REAL b) L COMPLEX
16. OP +\* = (L REAL a, L INT b) L COMPLEX

### 10.5.10 Operators with BITS operands

Consistent substitution applies to all operators using the L shorthand.

#### Monadic-operators

1. OP BIN = (L INT a) L BITS  
Mode conversion.
2. OP ABS = (L BITS a) L INT  
Mode conversion.
3. OP NOT = (L BITS a) L BITS  
Yields the bits obtained by inverting each bit in the **operand**.
4. OP LENG = (L BITS a) LONG L BITS  
Converts a bits value to the next longer precision by adding zero bits to the more significant end.
5. OP SHORTEN = (LONG L BITS a) L BITS  
Converts a bits value to a value of the next shorter precision.



**Dyadic-operators**

1. OP AND = (L BITS a, L BITS b) L BITS  
The logical "AND" of corresponding binary digits in a and b.
2. OP OR = (L BITS a, L BITS b) L BITS  
The logical "OR" of corresponding binary digits in a and b.
3. OP SHL = (L BITS a, INT b) L BITS  
The left **operand** shifted left by the number of bits specified by the right **operand**.  
New bits shifted in are zero. If the right **operand** is negative, shifting is to the right.
4. OP SHR = (L BITS a, INT b) L BITS  
The left **operand** shifted right by the number of bits specified by the right **operand**.  
New bits shifted in are zero. If the right **operand** is negative, shifting is to the left.
5. OP ELEM = (INT a, L BITS b) BOOL  
Yields TRUE if bit a is set, and FALSE if it is not set.
6. OP == = (L BITS a, L BITS b) BOOL  
Logical equality  $a = b$ .
7. OP /= = (L BITS a, L BITS b) BOOL  
Logical inequality  $a \neq b$ .
8. OP <= = (L BITS a, L BITS b) BOOL  
Yields TRUE a is a subset of b or FALSE otherwise:  $(a \text{ OR } b) = b$
9. OP >= = (L BITS a, L BITS b) BOOL  
Yields TRUE b is a subset of a or FALSE otherwise:  $(a \text{ OR } b) = a$

**10.5.11 Operators with CHAR operands**

The short hands in section 10.5.9 apply here.

1. OP ABS = (CHAR a) INT  
The integer equivalent of a character.
2. OP REPR = (INT a) CHAR  
The character representation of an integer. The **operand** should be in the range 0 ... max abs char.
3. OP + = (CHAR a, CHAR b) STRING  
The character b is appended to the character a (concatenation).

## PROGRAMMING WITH ALGOL 68 GENIE

4. OP E = (CHAR a, CHAR b) BOOL  
Equality or inequality of characters.
5. OP R = (CHAR a, CHAR b) BOOL  
Relative ordering of characters.

### 10.5.12 Operators with STRING operands

The short hands in section 10.5.9 apply here.

1. OP ELEM = (INT a, STRING b) CHAR  
Yields  $b[a]$ . This is an *ALGOL68C* operator.
2. OP + = (STRING a, STRING b) STRING  
String b is appended to string a (concatenation).
3. OP + = (CHAR a, STRING b) STRING  
String b is appended to character a.
4. OP + = (STRING a, CHAR b) STRING  
Character b is appended to string a.
5. OP \* = (INT a, STRING b) STRING  
Yields a times string b, concatenated.
6. OP \* = (STRING a, INT b) STRING  
Yields b times string a, concatenated.
7. OP \* = (INT a, CHAR b) STRING  
Yields a times character b, concatenated.
8. OP \* = (CHAR a, INT b) STRING  
Yields b times character a, concatenated.
9. OP E = (STRING a, STRING b) BOOL  
OP E = (CHAR a, STRING b) BOOL  
OP E = (STRING a, CHAR b) BOOL  
Equality or inequality of characters and strings.
10. OP R = (STRING a, STRING b) BOOL  
OP R = (CHAR a, STRING b) BOOL  
OP R = (STRING a, CHAR b) BOOL  
Alphabetic ordering of strings.

### 10.5.13 Operators with BYTES operands

The short hands in section 10.5.9 apply here.

1. OP LENG = (BYTES a) LONG BYTES  
Converts a bytes value to longer width by padding null characters.
2. OP SHORTEN = (LONG BYTES a) L BYTES  
Converts a value to normal width.
3. OP ELEM = (INT a, L BYTES b) CHAR  
Yields the  $a^{th}$  character in b.
4. OP + = (L BYTES a, L BYTES b) BYTES  
Concatenation  $a + b$
5. OP E = (L BYTES a, L BYTES b) BOOL  
Equality or inequality of byte strings.
6. OP R = (L BYTES a, L BYTES b) BOOL  
Alphabetic ordering of byte strings.

### 10.5.14 Operators combined with assignation

Consistent substitution applies to all operators containing the L shorthand.

1. +=

The operator is a shorthand for  $a := a + b$ .

| Left operand  | Right operand | Result        |
|---------------|---------------|---------------|
| REF L INT     | L INT         | REF L INT     |
| REF L REAL    | L INT         | REF L REAL    |
| REF L COMPLEX | L INT         | REF L COMPLEX |
| REF L REAL    | L REAL        | REF L REAL    |
| REF L COMPLEX | L REAL        | REF L COMPLEX |
| REF L COMPLEX | L COMPLEX     | REF L COMPLEX |
| REF STRING    | CHAR          | REF STRING    |
| REF STRING    | STRING        | REF STRING    |
| REF L BYTES   | L BYTES       | REF L BYTES   |

2. += :

The operator is a shorthand for  $b := a + b$ .

| Left operand | Right operand | Result      |
|--------------|---------------|-------------|
| STRING       | REF STRING    | REF STRING  |
| CHAR         | REF STRING    | REF STRING  |
| L BYTES      | REF L BYTES   | REF L BYTES |

## PROGRAMMING WITH ALGOL 68 GENIE

### 3. -=

The operator is a shorthand for  $a := a - b$ .

| Left operand  | Right operand | Result        |
|---------------|---------------|---------------|
| REF L INT     | L INT         | REF L INT     |
| REF L REAL    | L INT         | REF L REAL    |
| REF L COMPLEX | L INT         | REF L COMPLEX |
| REF L REAL    | L REAL        | REF L REAL    |
| REF L COMPLEX | L REAL        | REF L COMPLEX |
| REF L COMPLEX | L COMPLEX     | REF L COMPLEX |

### 4. \*:=

The operator is a shorthand for  $a := a * b$ .

| Left operand  | Right operand | Result        |
|---------------|---------------|---------------|
| REF L INT     | L INT         | REF L INT     |
| REF L REAL    | L INT         | REF L REAL    |
| REF L COMPLEX | L INT         | REF L COMPLEX |
| REF L REAL    | L REAL        | REF L REAL    |
| REF L COMPLEX | L REAL        | REF L COMPLEX |
| REF L COMPLEX | L COMPLEX     | REF L COMPLEX |
| REF STRING    | INT           | REF STRING    |

### 5. /:=

The operator is a shorthand for  $a := a / b$ .

| Left operand  | Right operand | Result        |
|---------------|---------------|---------------|
| REF L REAL    | L INT         | REF L REAL    |
| REF L REAL    | L REAL        | REF L REAL    |
| REF L COMPLEX | L INT         | REF L COMPLEX |
| REF L COMPLEX | L REAL        | REF L COMPLEX |
| REF L COMPLEX | L COMPLEX     | REF L COMPLEX |

### 6. OP %:= = (REF L INT a, L INT b) REF L INT

The operator is a shorthand for  $a := a \% b$ .

### 7. OP %\*:= = (REF L INT a, L INT b) REF L INT

The operator is a shorthand for  $a := a \%* b$ .

## 10.5.15 Synchronisation operators

a68g implements the **parallel-clause** on platforms that support POSIX threads {see ??}.

### 1. OP LEVEL = (INT a) SEMA

Yields a semaphore whose value is a.

2. OP LEVEL = (SEMA a) INT  
Yields the level of a, that is filed F OF a.
3. OP DOWN = (SEMA a) VOID  
The level of a is decremented. If it reaches 0, then the parallel **unit** that called this operator is hibernated until another parallel **unit** increments the level of a again.
4. OP UP = (SEMA a) VOID  
The level of a is incremented and all parallel **units** that were hibernated due to this semaphore being down are awakened.

## 10.6 Standard procedures

### 10.6.1 Procedures for real numbers

The shorthand L is used to simplify the list of procedures.

1. PROC L sqrt = (L REAL x) L REAL  
The square root of x provided that  $x \geq 0$ .
2. PROC L curt = (L REAL x) L REAL  
PROC L cbrt = (L REAL x) L REAL  
The cube root of x provided that  $x \geq 0$ .
3. PROC L exp = (L REAL x) L REAL  
Yields  $e^x$ .
4. PROC L ln = (L REAL x) L REAL  
The natural logarithm of x provided that  $x > 0$ .
5. PROC L log = (L REAL x) L REAL  
The logarithm of x to base 10.
6. PROC L sin = (L REAL x) L REAL  
The sine of x, with x in radians.
7. PROC L arcsin = (L REAL x) L REAL  
The inverse sine of x.
8. PROC L cos = (L REAL x) L REAL  
The cosine of x, with x in radians.
9. PROC L arccos = (L REAL x) L REAL  
The inverse cosine of x.

## PROGRAMMING WITH ALGOL 68 GENIE

10. PROC L tan = (L REAL x) L REAL  
The tangent of  $x$ , with  $x$  in radians.
11. PROC L arctan = (L REAL x)L REAL  
The inverse tangent of  $x$ .
12. PROC L arctan2(L REAL x, y) L REAL  
The angle whose tangent is  $y/x$ . The angle will be in range  $[-\pi, \pi]$ .
13. PROC L sinh = (L REAL x) L REAL  
The hyperbolic sine of  $x$ .
14. PROC L arcsinh = (L REAL x) L REAL  
The inverse hyperbolic sine of  $x$ .
15. PROC L cosh = (L REAL x) L REAL  
The hyperbolic cosine of  $x$ .
16. PROC L arccosh = (L REAL x) L REAL  
The inverse hyperbolic cosine of  $x$ .
17. PROC L tanh = (L REAL x)L REAL  
The hyperbolic tangent of  $x$ .
18. PROC L arctanh = (L REAL x)L REAL  
The inverse hyperbolic tangent of  $x$ .

### 10.6.2 Procedures for complex numbers

The shorthand L is used to simplify the list of procedures.

1. PROC L complex sqrt = (L COMPLEX z) L COMPLEX  
The square root of  $z$ .
2. PROC L complex exp = (L COMPLEX z) L COMPLEX  
Yields  $e^z$ .
3. PROC L complex ln = (L COMPLEX z) L COMPLEX  
The natural logarithm of  $z$ .
4. PROC L complex sin = (L COMPLEX z) L COMPLEX  
The sine of  $z$ .

5. PROC L complex arcsin = (L COMPLEX z) L COMPLEX  
The inverse sine of z.
6. PROC L complex cos = (L COMPLEX z) L COMPLEX  
The cosine of z.
7. PROC L complex arccos = (L COMPLEX z) L COMPLEX  
The inverse cosine of z.
8. PROC L complex tan = (L COMPLEX z)L COMPLEX  
The tangent of z.
9. PROC L complex arctan = (L COMPLEX z)L COMPLEX  
The inverse tangent of z.
10. PROC L complex sinh = (L COMPLEX z) L COMPLEX  
The hyperbolic sine of z.
11. PROC L complex arcsinh = (L COMPLEX z) L COMPLEX  
The inverse hyperbolic sine of z.
12. PROC L complex cosh = (L COMPLEX z) L COMPLEX  
The hyperbolic cosine of z.
13. PROC L complex arccosh = (L COMPLEX z) L COMPLEX  
The inverse hyperbolic cosine of z.
14. PROC L complex tanh = (L COMPLEX z) L COMPLEX  
The hyperbolic tangent of z.
15. PROC L complex arctanh = (L COMPLEX z) L COMPLEX  
The inverse hyperbolic tangent of z.

### 10.6.3 Random-number generator

a68g implements a Tausworthe generator that has a period of order  $2^{113}$ , from the GNU Scientific Library.

1. PROC first random = (INT seed) VOID  
Initialises the random number generator using seed.
2. PROC L next random = L REAL  
Generates the next random number. The result will be in  $[0 \dots 1 >$ .

## 10.6.4 Miscellaneous procedures and operators

1. PROC L bits pack = ([ ] BOOL a) L BITS

This routine packs *a* into a value of mode L BITS. In case *a* has more elements than L bits width, a runtime error occurs. In case *a* has less elements than L bits width, the result will be aligned to the right and be padded with F bits to the left. For example:

```
$ a68g -p "bitpack ((TRUE, FALSE, TRUE, TRUE, FALSE))"
FFFFFFFFFFFFFFFFFFFFFFFFTTTT
```

Only when *a* has L bits width elements, this relation holds:

```
a[i] = i ELEM L bits pack (a)
```

2. PROC L bytes pack = (STRING text) L BYTES

This routine converts *text* to a L BYTES representation. In case *text* has more elements than L bytes width, a runtime error occurs. In case the argument has less elements than L bytes width, the result is padded with null chars. The result of the routine is such that this relation holds:

```
text[i] = i ELEM L bytes pack (text)
```

3. PROC char in string = (CHAR c, REF INT p, STRING t) BOOL

If found, assigns position (counting from LWB t) of first occurrence of *c* in *t* to *p* if *p* is not NIL. Returns whether *c* is found.

4. PROC last char in string = (CHAR c, REF INT p, STRING t) BOOL

If found, assigns position (counting from LWB t) of last occurrence of *c* in *text* to *p* if *p* is not NIL. Returns whether *c* is found.

5. PROC string in string = (STRING c, REF INT p, STRING t) BOOL

If found, assigns position (counting from LWB t) of first occurrence of *c* in *t* to *p* if *p* is not NIL. Returns whether *c* is found.

6. OP SORT = ([ ] STRING row) [ ] STRING

Yields a copy of *row*, with elements sorted in ascending order. SORT employs a quick-sort algorithm.

7. PROC is alnum = (CHAR c) BOOL

Checks whether *c* is an alphanumeric character; it is equivalent to:

```
is alpha (c) OR is digit (c)
```

8. PROC is alpha = (CHAR c) BOOL

Checks whether *c* is an alphabetic character; it is equivalent to:

```
is upper (c) OR is lower (c)
```

In some locales, there may be additional characters for which this routine holds.



9. PROC `is cntrl = (CHAR c) BOOL`  
Checks whether `c` is a control character.
10. PROC `is digit = (CHAR c) BOOL`  
Checks whether `c` is a digit (0 through 9).
11. PROC `is graph = (CHAR c) BOOL`  
Checks whether `c` is a printable character except space
12. PROC `is lower = (CHAR c) BOOL`  
Checks whether `c` is a lower-case character.
13. PROC `is print = (CHAR c) BOOL`  
Checks whether `c` is a printable character including space.
14. PROC `is punct = (CHAR c) BOOL`  
Checks whether `c` is a printable character which is not a space or an alphanumeric character.
15. PROC `is space = (CHAR c) BOOL`  
Checks whether `c` is white-space characters. In the C and POSIX locales, these are space, form-feed, newline, carriage return, horizontal tab, and vertical tab.
16. PROC `is upper = (CHAR c) BOOL`  
Checks whether `c` is an upper-case letter.
17. PROC `is xdigit = (CHAR c) BOOL`  
Checks whether `c` is a hexadecimal digit, that is, one of `0123456789abcdefABCDEF`.
18. PROC `to lower = (CHAR c) CHAR`  
Converts the letter `c` to lower-case, if possible.
19. PROC `to upper = (CHAR c) CHAR`  
Converts the letter `c` to upper-case, if possible.
20. PROC `system = (STRING command) INT`  
Passes the string `command` to the operating system for execution. The operating system is expected to return an integer value that will be returned by `system`.
21. PROC `break = VOID`  
Raises `SIGINT`, after which `a68g` will stop in its monitor routine. The monitor may be entered on a **unit** executed after the actual **call** to `break`; see details on breakpoints. To enter the monitor immediately, call `debug` or `monitor`.

## PROGRAMMING WITH ALGOL 68 GENIE

22. PROC debug = VOID  
PROC monitor = VOID  
a68g will stop in its monitor routine immediately.
23. PROC evaluate = (STRING expression) STRING  
Evaluate expression in the monitor and returning the resulting value as a string.
24. PROC program idf = STRING  
Returns the name of the source file.
25. PROC seconds = REAL  
PROC clock = REAL  
PROC cpu time = REAL  
Returns time in seconds elapsed since interpreter started. Note that the time can wrap around; on a 32-bit system with a microsecond resolution of the clock for example, this routine will return the same value approximately every 72 minutes.
26. PROC sweep heap = VOID  
Invokes the garbage collector immediately. Sometimes Algol 68 Genie cannot sweep the heap when memory is required, for instance when copying stowed objects. In rare occasions this may lead to a **program** running out of memory. Calling this routine just before the position where the **program** ran out of memory might help to keep it running. Otherwise (or, alternatively) a larger heap size should be given to the **program**.
27. PROC pre emptive sweep = VOID  
PROC pre emptive sweep heap = VOID  
Invokes the garbage collector if the heap is "sufficiently full". Sometimes Algol 68 Genie cannot sweep the heap when memory is required, for instance when copying stowed objects. In rare occasions this may lead to a **program** running out of memory. Calling this routine just before the position where the **program** ran out of memory might help to keep it running. Otherwise (or, alternatively) a larger heap size should be given to the **program**.
28. PROC garbage = LONG INT  
Returns the number of bytes recovered by the garbage collector.
29. PROC collections = INT  
Returns the number of times the garbage collector was invoked.
30. PROC collect seconds = REAL  
Returns an estimate of the time used by the garbage collector.

31. `PROC system stack size = INT`  
Returns the size of the system stack.
32. `PROC system stack pointer = INT`  
Returns the (approximate) value of the system stack pointer.
33. `PROC stack pointer = INT`  
Returns the value of the evaluation stack pointer. (Note: this is an Algol 68 stack, not the system stack)

## 10.7 Transput

An introduction to Algol 68 transput is in chapter 7. The function of this section is to document all the transput **declarations** so that you can use it for reference purposes.

### 10.7.1 Transput modes

Only two modes are available:

1. `FILE`  
A structure containing details of a file.
2. `CHANNEL`  
A structure whose fields are routines returning truth values which determine the available methods of access to a file.

### 10.7.2 Standard channels

`a68g` initialises four channels at start-up:

1. `CHANNEL stand in channel`
2. `CHANNEL stand out channel`
3. `CHANNEL stand back channel`
4. `CHANNEL stand error channel`

## PROGRAMMING WITH ALGOL 68 GENIE

These four channels have similar properties because they use the same access procedures. The standard input channel is `stand in channel`. Files on this channel have the following properties:

| stand in channel |       |
|------------------|-------|
| reset possible   | FALSE |
| set possible     | FALSE |
| get possible     | TRUE  |
| put possible     | FALSE |
| bin possible     | FALSE |

The `stand out channel` is the standard output channel. Files on this channel have the following properties:

| stand out channel |       |
|-------------------|-------|
| reset possible    | FALSE |
| set possible      | FALSE |
| get possible      | FALSE |
| put possible      | TRUE  |
| bin possible      | FALSE |

The `stand back channel` is the standard input/output channel. Files on this channel have the following properties:

| stand back channel |      |
|--------------------|------|
| reset possible     | TRUE |
| set possible       | TRUE |
| get possible       | TRUE |
| put possible       | TRUE |
| bin possible       | TRUE |

The `stand error channel` is the standard error output channel. Files on this channel have the following properties:

| stand out channel |       |
|-------------------|-------|
| reset possible    | FALSE |
| set possible      | FALSE |
| get possible      | FALSE |
| put possible      | TRUE  |
| bin possible      | FALSE |

### 10.7.3 Standard files

Four standard files are provided:

1. REF FILE stand in  
On Linux this file is opened on `stdin` with `stand in` channel.
2. REF FILE stand out  
On Linux this file is opened on `stdout` with `stand out` channel.
3. REF FILE stand error  
On Linux this file is opened on `stderr` with `stand error` channel.
4. REF FILE stand back  
If this file is used, a temporary file will be established on the file system.

### 10.7.4 Opening files

These procedures are available for opening files:

1. PROC establish = (REF FILE *f*, STRING *n*, CHANNEL *c*) INT  
Establishes a new file *f* with file name *n* and channel *c*. If the file already exists, an `on open error` event occurs. The procedure yields zero on success, otherwise an integer denoting an error.
2. PROC open = (REF FILE *f*, STRING *n*, CHANNEL *c*) INT  
Establishes a new file *f* with file name *n* and channel *c*. On Linux, the routine yields zero if the file is a regular file and non-zero otherwise. If the file does not exist, it will be created when the file is actually written to.
3. PROC create = (REF FILE *f*, CHANNEL *c*) INT  
Creates a temporary file (on Linux with a unique identification in the directory `/tmp`) using the given channel *c*.
4. PROC associate = (REF FILE *f*, REF STRING *s*) VOID  
Associates file *f* with string *s*. Note that the Revised Report specifies a `REF [][][] CHAR` argument where `a68g` specifies a `REF STRING` argument. On putting, the string is dynamically lengthened and output is added at the end of the string. Attempted getting outside the string provokes an end of file condition, and `on file end` is invoked. When a file that is associated with a string is reset, getting restarts from the start of the associated string.

### 10.7.5 Closing files

Three procedures are provided.

## PROGRAMMING WITH ALGOL 68 GENIE

1. PROC close = (REF FILE f) VOID  
This is the common procedure for closing a file. The procedure checks whether the file is open.
2. PROC lock = (REF FILE f) VOID  
The Algol 68 Revised Report requires lock to close the file in such a manner that some system action is required before it can be reopened. a68g closes the file and then removes all access permissions.
3. PROC scratch = (REF FILE f) VOID  
The file is closed and then deleted from the file system.

### 10.7.6 Transput routines

The procedures in this section are responsible for the transput of actual values. Firstly, default-format transput is covered and then binary transput. In each section, the shorthand L is used for the various precisions of numbers and bits values.

#### Default-format transput

1. PROC put = (REF FILE f,  
[] UNION (OUTTYPE, PROC (REF FILE) VOID) items) VOID  
Writes items to file f using default formatting.
2. PROC print =  
([] UNION (OUTTYPE, PROC (REF FILE) VOID) items) VOID  
Writes items to file stdout using default formatting.
3. PROC get = (REF FILE f,  
[] UNION (INTYPE, PROC (REF FILE) VOID) items) VOID  
Reads items from file f using default formatting.
4. PROC read =  
([] UNION (INTYPE, PROC (REF FILE) VOID) items) VOID  
Reads items from file stdin using default formatting.

#### Formatted transput

1. PROC putf = (REF FILE f, [] UNION (OUTTYPE,  
PROC (REF FILE) VOID, FORMAT) items) VOID  
Writes items to file f using formatted transput.

2. PROC printf = ([] UNION (OUTTYPE,  
PROC (REF FILE) VOID, FORMAT) items) VOID  
Writes items to file stdout using formatted transput.
3. PROC getf = (REF FILE f, [] UNION (INTYPE,  
PROC (REF FILE) VOID, FORMAT) items) VOID  
Reads items from file f using formatted transput.
4. PROC readf = ([] UNION (INTYPE,  
PROC (REF FILE) VOID, FORMAT) items) VOID  
Reads items from file stdin using formatted transput.

## Binary transput

Binary transput performs no conversion, thus providing a means of storing data in a compact form in files or reading data.

1. PROC write bin = ([] OUTTYPE x) VOID  
This is equivalent to put bin (stand back, x).
2. PROC put bin =  
(REF FILE f, [] OUTTYPE x) VOID  
This procedure outputs data in a compact form. Then external size is the same as the internal size.
3. PROC read bin=([]SIMPLIN x)VOID  
This procedure is equivalent to:  
get bin (stand back, x)
4. PROC get bin = (REF FILE f, [] INTYPE x)VOID  
This procedure reads data in a compact form.

It should also be noted that the procedure `make term`, although usually used with default-format transput, can also be used with binary transput for reading a `STRING`.

## 10.7.7 Interrogating files

A number of procedures are available for interrogating the properties of files:

1. PROC bin possible = (REF FILE f) BOOL  
Yields TRUE if binary transput is possible.

## PROGRAMMING WITH ALGOL 68 GENIE

2. PROC put possible = (REF FILE f) BOOL  
Yields TRUE if data can be sent to the file.
3. PROC get possible = (REF FILE f) BOOL  
Yields TRUE if data can be got from the file.
4. PROC set possible = (REF FILE f) BOOL  
Yields TRUE if the position in the file for further transput can be set.
5. PROC reset possible = (REF FILE f) BOOL  
PROC rewind possible = (REF FILE f) BOOL  
Yields TRUE if the position in the file for further transput can be reset. The spelling rewind possible is an a68g extension.
6. PROC compressible = (REF FILE f) BOOL  
Yields TRUE if the file is compressible. This is a dummy routine supplied for backwards compatibility since a68g sets the compressible field of all channels to TRUE.
7. PROC reidf possible = (REF FILE f) BOOL  
Yields TRUE if the identification of the file can be changed.
8. PROC idf = (REF FILE f) STRING  
Yields the identification string of f, if it is set.
9. PROC term = (REF FILE file) STRING  
Yields the terminator string of f, if it is set.

### 10.7.8 File properties

1. PROC make term=(REF FILE f, STRING term) VOID  
Makes term the current string terminator.

### 10.7.9 Event routines

For each routine, the default behaviour will be described. In each case, if the user routine yields FALSE, the default action will be elaborated. If it yields TRUE, the action depends on the event.

1. PROC on file end =  
(REF FILE f, PROC (REF FILE) BOOL p) VOID  
PROC on logical file end =  
(REF FILE f, PROC (REF FILE) BOOL p) VOID  
PROC on physical file end =



## STANDARD-PRELUDE AND LIBRARY-PRELUDE

(REF FILE f, PROC (REF FILE) BOOL p) VOID

These procedure let you provide a routine to be called when end of file is reached on file *f*. The default action on file end is to produce a runtime error.

2. PROC on format end =

(REF FILE f, PROC (REF FILE) BOOL p) VOID

Format end events are caused when in formatted transput, the format gets exhausted. The procedure on format end lets you provide a procedure of mode PROC (REF FILE) BOOL. If the programmer-supplied routine yields TRUE, transput simply continues, otherwise the format that just ended is restarted and transput resumes.

3. PROC on line end =

(REF FILE f, PROC (REF FILE) BOOL p) VOID

PROC on page end =

(REF FILE f, PROC (REF FILE) BOOL p) VOID

Line end and page end events are caused when while reading, the end of line or end of page is encountered. These are events so you can provide a routine that for instance automatically counts the number of lines or pages read. The procedures on line end and on page end let the programmer provide a procedure whose mode must be PROC (REF FILE) BOOL. If the programmer-supplied routine yields TRUE, transput continues, otherwise a new line in case of end of line, or new page in case of end of page, is executed and then transput resumes.

Be careful when reading strings, since end of line and end of page are string terminators! If you provide a routine that mends the line - or page end, be sure to call new line or new page before returning TRUE. In case of a default action, new line or new page must be called explicitly, for instance in the read procedure, otherwise you will read nothing but empty strings as you do not eliminate the terminator.

4. PROC on open error =

(REF FILE f, PROC (REF FILE) BOOL p) VOID

Open error events are caused when a file cannot be opened as required. For instance, you want to read a file that does not exist, or write to a read-only file. The procedure on open error lets the programmer provide a procedure whose mode must be PROC (REF FILE) BOOL. If the programmer-supplied routine yields TRUE, the **program** continues, otherwise a runtime error occurs.

5. PROC on value error =

(REF FILE f, PROC (REF FILE) BOOL p) VOID

Value error events are caused when transputting a value that is not a valid representation of the mode of the object being transput. The procedure on value error lets the programmer provide a procedure whose mode must be PROC (REF FILE) BOOL. If you do transput on the file within the procedure ensure that a value error will not occur again! If the programmer-supplied routine yields TRUE, transput continues, otherwise a runtime error occurs.

6. PROC on format error =

(REF FILE f, PROC (REF FILE) BOOL p) VOID

Format error events are caused when an error occurs in a format, typically when patterns are provided without objects to transput. The procedure on format error lets the programmer provide a procedure whose mode must be PROC (REF FILE) BOOL. If the programmer-supplied routine yields TRUE, the **program** continues, otherwise a runtime error occurs.

7. PROC on transput error =

(REF FILE f, PROC (REF FILE) BOOL p) VOID

This event is caused when an error occurs in transput that is not covered by the other events, typically conversion errors (value out of range et cetera). The procedure on transput error lets the programmer provide a procedure with mode PROC (REF FILE) BOOL. If the programmer-supplied routine yields TRUE, the **program** continues, otherwise a runtime error occurs.

### 10.7.10 Formatting routines

There are four procedures for conversion of numbers to strings. The procedures `whole`, `fixed` and `float` will return a string of `error chars` if the number to be converted cannot be represented in the given width.

1. PROC `whole` = (NUMBER v, INT width) STRING

The procedure converts integer values. Leading zeros are replaced by spaces and a **sign** is included if *width* > 0. If *width* is zero, the shortest possible string is yielded. If a real number is supplied for the **parameter** v, then the call `fixed (v, width, 0)` is elaborated.

2. PROC `fixed` = (NUMBER v, INT width, after) STRING

The procedure converts real numbers to fixed point form, that is, without an exponent. The total number of characters in the converted value is given by the **parameter** *width* whose **sign** controls the presence of a **sign** in the converted value as for `whole`. The **parameter** *after* specifies the number of required digits after the **point-symbol**. From the values of *width* and *after*, the number of digits in front of the **point-symbol** can be calculated. If the space left in front of the **point-symbol** is insufficient to contain the integral part of the value being converted, digits after the **point-symbol** are sacrificed.

3. PROC `float` = (NUMBER v, INT width, after, exp) STRING

The procedure converts reals to floating-point form ("scientific notation"). The total number of characters in the converted value is given by the **parameter** *width* whose **sign** controls the presence of a **sign** in the converted value as for `whole`. Likewise, the **sign** of *exp* controls the presence of a preceding **sign** for the exponent. If *exp*

is zero, then the exponent is expressed in a string of minimum length. In this case, the value of `width` must not be zero. Note that `float` always leaves a position for the **sign**. If there is no **sign**, a blank is produced instead. The values of `width`, `after` and `exp` determine how many digits are available before the decimal point and, therefore, the value of the exponent. The latter value has to fit into the width specified by `exp` and so, if it cannot fit, decimal places are sacrificed one by one until either it fits or there are no more decimal places (and no **point-symbol**). If it still doesn't fit, digits before the decimal place are also sacrificed. If no space for digits remains, the whole string is filled with `error char`.

4. `PROC real = (NUMBER x, INT width, after, exp width, modifier) STRING`  
 Converts `x` to a `STRING` representation. If `modifier` is a positive number, the resulting string will present `x` with its exponent a multiple of `modifier`. If `modifier = 1`, the returned string is identical to that returned by `float`. A common choice for `modifier` is 3 which returns the so-called engineers notation of `x`. If `modifier ≤ 0`, the resulting string will present `x` with `ABS modifier` digits before the **point-symbol** and its exponent adjusted accordingly; compare this to Fortran `nP` syntax.

### 10.7.11 Miscellaneous transput routines

1. `PROC end of line = (REF FILE file) BOOL`  
 This routine yields `TRUE` if the file pointer of `file` is at the end of a line, or `FALSE` if it is not. You can advance the file pointer with `get (file, new line)` or `new line (file)`.
2. `PROC end of file = (REF FILE file) BOOL`  
 This routine yields `TRUE` if the file pointer of `file` is at the end of the file, or `FALSE` if it is not.
3. `PROC set = (REF FILE f, INT n) INT`  
 This routine deviates from the standard Algol 68 definition. It attempts to move the file pointer by `n` character positions with respect to the current position. If the file pointer would as a result of this move get outside the file, it is not changed and the routine `set by on file end` is called. If this routine returns `FALSE`, and end-of-file runtime error is produced. The routine returns an `INT` value representing system-dependent information on this repositioning.
4. `PROC reset = (REF FILE f) VOID`  
`PROC rewind = (REF FILE f) VOID`  
 Resets the file to the state it was in directly after `open`, `create` or `establish`. Default event-routines are installed and the file pointer is reset. The spelling `rewind` is an `a68g` extension.

5. PROC space = (REF FILE f) VOID  
The procedure advances the file pointer in file *f* by one character. It does *not* read or write a blank.
6. PROC backspace = (REF FILE f) VOID  
The procedure attempts to retract the file pointer in file *f* by one character. It executes:  

```
VOID (set (f, -1))
```
7. PROC new line = (REF FILE f) VOID  
On input, skips any remaining characters on the current line and positions the file pointer at the beginning of the next line. This means that all characters on input are skipped until a linefeed character is read. On output, the routine writes a linefeed character.
8. PROC new page = (REF FILE f) VOID  
On input, skips any remaining characters on the current page and positions the file pointer at the beginning of the next page. This means that all characters on input are skipped until a form feed character is read. On output, a form feed character is written.

## 10.8 The library-prelude

`a68g` supplies many routines, operators and constants, not described by the Revised Report, for linear algebra, Fourier transforms, Laplace transforms, drawing and plotting, PCM sounds and PostgreSQL database support. Many of these routines require optional libraries; if these libraries are not present on your system, `a68g` will not support library-prelude elements that require them.

## 10.9 *ALGOL68C* transput procedures

For compatibility with *ALGOL68C*, `a68g` offers a number of transput procedures. Following routines read an object of the resulting mode of the respective routine from `stand in`:

1. PROC read int = INT
2. PROC read long int = LONG INT
3. PROC read long long int = LONG LONG INT

## STANDARD-PRELUDE AND LIBRARY-PRELUDE

4. PROC read real = REAL
5. PROC read long real = LONG REAL
6. PROC read long long real = LONG LONG REAL
7. PROC read complex = COMPLEX
8. PROC read long complex = LONG COMPLEX
9. PROC read long long complex = LONG LONG COMPLEX
10. PROC read bool = BOOL
11. PROC read bits = BITS
12. PROC read long bits = LONG BITS
13. PROC read long long bits = LONG LONG BITS
14. PROC read char = CHAR
15. PROC read string = STRING

Following routines write an object of the **parameter** mode of the respective routine to stand out:

1. PROC print int = (INT k) VOID
2. PROC print long int = (LONG INT k) VOID
3. PROC print long long int = (LONG LONG INT k) VOID
4. PROC print real = (REAL x) VOID
5. PROC print long real = (LONG REAL x) VOID
6. PROC print long long real = (LONG LONG REAL x) VOID
7. PROC print complex = (COMPLEX c) VOID
8. PROC print long complex = (LONG COMPLEX c) VOID
9. PROC print long long complex = (LONG LONG COMPLEX c) VOID
10. PROC print bool = (BOOL b) VOID
11. PROC print bits = (BITS b) VOID
12. PROC print long bits = (LONG BITS b) VOID
13. PROC print long long bits = (LONG LONG BITS b) VOID
14. PROC print char = (CHAR c) VOID
15. PROC print string = (STRING s) VOID

## 10.10 Drawing and plotting

On systems that have installed the GNU Plotting Utilities, `a68g` can be linked with the `libplot` library to extend the standard-`prelude`. This section contains a list of routines supported by the current revision of `a68g`.

The GNU Plotting Utilities offer a number of routines for drawing 2-D graphics. This allows for drawing in an X window, postscript format, pseudo graphical interface format (pseudo-gif) and portable any-map format (pnm) from Algol 68. Note that `a68g` is not a full Algol 68 binding for `libplot`.

A plotter (window, postscript file, et cetera) can be considered as a stream to which plotting commands are written. Therefore `a68g` considers plotters to be objects of mode `REF FILE`. This seems consistent with the design of Algol 68 transput since a file is associated with a channel that describes the device, and therefore can be considered as a description of a device driver.

A plotter must be opened and closed, just like any file. To specify the file to be a drawing device, `stand draw channel` is declared, that yields `TRUE` when inspected by `draw possible`. To specify details on the plotter type and page size, the procedure `draw device` has to be called. A sample **program** is given below:

```
# 'window' is an X window with 600x400 pixels #
FILE window;
draw device (window, "X", "600x400");
open (window, "Hello!", stand draw channel);
# 'draw erase' makes the window appear on the screen #
draw erase (window);
# Goto pixel 0.25x600, 0.5x400 #
draw move (window, 0.25, 0.5);
# Colour will be 100% red, 0% green, 0% blue #
draw colour (window, 1, 0, 0);
# Write an aligned text #
draw text (window, "c", "c", "Hello world!");
# Flush the drawing command buffer -
only needed for real-time plotters like X #
draw show (window);
# Give audience a chance to read the message #
VOID (read char);
# Make the window disappear from the screen #
close (window);
```

### 10.10.1 Setting up a graphics device

## STANDARD-PRELUDE AND LIBRARY-PRELUDE

1. PROC draw device =  
(REF FILE file, STRING type, STRING params) BOOL  
PROC make device =  
(REF FILE file, STRING type, STRING params) BOOL

Set **parameters** for plotter file. **Parameter** type sets the plotter type. a68g supports "X" for X Window System, "ps" for postscript, "gif" for pseudo graphical interface format and "pnm" for portable any-map format. X plotters in a68g are only available with Linux. X plotters write graphics to an X Window System display rather than to an output stream. A runtime error results if the plotter could not be opened. Argument `params` is interpreted as an image size. For X, gif and pnm the syntax reads `x pixelsxy pixels`, for instance "500x500". For postscript, image size is the size of the page on which the graphics display will be positioned. Any ISO page size in the range "a0" - "a4" or ANSI page size in the range "a" - "e" may be specified. Recognised aliases are "letter" for "a" and "tabloid" for "b". Other valid page sizes are "legal", "ledger" and "b5". The graphics display will be a square region centred on the specified page and occupying its full width, with allowance being made for margins. This routine returns TRUE if it succeeds.

Note that some platforms cannot give `libplot` proper support for all plotters. Linux with the X window system lets `libplot` implement all plotters but for example one Win32 binary has problems with X and postscript plotters. For example, the Win32 executable provided for a68g will give runtime errors as X plotter missing or postscript plotter missing. On other platforms it is possible that a plotter produces garbage or gives a message as output stream jammed. Algol 68 Genie does not work around this deficiency.

2. PROC draw erase = (REF FILE file) VOID

Begins the next frame of a multi-frame page, by clearing all previously plotted objects from the graphics display, and filling it with the background colour (if any).

3. PROC draw show = (REF FILE file) VOID

Flushes all pending plotting commands to the display device. This is useful only if the currently selected plotter does real-time plotting, since it may be used to ensure that all previously plotted objects have been sent to the display and are visible to the user. It has no effect on plotters that do not do real-time plotting.

4. PROC draw move = (REF FILE file, REAL x, y) VOID

The graphics cursor is moved to  $(x,y)$ . The values of  $x$  and  $y$  are fractions 0 ... 1 of the page size in respectively the  $x$  and  $y$  directions.

5. PROC draw aspect = (REF FILE file) REAL

Yields the aspect ratio  $y\text{size}/x\text{size}$ .

6. PROC draw fill style = (REF FILE file, INT level) VOID

Sets the fill fraction for all subsequently drawn objects. A value of  $level = 0$  indicates that objects should be unfilled, or transparent. This is the default. A value in the range  $16r0001 \dots 16rffff$ , indicates that objects should be filled. A value of 1 signifies complete filling; the fill colour will be the colour specified by calling `draw colour`. If  $level = 16rffff$ , the fill colour will be white. Values  $16r0002 \dots 16rfffe$  represent a proportional saturation level, for instance  $16r8000$  denotes a saturation of 0.5.

7. PROC `draw linestyle = (REF FILE file, [] CHAR style) VOID`

Sets the line style for all lines subsequently drawn on the graphics display. The supported line styles are "solid", "dotted", "dotdashed", "shortdashed", "longdashed", "dotdotdashed", "dotdotdotdashed", and "disconnected". The first seven correspond to the following dash patterns:

```
"solid"           -----
"dotted"         - - - - -
"dotdashed"      --- - --- - ---
"shortdashed"    --- --- --- ---
"longdashed"     -----
"dotdotdashed"   --- - - --- - -
"dotdotdotdashed" --- - - - --- - -
```

In the preceding patterns, each hyphen stands for one line thickness. This is the case for sufficiently thick lines, at least. So for sufficiently thick lines, the distance over which a dash pattern repeats is scaled proportionately to the line thickness. The "disconnected" line style is special. A "disconnected" path is rendered as a set of filled circles, each of which has diameter equal to the nominal line thickness. One of these circles is centred on each of the juncture points of the path (i.e., the endpoints of the line segments or arcs from which it is constructed). Circles with "disconnected" line style are invisible. Disconnected circles are not filled. All line styles are supported by all plotters.

8. PROC `draw linewidth = (REF FILE file, REAL thickness) VOID`

Sets the thickness of all lines subsequently drawn on the graphics display. The value of `thickness` is a fraction  $0 \dots 1$  of the page size in the  $y$  direction. A negative value resets the thickness to the default. For plotters that produce bitmaps, i.e., X plotters, and pnm plotters, it is zero. By convention, a zero-thickness line is the thinnest line that can be drawn.

## 10.10.2 Specifying colours

In some versions of `libplot`, pseudo-gif plotters produce garbled graphics when more than 256 different colours are specified. Algol 68 Genie does not work around this problem in `libplot`.



1. PROC draw background colour =  
 (REF FILE file, REAL red, green, blue) VOID  
 PROC draw background color =  
 (REF FILE file, REAL red, green, blue) VOID

Sets the background colour for plotter *file* its graphics display, using fractional intensities in a 48-bit RGB colour model. The arguments *red*, *green* and *blue* specify the red, green and blue intensities of the background colour. Each is a real value in the range 0... 1. The choice 0,0,0 signifies black, and the choice 1,1,1 signifies white. This procedure affects only plotters that produce bitmaps, i. e., X plotters and pnm plotters. Its effect is that when the draw erase procedure is called for this plotter, its display will be filled with the specified colour.

2. PROC draw background colour name =  
 (REF FILE file, STRING name) VOID  
 PROC draw background color name =  
 (REF FILE file, STRING name) VOID

Sets the background colour for plotter *file* its graphics display. Argument *name* is a case insensitive string. Accepted names are essentially those supported by the X Window System.

3. PROC draw colour = (REF FILE file, REAL red, green, blue) VOID  
 PROC draw color = (REF FILE file, REAL red, green, blue) VOID

Sets the foreground colour for plotter *file* its graphics display, using fractional intensities in a 48-bit RGB colour model. The arguments *red*, *green* and *blue* specify the red, green and blue intensities of the background colour. Each is a real value in the range 0... 1. The choice 0,0,0 signifies black, and the choice 1,1,1 signifies white.

4. PROC draw colour name = (REF FILE file, STRING name) VOID  
 PROC draw color name = (REF FILE file, STRING name) VOID

Sets the foreground colour for plotter *file* its graphics display. Argument *name* is a case insensitive string. Accepted names are essentially those supported by the X Window System.

### 10.10.3 Drawing objects

1. PROC draw point = (REF FILE file, REAL x, y) VOID

The arguments specify the co-ordinates  $(x, y)$  of a point that will be drawn. The graphics cursor is moved to  $(x, y)$ . The values of *x* and *y* are fractions 0... 1 of the page size in respectively the *x* and *y* directions.

## PROGRAMMING WITH ALGOL 68 GENIE

2. PROC draw line = (REF FILE file, REAL x, y) VOID

The arguments specify the co-ordinates  $(x, y)$  of a line that will be drawn starting from the current graphics cursor. The graphics cursor is moved to  $(x, y)$ . The values of  $x$  and  $y$  are fractions 0... 1 of the page size in respectively the  $x$  and  $y$  directions.

3. PROC draw rect = (REF FILE file, REAL x, y) VOID

The arguments specify the corner coordinates  $(x, y)$  of a rectangle that will be drawn starting from the diagonally opposing corner represented by the current graphics cursor. The graphics cursor is moved to  $(x, y)$ . The values of  $x$  and  $y$  are fractions 0... 1 of the page size in respectively the  $x$  and  $y$  directions.

4. PROC draw circle = (REF FILE file, REAL x, y, r) VOID

The three arguments specifying the centre  $(x, y)$  and radius  $r$  of a circle that is drawn. The graphics cursor is moved to  $(x, y)$ . The values of  $x$  and  $y$  are fractions 0... 1 of the page size in respectively the  $x$  and  $y$  directions. The value of  $r$  is a fraction 0... 1 of the page size in the  $y$  direction.

In some versions of `libplot`, X plotters do not flush after every plotting operation. It may happen that a plotting operation does not show until `close` is called for that plotter. After closing, an X plotter window stays on the screen (as a forked process) until you type "q" while it has focus, or click in it. If you click the close-button of such window, you will get an error message like

```
XIO: fatal IO error 11 (Resource temporarily unavailable) on
X server ":0.0" after 198525 requests (198525 known processed) with 0
events remaining.
```

Algol 68 Genie does not work around this buffering mechanism in `libplot`.

Note that `libplot` plots its primitives without applying anti-aliasing. Example 11.16 demonstrates how to draw an anti-aliased circle using a method from X. Wu:

### 10.10.4 Drawing text

1. PROC draw text =  
(REF FILE file, CHAR h justify, v justify, [] CHAR s) VOID

The justified string  $s$  is drawn according to the specified justifications. If  $h$  justify is equal to "l", "c", or "r", then the string will be drawn with left, centre or right justification, relative to the current graphics cursor position. If  $v$  justify is equal to "b", "x", "c", or "t", then the bottom, baseline, centre or top of the string will be placed even with the current graphics cursor position. The graphics cursor is moved to the right end of the string if left justification is specified, and to the left end if right justification is specified. The string may contain escape sequences of various

sorts {see "Text string format and escape sequences" in the plotutils manual}, though it should not contain line feeds or carriage returns; in fact it should include only printable characters. The string may be plotted at a non-zero angle, if `draw text angle` has been called.

2. PROC `draw text angle = (REF FILE file, INT angle) VOID`

Argument `angle` specifies the angle in degrees counter clockwise from the  $x$  (horizontal) axis in the user coordinate system, for text strings subsequently drawn on the graphics display. The default angle is zero. The font for plotting strings is fully specified by calling `draw font name`, `draw font size`, and `draw text angle`.

3. PROC `draw font name = (REF FILE file, [] CHAR name) VOID`

The case-insensitive string `name` specifies the name of the font to be used for all text strings subsequently drawn on the graphics display. The font for plotting strings is fully specified by calling `draw font name`, `draw font size`, and `draw text angle`. The default font name depends on the type of plotter. It is "Helvetica" for X plotters, for pnm it is "HersheySerif". If the argument `name` is NIL or an empty string, or the font is not available, the default font name will be used. Which fonts are available also depends on the type of plotter; for a list of all available fonts {see "Available text fonts" in the plotutils manual}.

4. PROC `draw font size = (REF FILE file, INT size) VOID`

Argument `size` is interpreted as the size, in the user coordinate system, of the font to be used for all text strings subsequently drawn on the graphics display. A negative value for `size` sets the size to the default, which depends on the type of plotter. Typically, the default font size is 1/50 times the size (minimum dimension) of the display. The font for plotting strings is fully specified by calling `draw font name`, `draw font size`, and `draw text angle`.

## 10.11 Mathematical functions

### 10.11.1 COMPLEX functions

Next routines require that `a68g` is linked to the GNU Scientific Library.

1. PROC `complex sinh = (COMPLEX z) COMPLEX`  
Complex hyperbolic sine of argument `z`.
2. PROC `complex arc sinh = (COMPLEX z) COMPLEX`  
Complex inverse hyperbolic sine of argument `z`.

## PROGRAMMING WITH ALGOL 68 GENIE

3. PROC complex cosh = (COMPLEX z) COMPLEX  
Complex hyperbolic cosine of argument z.
4. PROC complex arc cosh = (COMPLEX z) COMPLEX  
Complex inverse hyperbolic cosine of argument z.
5. PROC complex tanh = (COMPLEX z) COMPLEX  
Complex hyperbolic tangent of argument z.
6. PROC complex arc tanh = (COMPLEX z) COMPLEX  
Complex inverse hyperbolic tangent of argument z.

### 10.11.2 Error and gamma functions

Next routines require that a68g is linked to the GNU Scientific Library, except inverse erf and inverse erfc.

1. PROC erf = (REAL x) REAL  
Error function of argument x.
2. PROC inverse erf = (REAL x) REAL  
Inverse error function of argument x.
3. PROC erfc = (REAL x) REAL  
Complementary error function of argument x, defined as  $1 - \text{erf}(x)$ .
4. PROC inverse erfc = (REAL x) REAL  
Inverse complementary error function of argument x.
5. PROC gamma = (REAL x) REAL  
Gamma function of argument  $\Gamma(x)$ .
6. PROC incomplete gamma = (REAL a, x) REAL  
Incomplete gamma function  $P(a, x)$ .
7. PROC ln gamma = (REAL x) REAL  
Natural logarithm of the gamma function of argument  $\ln\Gamma(x)$ .
8. PROC factorial = (REAL x) REAL  
Factorial of argument x, defined as  $\Gamma(1 + x)$ .

9. PROC beta = (REAL a, b) REAL

Beta function of arguments a and b, defined as  $\Gamma(a)\Gamma(b)/\Gamma(a+b)$ .

10. PROC incomplete beta = (REAL a, b, x) REAL

Normalised incomplete beta function of arguments a, b and x.

### 10.11.3 Airy functions

Next routines require that a68g is linked to the GNU Scientific Library.

1. PROC airy ai = (REAL x) REAL

Evaluates the airy function  $Ai(x)$ .

2. PROC airy bi = (REAL x) REAL

Evaluates the airy function  $Bi(x)$ .

3. PROC airy ai derivative = (REAL x) REAL

Evaluates the airy function  $Ai'(x)$ .

4. PROC airy bi derivative = (REAL x) REAL

Evaluates the airy function  $Bi'(x)$ .

### 10.11.4 Bessel functions

Next routines require that a68g is linked to the GNU Scientific Library.

1. PROC bessel jn = (REAL n, REAL x) REAL

Regular cylindrical Bessel function  $J_n(x)$ . Argument n must be integral.

2. PROC bessel yn = (REAL n, REAL x) REAL

Irregular cylindrical Bessel function  $Y_n(x)$ . Argument n must be integral.

3. PROC bessel in = (REAL n, REAL x) REAL

Regular modified cylindrical Bessel function  $I_n(x)$ . Argument n must be integral.

4. PROC bessel exp in = (REAL n, REAL x) REAL

Scaled regular modified cylindrical Bessel function  $e^{-|x|}I_n(x)$ . Argument n must be integral.

## PROGRAMMING WITH ALGOL 68 GENIE

5. PROC `bessel kn = (REAL n, REAL x) REAL`  
Irregular modified cylindrical Bessel function  $K_n(x)$ . Argument `n` must be integral.
6. PROC `bessel exp kn = (REAL n, REAL x) REAL`  
Scaled regular modified cylindrical Bessel function  $e^{-|x|}K_n(x)$ . Argument `n` must be integral.
7. PROC `bessel jl = (REAL l, REAL x) REAL` Evaluates the regular spherical Bessel function  $j_l(x)$ . Argument `l` must be integral.
8. PROC `bessel yl = (REAL l, REAL x) REAL`  
Irregular spherical Bessel function  $y_l(x)$ . Argument `l` must be integral.
9. PROC `bessel exp il = (REAL n, REAL x) REAL`  
Scaled regular modified spherical Bessel function  $e^{-|x|}i_l(x)$ . Argument `l` must be integral.
10. PROC `bessel exp kl = (REAL n, REAL x) REAL`  
Scaled irregular modified spherical Bessel function  $e^{-|x|}k_l(x)$ . Argument `l` must be integral.
11. PROC `bessel jnu = (REAL nu, REAL x) REAL`  
Fractional regular cylindrical Bessel function  $J_\nu(x)$ .
12. PROC `bessel ynu = (REAL nu, REAL x) REAL`  
Fractional irregular cylindrical Bessel function  $Y_\nu(x)$ .
13. PROC `bessel inu = (REAL nu, REAL x) REAL` Evaluates the fractional regular modified Bessel function  $J_\nu(x)$ .
14. PROC `bessel exp inu = (REAL nu, REAL x) REAL`  
Scaled fractional regular modified Bessel function  $e^{-|x|}I_\nu(x)$ .
15. PROC `bessel knu = (REAL nu, REAL x) REAL`  
Fractional irregular modified Bessel function  $Y_\nu(x)$ .
16. PROC `bessel exp knu = (REAL nu, REAL x) REAL`  
Scaled fractional irregular modified Bessel function  $e^{-|x|}K_\nu(x)$ .

## 10.11.5 Elliptic integrals

Next routines require that a68g is linked to the GNU Scientific Library.

1. PROC elliptic integral k = (REAL x) REAL  
Complete elliptic integral of the first kind  $K(x)$ .
2. PROC elliptic integral e = (REAL x) REAL  
Complete elliptic integral of the second kind  $E(x)$ .
3. PROC elliptic integral rf = (REAL x, y, z) REAL  
Carlson's elliptic integral of the first kind  $R_F(x, y, z)$ .
4. PROC elliptic integral rd = (REAL x, y, z) REAL  
Carlson's elliptic integral of the second kind  $R_D(x, y, z)$ .
5. PROC elliptic integral rj = (REAL x, y, z, rho) REAL  
Carlson's elliptic integral of the third kind  $R_J(x, y, z, \rho)$ .
6. PROC elliptic integral rc = (REAL x, y) REAL  
Carlson's elliptic integral of the third kind  $R_C(x, y)$ .

## 10.12 Linear algebra

Next routines require that a68g is linked to the GNU Scientific Library. These routines provide a simple vector and matrix interface to Algol 68 rows of mode:

```
[ ] REAL # vector #
[, ] REAL # matrix #
[] COMPLEX # complex vector #
[, ] COMPLEX # complex matrix #
```

Routines in this chapter convert Algol 68 rows to objects compatible with vector- and matrix formats used by BLAS routines. However, they are always passed to GSL routines.

### 10.12.1 Monadic-operators

1. OP + = ([ ] REAL u) [ ] REAL  
OP + = ([, ] REAL u) [, ] REAL  
OP + = ([ ] COMPLEX u) [ ] COMPLEX

## PROGRAMMING WITH ALGOL 68 GENIE

OP + = ([, ] COMPLEX u) [, ] COMPLEX

**Evaluate**  $+u$ .

2. OP - = ([] REAL u) [] REAL

OP - = ([, ] REAL u) [, ] REAL

OP - = ([] COMPLEX u) [] COMPLEX

OP - = ([, ] COMPLEX u) [, ] COMPLEX

**Evaluate**  $-u$ .

3. OP NORM = ([] REAL u) REAL

OP NORM = ([] COMPLEX u) REAL

**Euclidean norm of vector**  $u$ .

4. OP TRACE = ([, ] REAL u) REAL

OP TRACE = ([, ] COMPLEX u) COMPLEX

**Trace (sum of diagonal elements) of square matrix**  $u$ .

5. OP T = ([, ] REAL u) [, ] REAL

OP T = ([, ] COMPLEX u) [, ] COMPLEX

**Transpose of matrix**  $u$ . **GSL** requires the matrix to be square, unlike pseudo-operator **TRNSP**. Note that operator **T** yields a copy of the transpose of its argument, whereas pseudo-operator **TRNSP** yields a descriptor without copying its argument.

6. OP DET = ([, ] REAL u) REAL

OP DET = ([, ] COMPLEX u) COMPLEX

**Determinant of square matrix**  $u$  by *LU* decomposition.

7. OP INV = ([, ] REAL u) [, ] REAL

OP INV = ([, ] COMPLEX u) [, ] COMPLEX

**Inverse of square matrix**  $u$  by *LU* decomposition.

### 10.12.2 Dyadic-operators

1. OP = = ([] REAL u, v) BOOL

OP = = ([, ] REAL u, v) BOOL

OP = = ([] COMPLEX u, v) BOOL

OP = = ([, ] COMPLEX u, v) BOOL

**Evaluate whether**  $u$  equals  $v$ .



2. OP /= = ([ ] REAL u, v) BOOL  
 OP /= = ([, ] REAL u, v) BOOL  
 OP /= = ([ ] COMPLEX u, v) BOOL  
 OP /= = ([, ] COMPLEX u, v) BOOL

**Evaluate whether u does not equal v.**

3. OP DYAD = ([ ] REAL u, v) [, ] REAL  
 OP DYAD = ([ ] COMPLEX u, v) [, ] COMPLEX

**Dyadic (or tensor) product of u and v. The priority of DYAD is 3. This means that (assuming standard priorities) addition, subtraction, multiplication and division have priority over a dyadic product:**

$r + dr$  DYAD  $t * r$

**is equivalent to**

$(r + dr)$  DYAD  $(t * r)$ .

4. OP + = ([ ] REAL u, v) [ ] REAL  
 OP + = ([, ] REAL u, v) [, ] REAL  
 OP + = ([ ] COMPLEX u, v) [ ] COMPLEX  
 OP + = ([, ] COMPLEX u, v) [, ] COMPLEX

**Evaluate  $u + v$ .**

5. OP += ([ ] REAL u, v) [ ] REAL  
 OP += ([, ] REAL u, v) [, ] REAL  
 OP += ([ ] COMPLEX u, v) [ ] COMPLEX  
 OP += ([, ] COMPLEX u, v) [, ] COMPLEX

**Evaluate  $u := u + v$ .**

6. OP - = ([ ] REAL u, v) [ ] REAL  
 OP - = ([, ] REAL u, v) [, ] REAL  
 OP - = ([ ] COMPLEX u, v) [ ] COMPLEX  
 OP - = ([, ] COMPLEX u, v) [, ] COMPLEX

**Evaluate  $u - v$ .**

7. OP -= ([ ] REAL u, v) [ ] REAL  
 OP -= ([, ] REAL u, v) [, ] REAL  
 OP -= ([ ] COMPLEX u, v) [ ] COMPLEX  
 OP -= ([, ] COMPLEX u, v) [, ] COMPLEX

**Evaluate  $u := u - v$ .**

## PROGRAMMING WITH ALGOL 68 GENIE

8. OP \* = ([ ] REAL u, REAL v) [ ] REAL  
OP \* = ([, ] REAL u, REAL v) [, ] REAL  
OP \* = ([ ] COMPLEX u, COMPLEX v) [ ] COMPLEX  
OP \* = ([, ] COMPLEX u, COMPLEX v) [, ] COMPLEX

**Scaling of u by a scalar v:**  $u \times v$ .

9. OP \* = (REAL u, [ ] REAL v) [ ] REAL  
OP \* = (REAL u, [, ] REAL v) [, ] REAL  
OP \* = (COMPLEX u, [ ] COMPLEX v) [ ] COMPLEX  
OP \* = (COMPLEX u, [, ] COMPLEX v) [, ] COMPLEX

**Scaling of u by a scalar v:**  $u \times v$ .

10. OP \* = ([, ] REAL u, [ ] REAL v) [ ] REAL  
OP \* = ([, ] COMPLEX u, [ ] COMPLEX v) [ ] COMPLEX

**Matrix-vector product**  $u \cdot v$ .

11. OP \* = ([ ] REAL u, [, ] REAL v) [ ] REAL  
OP \* = ([ ] COMPLEX u, [, ] COMPLEX v) [ ] COMPLEX

**Vector-matrix product**  $u \cdot v$ , that equals  $v^T \cdot u$ .

12. OP \* = ([ ] REAL u, v) REAL  
OP \* = ([ ] COMPLEX u, v) COMPLEX

**Inner product of**  $u \cdot v$ .

13. OP \* = ([, ] REAL u, [, ] REAL v) [, ] REAL  
OP \* = ([, ] COMPLEX u, [, ] COMPLEX v) [, ] COMPLEX

**Matrix-matrix product**  $u \cdot v$ .

14. OP \*:= = (REF [ ] REAL u, REAL v) REF [ ] REAL  
OP \*:= = (REF [, ] REAL u, REAL v) REF [, ] REAL  
OP \*:= = (REF [ ] COMPLEX u, COMPLEX v) REF [ ] COMPLEX  
OP \*:= = (REF [, ] COMPLEX u, COMPLEX v) REF [, ] COMPLEX

**Scaling by a scalar**  $u := u * v$ .

15. OP / = ([ ] REAL u, REAL v) [ ] REAL  
OP / = ([, ] REAL u, REAL v) [, ] REAL  
OP / = ([ ] COMPLEX u, COMPLEX v) [ ] COMPLEX  
OP / = ([, ] COMPLEX u, COMPLEX v) [, ] COMPLEX

**Scaling by a scalar**  $u/v$ .

16. OP /:= = (REF [] REAL u, REAL v) REF [] REAL  
 OP /:= = (REF [, ] REAL u, REAL v) REF [, ] REAL  
 OP /:= = (REF [] COMPLEX u, COMPLEX v) REF [] COMPLEX  
 OP /:= = (REF [, ] COMPLEX u, COMPLEX v) REF [, ] COMPLEX  
 Evaluate `u := u / v.`

## 10.13 Solution of linear algebraic equations

Next routines require that `a68g` is linked to the GNU Scientific Library. This paragraph describes routines for solving linear algebraic equations. These are intended for "small" linear equations where simple algorithms are acceptable. Should you be interested in solving large linear equations please refer to for instance LAPACK.

### 10.13.1 *LU* decomposition through Gaussian elimination

A square matrix has an *LU* decomposition into upper and lower triangular matrices

$$P \cdot A = L \cdot U$$

where *A* is a square matrix, *P* is a permutation matrix, *L* is a unit lower triangular matrix and *U* is an upper triangular matrix. For square matrices this decomposition can be used to convert the linear equation  $A \cdot x = b$  into a pair of triangular equations

$$L \cdot y = P \cdot b, U \cdot x = y$$

which can be solved by forward- and back-substitution.

The algorithm that GSL uses in *LU* decomposition is Gaussian elimination with partial pivoting. Advantages of *LU* decomposition are that it works for any square matrix and will efficiently produce all solutions for a linear equation. A disadvantage of *LU* decomposition is that that it cannot find approximate (least-square) solutions in case of singular or ill-conditioned matrices, that are better solved by a singular value decomposition.

1. PROC `lu decomp =`  
`([, ] REAL a, REF [] INT p, REF INT sign) [, ] REAL`  
 PROC `complex lu decomp =`  
`([, ] COMPLEX a, REF [] INT p, REF INT sign) [, ] COMPLEX`

These routines factorise the square matrix *a* into the *LU* decomposition  $P \cdot A = L \cdot U$ . The diagonal and upper triangular part of the returned matrix contain *U*. The lower triangular part of the returned matrix holds *L*. Diagonal elements of *L* are unity, and

are not stored. Permutation matrix  $P$  is encoded in the permutation  $p$ . The **sign** of the permutation is given by  $sign$ . It has the value  $(-1)^n$ , where  $n$  is the number of row interchanges in the permutation.

2. PROC lu solve = ([, ] REAL a, lu, [] INT p, [] REAL b) [] REAL  
 PROC complex lu solve =  
 ([, ] COMPLEX a, lu, [] INT p, [] COMPLEX b) [] COMPLEX

These routines solve the equation  $A \cdot x = b$  and apply an iterative improvement to  $x$ , using the  $LU$  decomposition of  $a$  into  $lu, p$  as calculated by [complex] lu decomp.

3. PROC lu inv = ([, ] REAL lu, [] INT p) [, ] REAL  
 PROC complex lu inv = ([, ] COMPLEX lu, [] INT p) [, ] COMPLEX

These routines yield the inverse of a matrix from its  $LU$  decomposition  $lu, p$  as calculated by [complex] lu decomp. The inverse is computed by solving  $A \cdot A^{inv} = 1$ . It is not recommended to use the inverse matrix to solve a linear equation  $A \cdot x = b$  by applying  $x = A^{inv} \cdot b$ ; use [complex] lu solve for better precision instead.

4. PROC lu det = ([, ] REAL lu, INT sign) REAL  
 PROC complex lu det = ([, ] COMPLEX lu, INT sign) COMPLEX

These routines yield the determinant of a matrix from its  $LU$  decomposition  $lu, sign$  as calculated by [complex] lu decomp. The determinant is computed as the product of the diagonal elements of  $U$  and the **sign** of the row permutation  $sign$ .

### 10.13.2 Singular value decomposition

A  $M \times N$  matrix  $A$  has a singular value decomposition in the product of a  $M \times N$  orthogonal matrix  $U$ , a  $N \times N$  diagonal matrix  $S$  and the transpose of a  $N \times N$  orthogonal square matrix  $V$ , such that

$$A = U \cdot S \cdot V^T$$

The singular values  $S_{ii}$  are zero or positive, off-diagonal elements are zero.

Singular value decomposition has many practical applications. The ratio of the largest to the smallest singular value is called the condition number; a high ratio means that the equation is ill conditioned. Zero singular values indicate a singular matrix  $A$ , the number of non-zero singular values is the rank of  $A$ . Small singular values should be edited by choosing a suitable tolerance since finite numerical precision may result in a singular value to be close to, but not equal to, zero.

1. PROC sv decomp =  
 ([, ] REAL a, REF [, ] REAL v, REF [] REAL s) [, ] REAL

This routine factorises a  $M \times N$  matrix  $A$  into the singular value decomposition for  $M \geq N$ . The routine yields  $U$ . The diagonal elements of singular value matrix  $S$  are stored in vector  $s$ . The singular values are non-negative and form a non-increasing sequence from  $s_1$  to  $s_N$ . Matrix  $v$  contains on completion the elements of  $V$  in untransposed form. To form the product

$$U \cdot S \cdot V^T$$

it is necessary to take the transpose of  $V$ . This routine uses the Golub-Reinsch SVD algorithm.

2. PROC svd solve = ([, ] REAL u, v, [] REAL s, [] REAL b) [] REAL

This routine solves the system  $A \cdot x = b$  using the singular value decomposition  $u, s, v$  of  $A$  computed by svd decomp. Only non-zero singular values are used in calculating the solution. The parts of the solution corresponding to singular values of zero are ignored. Other singular values can be edited out by setting them to zero before calling this routine. In the overdetermined case where  $A$  has more rows than columns the routine returns solution  $x$  which minimises  $\|A \cdot x - b\|^2$ .

### 10.13.3 QR decomposition

A  $M \times N$  matrix  $A$  has a decomposition into the product of an orthogonal  $M \times M$  square matrix  $Q$ , where  $Q^T \cdot Q = I$ , and an  $M \times N$  right-triangular matrix  $R$ , such that  $A = Q \cdot R$ . A linear equation  $A \cdot x = b$  can be converted to the triangular equation  $R \cdot x = Q^T \cdot b$ , which can be solved by back-substitution.

QR decomposition can be used to determine an orthonormal basis for a set of vectors since the first  $N$  columns of  $Q$  form an orthonormal basis for the range of  $A$ .

1. PROC qr decomp = ([, ] REAL a, REF [] REAL t) [, ] REAL

This routine factorises the  $M \times N$  matrix  $a$  into the QR decomposition  $A = Q \cdot R$ . The diagonal and upper triangular part of the returned matrix contain matrix  $R$ . The vector  $t$  and the columns of the lower triangular part of matrix  $a$  contain the Householder coefficients and Householder vectors which encode the orthogonal matrix  $Q$ . Vector  $t$  must have length  $\min(M, N)$ . The algorithm used to perform the decomposition is Householder QR (Golub and Van Loan, *Matrix Computations*, Algorithm 5.2.1).

2. PROC qr solve ([, ] REAL a, [] REAL t, [] REAL b) [] REAL

This routine solves the square system  $A \cdot x = b$  using the QR decomposition of  $A$  into  $a, t$  computed by qr decomp. The least-squares solution for rectangular equations can be found using qr ls solve.

3. PROC qr ls solve ([, ] REAL a, [] REAL t, [] REAL b) [] REAL

This routine finds the least squares solution to the overdetermined system  $A \cdot x = b$  where  $M \times N$  matrix  $A$  has more rows than columns, id est  $M > N$ . The least squares solution minimises the Euclidean norm of the residual  $\|A \cdot x - b\|^2$ . The routine uses the QR decomposition of  $A$  into  $a, t$  computed by qr decomp.

### 10.13.4 Cholesky decomposition

A symmetric, positive definite square matrix  $A$  has a Cholesky decomposition into a product of a lower triangular matrix  $L$  and its transpose  $L^T$ ,  $A = L \cdot L^T$ . This is sometimes referred to as taking the square root of a matrix. The Cholesky decomposition can only be carried out when all the eigen values of the matrix are positive. This decomposition can be used to convert the linear system  $A \cdot x = b$  into a pair of triangular equations  $L \cdot y = b$ ,  $L^T \cdot x = y$ , which can be solved by forward and back-substitution.

1. PROC cholesky decomp = ([, ] REAL a) [, ] REAL

This routine factorises the positive-definite symmetric square matrix  $a$  into the Cholesky decomposition  $A = L \cdot L^T$ . The diagonal and lower triangular part of the returned matrix contain matrix  $L$ . The upper triangular part of the input matrix contains  $L^T$ , the diagonal terms being identical for both  $L$  and  $L^T$ . If the matrix is not positive-definite then the decomposition will fail.

2. PROC cholesky solve = ([, ] REAL a, [] REAL b) [] REAL

This routine solves the system  $A \cdot x = b$  using the Cholesky decomposition of  $A$  into matrix  $a$  computed by cholesky decomp.

## 10.14 Fourier transforms

This section describes mixed-radix discrete fast Fourier transform (FFT) algorithms for complex data. Next routines require that `a68g` is linked to the GNU Scientific Library. The forward transform (to the time domain) is defined by

$$F_k = \sum_{j=0}^{N-1} e^{-2\pi i k j / N} f_j$$

while the inverse transform (to the frequency domain) is defined by

$$f_k = \frac{1}{N} \sum_{j=0}^{N-1} e^{2\pi i k j / N} F_j$$

and the backward transform is an unscaled inverse transform defined by

$$f_k = \sum_{j=0}^{N-1} e^{2\pi i k j / N} F_j$$

The mixed-radix procedures work for FFTs of any length. They are a reimplementaion of Paul Swarztrauber's FFTPACK library.

The mixed-radix algorithm is based on sub-transform modules - highly optimised small length FFTs which are combined to create larger FFTs. There are efficient modules for factors of 2, 3, 4, 5, 6 and 7. The modules for the composite factors of 4 and 6 are faster than combining the modules for  $2 \times 2$  and  $2 \times 3$ .

For factors which are not implemented as modules there is a general module which uses Singleton's method. Lengths which use the general module will still be factorised as much as possible. For example, a length of 143 will be factorised into  $11 \times 13$ . Large prime factors, e.g. 99991, should be avoided because of the  $\mathcal{O}(n^2)$  complexity of the general module. The procedure `prime factors` can be used to detect inefficiencies in computing a FFT.

For physical applications it is important to note that the index appearing in the discrete Fourier transform (DFT) does not correspond directly to a physical frequency. If the time-step of the DFT reads `dt` then the frequency-domain includes both positive and negative frequencies, ranging from  $-1/(2dt)$  to  $+1/(2dt)$ . The positive frequencies are stored from the beginning of the row up to the middle, and the negative frequencies are stored backwards from the end of the row. When  $N$  is even, next frequency table holds:

| Index     | Time          | Frequency          |
|-----------|---------------|--------------------|
| 1         | 0             | 0                  |
| 2         | $dt$          | $1/(Ndt)$          |
| ...       | ...           | ...                |
| $N/2$     | $(N/2 - 1)dt$ | $(N/2 - 1)/(Ndt)$  |
| $N/2 + 1$ | $(N/2)dt$     | $\pm 1/(2dt)$      |
| $N/2 + 2$ | $(N/2 + 1)dt$ | $-(N/2 - 1)/(Ndt)$ |
| ...       | ...           | ...                |
| $N - 1$   | $(N - 2)dt$   | $-2/(Ndt)$         |
| $N$       | $(N - 1)dt$   | $-1/(Ndt)$         |

When the length  $N$  of the row is even, the location  $N/2 + 1$  contains the most positive and negative frequencies  $\pm 1/(2dt)$  that are equivalent. If  $N$  is odd, this central value does not appear, but the general structure of above table still applies.

1. PROC `prime factors = (INT n) [] INT`

Factorises argument `n` and yields the factors as a row of integral values. This routine can be used to determine whether the Fourier transform of a row of length `n` can be calculated in an efficient way.

## PROGRAMMING WITH ALGOL 68 GENIE

```
2. PROC fft complex forward = ([] COMPLEX) [] COMPLEX
PROC fft complex backward = ([] COMPLEX) [] COMPLEX
PROC fft complex inverse = ([] COMPLEX) [] COMPLEX
PROC fft forward = ([] REAL) [] COMPLEX
PROC fft backward = ([] COMPLEX) [] REAL
PROC fft inverse = ([] COMPLEX) [] REAL
```

These procedures compute forward, backward (i.e., an unscaled inverse) and inverse FFTs, using a mixed radix decimation-in-frequency algorithm. There is no restriction on the length of the argument rows. Efficient modules are provided for sub transforms of length 2, 3, 4, 5, 6 and 7. Any remaining factors are computed with a slow,  $O(N^2)$ , module.

## 10.15 Laplace transform

Next routine requires the GNU Scientific Library. The Laplace transform  $F(s)$  of a real-valued function  $f(t)$  is defined by

$$F(s) = \int_0^{\infty} dt f(t) e^{-st}$$

a68g can calculate this transform:

```
1. PROC laplace = (PROC (REAL) REAL f, REAL s, REF REAL error) REAL
```

This procedure computes the Laplace transform of  $f$  at value  $s$ . Argument *error* must be set to the desired error in the integral at beforehand. If *error* is positive, it is taken as an absolute error. If *error* is negative, its absolute value is passed as a relative error. Upon return, *error* contains the estimated *absolute* error in the calculated integral. Routine `laplace` calls a GSL function to compute an integral over the semi-infinite interval  $[0, \infty]$  by mapping onto the semi-open interval  $[0, 1]$  using the transformation

$$\int_0^{\infty} dt g(t) = \int_0^1 du g\left(\frac{1-u}{u}\right) \frac{1}{u^2}$$

For example:

```
PROC exp nx = (INT n, REAL x) REAL: exp (-n * x);

FOR n TO 3
DO print ("exp (-", whole (n, 0), "t)", new line));
```



```

FOR k TO 10
DO REAL error := 1e-12;
  print ((laplace (exp nx (n, ), k, error), blank, error,
        blank, 1 / (n + k), new line))
OD
OD;

```

## 10.16 Constants

This paragraph describes physical constants such as the speed of light or the gravitational constant. The constants are available in the MKSA system (meter-kilograms-seconds-ampere) and the CGS system (centimetre-grams-seconds).

### 10.16.1 Fundamental constants

1. REAL mksa speed of light • REAL cgs speed of light  
Speed of light in vacuum,  $c$ .
2. REAL mksa vacuum permeability  
Permeability of vacuum  $\mu_0$ .
3. REAL mksa vacuum permittivity  
Permittivity of free space  $\epsilon_0$ .
4. REAL num avogadro  
Avogadro's number  $N_a$ .
5. REAL mksa faraday • REAL cgs faraday  
Molar charge of one Faraday.
6. REAL mksa boltzmann • REAL cgs boltzmann  
Boltzmann's constant  $k_B$ .
7. REAL mksa molar gas • REAL cgs molar gas  
Molar gas constant  $R$ .
8. REAL mksa standard gas volume • REAL cgs standard gas volume  
Standard gas volume  $V_0$ .
9. REAL mksa planck constant • REAL cgs planck constant  
Planck constant  $h$ .
10. REAL mksa planck constant bar • REAL cgs planck constant bar  
Planck constant  $\hbar = h/(2\pi)$ .

## PROGRAMMING WITH ALGOL 68 GENIE

11. REAL mksa gauss • REAL cgs gauss  
Magnetic field strength of one Gauss.

### 10.16.2 Length and velocity

1. REAL mksa micron • REAL cgs micron  
Length of one micron.
2. REAL mksa hectare • REAL cgs hectare  
Area of one hectare.
3. REAL mksa miles per hour • REAL cgs miles per hour  
Speed of one mile per hour.
4. REAL mksa kilometres per hour • REAL cgs kilometres per hour  
Speed of one kilometre per hour.

### 10.16.3 Astronomy and astrophysics

1. REAL mksa astronomical unit • REAL cgs astronomical unit  
Length of one astronomical unit *au*.
2. REAL mksa gravitational constant •  
REAL cgs gravitational constant  
Gravitational constant *G*.
3. REAL mksa light year • REAL cgs light year  
Distance of one light-year *ly*.
4. REAL mksa parsec • REAL cgs parsec  
Distance of one parsec *pc*.
5. REAL mksa grav accel • REAL cgs grav accel  
Gravitational acceleration on Earth *g*.
6. REAL mksa solar mass • REAL cgs solar mass  
Mass of the Sun.

### 10.16.4 Atomic and nuclear physics

1. REAL mksa electron charge • REAL cgs electron charge  
Charge of an electron *e*.

## STANDARD-PRELUDE AND LIBRARY-PRELUDE

2. REAL mksa electron volt • REAL cgs electron volt  
**Energy of one electron volt  $eV$ .**
3. REAL mksa unified atomic mass • REAL cgs unified atomic mass  
**Unified atomic mass  $amu$ .**
4. REAL mksa mass electron • REAL cgs mass electron  
**Mass of an electron  $m_e$ .**
5. REAL mksa mass muon • REAL cgs mass muon  
**Mass of a muon  $m_\mu$ .**
6. REAL mksa mass proton • REAL cgs mass proton  
**Mass of a proton  $m_p$ .**
7. REAL mksa mass neutron • REAL cgs mass neutron  
**Mass of a neutron  $m_n$ .**
8. REAL num fine structure  
**Electromagnetic fine structure constant,  $\alpha$ .**
9. REAL mksa rydberg • REAL cgs rydberg  
**The Rydberg constant  $Ry$  in units of energy. This is related to the Rydberg inverse wavelength  $R$  by  $Ry = hcr$ .**
10. REAL mksa bohr radius • REAL cgs bohr radius  
**Bohr radius  $a_0$ .**
11. REAL mksa angstrom • REAL cgs angstrom  
**Length of one Ångstrom.**
12. REAL mksa barn • REAL cgs barn  
**Area of one barn.**
13. REAL mksa bohr magneton • REAL cgs bohr magneton  
**Bohr magneton  $\mu_b$ .**
14. REAL mksa nuclear magneton • REAL cgs nuclear magneton  
**Nuclear magneton  $\mu_n$ .**
15. REAL mksa electron magnetic moment •  
REAL cgs electron magnetic moment  
**Absolute value of the magnetic moment of an electron  $\mu_e$ . The physical magnetic moment of an electron is negative.**
16. REAL mksa proton magnetic moment •  
REAL cgs proton magnetic moment  
**Magnetic moment of a proton  $\mu_p$ .**

### 10.16.5 Time

1. REAL mksa minute • REAL cgs minute  
Number of seconds in one minute.
2. REAL mksa hour • REAL cgs hour  
Number of seconds in one hour.
3. REAL mksa day • REAL cgs day  
Number of seconds in one day.
4. REAL mksa week • REAL cgs week  
Number of seconds in one week.

### 10.16.6 Imperial units

1. REAL mksa inch • REAL cgs inch  
Length of one inch.
2. REAL mksa foot • REAL cgs foot  
Length of one foot.
3. REAL mksa yard • REAL cgs yard  
Length of one yard.
4. REAL mksa mile • REAL cgs mile  
Length of one mile.
5. REAL mksa mil • REAL cgs mil  
Length of one mil (1/1000th of an inch).

### 10.16.7 Nautical units

1. REAL mksa nautical mile • REAL cgs nautical mile  
Length of one nautical mile.
2. REAL mksa fathom • REAL cgs fathom  
Length of one fathom.
3. REAL mksa knot • REAL cgs knot  
Speed of one knot.

### 10.16.8 Volume

1. REAL mksa acre • REAL cgs acre  
Area of one acre.
2. REAL mksa liter • REAL cgs liter  
Volume of one liter.
3. REAL mksa us gallon • REAL cgs us gallon  
Volume of one us gallon.
4. REAL mksa canadian gallon • REAL cgs canadian gallon  
Volume of one canadian gallon.
5. REAL mksa uk gallon • REAL cgs uk gallon  
Volume of one uk gallon.
6. REAL mksa quart • REAL cgs quart  
Volume of one quart.
7. REAL mksa pint • REAL cgs pint  
Volume of one pint.

### 10.16.9 Mass and weight

1. REAL mksa pound mass • REAL cgs pound mass  
Mass of one pound.
2. REAL mksa ounce mass • REAL cgs ounce mass  
Mass of one ounce.
3. REAL mksa ton • REAL cgs ton  
Mass of one ton.
4. REAL mksa metric ton • REAL cgs metric ton  
Mass of one metric ton (1000 kg).
5. REAL mksa uk ton • REAL cgs uk ton  
Mass of one uk ton.
6. REAL mksa troy ounce • REAL cgs troy ounce  
Mass of one troy ounce.
7. REAL mksa carat • REAL cgs carat  
Mass of one carat.

## PROGRAMMING WITH ALGOL 68 GENIE

8. REAL mksa gram force • REAL cgs gram force  
Force of one gram weight.
9. REAL mksa pound force • REAL cgs pound force  
Force of one pound weight.
10. REAL mksa kilopound force • REAL cgs kilopound force  
Force of one kilopound weight.
11. REAL mksa poundal • REAL cgs poundal  
Force of one poundal.

### 10.16.10 Thermal energy and power

1. REAL mksa calorie • REAL cgs calorie  
Energy of one Calorie.
2. REAL mksa btu • REAL cgs btu  
Energy of one British Thermal Unit.
3. REAL mksa therm • REAL cgs therm  
Energy of one therm.
4. REAL mksa horsepower • REAL cgs horsepower  
Power of one horsepower.

### 10.16.11 Pressure

1. REAL mksa bar • REAL cgs bar  
Pressure of one Bar.
2. REAL mksa std atmosphere • REAL cgs std atmosphere  
Pressure of one standard atmosphere.
3. REAL mksa torr • REAL cgs torr  
Pressure of one Torricelli.
4. REAL mksa meter of mercury • REAL cgs meter of mercury  
Pressure of one meter of mercury.
5. REAL mksa inch of mercury • REAL cgs inch of mercury  
Pressure of one inch of mercury.
6. REAL mksa inch of water • REAL cgs inch of water  
Pressure of one inch of water.

7. REAL mksa psi • REAL cgs psi  
pressure of one pound per square inch.

### 10.16.12 Viscosity

1. REAL mksa poise • REAL cgs poise  
Dynamic viscosity of one Poise.
2. REAL mksa stokes • REAL cgs stokes  
Kinematic viscosity of one Stokes.

### 10.16.13 Light and illumination

1. REAL mksa stilb • REAL cgs stilb  
Luminance of one stilb.
2. REAL mksa lumen • REAL cgs lumen  
Luminous flux of one lumen.
3. REAL mksa lux • REAL cgs lux  
Illuminance of one lux.
4. REAL mksa phot • REAL cgs phot  
Illuminance of one phot.
5. REAL mksa footcandle • REAL cgs footcandle  
Illuminance of one footcandle.
6. REAL mksa lambert • REAL cgs lambert  
Luminance of one lambert.
7. REAL mksa footlambert • REAL cgs footlambert  
Luminance of one footlambert.

### 10.16.14 Radioactivity

1. REAL mksa curie • REAL cgs curie  
Activity of one Curie.
2. REAL mksa roentgen • REAL cgs roentgen  
Exposure of one Röntgen.
3. REAL mksa rad • REAL cgs rad  
Absorbed dose of one Rad.

### 10.16.15 Force and energy

1. REAL mksa newton • REAL cgs newton  
SI unit of force, one Newton.
2. REAL mksa dyne • REAL cgs dyne  
Force of one Dyne.
3. REAL mksa joule • REAL cgs joule  
SI unit of energy, one Joule.
4. REAL mksa erg • REAL cgs erg  
Energy of one Erg.

### 10.16.16 Prefixes

1. REAL num yotta  $10^{24}$
2. REAL num zetta  $10^{21}$
3. REAL num exa  $10^{18}$
4. REAL num peta  $10^{15}$
5. REAL num tera  $10^{12}$
6. REAL num giga  $10^9$
7. REAL num mega  $10^6$
8. REAL num kilo  $10^3$
9. REAL num milli  $10^{-3}$
10. REAL num micro  $10^{-6}$
11. REAL num nano  $10^{-9}$
12. REAL num pico  $10^{-12}$
13. REAL num femto  $10^{-15}$
14. REAL num atto  $10^{-18}$
15. REAL num zepto  $10^{-21}$
16. REAL num yocto  $10^{-24}$



### 10.16.17 Printer units

1. `REAL mksa point • REAL cgs point`  
Length of one printer's point 1/72".
2. `REAL mksa texpoint • REAL cgs texpoint`  
Length of one `TEX` point 1/72.27".

## 10.17 Linux extensions

`a68g` maps to Linux as to provide basic means with which a **program** can

1. access command line arguments,
2. access directory - and file information,
3. access environment variables,
4. access system error information,
5. start child processes and communicate with them by means of pipes. `a68g` transput procedures as `getf` or `putf` operate on pipes,
6. send a request to a HTTP server, fetch web page contents, and match regular expressions in a string.

### 10.17.1 General definitions

1. `MODE PIPE = STRUCT (REF FILE read, write, INT pid)`  
Used to set up communication between processes. See for instance `execve child pipe`.
2. `PROC utc time = [] INT`  
Returns UTC calendar time, or an empty row if the function fails. Respective array elements are year, month, day, hours, minutes, seconds, day of week and daylight-saving-time flag.
3. `PROC local time = [] INT`  
Returns local calendar time, or an empty row if the function fails. Respective array elements are year, month, day, hours, minutes, seconds, day of week and daylight-saving-time flag.
4. `PROC argc = INT`  
Returns the number of command-line arguments.

## PROGRAMMING WITH ALGOL 68 GENIE

5. PROC argv = (INT k) STRING  
Returns command-line argument `k` or an empty string when `k` is not valid.
6. PROC get env = (STRING var) STRING  
Returns the value of environment variable `var`.

For accessing system error information through `errno` next routines are available:

1. PROC reset errno = VOID  
Resets the global error variable `errno`.
2. PROC errno = INT Returns the global error variable `errno`.
3. PROC strerror = (INT errno) STRING  
Returns a string describing the error code passed as `errno`.

### 10.17.2 File system

1. PROC file is block device = (STRING file) BOOL  
Returns whether `file` is a block device. If the file does not exist or if an error occurs, FALSE is returned and `errno` is set.
2. PROC file is char device = (STRING file) BOOL  
Returns whether `file` is a character device. If the file does not exist or if an error occurs, FALSE is returned and `errno` is set.
3. PROC file is directory = (STRING file) BOOL  
Returns whether `file` is a directory. If the file does not exist or if an error occurs, FALSE is returned and `errno` is set.
4. PROC file is regular = (STRING file) BOOL  
Returns whether `file` is a regular file. If the file does not exist or if an error occurs, FALSE is returned and `errno` is set.
5. PROC file is fifo = (STRING file) BOOL  
Returns whether `file` is a first-in-first-out file, for instance a pipe. If the file does not exist or if an error occurs, FALSE is returned and `errno` is set.
6. PROC file is link = (STRING file) BOOL  
Returns whether `file` is a first-in-first-out file, for instance a pipe. If the file does not exist or if an error occurs, FALSE is returned and `errno` is set.
7. PROC file mode = (STRING file) BITS  
Yields file mode of file `file`. If the file does not exist or if an error occurs, `2r0` is returned and `errno` is set.

8. PROC get pwd = STRING  
Returns the current working directory.
9. PROC set pwd = (STRING dir) VOID  
Sets the current working directory to *dir*, which must be a valid directory name.
10. PROC get directory = (STRING dir) [] STRING  
Retrieves the file names from directory *dir*, which must be a valid directory name. Note that the order of entries in the returned [] STRING is arbitrary. Example:  

```
PROC ls = (STRING dir name) VOID:
  IF is directory (dir name)
  THEN printf (($lg$, SORT get directory (dir name)))
  ELSE print ("'", dir name, "' is not a directory")
  FI
```

### 10.17.3 Processes

1. PROC fork = INT  
Creates a child process that differs from the parent process only in its PID and PPID, and that resource utilisations are set to zero. File locks and pending signals are not inherited. Under Linux, fork is implemented using copy-on-write pages, so the only penalty incurred by fork is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child. On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and *errno* will be set appropriately.
2. PROC wait pid = (INT pid) VOID  
Waits until child process *pid* ends.
3. PROC execve =  
(STRING filename, [] STRING argv, [] STRING envp) INT  
Executes *filename* that must be either a binary executable, or a script starting with a line of the form:  

```
#! interpreter [argv]
```

In the latter case, *interpreter* must a valid path-name for an executable which is not itself a script, which will be invoked as *interpreter [arg] filename*. Argument *argv* is a row of argument strings that is passed to the new program. Argument *envp* is a row of strings, with elements conventionally of the form *KEY=VALUE*, which are passed as environment to the new program. The argument vector and environment can be accessed by the called program's *main* function, when it is defined as:

```
int main (int argc, char *argv[], char *envp[])
```

The procedure does not return on success, and the text, data, bss, and stack of the

## PROGRAMMING WITH ALGOL 68 GENIE

calling process are overwritten by that of the program loaded. The program invoked inherits the calling process's PID, and any open file descriptors that are not set to close on exec. Signals pending on the calling process are cleared. Any signals set to be caught by the calling process are reset to their default behaviour. The SIGCHLD signal (when set to SIG\_IGN) may or may not be reset to SIG\_DFL.

4. PROC `execve child =`  
(STRING filename, [] STRING argv, [] STRING envp) INT  
Executes a command in a child process. Algol 68 Genie is not superseded by the new process. For details on arguments see `execve`.

5. PROC `execve output =`  
(STRING filename, [] STRING argv, [] STRING envp,  
REF STRING output) INT  
Executes a command in a child process. Algol 68 Genie is not superseded by the new process. For details on arguments see `execve`. Output written to `stdout` by the child process is collected in a string which is assigned to `output` unless `output` is NIL. The routine yields the return code of the child process on success or -1 on failure. Next code fragment shows how to execute a command in the shell and retrieve output from both `stdout` and `stderr`:

```
PROC get output = (STRING cmd) VOID:  
  IF STRING sh cmd = " " + cmd + " ; 2>&1";  
    STRING output;  
    execve output ("/bin/sh", ("sh", "-c", sh cmd),  
      "", output) >= 0  
  THEN print (output)  
  FI;
```

```
get output ("ps | grep a68g")
```

6. PROC `execve child pipe =`  
(STRING filename, [] STRING argv, [] STRING envp) PIPE  
Executes a command in a child process. Algol 68 Genie is not superseded by the new process. For details on arguments see `execve`. The child process redirects standard input and standard output to the pipe that is delivered to the parent process. Therefore, the parent can write to the `write` field of the pipe, and this data can be read by the child from standard input. The child can write to standard output, and this data can be read by the parent process from the `read` field of the pipe. The files in the pipe are opened. The channels assigned to the read - and write end of the pipe are `standin channel` and `standout channel`, respectively. This procedure calls `execve` in the child process, for a description of the arguments refer to that procedure. For example:

```
PROC eop handler = (REF FILE f) BOOL:  
  BEGIN put (stand error, ("end of pipe", new line));
```

## STANDARD-PRELUDE AND LIBRARY-PRELUDE

```
GO TO end
END;

PIPE p = execve child pipe ("/bin/cat", ("cat", program idf), "");

IF pid OF p < 0
THEN put (stand error, ("pipe not valid: ", strerror (errno)));
stop
FI;

on logical file end (read OF p, eop handler);

DO STRING line;
get (read OF p, (line, newline));
put (stand out, ("From pipe:  ", line, " ", newline))
OD;

end:
close (read OF p);
close (write OF p)
```

### 10.17.4 Fetching web page contents and sending requests

1. PROC http content =  
(REF STRING content, STRING domain, STRING path, INT port) INT  
**Gets content from web page specified by domain and path, and stores it in content. Both domain and path must be properly MIME encoded. The routine connects to server domain using TCP and sends a straightforward GET request for path using HTTP 1.0, and reads the reply from the server. If port number port is 0 the default port number (80) will be used. The procedure returns 0 on success or an appropriate error code upon failure. For example:**

```
PROC good page = (REF STRING page) BOOL:
IF grep in string ("^HTTP/[0-9.]* 200", page, NIL, NIL) = 0
THEN TRUE
ELSE IF INT start, end;
grep in string ("^HTTP/[0-9.]* [0-9]+ [a-zA-Z ]*", page,
start, end) = 0
THEN print (page[start : end])
ELSE print ("unknown error retrieving page")
FI;
```

## PROGRAMMING WITH ALGOL 68 GENIE

```
FALSE
FI;

IF STRING reply;
  INT rc =
    http content (reply, "www.gnu.org", "http://www.gnu.org/", 0);
  rc = 0 AND good page (reply)
THEN print (reply)
ELSE print (strerror (rc))
FI
```

2. PROC tcp request =  
(REF STRING reply, STRING domain, STRING request, INT port) INT  
**Send request request to server specified by domain, and stores the server's reply in reply. Both domain and request must be properly MIME encoded. The routine connects to server domain using TCP. If port number port is 0 the default port number (80) will be used. The procedure returns 0 on success or an appropriate error code upon failure. For example:**

```
STRING reply;
IF INT rc = tcp request (reply, "www.kernel.org",
  "GET / HTTP/1.0" + newline char +
  "Range: bytes=0-1023" + 2 * newline char, 0);
  rc = 0
THEN print (reply)
ELSE print (strerror (rc))
FI;
```

## 10.18 Regular expressions in string manipulation

1. PROC grep in string =  
(STRING pattern, string, REF INT start, end) INT  
**Matches pattern with string and assigns first character position (counting from LWB string) and last character position of the widest leftmost matching sub string to start and end when they are not NIL. The routine yields 0 on match, 1 on no-match, 2 on out-of-memory error and 3 on other errors. If the string contains newline character, it is considered as end-of-line ("\$\$") and the following character is considered as start-of-line ("^"). Compare this function with AWK function match.**
2. PROC sub in string =  
(STRING pattern, replacement, REF STRING string) INT  
**Matches pattern with string and replaces the widest leftmost matching sub string with**

replacement. The routine yields 0 on match, 1 on no-match, 2 on out-of-memory error and 3 on other errors. If the string contains newline character, it is considered as end-of-line ("\$\$") and the following character is considered as start-of-line ("^"). Compare this function with AWK function `sub`. An equivalent to AWK function `gsub` can be programmed easily in Algol 68 Genie:

```
PROC awk gsub = (STRING pattern, replacement, REF STRING string) INT:
  BEGIN INT return code;
  DO return code :=
    sub in string (pattern, replacement, string);
    UNTIL return code /= 0
  OD;
  return code
END
```

`grep in string` and `sub in string` employ extended POSIX syntax for regular expressions. Traditional UNIX regular expression syntax is obsoleted by POSIX (but is still widely used for the purposes of backwards compatibility). POSIX extended regular expressions are similar to traditional UNIX regular expressions. Most characters are treated as literals that match themselves: "a" matches "a", "(bc" matches "(bc", et cetera. Meta characters "(", ")", "[", "]", "", "", ".", "\*", "?", "+", "^" and "\$" are used as special symbols that have to be marked by a preceding "\ when they are to be matched literally.

### 10.18.1 Definition of meta characters

1. `.`  
Matches any single character.
2. `*`  
Matches the preceding expression zero or more times. For example, "[xyz]\*" matches "", "x", "y", "zx", "zyx", and so on.
3. `+`  
Matches the preceding expression one or more times - "ba+" matches "ba", "baa", "baaa" et cetera.
4. `?`  
Matches the preceding expression zero or one times - "ba?" matches "b" or "ba".
5. `{x, y}` Matches the preceding expression at least `x` and not more than `y` times. For example, "a{3, 5}" matches "aaa", "aaaa" or "aaaaa". Note that this is not implemented on all platforms.
6. `|`  
The choice (or set union) operator: match either the expression before or the expression after the operator - "abc|def" matches "abc" or "def".

## PROGRAMMING WITH ALGOL 68 GENIE

### 7. [ ]

Matches characters that contained within [ ]. For example, [abc] matches "a", "b", or "c", and [a-z] matches any lower-case letter. These can be mixed: [abcq-z] matches a, b, c, q, r, s, t, u, v, w, x, y, z, and so does [a-cq-z]. A "-" character is a literal only if it is the first or last character within the brackets: [abc-] or [-abc]. To match an "[" or "]" character, put the closing bracket first in the enclosing square brackets: [][ab] matches "]" , "[", "a" or "b".

### 8. [^]

Matches a single character that is not contained within [ ]. For example, [^abc] matches any character other than "a", "b", or "c". [^a-z] matches any single character that is not a lower-case letter. As with [ ], these can be mixed.

### 9. ^

Matches start of the line.

### 10. \$

Matches end of the line.

### 11. ( )

Matches a sub-expression. A sub-expression followed by "\*" is invalid on most platforms.

Since many ranges of characters depend on the chosen locale setting (e.g., in some settings letters are organised as abc ... yzABC ... YZ while in some others as aAbBcC ... yYzZ) the POSIX standard defines categories of characters as shown in the following table:

| Category     | Compare to          | Matches                               |
|--------------|---------------------|---------------------------------------|
| [ :upper: ]  | [A-Z]               | Upper-case letters                    |
| [ :lower: ]  | [a-z]               | Lower-case letters                    |
| [ :alpha: ]  | [A-Za-z]            | Upper- and lower-case letters         |
| [ :alnum: ]  | [A-Za-z0-9]         | Digits, upper- and lower-case letters |
| [ :digit: ]  | [0-9]               | Digits                                |
| [ :xdigit: ] | [0-9A-Fa-f]         | Hexadecimal digits                    |
| [ :punct: ]  | [.,!?:...]          | Punctuation marks                     |
| [ :blank: ]  | [ \t ]              | Space and Tab                         |
| [ :space: ]  | [ \t \n \r \f \v ]  | Typographical display features        |
| [ :cntrl: ]  | —                   | Control characters                    |
| [ :graph: ]  | [^ \t \n \r \f \v ] | Printed characters                    |
| [ :print: ]  | [^ \t \n \r \f \v ] | Printed characters and space          |

Examples:

1. ".at" matches any three-character string ending with at,
2. "[hc]at" matches hat and cat,



3. "[^]at" matches any three-character string ending with `at` and not beginning with `b`,
4. "^[hc]at" matches `hat` and `cat` but only at the beginning of a line,
5. "[hc]at\$" matches `hat` and `cat` but only at the end of a line,
6. "[hc]+at" matches with `"hat"`, `"cat"`, `"hhat"`, `"chat"`, `"heat"`, `"ccchat"` et cetera,
7. "[hc]?at" matches `"hat"`, `"cat"` and `"at"`,
8. "([cC]at)([dD]og)" matches `"cat"`, `"Cat"`, `"dog"` and `"Dog"`,
9. "[[:upper:]ab]" matches upper-case letters and `"a"`, `"b"`.

## 10.19 Curses support

On systems that have installed the curses library, and most systems do, `a68g` can link with that library to extend the standard-prelude. Curses support is very basic.

1. `PROC curses start = VOID`  
Starts curses support.
2. `PROC curses end = VOID`  
Stops curses support.
3. `PROC curses clear = VOID`  
Clears the screen.
4. `PROC curses refresh = VOID`  
Refreshes the screen (by writing all changes since the last **call** to this routine).
5. `PROC curses getchar = CHAR`  
Check if a key was pressed on the keyboard. If so, the routine returns the corresponding character, if not, on UNIX it returns `null char`.
6. `PROC curses putchar = (CHAR ch) VOID`  
Writes `ch` at the current cursor position, and advances the cursor. If the advance is at the right margin, the cursor automatically wraps to the beginning of the next line. If the next line is off the screen, a runtime error occurs.
7. `PROC curses move = (INT line, column) VOID`  
Sets the cursor position to `line` and `column`. A runtime error occurs if you move the cursor off screen.

8. PROC `curses lines = INT`

This function returns the number of lines on the screen. The line numbers on the screen are  $0 \dots \text{curses lines} - 1$ . A runtime error occurs if you move the cursor off screen.

9. PROC `curses columns = INT`

This function returns the number of columns on the screen. The column numbers on the screen are  $0 \dots \text{curses columns} - 1$ . A runtime error occurs if you move the cursor off screen.

## 10.20 PostgreSQL support

PostgreSQL (<http://www.postgresql.org>) is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley computer science Department. PostgreSQL is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features. PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs)

1. A server process, which manages the database files, accepts connections to the database from client applications, and performs actions on the database on behalf of the clients. The database server program is called `postmaster`.
2. The user's client (front-end) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialised database maintenance tool.

As is typical of client/server applications, the client and the server can be on different hosts. In that case they communicate over a TCP/IP network connection. You should keep this in mind, because the files that can be accessed on a client machine might not be accessible (or might only be accessible using a different file name) on the database server machine.

The PostgreSQL server can handle multiple concurrent connections from clients. For that purpose it starts ("forks") a new process for each connection. From that point on, the client and the new server process communicate without intervention by the original `postmaster` process. Thus, the `postmaster` is always running, waiting for client connections, whereas client and associated server processes come and go.

The routines in this section enable basic client interactions from Algol 68 with a PostgreSQL back end server. These routines allow client programs to pass queries to a PostgreSQL back end server and to receive the results of these queries.

Algol 68 Genie uses `libpq`, the C application programmer's interface to PostgreSQL (version 8.1.4 or compatible). The `libpq` library is also used in other PostgreSQL application interfaces, including those for C++, Perl, Python, Tcl and ECPG. An example application is in 11.17.

Connection to a database server is established through a `REF FILE` variable. String results are communicated through a `REF STRING` buffer that is associated with the `REF FILE` variable upon establishing a connection with a PostgreSQL server. This allows for easy mode conversion of text-formatted data from tables using `get`. Currently, Algol 68 Genie does not support retrieving data in binary format.

Next routines return 0 or positive on success, `-1` when no connection was established, `-2` when no result was available, or `-3` on other errors.

### 10.20.1 Connecting to a server

1. PROC `pq connect db =`  
`(REF FILE f, STRING conninfo, REF STRING buffer) INT`  
 Connect to a PostgreSQL back end server and open a new database connection in a way specified by `conninfo`, associate the connection with file `f` and associate `buffer` with file `f`.

The passed string can be empty to use all default parameters, or it can contain one or more parameter settings separated by white space. Each parameter setting is in the form `keyword = value`. Spaces around the equal **sign** are optional. To write an empty value or a value containing spaces, surround it with single quotes, e.g., `keyword = 'a value'`. Single quotes and backslashes within the value must be escaped with a backslash, i.e., `\'` and `\\`.

The currently recognised parameter keywords are:

`host`

Name of host to connect to. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored. The default behaviour when `host` is not specified is to connect to a Unix-domain socket in `/tmp` (or whatever socket directory was specified when PostgreSQL was built). On machines without Unix-domain sockets, the default is to connect to `localhost`.

`hostaddr`

Numeric IP address of host to connect to. This should be in the standard IPv4 address format, e.g., `172.28.40.9`. If your machine supports IPv6, you can also use those addresses. TCP/IP communication is always used when a non-empty string is specified for this parameter. Using `hostaddr` instead of `host` allows the application to avoid a host name look-up, which may be important in applications with time constraints. However, Kerberos authentication requires the host name. The following

therefore applies: If `host` is specified without `hostaddr`, a host name lookup occurs. If `hostaddr` is specified without `host`, the value for `hostaddr` gives the remote address. When Kerberos is used, a reverse name query occurs to obtain the host name for Kerberos. If both `host` and `hostaddr` are specified, the value for `hostaddr` gives the remote address; the value for `host` is ignored, unless Kerberos is used, in which case that value is used for Kerberos authentication. (Note that authentication is likely to fail if `libpq` is passed a host name that is not the name of the machine at `hostaddr`.) Also, `host` rather than `hostaddr` is used to identify the connection in `/.pgpass`. Without either a host name or host address, `libpq` will connect using a local Unix-domain socket; or on machines without Unix-domain sockets, it will attempt to connect to `localhost`.

`port`

Port number to connect to at the server host, or socket file name extension for Unix-domain connections.

`dbname`

The database name. Defaults to be the same as the user name.

`user`

PostgreSQL user name to connect as. Defaults to be the same as the operating system name of the user running the application.

`password`

Password to be used if the server demands password authentication.

`connect_timeout`

Maximum wait for connection, in seconds (write as a decimal integer string). Zero or not specified means wait indefinitely. It is not recommended to use a time-out of less than 2 seconds.

`options`

Command-line options to be sent to the server.

`sslmode`

This option determines whether or with what priority an SSL connection will be negotiated with the server. There are four modes: `disable` will attempt only an unencrypted SSL connection; `allow` will negotiate, trying first a non-SSL connection, then if that fails, trying an SSL connection; `prefer` (the default) will negotiate, trying first an SSL connection, then if that fails, trying a regular non-SSL connection; `require` will try only an SSL connection. If PostgreSQL is compiled without SSL support, using option `require` will cause an error, while options `allow` and `prefer` will be accepted but `libpq` will not in fact attempt an SSL connection.

`requiressl`

This option is deprecated in favour of the `sslmode` setting. If set to 1, an SSL connection to the server is required (this is equivalent to `sslmode require`). `libpq` will then refuse to connect if the server does not accept an SSL connection. If set to 0 (default), `libpq` will negotiate the connection type with the server (equivalent to

`sslmode prefer`). This option is only available if PostgreSQL is compiled with SSL support.

`krbsrvname`

Kerberos service name to use when authenticating with Kerberos 5. This must match the service name specified in the server configuration for Kerberos authentication to succeed.

`service`

Service name to use for additional parameters. It specifies a service name in

`pg_service.conf`

that holds additional connection parameters. This allows applications to specify only a service name so connection parameters can be centrally maintained. See

`share/pg_service.conf.sample`

in the installation directory for information on how to set up the file. If any parameter is unspecified, then the corresponding environment variable is checked. If the environment variable is not set either, then the indicated built-in defaults are used. The following environment variables can be used to select default connection parameter values, which will be used by `PQconnectdb` if no value is directly specified by the calling code. These are useful to avoid hard-coding database connection information into simple client applications, for example.

`PGHOST`

`PGHOST` sets the database server name. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is then the name of the directory in which the socket file is stored (in a default installation set-up this would be `/tmp`).

`PGHOSTADDR`

`PGHOSTADDR` specifies the numeric IP address of the database server. This can be set instead of or in addition to `PGHOST` to avoid DNS lookup overhead. See the documentation of these parameters, under `PQconnectdb` above, for details on their interaction. When neither `PGHOST` nor `PGHOSTADDR` is set, the default behaviour is to connect using a local Unix-domain socket; or on machines without Unix-domain sockets, `libpq` will attempt to connect to `localhost`.

`PGPORT`

`PGPORT` sets the TCP port number or Unix-domain socket file extension for communicating with the PostgreSQL server.

`PGDATABASE`

`PGDATABASE` sets the PostgreSQL database name.

`PGUSER`

`PGUSER` sets the user name used to connect to the database.

`PGPASSWORD`

`PGPASSWORD` sets the password used if the server demands password authentication. Use of this environment variable is not recommended for security reasons (some op-

erating systems allow non-root users to see process environment variables via `ps`); instead consider using the `/.pgpass` file.

`PGPASSFILE`

`PGPASSFILE` specifies the name of the password file to use for lookups. If not set, it defaults to `/.pgpass`.

`PGSERVICE`

`PGSERVICE` sets the service name to be looked up in `pg_service.conf`. This offers a shorthand way of setting all the parameters.

`PGREALM`

`PGREALM` sets the Kerberos realm to use with PostgreSQL, if it is different from the local realm. If `PGREALM` is set, `libpq` applications will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. This environment variable is only used if Kerberos authentication is selected by the server.

`PGOPTIONS`

`PGOPTIONS` sets additional run-time options for the PostgreSQL server.

`PGSSLMODE`

`PGSSLMODE` determines whether and with what priority an SSL connection will be negotiated with the server. There are four modes: `disable` will attempt only an unencrypted SSL connection; `allow` will negotiate, trying first a non-SSL connection, then if that fails, trying an SSL connection; `prefer` (the default) will negotiate, trying first an SSL connection, then if that fails, trying a regular non-SSL connection; `require` will try only an SSL connection. If PostgreSQL is compiled without SSL support, using option `require` will cause an error, while options `allow` and `prefer` will be accepted but `libpq` will not in fact attempt an SSL connection.

`PGREQUIRESSL`

`PGREQUIRESSL` sets whether or not the connection must be made over SSL. If set to 1, `libpq` will refuse to connect if the server does not accept an SSL connection (equivalent to `sslmode prefer`). This option is deprecated in favour of the `sslmode` setting, and is only available if PostgreSQL is compiled with SSL support.

`PGKRBSRVNAME`

`PGKRBSRVNAME` sets the Kerberos service name to use when authenticating with Kerberos 5.

`PGCONNECT_TIMEOUT`

`PGCONNECT_TIMEOUT` sets the maximum number of seconds that `libpq` will wait when attempting to connect to the PostgreSQL server. If unset or set to zero, `libpq` will wait indefinitely. It is not recommended to set the time-out to less than 2 seconds.

## 2. PROC `pq finish = (REF FILE f) INT`

Closes the connection to the server. Also frees memory used. Note that even if the server connection attempt fails, the application should call `pq finish` to free memory used. The connection must not be used again after `pq finish` has been called.

3. PROC pq reset = (REF FILE f) INT  
Resets the communication channel to the server. This function will close the connection to the server and attempt to re-establish a new connection to the same server, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost.

4. PROC pq parameter status = (REF FILE f, STRING parameter) INT  
Assigns current parameter value to the string associated with f. Certain parameter values are reported by the server automatically at connection start-up or whenever their values change. pq parameter status can be used to interrogate these settings. It returns the current value of a parameter if known, or an empty string if the parameter is not known.

Parameters reported as of the current release include

```
server_version,
server_encoding,
client_encoding,
is_superuser,
session_authorization,
DateStyle,
TimeZone,
integer_datetimes, and
standard_conforming_strings.
```

Note that

```
server_version,
server_encoding and
integer_datetimes
cannot change after start-up.
```

Pre-3.0-protocol servers do not report parameter settings, but libpq includes logic to obtain values for server\_version and client\_encoding anyway. Note that on a pre-3.0 connection, changing client\_encoding via SET after connection start-up will not be reflected by pq parameter status.

If no value for standard\_conforming\_strings is reported, applications may assume it is false, that is, backslashes are treated as escapes in string literals. Also, the presence of this parameter may be taken as an indication that the escape string syntax (E' . . . ') is accepted.

## 10.20.2 Sending queries and retrieving results

1. PROC pq exec = (REF FILE f, STRING query) INT  
Submits a command to the server and waits for the result. A zero result will generally be returned except in out-of-memory conditions or serious errors such as inability to

send the command to the server. If a zero result is returned, it should be treated like a fatal error. Use `pq_error` message to get more information about such errors.

It is allowed to include multiple SQL commands (separated by **semicolon-symbols**) in the command string. Multiple queries sent in a single `pq_exec` call are processed in a single transaction, unless there are explicit `BEGIN/COMMIT` commands included in the query string to divide it into multiple transactions. Note however that the returned `PGresult` structure describes only the result of the last command executed from the string. Should one of the commands fail, processing of the string stops with it and the returned `PGresult` describes the error condition.

2. `PROC pq_ntuples = (REF FILE f) INT`  
Returns the number of rows (tuples) in the query result.
3. `PROC pq_nfields = (REF FILE f) INT`  
Returns the number of columns (fields) in each row of the query result.
4. `PROC pq_fname = (REF FILE f, INT index) INT`  
Returns the column name associated with the given column number. Column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscript the first element).
5. `PROC pq_fnumber = (REF FILE f, INT index) INT`  
Returns the column number associated with the given column name. Column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscript the first element). The given name is treated like an **identifier** in an SQL command, that is, it is down-cased unless quoted.
6. `PROC pq_fformat = (REF FILE f, INT index) INT`  
Returns the format code indicating the format of the given column. Column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscript the first element). Format code zero indicates textual data representation, while format code one indicates binary representation. (Other codes are reserved for future definition.) Column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscript the first element).
7. `PROC pq_get_is_null = (REF FILE f, INT row, column) INT`  
Tests a field for a null value. Row and column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscript the first element). This function returns 1 if the field is null and 0 if it contains a non-null value.
8. `PROC pq_get_value = (REF FILE f, INT row, column) INT`  
Assigns a single field value (in text format) of one row to the string associated with `f`. Row and column numbers start at 1 (which deviates from the `libpq` convention, where 0 subscript the first element).

Currently, `pq_get_value` does not support retrieving data in binary format.



An empty string is returned if the field value is null.

File `f` is reset, hence after a **call** to `pq get value`, conversion of text data to other data types than row-of-character can be easily accomplished using `get`.

9. `PROC pq cmd status = (REF FILE f) INT`  
 Assigns the command status tag of the last SQL command sent to `f` to the string associated with `f`. Commonly this is just the name of the command, but it may include additional data such as the number of rows processed.
10. `PROC pq cmd tuples = (REF FILE f) INT`  
 Assigns the number of rows affected by the last SQL command sent to `f` to the string associated with `f`, following the execution of an `INSERT`, `UPDATE`, `DELETE`, `MOVE`, or `FETCH` phrase.

### 10.20.3 Connection status information

1. `PROC pq error message = (REF FILE f) INT`  
 Assigns the error message most recently generated by an operation on the connection to the string associated with `f`.
2. `PROC pq result error message = (REF FILE f) INT`  
 Assigns the error message associated with the command (or an empty string if there was no error) to the string associated with `f`. Immediately following a `pq exec` call, `pq error message` (on the connection) will return the same string as `pq result error message` (on the result). However, a query result will retain its error message until destroyed, whereas the connection's error message will change when subsequent operations are done.
3. `PROC pq db = (REF FILE f) INT`  
 Assigns the database name of the connection to the string associated with `f`.
4. `PROC pq user = (REF FILE f) INT`  
 Assigns the user name of the connection to the string associated with `f`.
5. `PROC pq pass (REF FILE f) INT`  
 Assigns the password of the connection to the string associated with `f`.
6. `PROC pq host = (REF FILE f) INT`  
 Assigns the server host name to the string associated with `f`.
7. `PROC pq port = (REF FILE f) INT`  
 Assigns the port of the connection to the string associated with `f`.
8. `PROC pq options = (REF FILE f) INT`  
 Assigns the command-line options passed in the connection request to the string associated with `f`.

9. PROC pq protocol version = (REF FILE f) INT  
Returns the front-end/back-end protocol being used. Applications may wish to use this to determine whether certain features are supported. Currently, the possible values are 2 (2.0 protocol), 3 (3.0 protocol), or zero (connection bad). This will not change after connection start-up is complete, but it could theoretically change during a connection reset. The 3.0 protocol will normally be used when communicating with PostgreSQL 7.4 or later servers; pre-7.4 servers support only protocol 2.0. (Protocol 1.0 is obsolete and not supported by libpq.)
10. PROC pq server version = (REF FILE f) INT  
Returns an integer representing the back end version. Applications may use this to determine the version of the database server they are connected to. The number is formed by converting the major, minor, and revision numbers into two-decimal-digit numbers and appending them together. For example, version 7.4.2 will be returned as 70402, and version 8.1 will be returned as 80100 (leading zeroes are not shown). Zero is returned if the connection is bad
11. PROC pq socket = (REF FILE f) INT  
Obtains the file descriptor number of the connection socket to the server. A valid descriptor will be greater than or equal to 0; a result of -1 indicates that no server connection is currently open. (This will not change during normal operation, but could change during connection set-up or reset.)
12. PROC pq back end pid = (REF FILE f) INT  
Returns the process ID (PID) of the backed server process handling this connection. The back end PID is useful for debugging purposes and for comparison to NOTIFY messages (which include the PID of the notifying back end process). Note that the PID belongs to a process executing on the database server host, not the local host.

### 10.20.4 Behaviour in threaded programs

libpq is re-entrant and thread-safe if the configure command-line option

```
--enable-thread-safety
```

was used when the PostgreSQL distribution was built.

One restriction is that no two threads attempt to manipulate the same PGconn object at the same time. In particular, you cannot issue concurrent commands from different threads through the same connection object. If you need to run concurrent commands, use multiple connections. PGresult objects are read-only after creation, and so can be passed around freely between threads.

In Algol 68 Genie, this means that in a parallel-clause, two concurrent **units** should not manipulate a same FILE variable that is used to hold a connection to a database. Use

multiple `FILE` variables to set up multiple connections.

For more information please refer to the PostgreSQL documentation.

## 10.21 PCM Sounds

Algol 68 Genie has basic support for manipulating sounds. In `a68g`, `SOUND` values can be generated, read and written and the sampled sound data can be accessed and altered. You can for example write an application to concatenate sound files, or to filter a sound using Fourier transforms, {see 11.18}. Currently Algol 68 Genie supports linearly encoded PCM (for example used for CD audio). Should you wish to manipulate sound data in another format, then one option is to convert formats using a tool as for instance *SoX*.

### 10.21.1 Data structure and routines

`a68g` allows the **declaration** of values of mode `SOUND` that will contain a sound. The standard-prelude contains a **declaration**:

```
MODE SOUND = STRUCT ( ... );
```

with structured fields that cannot be directly accessed. The actual sample data is stored in the heap. `a68g` takes care of proper copying of sample data in case of assignation of `SOUND` values, or manipulation of stowed objects containing `SOUND` values, et cetera. Currently next routines are declared for values of mode `SOUND`:

1. `PROC new sound = (INT resolution, rate, channels, samples) SOUND`  
Returns a sound value where `resolution` is the number of bits per sample (8, 16, ...), `rate` is the sample rate, `channels` is the number of channels (1 for mono, 2 for stereo) and `samples` is the number of samples per channel.
2. `OP RESOLUTION = (SOUND s) INT`  
Returns the number of bits in a sample of `s`.
3. `OP CHANNELS = (SOUND s) INT`  
Returns the number of channels of `s`.
4. `OP RATE = (SOUND s) INT`  
Returns the sample rate (as samples per second per channel) of `s`. Common values are 8000, 11025, 22050 or 44100.
5. `OP SAMPLES = (SOUND s) INT`  
Returns the number of samples per channel of `s`.

## PROGRAMMING WITH ALGOL 68 GENIE

6. PROC get sound = (SOUND s, INT channel, sample) INT

Returns the sample indexed by channel and sample of s. Note that under RIFF, samples with size up to 8 bits are normally unsigned while samples of larger size are signed.

7. PROC set sound = (SOUND s, INT channel, sample, value) INT

Sets the sample indexed by channel and sample of s to value. Note that under RIFF, samples with size up to 8 bits are normally unsigned while samples of larger size are signed.

### 10.21.2 Transput of SOUND values

SOUND values can be read and written using standard-prelude transput routines. Formatted transput of SOUND values is undefined. Currently, Algol 68 Genie supports transput of files according to the RIFF PCM/WAVE format specification. Note that chunks other than WAVE, fmt or data are stripped upon reading, and do not appear upon writing.

An example of transput of a SOUND value is:

```
SOUND s;
FILE in, out;
open (in, "source.wav", standback channel);
open (out, "destination.wav", standback channel);
get bin (in, s);
put bin (out, s);
close (in);
close (out);
```

### 10.21.3 Playing sounds

Playing a sound depends on actual hardware details of a platform. Therefore, Algol 68 Genie does not have a standard procedure to play SOUND values. It is however possible to write a procedure to play a sound in Algol 68 Genie, making use of its extensions to print to pipes and to fork, as is demonstrated in next example that makes a SOUND value to be played by the play program that comes with SoX:

```
PR heap=256M PR # ... to play BIG audio tracks #
PROC play sound = (SOUND s) VOID:
  IF PIPE p = execve child pipe ("/usr/bin/play",
    ("play", "-t", "wav", "-q", "-"), "");
    pid OF p < 0
  THEN put (stand error, ("pipe not valid: ", strerror (errno)));
  ELSE put (write OF p, s); # put must wait for play to empty pipe #
```

## STANDARD-PRELUDE AND LIBRARY-PRELUDE

```
    close (read OF p);
    close (write OF p)
FI;

print ("File: ");
STRING name = read string;
open (stdin, name + ".wav", stdin channel);
SOUND s;
read (s);
close (stdin);
play sound (s)
```



## **Part III**

### **Example a68g programs**





# Chapter 11

## Example a68g programs

### 11.1 Lucas numbers

This sequence is named after François Édouard Anatole Lucas (1842-1891). Like Fibonacci numbers, a Lucas number is the sum of the two preceding terms in the sequence. Consequently, the ratio between two consecutive Lucas numbers converges to the golden ratio.

```
determine first generation;
WHILE can represent next generation
DO calculate next generation;
   print next generation
OD.

determine first generation:
   INT previous := 1, current := 3.

can represent next generation:
   current <= max int - previous.

calculate next generation:
   INT new = current + previous;
   previous := current;
   current := new.

print next generation:
   printf (($1z", "3z", "3z", "2z-d$, current,
           $xz", "3z", "3z", "2z-d$, previous,
           $xd.n (real width - 1)d$, current / previous)).
```

## 11.2 Roman numbers

The Roman numeral system is decimal but not positional and is related to the Etruscan number system. The letters derive from earlier non-alphabetical symbols.

```

BEGIN # Conversion decimal to Roman notation #
  OP ROMAN = (INT number) STRING:
    BEGIN INT n:= number, STRING sum,
      [] STRUCT (INT value, STRING r) table =
        ((1000, "M"), (900, "CM"), (500, "D"), (400, "CD"),
          (100, "C"), (90, "XC"), (50, "L"), (40, "XL"),
          (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I"));
      FOR i TO UPB table
        DO INT v = value OF table[i], STRING r = r OF table[i];
          WHILE v <= n
            DO (sum +:= r, n -= v)
              OD
            OD;
          sum
        END,
      OP DECIMAL = (STRING text) INT:
        IF ELEMS text = 0
          THEN 0
        ELSE
          OP ABS = (CHAR s) INT:
            CASE INT p;
              VOID (char in string (s,p, "IVXLCDM"));
              p
            IN 1, 5, 10, 50, 100, 500, 1000
            ESAC,
          INT v, maxv := 0, maxp;
          FOR p TO UPB text
            DO IF (v := ABS text[p]) > maxv
              THEN maxp := p;
                maxv := v
              FI
            OD;
          maxv - DECIMAL text[: maxp - 1] + DECIMAL text[maxp + 1:]
          FI;
          print(ROMAN 1968); # MCMLXVIII #
          print(DECIMAL "MMX") # 2010 #
        END
      END
    END
  END

```

## 11.3 Holidays related to Easter

From 325 on, Easter Sunday is the Sunday following the first Ecclesiastical full moon not earlier than the equinox date in 325 — 21 March. The Ecclesiastical full moon does not always correspond to the astronomical full moon since in 325 fine details of Lunar dynamics were not yet fully understood.

```
printf (($"Christian holidays, related to Easter, in the 21th century:"$));
FOR year FROM 2001 TO 2100
DO
  printf (($14d$, year));
  INT month, day;
  BEGIN
    COMMENT
      Easter date algorithm from J.M. Oudin (1940), reprinted in:
      P.K. Seidelmann ed., "Explanatory Supplement to the Astronomical
      Almanac" [1992] (Chapter 12, "Calendars", by L.E. Doggett)
    COMMENT
      INT c = year % 100, n = year %* 19;
      INT i := (c - c % 4 - (c - (c - 17) % 25) % 3 + 19 * n + 15) %* 30;
      i -= (i % 28) * (1 - (i % 28) * (29 % (i + 1)) * ((21 - n) % 11));
      INT l = i - (year + year % 4 + i + 2 - c + c % 4) %* 7;
      month := 3 + (l + 40) % 44;
      day := l + 28 - 31 * (month % 4)
    END;
  PROC print date = (STRING holiday, INT year, month, day) VOID:
  BEGIN
    PROC weekday = (INT year0, month0, day0) INT:
      # Zeller's Congruence algorithm from 1887. #
      BEGIN INT year := year0, month := month0, day := day0, c;
        (month <= 2 | (month += 12, year -= 1));
        c := year OVER 100;
        year %*:= 100;
        1 + (day + ((month + 1) * 26) OVER 10
          + year + year OVER 4 + c OVER 4 - 2 * c) %* 7
      END;
    print f (($xg": "
      c("Sat", "Sun", "Mon", "Tue", "Wed", "Thu", "Fri")xz-dx
      c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")$,
      holiday, weekday(year, month, day), day, month))
  END;
# Print Easter date, always a Sunday. #
```

## PROGRAMMING WITH ALGOL 68 GENIE

```
print date ("Easter", year, month, day);
PROC advance = (INT number) VOID:
  BEGIN PROC days = (INT year, month) INT:
    (month | 31,
      28 + (year %* 4 = 0 AND year %* 400 /= 0 | 1 | 0),
      31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
    TO number
    DO day += 1;
      (day > days (year, month) | (day := 1, month += 1))
    OD
  END;
# Ascension day is 39 days after Easter.#
advance (39);
print date ("Ascension", year, month, day);
# Pentecost is 10 days after Ascension day.#
advance (10);
print date ("Pentecost", year, month, day);
# Trinity is 7 days after Pentecost.#
advance (7);
print date ("Trinity", year, month, day);
# Corpus Christi is 4 days after Trinity.#
advance (4);
print date ("Corpus", year, month, day)
OD
```

## 11.4 The Sierpinski triangle

The Sierpinski triangle, also called the Sierpinski gasket or the Sierpinski sieve, is a fractal named after the Polish mathematician Sierpinski who described it in 1915. Originally constructed as a curve, this is one of the basic examples of self-similar sets, id est it is a mathematically generated pattern that can be reproduced at any magnification or reduction.

```
PROC sierpinski = (INT n) [] STRING:
  BEGIN
    FLEX [1] STRING d := "*";
    FOR i TO n
      DO [2 * UPB d] STRING next;
        FOR k TO UPB d
          DO next[k] := 2 ** (i - 1) * blank + d[k] + 2 ** (i - 1) * blank;
            next[UPB d + k] := d[k] + blank + d[k]
          OD;
        d := next
      OD
    END
```

```

OD;
d
END;

```

```
printf(($gl$,sierpinski(4))
```

This code produces next output:

```

      *
     * *
    *  *
   * * * *
  *      *
 * *      * *
*   *   *   *
 * * * * * * * *
*           *
 * *           * *
 *   *           *   *
 * * * *           * * * *
 *           *           *           *
 * *           * *           * *           * *
 *   *   *   *   *   *   *   *   *
* * * * * * * * * * * * * * * *

```

## 11.5 Whetstone benchmark

The Whetstone benchmark is a synthetic benchmark for evaluating the performance of computers. It primarily measures the floating-point arithmetic performance. A similar benchmark for integer and string operations is the Dhrystone benchmark. The Whetstone benchmark was first written in Algol 60 in 1972 at the National Physical Laboratory in the United Kingdom using a modified version of the Whetstone Algol 60 compiler which was written at the Atomic Power Division of the English Electric Company in Whetstone, Leicestershire, England. The program's behaviour replicated that of a typical scientific program and was designed to defeat compiler optimisations.

```

CO
  Synthetic benchmark following Curnow & Wichmann.
  Vintage Algol 68 results (MWHIPS):
  ICL 1906A A68R Compiler 0.6
CO
[1 : 4] REAL e1,

```

## PROGRAMMING WITH ALGOL 68 GENIE

```
REAL t, t1, t2, cpul, time, x1, x2, x3, x4, x, y, z,
INT j, k, l, i, ii;

PROC pa = (REF [] REAL e) VOID:
  TO 6
  DO e[1] := (e[1] + e[2] + e[3] - e[4]) * t;
     e[2] := (e[1] + e[2] - e[3] + e[4]) * t;
     e[3] := (e[1] - e[2] + e[3] + e[4]) * t;
     e[4] := (- e[1] + e[2] + e[3] + e[4]) / t2
  OD;

PROC po = VOID:
  (e1[j] := e1[k]; e1[k] := e1[l]; e1[l] := e1[j]);

PROC p3 = (REF REAL x, y, z) VOID:
  (x := t * (x + y); y := t * (x + y); z := (x + y) / t2);

cpul := seconds;
INT max = 10;
FOR cycles TO max
DO # Initialise constants #

  t := 0.499975; t1 := 0.50025; t2 := 2.0;
  # If i = 10 we have one million whetstone instructions
  # per cycle #
  i := 100; ii := i;
  INT n2 = 12 * i, n3 = 14 * i, n4 = 345 * i, n6 = 210 * i,
      n7 = 32 * i, n8 = 899 * i, n9 = 616 * i, n11 = 93 * i;

  # MODULE 1. Simple identifiers #
  x1 := 1.0; x2 := x3 := x4 := -1.0;

  # MODULE 2. Array elements #
  e1[1] := 1.0; e1[2] := e1[3] := e1[4] := -1.0;
  TO n2
  DO e1[1] := (e1[1] + e1[2] + e1[3] - e1[4]) * t;
     e1[2] := (e1[1] + e1[2] - e1[3] + e1[4]) * t;
     e1[3] := (e1[1] - e1[2] + e1[3] + e1[4]) * t;
     e1[4] := (- e1[1] + e1[2] + e1[3] + e1[4]) * t
  OD;

  # MODULE 3. Array parameters #
  TO n3 DO pa (e1) OD;

  # MODULE 4. Conditional jumps #
  j := 1;
  TO n4
  DO (j = 1 | j := 2 | j := 3);
     (j > 2 | j := 0 | j := 1);
     (j < 1 | j := 1 | j := 0)
  OD;
```

```

# MODULE 5. Omitted #

# MODULE 6. Integers #
j := 1; k := 2; l := 3;
TO n6
DO j := j * (k - j) * (l - k);
   k := l * k - (l - j) * k;
   l := (l - k) * (k + j);
   e1[l - 1] := j + k + l;
   e1[k - 1] := j * k * l
OD;

# MODULE 7. Trigonometry #
x := y := 0.5;

TO n7
DO x := t * arctan (t2 * sin (x) * cos (x) /
                   (cos (x + y) + cos (x - y) - 1.0));
   y := t * arctan (t2 * sin (y) * cos (y) /
                   (cos (x + y) + cos (x - y) - 1.0))
OD;

# MODULE 8. Calls #
x := y := z := 1.0;
TO n8
DO p3(x, y, z)
OD;

# MODULE 9. Array references #
j := 1; k := 2; l := 3;
e1[1] := 1.0; e1[2] := 2.0; e1[3] := 3.0;
TO n9
DO po
OD;

# MODULE 11. Standard functions #
x := 0.75;
TO n11
DO x := sqrt (exp (ln (x) / t1))
OD;

time := (seconds - cpu1) / cycles;
printf (($lz-d.2d, xz-d.2d$, time, 1 / (time / (ii / 10))))
OD

```

## 11.6 Prime numbers

### 11.6.1 Parallel sieve of Erathostenes

This is a gratuitously parallel implementation of the sieve of Eratosthenes after a program contributed by Lawrence D'Oliveiro; the number of primes it can output is limited only by the number of threads a68g can create.

```

MODE SIEVER = STRUCT (SEMA full, empty, REF INT n);
PROC make siever = SIEVER: (LEVEL 0, LEVEL 1, NEW INT);

PROC start siever = (SIEVER s, INT n) VOID:
  (n MOD 2 /= 0 | DOWN empty OF s; n OF s := n; UP full OF s);

PROC next unmarked = (SIEVER s) INT:
  (DOWN full OF s; INT n = n OF s; UP empty OF s; n);

PROC sieve = (SIEVER current) VOID:
  (INT n = next unmarked (current);
  print ((whole (n, -int width), new line));
  SIEVER new = make siever;
  PAR (
    DO IF INT i = next unmarked (current);
      i MOD n /= 0
      THEN start siever (new, i)
      FI
    OD,
    sieve (new)
  )
);

SIEVER first = make siever;
INT n := 1;

PAR (DO n PLUSAB 1; start siever (first, n) OD, sieve (first))

```

### 11.6.2 Rabin-Miller primality test

A primality test is an algorithm for determining whether an input number is prime. Amongst other fields of mathematics, it is used for cryptography. The difference with integer factorisation is that a primality test does not necessarily give prime factors. Factorization is a computationally hard problem, whereas primality testing is comparatively easy as its running time is polynomial. Some primality tests prove that a number is prime, while others prove that a number is composite. This example implements the probabilistic Rabin-Miller primality test in Algol 68. An odd integer  $p$  can be stated as  $p - 1 = 2^s \times d$  with  $s$  odd. If  $p$



is prime, then  $\forall a \in [1, p >$  this relation holds:

$$\{a^d \equiv 1 \pmod{n}\} \vee \{a^{2^r} \times d \equiv -1 \pmod{n}, \exists r \in [0, s >\}$$

Next algorithm tries *max trials* random choices for *a* and if a number *p* passes these trials, it is prime with approximate probability:

$$1 - \frac{1}{4^{\text{max trials}}}$$

If a number does not pass, it is not prime. Literature gives better choices for test numbers *a* than used here. To compute an exponentiation  $u^v \pmod{w}$  the algorithm uses the equality:

$$(a \ b) \pmod{m} = \{(a \ \pmod{m})(b \ \pmod{m})\} \pmod{m}$$

```

INT max trials = 10,
UNT p, m, pow := 2, INT b, MODE UNT = LONG LONG INT;
read (p);

# Dissect the number under test in form p = (2 ^ b) * m + 1 #

FOR n WHILE pow < p
DO IF (p - 1) MOD pow = 0
  THEN b := n;
  m := (p - 1) OVER pow
  FI;
  pow *:= 2
OD;
printf (($lg (0), " = 2 ^ ", g (0), " * ", g (0), " + 1"$, p, b, m));

# Test primality #
BOOL is prime := TRUE;
TO max trials WHILE is prime
DO UNT a = 1 + ENTIER (next random * (p - 1));
  IF OP GCD = (UNT a, b) UNT:
    IF b = 0
      THEN ABS a
    ELSE b GCD (a MOD b)
    FI;
  PRIO GCD = 6;
  p GCD a = 1
  THEN is prime := FALSE
  ELIF PROC up mod = (UNT a, b, m) UNT: # (a ^ b) MOD m #
    (UNT prod := 1, expo := b, pow := a MOD m;
    WHILE expo = 0
    DO IF expo MOD 2 = 1
      THEN prod := (prod * pow) MOD m
    FI;
    pow := pow ^ 2 MOD m;

```

## PROGRAMMING WITH ALGOL 68 GENIE

```
        expo OVERAB 2
    OD;
    prod
    );
    UNT z := up mod (a, m, p);
    z = 1 OR z = p - 1
    THEN SKIP # pass #
    ELSE BOOL pass := FALSE;
    TO b WHILE NOT pass
    DO z := up mod (z, 2, p);
    pass := z = p - 1
    OD;
    is prime := is prime AND pass
    FI
OD;

IF is prime
THEN printf (($lg (0)" is prime with approximate probability ",
    h$, p, 1.0 - 0.25 ^ max trials))
ELSE printf (($lg (0), " is a composite number"$, p))
FI
```

### 11.6.3 Lucas-Lehmer primality test

This program implements the Lucas-Lehmer test: for  $p$  a prime, the Mersenne number  $2^p - 1$  is prime if and only if  $2^p - 1$  divides  $S(p - 1)$  where:

$$S_{n+1} = S_n^2 - 2; S_1 = 4$$

This example verifies that  $M_{11213} = 2^{11213} - 1$  is prime.

```
INT p = 11213; PR precision=6800 timelimit=120 PR
printf (($l"M_", g (0), b (" is", " is not"), " prime"$, p,
    (
    LONG LONG INT m p = LONG LONG 2 ** p - 1,
    LONG LONG INT s := 4;
    FROM 3 TO p
    DO s := (s * s - 2) MOD m p
    OD;
    s = 0
    )
))
```

## 11.7 Hilbert matrix using fractions

In linear algebra, a Hilbert matrix is a square matrix having unit fraction entries:

$$H_{ij} = \frac{1}{i+j-1}$$

Hilbert matrices are canonical examples of ill-conditioned matrices, making them notoriously difficult to use in numerical computation. In this example we tackle these difficulties by using multi-precision fractions to calculate the exact determinant of Hilbert matrices.

COMMENT

An application for multi-precision LONG LONG INT.

A fraction has positive denominator; the nominator holds the sign.

COMMENT

```
MODE FRAC = STRUCT (NUM nom, den), NUM = LONG LONG INT;
```

```
OP N = (FRAC u) NUM: nom OF u,
```

```
  D = (FRAC u) NUM: den OF u;
```

```
FRAC zero = 0 DIV 1, one = 1 DIV 1;
```

```
PR precision=101 PR # NUM can hold a googol! #
```

```
OP DIV = (NUM n, d) FRAC:
```

```
  IF d = 0
```

```
  THEN print ("Zero denominator"); stop
```

```
  ELSE NUM gcd = ABS n GCD ABS d;
```

```
    (SIGN n * SIGN d * ABS n OVER gcd, ABS d OVER gcd)
```

```
  FI;
```

```
OP DIV = (INT n, d) FRAC: NUM (n) DIV NUM (d);
```

```
PRIO DIV = 2;
```

```
OP GCD = (NUM a, b) NUM:
```

```
  IF b = 0
```

```
  THEN ABS a
```

```
  ELSE b GCD (a MOD b)
```

```
  FI;
```

```
PRIO GCD = 8;
```

```
# Basic operators for fractions. #
```

```
OP - = (FRAC u) FRAC: - N u DIV D u;
```

```
OP + = (FRAC u, v) FRAC:
```

```
  N u * D v + N v * D u DIV D u * D v;
```

```
OP - = (FRAC u, v) FRAC: u + - v;
```

```
OP * = (FRAC u, v) FRAC: N u * N v DIV D u * D v;
```

```
OP / = (FRAC u, v) FRAC: u * (D v DIV N v);
```

```
OP += = (REF FRAC u, FRAC v) REF FRAC: u := u + v;
```

```
OP -= = (REF FRAC u, FRAC v) REF FRAC: u := u - v;
```

## PROGRAMMING WITH ALGOL 68 GENIE

```
OP *:= = (REF FRAC u, FRAC v) REF FRAC: u := u * v;
OP /:= = (REF FRAC u, FRAC v) REF FRAC: u := u / v;
OP = = (FRAC u, v) BOOL: N u = N v ANDF D u = D v;
OP /= = (FRAC u, v) BOOL: NOT (u = v);

# Matrix algebra. #
OP INNER = ([ ] FRAC u, v) FRAC:
  # Inner product of two arrays of rationals #
  BEGIN FRAC s := zero;
    FOR i TO UPB u
      DO s += u[i] * v[i]
    OD;
  s
END;
PRIO INNER = 8;

PROC lu decomposition = (REF [, ] FRAC a, REF [ ] INT p) VOID:
  # LU-decomposition cf. Crout, of a matrix of rationals. #
  BEGIN INT n = 1 UPB a;
    FOR k TO n
      DO FRAC piv := zero, INT k1 := k - 1;
        REF INT pk = p[k];
        REF [ ] FRAC aik = a[, k], aki = a[k, ];
        FOR i FROM k TO n
          DO aik[i] -= a[i, 1 : k1] INNER aik[1 : k1];
            IF piv = zero
              THEF aik[i] /= zero
              THEN piv := aik[i];
                pk := i
              FI
            OD;
          IF piv = zero
            THEN print ((new line, "Singular matrix")); stop
            FI;
          IF pk /= k
            THEN FOR i TO n
              DO FRAC r = aki[i];
                aki[i] := a[pk, i];
                a[pk, i] := -r
              OD
            FI;
          FOR i FROM k + 1 TO n
            DO aki[i] -=
              aik[1 : k1] INNER a[1 : k1, i] /= piv
            OD
          OD
        END;
      END;

PROC determinant = ([, ] FRAC a) FRAC:
  # Determinant of a decomposed matrix is its trace. #
```

```

BEGIN FRAC d := one;
  FOR i TO 1 UPB a
    DO d *:= a[i, i]
    OD;
  d
END;

# Table of required results. #
[] NUM table = BEGIN
  1,
  12,
  2 160,
  6 048 000,
  266 716 800 000,
  186 313 420 339 200 000,
  2 067 909 047 925 770 649 600 000,
  365 356 847 125 734 485 878 112 256 000 000,
  1 028 781 784 378 569 697 887 052 962 909 388 800 000 000
  46 206 893 947 914 691 316 295 628 839 036 278 726 983 680 000 000 000
END;

# Compute determinant of Hilbert matrix of increasing rank. #
FOR n TO UPB table
DO [1 : n, 1 : n] FRAC a;
  FOR i TO n
    DO a[i, i] := 1 DIV 2 * i - 1;
      FOR j FROM i + 1 TO n
        DO a[i, j] := a[j, i] := 1 DIV i + j - 1
        OD
      OD;
    lu decomposition (a, LOC [1 : n] INT);
    FRAC det = determinant (a);
    print (("Rank ", whole (n, 0), ", determinant ",
      whole (N det, 0), " / ", whole (D det, 0),
      IF N det = 1 AND D det = table[n]
      THEN ", ok"
      ELSE ", not ok"
      FI, new line))
  OD

```

## 11.8 Mastermind code breaker

Mastermind is a simple code-breaking board game for two players. The modern game with coloured pegs was devised in 1970 but the game resembles an earlier pencil and paper

## PROGRAMMING WITH ALGOL 68 GENIE

game called *bulls and cows* that may date back a century or more. The program guesses the pattern you think of, in both order and color. It demonstrates the use of a flexible row. Peg colour is replaced by a small integral value. After each guess, you rate the score with a series of "w"s and "b"s stating how many pegs are of the right colour, and also how many pegs are in the correct position.

```
# This breaks a code of 'pegs' unique digits that you think of. #
INT pegs = 4, max digit = 6;

MODE LIST = FLEX [1 : 0] COMBINATION,
  COMBINATION = [pegs] DIGIT,
  DIGIT = INT;

OP += = (REF LIST u, COMBINATION v) REF LIST:
  # Add one combination to a list. #
  (sweep heap;
  [UPB u + 1] COMBINATION w;
  w[: UPB u] := u;
  w[UPB w] := v;
  u := w
  );

PROC gen = (REF COMBINATION part, INT peg) VOID:
  # Generate all unique [max digit!/(max digit-pegs)!] combinations. #
  IF peg > pegs
  THEN all combs += part
  ELSE FOR i TO max digit
    DO IF BOOL unique := TRUE;
      FOR j TO peg - 1 WHILE unique
        DO unique := part[j] = i
          OD;
        unique
      THEN part[peg] := i;
        gen (part, peg + 1)
    FI
  OD
  FI;

LIST all combs;
gen (LOC COMBINATION, 1);

PROC break code = (LIST sieved) VOID:
  # Present a trial and sieve the list with entered score. #
  CASE UPB sieved + 1
  IN printf ("%Inconsistent scores"$),
    printf ("%Solution is "4(d)$, sieved[1])
  OUT printf ((%g (0), " solutions in set; Guess is "4(d)": "$,
    UPB sieved, sieved[1]));
  # Read the score as a sequence of "w" and "b" #
  INT col ok := 0, pos ok := 0, STRING z := "";
```

```

WHILE z = ""
DO read ((z, new line))
OD;
FOR i TO UPB z
DO (z[i] = "w" | col ok | : z[i] = "b" | pos ok) += 1
OD;
(pos ok = pegs | stop);
# Survivors are combinations with score as entered. #
LIST survivors;
FOR i FROM 2 TO UPB sieved
DO INT col ok i := 0, pos ok i := 0;
  FOR u TO pegs
  DO FOR v TO pegs
  DO IF sieved[1][u] = sieved[i][v]
  THEN (u = v | pos ok i | col ok i) += 1
  FI
  OD
  OD;
  IF col ok = col ok i AND pos ok = pos ok i
  THEN survivors += sieved[i]
  FI
  OD;
break code (survivors)
ESAC;

printf (($"This breaks a unique code of ", g (0),
" digits in range 1 ... ", g (0),
" that you think of.", 2l,
"Enter score as a series of w (hite) and b (lack)", 1,
"for instance www, wbw or bbbb.", 2l
$, pegs, max digit));

break code (all combs)

```

## 11.9 Mandelbrot set

The Mandelbrot set, named after Benoît Mandelbrot, is a set the of points  $z_0$  in the complex plane, for which the sequence  $z_{n+1} := z_n^2 + z_0$ ;  $n \geq 0$  is bounded. The boundary of the set forms the well-known fractal.

```

INT pix = 150, max iter = 256, REAL zoom = 0.33 / pix;
[-pix : pix, -pix : pix] INT plane;
COMPL ctr = 0.05 I 0.75 # center of set #;

# Compute the length of an orbit. #
PROC iterate = (COMPL z0) INT:

```

## PROGRAMMING WITH ALGOL 68 GENIE

```
BEGIN COMPL z := z0, INT iter := 1;
  WHILE (iter += 1) < max iter # not converged # AND
    ABS z < 2 # not diverged #
  DO z := z * z + z0
  OD;
  iter
END;

# Compute set and find maximum orbit length. #
INT max col := 0;
FOR x FROM -pix TO pix
DO FOR y FROM -pix TO pix
  DO COMPL z0 = ctr + (x * zoom) I (y * zoom);
    IF (plane [x, y] := iterate (z0)) < max iter
      THEN (plane [x, y] > max col | max col := plane [x, y])
    FI
  OD
OD;

# Make a plot. #
FILE plot;
INT num pix = 2 * pix + 1;
make device (plot, "gif", whole (num pix, 0) + "x" + whole (num pix, 0));
open (plot, "mandelbrot.gif", stand draw channel);
FOR x FROM -pix TO pix
DO FOR y FROM -pix TO pix
  DO INT col = (plane [x, y] > max col | max col | plane [x, y]);
    REAL c = sqrt (1- col / max col); # sqrt to enhance contrast #
    draw colour (plot, c, c, c);
    draw point (plot, (x + pix) / (num pix - 1),
      (y + pix) / (num pix - 1))
  OD
OD;
close (plot)
```

### 11.10 Knight traversing a chess board

This example show paths by which a knight can traverse a  $n * n$  board without passing a square twice.

```
INT n = 8, empty = 0;
[] STRUCT (INT row, col) moves =
  ((1, 2), (2, 1), (2, -1), (1, -2),
  (-1, -2), (-2, -1), (-1, 2), (-2, 1));
[n, n] INT board;
```



```

FOR i TO n
DO FOR j TO n
  DO board[i, j] := empty
  OD
OD;

INT sols := 0;

PROC go = (INT u, v, num) VOID:
  # Backtracking routine to traverse board. #
  IF (u >= 1 AND u <= n AND v >= 1 AND v <= n)
  THEN board[u, v] = empty
  THEN board[u, v] := num;
    IF num = n * n
    THEN printf (($1"solution" xg (0)n (n)(ln (n)(x2d))$, sols += 1, board))
    ELSE FOR k TO UPB moves
      DO go (u + row OF moves[k], v + col OF moves[k], num + 1)
      OD
    FI;
  board[u, v] := empty
FI;

go (1, 1, 1)

```

## 11.11 Decision tree

This program builds a library of objects together with a question distinguishing each object, while it guesses an object you have in mind using that library. This was a popular programming exercise among students in the 1980's.

```

ENTRY library := get reply ("give an initial object");
DO guess object (library);
  UNTIL NOT ask ("again")
OD;

# Data structure and access operators #
MODE ENTRY = UNION (STRING, FORK),
  FORK = STRUCT (STRING text, REF ENTRY has, hasnt);

OP TEXT = (FORK d) STRING: text OF d,
  HAS = (FORK d) REF ENTRY: has OF d,
  HASNT = (FORK d) REF ENTRY: hasnt OF d;
PROC new fork = (STRING text, ENTRY has, hasnt) FORK:
  (HEAP STRING := text, HEAP ENTRY := has, HEAP ENTRY := hasnt);

# Guessing and extending library #
PROC guess object = (REF ENTRY sub lib) VOID:

```

## PROGRAMMING WITH ALGOL 68 GENIE

```
# How to guess an object #
CASE sub lib
IN (STRING s): (ask (s) | SKIP | sub lib := learn (s)),
(FORK d): guess object ((ask (TEXT d) | HAS d | HASNT d))
ESAC;

PROC learn = (STRING guess) ENTRY:
# Introduce new entry in library #
IF STRING object = get reply ("what is the object"),
question = get reply ("give a question to distinguish "
+ object);
ask ("does " + question + " apply to " + object)
THEN new fork (question, object, guess)
ELSE new fork (question, guess, object)
FI;

# Interaction #

PROC get reply = (STRING prompt) STRING:
BEGIN STRING s;
printf (($gl$, prompt));
readf (($gl$, s));
s
END;

PROC ask = (STRING question) BOOL:
IF STRING s = get reply (question);
UPB s > 0
THEN s[1] = "y" ORF s[1] = "Y"
ELSE ask (question)
FI;

SKIP
```

### 11.12 Peano curve

This is the complete algorithm as discussed in 5.8.2. This program writes instructions to draw a Hilbert curve (a particular Peano curve) from a picture-environment in  $\text{\LaTeX}$ .

```
INT points = 512;

PROC draw member = (INT n) VOID:
BEGIN INT d = points OVER (2 ^ n);
MODE POINT = STRUCT (INT x, y);
POINT origin := (d OVER 2, d OVER 2);
print (("\begin{picture}(", whole (points, 0), ", ",
whole(points, 0), ")", newline));
```

```

go right (n);
print ("\end{picture}");

PROC line to = (POINT end) VOID:
  (POINT vector = (x OF end - x OF origin, y OF end - y OF origin);
  print ("\put (", whole (x OF origin, 0), ", ",
    whole (y OF origin, 0), ")",
    "\line (", whole (SIGN x OF vector, 0), ", ",
    whole (SIGN y OF vector, 0), ")",
    whole (ENTIER sqrt (x OF vector ^ 2 +
    y OF vector ^ 2), 0), "", newline));
  origin := end
  );

PROC go up = (INT n) VOID:
  IF n > 0
  THEN go right (n - 1);
    line to ((x OF origin + d, y OF origin));
    go up (n - 1);
    line to ((x OF origin, y OF origin + d));
    go up (n - 1);
    line to ((x OF origin - d, y OF origin));
    go left (n - 1)
  FI;

PROC go down = (INT n) VOID:
  IF n > 0
  THEN go left (n - 1);
    line to ((x OF origin - d, y OF origin));
    go down (n - 1);
    line to ((x OF origin, y OF origin - d));
    go down (n - 1);
    line to ((x OF origin + d, y OF origin));
    go right (n - 1)
  FI;

PROC go left = (INT n) VOID:
  IF n > 0
  THEN go down (n - 1);
    line to ((x OF origin, y OF origin - d));
    go left (n - 1);
    line to ((x OF origin - d, y OF origin));
    go left (n - 1);
    line to ((x OF origin, y OF origin + d));
    go up (n - 1)
  FI;

PROC go right = (INT n) VOID:
  IF n > 0
  THEN go up (n - 1);
    line to ((x OF origin, y OF origin + d));

```

## PROGRAMMING WITH ALGOL 68 GENIE

```
    go right (n - 1);
    line to ((x OF origin + d, y OF origin));
    go right (n - 1);
    line to ((x OF origin, y OF origin - d));
    go down (n - 1)
  FI;
  SKIP
END;
```

draw member (5)

### 11.13 Formula manipulation

This example is based on example 26<sub>11.10</sub> from the Revised Report on Algol 68.

```
# Data structure #
MODE FORMULA = UNION (REF CONST, REF VAR, REF DYADIC, REF MONADIC,
  REF CALL),
  DYADIC = STRUCT (FORMULA lhs, INT operator, FORMULA rhs),
  MONADIC = STRUCT (INT operator, FORMULA rhs),
  CALL = STRUCT (REF FUNCTION name, FORMULA parameter),
  FUNCTION = STRUCT (REF VAR bound var, FORMULA body),
  VAR = STRUCT (STRING name, NUMBER value),
  CONST = STRUCT (NUMBER value),
  NUMBER = LONG LONG REAL;

# Access operators #
OP VALUE = (REF CONST c) REF NUMBER: c (value),
  VALUE = (REF VAR v) REF NUMBER: v (value),
  NAME = (REF VAR v) REF STRING: v (name),
  NAME = (REF CALL c) REF FUNCTION: c (name),
  PARAMETER = (REF CALL c) REF FORMULA: c (parameter),
  LHS = (REF DYADIC t) REF FORMULA: t (lhs),
  RHS = (REF DYADIC t) REF FORMULA: t (rhs),
  RHS = (REF MONADIC t) REF FORMULA: t (rhs),
  OPERATOR = (REF DYADIC t) REF INT: t (operator),
  OPERATOR = (REF MONADIC t) REF INT: t (operator),
  BOUND = (REF FUNCTION f) REF REF VAR: f (bound var),
  BODY = (REF FUNCTION f) REF FORMULA: f (body);

# Generate objects #
PROC make dyadic = (FORMULA u, INT op, FORMULA v) REF DYADIC:
  NEW DYADIC := (u, op, v);
PROC make monadic = (INT op, FORMULA v) REF MONADIC:
  NEW MONADIC := (op, v);
PROC make call = (REF FUNCTION name, FORMULA parameter) REF CALL:
  NEW CALL := (name, parameter);
PROC make function = (REF VAR bound var, FORMULA body) REF FUNCTION:
```

## EXAMPLE a68g PROGRAMS

```

NEW FUNCTION := (bound var, body);
PROC make var = (STRING name, NUMBER value) REF VAR:
  NEW VAR := (name, value);
PROC make const = (NUMBER x) REF CONST: (NEW CONST c; VALUE c := x; c);
PROC zero = REF CONST: make const (0),
  one = REF CONST: make const (1),
  two = REF CONST: make const (2);

# Basic routines and operators #

PROC is var = (FORMULA f) BOOL: (f | (REF VAR): TRUE | FALSE);
PROC is const = (FORMULA f) BOOL: (f | (REF CONST): TRUE | FALSE);
PROC const value = (FORMULA f) NUMBER: (f | (REF CONST v): VALUE v);
PROC var name = (FORMULA f) STRING: (f | (REF VAR v): NAME v);
INT plus = 1, minus = 2, times = 3, divide = 4, up = 5;

OP COPY = (FORMULA u) FORMULA:
  CASE u
  IN (REF CONST v): make const (VALUE v),
    (REF VAR v): make var (NAME v, VALUE v),
    (REF DYADIC v): make dyadic (COPY LHS v, OPERATOR v, COPY RHS v),
    (REF MONADIC v): make monadic (OPERATOR v, COPY RHS v),
    (REF CALL v): make call (NAME v, PARAMETER v)
  ESAC;

OP = = (REF FUNCTION u, v) BOOL: BOUND u = BOUND v AND BODY u = BODY v;
OP = = (FORMULA a, b) BOOL:
  CASE a
  IN (REF CONST u): (b | (REF CONST v): VALUE u = VALUE v | FALSE),
    (REF VAR u): (b | (REF VAR v): NAME u = NAME v | FALSE),
    (REF DYADIC u):
      CASE b
      IN (REF DYADIC v):
          LHS u = LHS v AND RHS u = RHS v AND OPERATOR u = OPERATOR v
        OUT FALSE
      ESAC,
    (REF MONADIC u):
      CASE b
      IN (REF MONADIC v): RHS u = RHS v AND OPERATOR u = OPERATOR v
        OUT FALSE
      ESAC,
    (REF CALL u):
      CASE b
      IN (REF CALL v): NAME u = NAME v AND PARAMETER u = PARAMETER v
        OUT FALSE
      ESAC
  OUT FALSE
  ESAC;
OP /= = (FORMULA a, b) BOOL: NOT (a = b);

# Basic math #

```

## PROGRAMMING WITH ALGOL 68 GENIE

```
OP + = (FORMULA a) FORMULA: a;
OP - = (FORMULA a) FORMULA:
  (a = zero | a | make monadic (minus, a));
OP + = (FORMULA a, b) FORMULA:
  (a = zero | b | b = zero | a | make dyadic (a, plus, b));
OP - = (FORMULA a, b) FORMULA:
  (b = zero | a | make dyadic (a, minus, b));
OP * = (FORMULA a, b) FORMULA:
  IF a = zero OR b = zero
  THEN zero
  ELSE (a = one | b | b = one | a | make dyadic (a, times, b))
  FI;
OP / = (FORMULA a, b) FORMULA:
  IF a = zero AND NOT (b = zero)
  THEN zero
  ELSE (b = one | a | make dyadic (a, divide, b))
  FI;
OP ^ = (FORMULA a, REF CONST b) FORMULA:
  IF a = one OR (b IS zero)
  THEN one
  ELSE (b IS one | a | make dyadic (a, up, b))
  FI;
```

# Applications of above definitions:  
derivative, evaluation, simplification #

```
PROC derivative = (FORMULA e, # with respect to # REF VAR x) FORMULA:
  # derivative a formula #
  CASE e
  IN (REF CONST): zero,
  (REF VAR v): (v = x | one | zero),
  (REF DYADIC f):
    CASE FORMULA u = LHS f, v = RHS f;
    FORMULA deriv u = derivative (u, x),
    deriv v = derivative (v, x);
    OPERATOR f
  IN deriv u + deriv v,
  deriv u - deriv v,
  u * deriv v + deriv u * v,
  (deriv u - f * deriv v) / v,
  CASE v
  IN (REF CONST c):
    v * u ^ make const (VALUE c - 1) * deriv u
  ESAC
  ESAC,
  (REF MONADIC f):
    CASE FORMULA v = RHS f;
    FORMULA deriv v = derivative (v, x);
    OPERATOR f
  IN + deriv v,
```

```

- deriv v
ESAC,
(REF CALL c):
BEGIN
  REF FUNCTION f = NAME c;
  FORMULA g = PARAMETER c;
  REF VAR y = BOUND f;
  REF FUNCTION deriv f =
    make function (y, derivative (BODY f, y));
    (make call (deriv f, g)) * derivative (g, x)
  END
ESAC;

PROC evaluate = (FORMULA e) NUMBER:
# Value of a formula #
CASE e
IN (REF CONST c): VALUE c,
(REF VAR v): VALUE v,
(REF DYADIC f):
  CASE NUMBER u = evaluate (LHS f), v = evaluate (RHS f);
  OPERATOR f
  IN u + v, u - v, u * v, u / v, u ^ SHORTEN SHORTEN ROUND v
  ESAC,
(REF MONADIC f):
  CASE NUMBER v = evaluate (RHS f);
  OPERATOR f
  IN v, - v
  ESAC,
(REF CALL c):
  BEGIN
    REF FUNCTION f = NAME c;
    VALUE BOUND f := evaluate (PARAMETER c);
    evaluate (BODY f)
  END
ESAC;

OP SIMPLIFY = (FORMULA a) FORMULA:
IF OP SIMPLER = (FORMULA u) FORMULA:
# Example simplifications - extend as you see fit #
CASE u
IN (REF CONST v): make const (VALUE v),
(REF VAR v): make var (NAME v, VALUE v),
(REF DYADIC v):
  IF FORMULA f = SIMPLIFY LHS v, g = SIMPLIFY RHS v;
  is const (f)
  THEF is const (g)
  THEN make const (evaluate (make dyadic (f, OPERATOR v, g)))
  ELIF OPERATOR v = plus
  THEN IF f = g
    THEN make dyadic (two, times, f)

```

## PROGRAMMING WITH ALGOL 68 GENIE

```
        ELSE make dyadic (f, plus, g)
        FI
    ELIF OPERATOR v = minus
    THEN IF is const (f)
        THEF const value (f) = 0
        THEN make monadic (minus, g)
        ELIF f = g
        THEN zero
        ELSE make dyadic (f, minus, g)
        FI
    ELIF OPERATOR v = times
    THEN IF is const (g)
        THEN make dyadic (g, times, f)
        ELIF f = g
        THEN make dyadic (f, up, two)
        ELSE make dyadic (f, times, g)
        FI
    ELSE make dyadic (f, OPERATOR v, g)
    FI,
    (REF MONADIC v):
        IF FORMULA g = SIMPLIFY RHS v;
            is const (g)
        THEN make const (evaluate (make monadic (OPERATOR v, g)))
        ELSE make monadic (OPERATOR v, g)
        FI,
    (REF CALL v): make call (NAME v, PARAMETER v)
    ESAC;
    FORMULA b = SIMPLER a;
    a /= b
    THEN SIMPLIFY b
    ELSE a
    FI;

# A small demonstration #

OP FMT = (NUMBER x) STRING:
    (x = ENTIER x | whole (x, 0) | fixed (x, 0, 4));

PROC write = (FORMULA e) VOID:
    CASE e
    IN (REF CONST c): print (FMT VALUE c),
    (REF VAR v): print (NAME v),
    (REF DYADIC f):
        BEGIN
            print "(";
            write (LHS f);
            print ((OPERATOR f | " + ", " - ", " * ", " / ", " ^ "));
            write (RHS f);
            print ")"
        END,
```



```

(REF MONADIC f):
  BEGIN
    print (("(", (OPERATOR f | "+ ", "- ")));
    write (RHS f);
    print (")")
  END
ESAC;

REF VAR x = make var ("x", -1), y = make var ("y", 1);

PROC demo = (FORMULA f, REF VAR z) VOID:
  BEGIN
    write (f);
    FORMULA g = SIMPLIFY f;
    IF f /= g
    THEN print (" = ");
         write (g)
    FI;
    print (" = ", FMT evaluate (g), new line));
    FORMULA u := derivative (f, z);
    print ("df/d", NAME z, " = ");
    write (u);
    FORMULA v := SIMPLIFY u;
    IF u /= v
    THEN print (" = ");
         write (v)
    FI;
    print (" = ", FMT evaluate (v), new line, new line)
  END;

printf (($"x = ", g, " y = ", gl$, FMT VALUE x, FMT VALUE y));
demo (x + x + zero * y, x);
demo (x * two, x);
demo (x * x * x + y * y, x);
demo (x * x + two * x * y + y * y, x);
demo (x + y / x, x)

COMMENT Output produced:
x = -1 y = 1
(x + x) = (2 * x) = -2
df/dx = (1 + 1) = 2 = 2

(x * 2) = (2 * x) = -2
df/dx = 2 = 2

((x * x) * x) + (y * y) = ((x ^ 2) * x) + (y ^ 2) = 0
df/dx = ((x * x) + ((x + x) * x)) = ((x ^ 2) + ((2 * x) * x)) = 3

```

## PROGRAMMING WITH ALGOL 68 GENIE

```
((x * x) + ((2 * x) * y)) + (y * y) = ((x ^ 2) + ((2 * x) * y))
+ (y ^ 2)) = 0
df/dx = ((x + x) + (2 * y)) = ((2 * x) + (2 * y)) = 0

(x + (y / x)) = -2
df/dx = (1 + ((0 - (y / x)) / x)) = (1 + ((- (y / x)) / x)) = 0
COMMENT
```

### 11.14 Fibonacci grammar

This example computes Fibonacci numbers by counting the number of derivations of the "Fibonacci grammar":

```
fib: "a"; "a", fib; "aa", fib.
```

The purpose for this is to illustrate the use of procedure closures which we call continuations. We use this to generate a recursive descent with backup parser following a simple translation from grammar rules to procedures. This program is contributed by Eric Voss.

```
MODE CONT = PROC (INT) VOID;

PROC grammar fib = (INT i, CONT q) VOID:
  BEGIN terminal (i, "a", q);
    terminal (i, "a", (INT j) VOID: grammar fib (j, q));
    terminal (i, "aa", (INT j) VOID: grammar fib (j, q))
  END;

PROC terminal = (INT i, STRING a, CONT q) VOID:
  IF INT u = i + UPB a - 1;
    u > UPB sentence
  THEN SKIP
  ELIF sentence [i : u] = a
  THEN q (u + 1)
  FI;

STRING sentence;
FOR k TO 10
DO sentence := k * "a";
  INT nr derivations := 0;
  grammar fib (1, (INT j) VOID: (j > UPB sentence | nr derivations += 1));
  print (("Fibonacci number ", UPB sentence, " = ", nr derivations,
    new line))
OD
```

## 11.15 Formatting routines

Next listing is a specification of `whole`, `fixed`, `float` and `real` implemented by a68g. It closely follows the Revised Report definition (25<sub>10</sub>.3.2.1), that however does not define the routine `real`.

```
# Actual implementation also includes LONG and LONG LONG modes #
MODE NUMBER = UNION (INT, REAL);
CHAR error char = "*";

# REM is not the same as MOD #
OP REM = (INT i, j) INT: (i - j * (i OVER j));
PRIO REM = 7;

PROC whole = (NUMBER v, INT width) STRING:
  CASE v IN
    (INT x):
      (INT length := ABS width - (x < 0 OR width > 0 | 1 | 0),
        INT n := ABS x;
        IF width = 0 THEN
          INT m := n; length := 0;
          WHILE m OVERAB 10; length += 1; m /= 0
            DO SKIP OD
          FI;
          STRING s := sub whole (n, length);
          IF length = 0 OR char in string (error char, LOC INT, s)
            THEN ABS width * error char
          ELSE
            (x < 0 | "-" | : width > 0 | "+" | "") PLUSTO s;
            (width /= 0 | (ABS width - UPB s) * " " PLUSTO s);
            s
          FI),
      (REAL x): fixed (x, width, 0)
  ESAC;

PROC fixed = (NUMBER v, INT width, after) STRING:
  CASE v IN
    (REAL x):
      IF INT length := ABS width - (x < 0 OR width > 0 | 1 | 0);
        after >= 0 AND (length > after OR width = 0)
      THEN REAL y = ABS x;
        IF width = 0
          THEN length := (after = 0 | 1 | 0);
          WHILE y + .5 * .1 ^ after > 10 ^ length
            DO length += 1 OD;
          length += (after = 0 | 0 | after + 1)
        FI;
        STRING s := sub fixed (y, length, after);
        IF NOT char in string (error char, LOC INT, s)
```

## PROGRAMMING WITH ALGOL 68 GENIE

```

THEN (length > UPB s AND y < 1.0 | "0" PLUSTO s);
    (x < 0 | "-" |: width > 0 | "+" | "") PLUSTO s;
    (width /= 0 | (ABS width - UPB s) * " " PLUSTO s);
    s
ELIF after > 0
THEN fixed (v, width, after - 1)
ELSE ABS width * error char
FI
ELSE ABS width * error char
FI,
(INT x): fixed (REAL(x), width, after)
ESAC;

PROC real = (NUMBER v, INT width, after, exp, modifier) STRING:
CASE v IN
(REAL x):
IF INT before = ABS width - ABS exp -
    (after /= 0 | after + 1 | 0) - 2;
    INT mod aft := after;
    SIGN before + SIGN after > 0
THEN STRING s, REAL y := ABS x, INT p := 0;
    standardize (y, before, after, p);
    IF modifier > 0
    THEN WHILE p REM modifier = 0
        DO y *:= 10;
            p -= 1;
            IF mod aft > 0
            THEN mod aft -= 1
            FI
        OD
    ELSE WHILE y < 10.0 ^ (- modifier - 1)
        DO y *:= 10;
            p -= 1;
            IF mod aft > 0
            THEN mod aft -= 1
            FI
        OD;
    WHILE y > 10.0 ^ - modifier
        DO y /= 10;
            p += 1;
            IF mod aft > 0
            THEN mod aft += 1
            FI
        OD
    FI;
s := fixed (SIGN x * y,
    SIGN width * (ABS width - ABS exp - 1), mod aft) +
    "E" + whole (p, exp);
IF exp = 0 OR char in string (error char, LOC INT, s)
THEN float (x, width,

```

```

        (mod aft /= 0 | mod aft - 1 | 0),
        (exp > 0 | exp + 1 | exp - 1))
    ELSE s
    FI
    ELSE ABS width * error char
    FI,
    (INT x): float (REAL(x), width, after, exp)
ESAC;

PROC float = (NUMBER v, INT width, after, exp) STRING:
    real (v, width, after, exp, 1);

PROC sub whole = (NUMBER v, INT width) STRING:
    CASE v IN
        (INT x):
            BEGIN STRING s, INT n := x;
                WHILE dig char (n MOD 10) PLUSTO s;
                    n OVERAB 10; n /= 0
                DO SKIP OD;
                (UPB s > width | width * error char | s)
            END
    ESAC;

PROC sub fixed = (NUMBER v, INT width, after) STRING:
    CASE v IN
        (REAL x):
            BEGIN STRING s, INT before := 0;
                REAL y := x + .5 * .1 ^ after;
                PROC choosedig = (REF REAL y) CHAR:
                    dig char ((INT c := ENTIER (y *:= 10.0);
                        (c > 9 | c := 9); y -=: c; c));
                WHILE y >= 10.0 ^ before DO before +=: 1 OD;
                y /=: 10.0 ^ before;
                TO before DO s PLUSAB choose dig (y) OD;
                (after > 0 | s PLUSAB ".");
                TO after DO s PLUSAB choose dig (y) OD;
                (UPB s > width | width * errorchar | s)
            END
    ESAC;

PROC standardize = (REF REAL y, INT before, after, REF INT p) VOID:
    BEGIN
        REAL g = 10.0 ^ before, REAL h := g * .1;
        WHILE y >= g DO y *:= .1; p +=: 1 OD;
        (y /= 0.0 | WHILE y < h DO y *:= 10.0; p -=: 1 OD);
        (y + .5 * .1 ^ after >= g | y := h; p +=: 1)
    END;

PROC dig char = (INT x) CHAR: "0123456789abcdef" [x + 1];

SKIP

```

## 11.16 Anti-aliasing in drawing

This example draws an anti-aliased circle using the Wu algorithm.

```

FILE f;
INT resolution = 1000;
open (f, "circle.pnm", stand draw channel);
make device (f, "pnm", whole (resolution, 0) + "x" + whole (resolution, 0));

PROC circle anti aliased = (INT x0, y0, r) VOID:
(# This is the Wu algorithm.#
  REAL rgb r = 0.5, rgb g = .5, rgb b = .5, alpha = 0.6;
  PROC point = (INT x, x0, y, y0) VOID:
    draw point (f, (x + x0) / resolution, (y + y0) / resolution);
    draw point (f, (y + x0) / resolution, (x + y0) / resolution);
    draw point (f, (x0 - x) / resolution, (y + y0) / resolution);
    draw point (f, (x0 - y) / resolution, (x + y0) / resolution);
    draw point (f, (x + x0) / resolution, (-y + y0) / resolution);
    draw point (f, (y + x0) / resolution, (-x + y0) / resolution);
    draw point (f, (x0 - x) / resolution, (-y + y0) / resolution);
    draw point (f, (x0 - y) / resolution, (-x + y0) / resolution)
  );
  FOR x FROM 0 TO r
  WHILE REAL y = sqrt (r * r - x * x);
    INT y1 = ENTIER y;
    x <= y
  DO IF y1 = y
    THEN draw colour (f, alpha * rgb r, alpha * rgb g, alpha * rgb b);
      point (x, x0, y1, y0)
    ELSE REAL d0 = ABS (y1 - y) * alpha;
      draw colour (f, d0 * rgb r, d0 * rgb g, d0 * rgb b);
      point (x, x0, y1 + 1, y0)
    FI;
  FOR y2 FROM y1 DOWNT0 0
  DO REAL d = (1 - alpha) * sqrt (r * r - x * x - y2 * y2) / r + alpha;
    draw colour (f, d * rgb r, d * rgb g, d * rgb b);
    point (x, x0, y2, y0)
  OD
OD
);

circle anti aliased (resolution % 2, resolution % 2, resolution % 3);
close (f)

```

## 11.17 Accessing a PostgreSQL database

This example assumes a database `mydb` owned by postgres user `marcel` and a table `weather` constructed following the example in the PostgreSQL 8.1 manual, chapter 2.

```

FILE z; # Holds the connection #
STRING str; # The string associated with 'z' to do IO with the server #

# Connect to the server #
IF pq connectdb (z, "dbname=mydb user=marcel", str) = 0
THEN print ((pq result error message (z); str));
    exit
ELSE printf (($"protocol="g (0)x"server="g (0)x"socket="g (0)x"pid="g (0)l$,
    pq protocol version (z),
    pq server version (z),
    pq socket (z),
    pq back end pid (z)))
FI;

# Get the complete table #
IF pq exec (z, "SELECT * FROM weather") = 0
THEN print ((pq result error message (z); str));
    exit
FI;

printf (($"tuples="g (0)x"fields="g (0)l$,
    pq ntuples (z), pq nfields (z)));

# Print column names #
FOR i TO pq nfields (z)
DO IF pq fname (z, i) = 0
    THEN print ((str, blank))
    FI
OD;
newline (standout);

# Print row entries #
FOR i TO pq ntuples (z)
DO FOR k TO pq nfields (z)
    DO IF pq getisnull (z, i, k) = 0
        THEN print ("null")
        ELIF pq getvalue (z, i, k) = 0
        THEN print ((str, blank))
        ELSE print ("error")
        FI
    OD;
    newline (standout)
OD;

# Print average high temperature, convert mode as a demo #
IF pq exec (z, "SELECT avg (temp_hi) FROM weather") = 0
THEN print ((pq result error message (z); str));

```

## PROGRAMMING WITH ALGOL 68 GENIE

```
    exit
FI;
printf (($"tuples="g (0)x"fields="g (0)/$, pq ntuples (z), pq nfields (z)));
IF pq getisnull (z, 1, 1) = 0
THEN print ("(null)")
ELIF pq getvalue (z, 1, 1) = 0
THEN print ((REAL avg temp hi; get (z, avg temp hi); avg temp hi))
ELSE print ("error")
FI;

# Close connection to the server #
exit:
pq finish (z)
```

### 11.18 Filter for SOUND values

Next application demonstrates how to program applications manipulating sound values. This particular example filters a sound by Fourier-transformation using routines described in section 10.21:

```
COMMENT
Band-pass filter for a sound file on a second-per-second basis using FFT.
Limitations:
(1) sound duration must be a whole number of seconds
(2) sample rate must be even
COMMENT

# Get the sound file #
SOUND w, FILE in, out;
print ((newline, "File:  "));
STRING fn = read string;
open (in, fn + ".wav", standback channel);
get bin (in, w);
close (in);
print ((newline, "rate=", RATE w));
print ((newline, "samples=", SAMPLES w));
print ((newline, "bits=", RESOLUTION w));
print ((newline, "channels=", CHANNELS w));
INT secs = SAMPLES w OVER RATE w;
print ((newline, "duration=", secs));
# If next number is large, FFT is inefficient #
print ((newline, "prime factors=", UPB prime factors (RATE w)));
# Get filter parameters #
print ((newline, "low frequency:  "));
```



```

REAL low f = read real;
print ((newline, "high frequency:  "));
REAL high f = read real;
# Apply a band filter.
Not the most efficient code, but obvious #
FOR i TO secs
DO print ((newline, i, collections, garbage));
  FOR j TO CHANNELS w
  DO # Get data for this second #
    [RATE w] REAL samples;
    FOR k TO RATE w
    DO samples[k] := get sound (w, j, (i - 1) * RATE w + k)
    OD;
    # Forward transform to determine spectrum #
    [RATE w] COMPLEX spectrum := fft forward (samples);
    # Apply filter.
    Simplifications apply since the FFT is over one second,
    hence N*dt=1 and the frequencies are whole numbers #
    FOR m TO RATE w OVER 2 + 1
    DO # f2 is the corresponding negative frequency with f1 #
      INT f1 = m, f2 = (f1 = 1 | 1 | RATE w - f1 + 2);
      IF f1 < low f OR f1 > high f
      THEN spectrum[f1] := spectrum[f2] := 0
      FI
    OD;
    # Restore sample data for this second #
    samples := fft inverse (spectrum);
    FOR k TO RATE w
    DO set sound (w, j, (i - 1) * RATE w + k, ENTIER samples[k])
    OD
  OD
OD;
# Put the filtered sound file #
open (out, fn + ".fft.wav", standback channel);
put bin (out, w);
close (out)

```

## 11.19 Monte Carlo simulation

In chemistry and physics, Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to compute their results. The term was coined in the 1940s by scientists at the Los Alamos National Laboratory. Monte Carlo methods

## PROGRAMMING WITH ALGOL 68 GENIE

are used when it is unfeasible or impossible to compute an exact result with a deterministic algorithm; Monte Carlo is especially useful in studying systems with a large number of coupled degrees of freedom, such as fluids, disordered materials, strongly coupled solids, and cellular structures. In a wider perspective, Monte Carlo methods are useful for modeling phenomena with significant uncertainty in input values, such as business risk-calculations. These methods are also widely used in mathematics: a classic use is the evaluation of definite integrals, particularly multidimensional integrals with complicated boundary conditions.

This program generates trajectories for atoms in a binary Lennard-Jones TVN ensemble through the Metropolis Monte Carlo method. Instead of a Verlet table this program uses a sorted list along the  $z$ -axis. This gives a comparable set estimate for interacting neighbours. The program stores the energy for each particle to calculate initial energy quickly. If a state is accepted then the energies are updated. This saves an energy calculation if a state is not accepted.

```

BOOL continuing = FALSE; # FALSE means starting from FCC lattice #
STRING jobname = "rlj10", INT states = 100;
INT n x = 3, n y = 3, n z = 3;
INT n atoms = 4 * n x * n y * n z;
REAL rho = 1.0, cut = 21/6, e over kt = 2.5;
REAL u0 = - 1j e 12 6 (e over kt, 1, cut);

# Basic vector and matrix stuff #

MODE VECTOR = STRUCT (REAL x, y, z), VECTOR zero = (0, 0, 0);
OP + = (VECTOR u, v) VECTOR:
    (x OF u + x OF v, y OF u + y OF v, z OF u + z OF v),
+:= = (REF VECTOR u, VECTOR v) REF VECTOR: u := u + v,
- = (VECTOR u, v) VECTOR:
    (x OF u - x OF v, y OF u - y OF v, z OF u - z OF v),
-:= = (REF VECTOR u, VECTOR v) REF VECTOR: u := u - v,
* = (VECTOR u, REAL s) VECTOR:
    (s * x OF u, s * y OF u, s * z OF u),
*:= = (REF VECTOR u, REAL s) REF VECTOR: u := u * s,
/ = (VECTOR u, INT v) VECTOR: u * (1 / v),
/ = (VECTOR u, REAL v) VECTOR: u * (1 / v);

OP NORM = (VECTOR u) REAL: sqrt (u * u),
VOL = (VECTOR u) REAL: x OF u * y OF u * z OF u,
TRACE = (VECTOR u) REAL: x OF u + y OF u + z OF u;

# Operators for periodic boundary conditions #

OP NEAREST = (REAL r, REAL l) REAL:
    (REAL hl = l * 0.5; r > hl | r - l | : r < - hl | r + l | r),
NEAREST = (VECTOR r, VECTOR l) VECTOR:
    (x OF r NEAREST x OF l, y OF r NEAREST y OF l, z OF r NEAREST z OF l),
```

```

INSIDE = (REAL r, REAL l) REAL:
  (r < 0 | r + l | : r >= l | r - l | r),
INSIDE = (VECTOR r, VECTOR l) VECTOR:
  (x OF r INSIDE x OF l, y OF r INSIDE y OF l, z OF r INSIDE z OF l),
PRIO NEAREST = 2, INSIDE = 2;

# Atoms, ensembles, et cetera #

MODE ATOM = STRUCT (VECTOR pos, LONG REAL u),
  NEIGHBOUR = STRUCT (INT n, VECTOR dr, REAL r, REAL u),
  ENSEMBLE = STRUCT ([n atoms] ATOM parts, [n atoms] INT admin, link,
    VECTOR box, INT steps, accepted,
    REAL step size, energy, VECTOR stress);

# Basic routines to work with an ensemble #

PROC dump box = (REF ENSEMBLE s) VOID:
  (FILE out;
  open (out, job name + ".bin", stand back channel);
  put bin (out, s);
  close (out)
  );

PROC read box = (REF ENSEMBLE s) VOID:
  (FILE in;
  open (in, job name + ".bin", stand back channel);
  get bin (in, s);
  close (in)
  );

PROC print stats = (REF FILE f, REF ENSEMBLE s) VOID:
  (putf (f, ($lg(-6)x3(g(0, 4)x2(h(12, 4, 3)x)$,
  steps OF ensemble / n atoms,
  accepted OF ensemble / steps OF ensemble,
  step size OF ensemble,
  n atoms / VOL (box OF s),
  energy OF s / n atoms,
  - TRACE stress OF s / VOL (box OF s) / 3.0))
  );

# Routines to work with a neighbour list #

PROC swap = (REF INT i, REF INT j) VOID:
  (INT k = i; i := j; j := k);

PROC sort = (REF ENSEMBLE s) VOID:
  (REF [] REAL z = z OF pos OF parts OF s;
  REF [] INT admin = admin OF s, link = link OF s;
  FOR i TO UPB admin
  DO admin[i] := i OD;
  FOR i TO UPB z - 1
  DO FOR j FROM i + 1 TO UPB z

```

## PROGRAMMING WITH ALGOL 68 GENIE

```
        DO IF z[admin[i]] > z[admin[j]]
            THEN swap (admin[i], admin[j])
            FI
        OD
    OD;
    FOR i TO UPB z
    DO link[admin[i]] := i OD
    );

PROC resort = (REF ENSEMBLE s, INT m) VOID:
    (# Rearrange the neighbour list assuming small displacements #
    REF [] REAL z = z OF pos OF parts OF s;
    REAL z m = z[m];
    REF [] INT admin = admin OF s, link = link OF s;
    INT i := link[m];
    # Shift particle up the list #
    WHILE i < n atoms ANDF z m > z[admin[i + 1]]
    DO admin[i] := admin[i + 1]; link[admin[i]] := i;
        i += 1;
        admin[i] := m; link[m] := i
    OD;
    # Shift particle down the list #
    WHILE i > 1 ANDF z m < z[admin[i - 1]]
    DO admin[i] := admin[i - 1]; link[admin[i]] := i;
        i -= 1;
        admin[i] := m; link[m] := i
    OD
    );

# Routines to work with energy tables #

PROC atom energy = (REF [] NEIGHBOUR list, INT atom) LONG REAL:
    (# Total energy of particle 'atom' with neighbour list 'list' #
    LONG REAL sum := 0;
    FOR i TO UPB list
    DO REF NEIGHBOUR n = list[i];
        sum += (u OF n := u0 + l j e 12 6 (e over kt, 1.0, r OF n))
    OD;
    sum
    );

PROC atom stress = ([] NEIGHBOUR list, INT atom) VECTOR:
    (# Stress of particle 'atom' with neighbour list 'list' #
    VECTOR sum := zero;
    FOR i TO UPB list
    DO NEIGHBOUR n = list[i];
        IF atom < n OF n
        THEN VECTOR dr = dr OF n, REAL r = r OF n;
            VECTOR f = dr / r * l j f 12 6 (e over kt, 1.0, r);
            # Sum diagonal tensor elements only #
```

## EXAMPLE a68g PROGRAMS

```

        sum +=
            VECTOR (x OF dr * x OF f, y OF dr * y OF f, z OF dr * z OF f)
    FI
OD;
sum
);

PROC collect = (REF ENSEMBLE s, INT atom) REF [] NEIGHBOUR:
    (REF [] ATOM p = parts OF s;
    REF [] INT a = admin OF s, l = link OF s;
    NEW [n atoms] NEIGHBOUR list;
    INT k, n := 0;
    # Work down the neighbour list, periodically #
    PROC prev = (INT n) INT: (n = 1 | n atoms | n - 1);
    k := prev (l[atom]);
    WHILE VECTOR dr = (pos OF p[atom] - pos OF p[a[k]]) NEAREST box OF s;
        ABS (z OF dr) < cut
    DO IF REAL r = NORM dr; r < cut
        THEN list[n += 1] := (a[k], dr, r, SKIP)
        FI;
        k := prev (k)
    OD;
    # Work up the neighbour list, periodically #
    PROC next = (INT n) INT: (n = n atoms | 1 | n + 1);
    k := next (l[atom]);
    WHILE VECTOR dr = (pos OF p[atom] - pos OF p[a[k]]) NEAREST box OF s;
        ABS (z OF dr) < cut
    DO IF REAL r = NORM dr; r < cut
        THEN list[n += 1] := (a[k], dr, r, SKIP)
        FI;
        k := next (k)
    OD;
    list[1 .. n]
);

PROC energy and stress = (REF ENSEMBLE s) VOID:
    (REF [] ATOM p = parts OF s;
    energy OF s := 0;
    stress OF s := zero;
    FOR i TO n atoms
    DO REF [] NEIGHBOUR list = collect (s, i);
        energy OF s += SHORTEN (u OF p[i] := atom energy (list, i));
        stress OF s += atom stress (list, i)
    OD
);

PROC rework u table = (REF ENSEMBLE s, REF [] NEIGHBOUR init, final) VOID:
    (REF [] ATOM p = parts OF s;
    FOR i TO UPB init
    DO REF NEIGHBOUR neigh = init[i];

```

## PROGRAMMING WITH ALGOL 68 GENIE

```

    u OF p[n OF neigh] -= u OF neigh
OD;
FOR i TO UPB final
DO REF NEIGHBOUR neigh = final[i];
    u OF p[n OF neigh] += u OF neigh
OD
);

PROC mc step = (REF ENSEMBLE s) VOID:
  (# Generate a new state by moving all atoms a little bit #
  REF [] ATOM p = parts OF s;
  FOR k TO n atoms
  DO LONG REAL u init = u OF p[k];
    PROC rnd step = REAL: (2 * random - 1) * step size OF s;
    VECTOR pos init := pos OF p[k];
    # Move to new position #
    VECTOR disp = (rnd step, rnd step, rnd step);
    VECTOR pos final := (pos init + disp) INSIDE box OF s;
    pos OF p[k] := pos final; resort (s, k);
    REF [] NEIGHBOUR l final = collect (s, k);
    LONG REAL u final = atom energy (l final, k);
    # Restore old situation #
    pos OF p[k] := pos init; resort (s, k);
    # Use Boltzmann's criterion #
    REAL u diff = SHORTEN (u final - u init);
    IF u diff < 0 ORF exp (- u diff) > random
    THEN accepted OF s += 1;
      REF [] NEIGHBOUR l init = collect (s, k);
      LONG REAL u init 2 = atom energy (l init, k);
      IF ABS (u init - u init 2) > 0.01
      THEN print ((newline, "u table error: ", u init, u init 2))
      FI;
      rework u table (s, l init, l final);
      u OF p[k] := u final;
      pos OF p[k] := pos final; resort (s, k)
    FI;
    steps OF s += 1
  OD;
  # Adapt step size according to optimum for hard sphere system #
  IF accepted OF s / steps OF s > 0.2
  THEN (step size OF s < 1 | step size OF s *= 1.05)
  ELSE (step size OF s > 1e-6 | step size OF s /= 1.05)
  FI
);

PROC gen fcc = (REF ENSEMBLE s) VOID:
  (INT m := 0;
  REF [] ATOM p = parts OF s;
  REAL frac = (4 / rho)^(1/3);
  box OF s := (n x * frac, n y * frac, n z * frac);

```

```

FOR k FROM 0 TO n z - 1
DO FOR i FROM 0 TO n x - 1
  DO FOR j FROM 0 TO n y - 1
    DO p[m += 1] := ((i, j, k), 0);
      p[m += 1] := ((i + 0.5, j + 0.5, k), 0);
      p[m += 1] := ((i + 0.5, j, k + 0.5), 0);
      p[m += 1] := ((i, j + 0.5, k + 0.5), 0)
    OD
  OD
OD;
FOR i TO m
DO pos OF p[i] :=
  (pos OF p[i] + VECTOR (0.25, 0.25, 0.25)) * frac INSIDE box OF s
OD
);

ENSEMBLE ensemble;
# Read old box or generate a new one #
IF continuing
THEN read box (ensemble)
ELSE steps OF ensemble := accepted OF ensemble := 0;
  step size OF ensemble := 0.5;
  gen fcc (ensemble);
  sort (ensemble);
  energy and stress (ensemble)
FI;
# Monte Carlo simulation #
FOR i TO states
DO TO 10
  DO mc step (ensemble)
  OD;
  energy and stress (ensemble);
  print stats (standout, ensemble)
OD;
dump box (ensemble)

```

## 11.20 Hamming numbers

Hamming numbers are of the form  $2^i \times 3^j \times 5^k$ . This program generates them in a trivial iterative way but keeps the series needed to generate the numbers as short as possible using flexible rows; on the downside, it spends considerable time on garbage collection.

```

PR precision=100 PR
MODE SERIES = FLEX [1 : 0] UNT; # Initially, no elements #

```





## **Part IV**

# **Algol 68 Revised Report**



# REVISED REPORT ON THE ALGORITHMIC LANGUAGE

## ALGOL 68

Edited by

A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster,  
M. Sintzoff, C.H. Lindsey, L.G.T. Meertens and R.G. Fisker

---

This Report has been accepted by Working Group 2.1, reviewed by Technical Committee 2 on Programming and approved for publication by the General Assembly of The International Federation for Information Processing. Reproduction of the Report, for any purpose, but only of the whole text, is explicitly permitted without formality.

Chapter 20, "Specification of partial parametrization proposal", is not a part of the Algol 68 Revised Report, and is distributed with kind permission of the author of this proposal, C.H. Lindsey.

---

The Revised Report on Algol 68 has been previously published as:

- Mathematical Centre Tracts **50** Mathematisch Centrum Amsterdam [1976].
- Acta Informatica **5** 1-236 [1976].
- SIGPLAN Notices **12**(5) 1-70 [1977].
- Springer-Verlag, ISBN 3-540-07592-5, ISBN 0-387-07592-5 [1976].



# Chapter 12

## Foreword to the translation

This is a translation {15<sub>1</sub>.1.5} of the Algol 68 Revised Report into L<sup>A</sup>T<sub>E</sub>X, for distribution with Algol 68 Genie, an Open Source Algol 68 compiler-interpreter. Since Algol 68 Genie became available, it has become clear that there still are many people who are interested in an Algol 68 implementation; hence it is expected that there are also people interested in having access to the formal defining document, the Algol 68 Revised Report, which is unfortunately out of print. It is believed that a L<sup>A</sup>T<sub>E</sub>X translation may well meet this need and enable people to study the formal description of Algol 68.

This L<sup>A</sup>T<sub>E</sub>X translation is based on an HTML translation that was formerly distributed with Algol 68 Genie. Since the translation is a part of the Algol 68 Genie documentation, chapter numbering differs from that of the original report and therefore cross-references have also been renumbered. Chapter numbers in cross-references are appended with the corresponding chapter number in the original Revised Report: for instance 24<sub>9</sub>.4.d refers to chapter 24 in this translation, but to chapter 9 in the original Revised Report. In cross-referencing {15<sub>1</sub>.1.3.4.f} a final 1 is omitted as is done in the original Revised Report, but since L<sup>A</sup>T<sub>E</sub>X does not support 10 to appear as *A* in cross-references, the substitution of 10 for *A* is not maintained in this translation, and points are not omitted as to improve legibility of cross-references. For instance, a cross-reference {A341A} in the original Revised Report will appear as {25<sub>10</sub>.3.4.1.A} in this translation.

Lay-out may differ in places from that of the original publication. The name *ALGOL 68* was replaced by *Algol 68* since the latter is the preferred way of writing the name these days. Code examples in this translation adopt upper-stropping for bold symbols since this has been the predominant notation in actual Algol 68 programming over the years; for instance the **bold-begin-symbol** {15<sub>1</sub>.3.3.d} is typeset as BEGIN in this translation.

This translation contains comments which are not part of the original text and are con-

## ALGOL 68 REVISED REPORT

tained in footnotes, for instance errata from ALGOL BULLETIN as well as a-posteriori comments or - remarks.

# Chapter 13

## Acknowledgments

{Habent sua fata libelli.  
De litteris. Terentianus Maurus. }

Working Group 2.1 on ALGOL of the International Federation for Information Processing has discussed the development of "Algol X", a successor to Algol 60 [3] since 1963. At its meeting in Princeton in May 1965, WG 2.1 invited written descriptions of the language based on the previous discussions. At the meeting in St Pierre de Chartreuse near Grenoble in October 1965, three reports describing more or less complete languages were amongst the contributions, by Niklaus Wirth [8], Gerhard Seegmüller [6], and Aad van Wijngaarden [9]. In [6] and [8], the descriptive technique of [3] was used, whereas [9] featured a new technique for language design and definition. Other significant contributions available were papers by Tony Hoare [2] and Peter Naur [4, 5].

At subsequent meetings between April 1966 and December 1968, held in Kootwijk near Amsterdam, Warsaw, Zandvoort near Amsterdam, Tirrenia near Pisa and North Berwick near Edinburgh, a number of successive approximations to a final report, commencing with [10] and followed by a series numbered MR 88, MR 92, MR 93, MR 95, MR 99 and MR 100, were submitted by a team working in Amsterdam, consisting first of A. van Wijngaarden and Barry Mailloux, later reinforced by John Peck, and finally by Kees Koster. Versions were used during courses on the language held at various centres, and the experience gained in explaining the language to skilled audiences and the reactions of the students influenced the succeeding versions. The final version, MR 101 [11], was adopted by the Working Group on December 20<sup>th</sup> 1968 in Munich, and was subsequently approved for publication by the General Assembly of IFIP. Since that time, it has been published in Numerische Mathematik [12], and translations have been made into Russian [13], into German [14], into French [15], and into Bulgarian [16]. An "Informal Introduction", for the benefit of the uninitiated reader, was also prepared at the request of the Working Group [18].

## ALGOL 68 REVISED REPORT

The original authors acknowledged with pleasure and thanks the wholehearted cooperation, support, interest, criticism and violent objections from members of WG 2.1 and many other people interested in Algol. At the risk of embarrassing omissions, special mention should be made of Jan Garwick, Jack Merner, Peter Ingerman and Manfred Paul for [1], the Brussels group consisting of M. Sintzoff, P. Branquart, J. Lewi and P. Wodon for numerous brainstormings, A.J.M. van Gils of Apeldoorn, G. Goos and his group at Munich, also for [7], G.S. Tseytin of Leningrad, and L.G.L.T. Meertens and J.W. de Bakker of Amsterdam. An occasional choice of a, not inherently meaningful, identifier in the sequel may compensate for not mentioning more names in this section.

Since the publication of the Original Report, much discussion has taken place in the Working Group concerning the further development of the language. This has been influenced by the experience of many people who saw disadvantages in the original proposals and suggested revised or extended features. Amongst these must be mentioned especially: I.R. Currie, Susan G. Bond, J.D. Morison and D. Jenkins of Malvern (see in [17]), in whose dialect of Algol 68 many features of this Revision may already be found; P. Branquart, J.P. Cardinael and J. Lewi of Brussels, who exposed many weaknesses (see in [17]); Ursula Hill, H. Woessner and H. Scheidig of Munich, who discovered some unpleasant consequences: the contributors to the Rapport d'Evaluation [19]: and the many people who served on the Working Group subcommittees on Maintenance and Improvements (convened by M. Sintzoff) and on Transput (convened by C.H. Lindsey). During the later stages of the revision, much helpful advice was given by H. Boom of Edmonton, W. Freeman of York, W.J. Hansen of Vancouver, Mary Zosel of Livermore, N. Yoneda of Tokyo, M. Rain of Trondheim, L. Ammeraal, D. Grune, H. van Vliet and R. van Vliet of Amsterdam, G. van der Mey of Delft, and A.A. Baehrs and A.F. Rar of Novosibirsk. The editors of this revision also wish to acknowledge that the wholehearted cooperation, support, interest, criticism and violent objections on the part of the members of WG 2.1 have continued unabated during this time.

- [1] J.V. Garwick, J.M. Merner, P.Z. Ingerman and M. Paul, Report of the ALGOL-X - I-O Subcommittee, WG 2.1 Working Paper, July 1966.
- [2] C.A.R. Hoare, Record Handling, WG 2.1 Working Paper, October 1965; also AB.21.3.6. November 1965.
- [3] P. Naur (Editor), Revised Report on the Algorithmic Language ALGOL 60, Regne-centralen, Copenhagen, 1962, and elsewhere.
- [4] P. Naur, Proposals for a new language, AB.18.3.9, October 1964.
- [5] P. Naur, Proposals for introduction on aims, WG 2.1 Working Paper, October 1965.
- [6] G. Seegmueller, A proposal for a basis for a Report on a Successor to ALGOL 60. Bavarian Acad. Sci., Munich, October 1965.
- [7] G. Goos, H. Scheidig, G. Seegmueller and H. Walther, Another proposal for ALGOL 67. Bavarian Acad. Sci., Munich, May 1967.



## ACKNOWLEDGMENTS

- [8] N. Wirth, A Proposal for a Report on a Successor of ALGOL 60, Mathematisch Centrum, Amsterdam, MR 75, October 1965.
- [9] A. van Wijngaarden, Orthogonal Design and Description of a Formal Language, Mathematisch Centrum, Amsterdam, MR 76, October 1965.
- [10] A. van Wijngaarden and B.J. Mailloux, A Draft Proposal for the Algorithmic Language ALGOL X, WG 2.1 Working Paper, October 1966.
- [11] A. van Wijngaarden (Editor), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Report on the Algorithmic Language ALGOL 68, Mathematisch Centrum. Amsterdam, MR 101, February 1969.
- [12] idem, Numerische Mathematik, Vol. 14, pp. 79-218. 1969.
- [13] Soobshchenie ob algoritmicheskom yazyke ALGOL 68, translation into Russian by A.A. Baehrs, A.P. Ershov, L.L. Zmievsckaya and A.F. Rar. Kybernetika, Kiev, Part 6 of 1969 and Part 1 of 1970.
- [14] Bericht ueber die Algorithmische Sprache ALGOL 68, translation into German by I.O. Kerner, Akademie-Verlag, Berlin, 1972.
- [15] Définition du Langage Algorithmique ALGOL 68, translation into French by J. Buffet, P. Arnal, A. Quéré (Eds.), Hermann, Paris, 1972.
- [16] Algoritmichniyat yezik ALGOL 68, translation into Bulgarian by D. Toshkov and St. Buchvarov, Nauka i Yzkustvo, Sofia, 1971.
- [17] J.E.L. Peck (Ed.), ALGOL 68 Implementation (proceedings of the IFIP working conference held in Munich in 1970), North Holland Publishing Company, 1971.
- [18] C.H. Lindsey and S.G. van der Meulen, Informal introduction to ALGOL 68, North Holland Publishing Company 1971.
- [19] J.C. Boussard and J.J. Duby (Eds.), Rapport d'Evaluation ALGOL 68, Revue d'Informatique et de Recherche Opérationelle, B2, Paris, 1970.

# ALGOL 68 REVISED REPORT

# Chapter 14

## Introduction

### 14.1 Aims and principles of design

a) In designing the Algorithmic Language Algol 68, Working Group 2.1 on ALGOL of the International Federation for Information Processing expresses its belief in the value of a common programming language serving many people in many countries.

b) Algol 68 is designed to communicate algorithms, to execute them efficiently on a variety of different computers, and to aid in teaching them to students.

c) This present Revision of the language is made in response to the directive of the parent committee, IFIP TC 2, to the Working Group to "keep continually under review experience obtained as a consequence of this original publication, so that it may institute such corrections and revisions to the Report as become desirable". In deciding to bring forward this Revision at the present time, the Working Group has tried to keep in balance the need to accumulate the maximum amount of experience of the problems which arose in the language originally defined, as opposed to the needs of the many teams at present engaged in implementation, for whom an early and simple resolution of those problems is imperative.

d) Although the language as now revised differs in many ways from that defined originally, no attempt has been made to introduce extensive new features and, it is believed, the revised language is still clearly recognizable as "Algol 68". The Working Group has decided that this present revision should be "the final definition of the language Algol 68", and the hope is expressed that it will be possible for implementations at present in preparation to be brought into line with this standard.

e) The Working Group may, from time to time, define sublanguages and extended capabilities, by means of Addenda to this Report, but these will always be built on the language here defined as a firm foundation. Moreover, variants more in conformity with natural languages other than English may be developed. To coordinate these activities, and to

maintain contact with implementers and users, a Subcommittee on Algol 68 Support has been established by the Working Group.

f) The members of the Group, influenced by several years of experience with Algol 60 and other programming languages, have always accepted the following as their aims:

### 14.1.1 Completeness and clarity of description

The Group wishes to contribute to the solution of the problems of describing a language clearly and completely. The method adopted in this Report is based upon a formalized two-level grammar, with the semantics expressed in natural language, but making use of some carefully and precisely defined terms and concepts. It is recognized, however, that this method may be difficult for the uninitiated reader.

### 14.1.2 Orthogonal design

The number of independent primitive concepts has been minimized in order that the language be easy to describe, to learn, and to implement. On the other hand, these concepts have been applied "orthogonally" in order to maximize the expressive power of the language while trying to avoid deleterious superfluities.

### 14.1.3 Security

Algol 68 has been designed in such a way that most syntactical and many other errors can be detected easily before they lead to calamitous results. Furthermore, the opportunities for making such errors are greatly restricted.

### 14.1.4 Efficiency

Algol 68 allows the programmer to specify programs which can be run efficiently on present-day computers and yet do not require sophisticated and time-consuming optimization features of a compiler; see, e.g., 26<sub>11</sub>.7.

#### 14.1.4.1 Static mode checking

The syntax of Algol 68 is such that no mode checking during run time is necessary, except when the programmer declares a **UNITED-variable** and then, in a **conformity-clause**, explicitly demands a check on its mode.

#### 14.1.4.2 Mode-independent parsing

The syntax of Algol 68 is such that the parsing of a program can be performed independently of the modes of its constituents. Moreover, it can be determined in a finite number of steps whether an arbitrary given sequence of symbols is a program.

#### 14.1.4.3 Independent compilation

The syntax of Algol 68 is such that the main-line programs and procedures can be compiled independently of one another without loss of object-program efficiency provided that, during each independent compilation, specification of the mode of all nonlocal quantities is provided: see the remarks after 16<sub>2</sub>.2.2.c.

#### 14.1.4.4 Loop optimization

Iterative processes are formulated in Algol 68 in such a way that straightforward application of well-known optimization techniques yields large gains during run time without excessive increase of compilation time.

#### 14.1.4.5 Representations

Representations of Algol 68 symbols have been chosen so that the language may be implemented on computers with a minimal character set. At the same time implementers may take advantage of a larger character set, if it is available.

## 14.2 Comparison with Algol 60

a) Algol 68 is a language of wider applicability and power than Algol 60. Although influenced by the lessons learned from Algol 60, Algol 68 has not been designed as an expansion of Algol 60 but rather as a completely new language based on new insight into the essential, fundamental concepts of computing and a new description technique.

b) The result is that the successful features of Algol 60 reappear in Algol 68 but as special cases of more general constructions, along with completely new features. It is, therefore, difficult to isolate differences between the two languages: however, the following sections are intended to give insight into some of the more striking differences.

### 14.2.1 Values in Algol 68

- a) Whereas Algol 60 has values of the types **integer**, **real** and **Boolean**, Algol 68 features an infinity of "modes", i.e., generalizations of the concept "type".
- b) Each plain value is either arithmetic, i.e., of '**integral**' or '**real**' mode and then it is of one of several sizes, or it is of '**boolean**' or '**character**' or '**void**' mode. Machine words, considered as sequences of bits or of bytes, may also be handled.
- c) In Algol 60, values can be composed into arrays, whereas in Algol 68, in addition to such "multiple" values, also "structured" values, composed of values of possibly different modes, are defined and manipulated. An example of a multiple value is the character array, which corresponds approximately to the Algol 60 string; examples of structured values are complex numbers and symbolic formulae.
- d) In Algol 68 the concept of a "name" is introduced, i.e., a value which is said to "refer to" another value; such a name-value pair corresponds to the Algol 60 variable. However, a name may take the value position in a name-value pair, and thus chains of indirect addresses can be built up.
- e) The Algol 60 concept of procedure body is generalized in Algol 68 to the concept of "routine", which includes also the formal parameters, and which is itself a value and therefore can be manipulated like any other value.
- f) In contrast with plain values, the significance of a name or routine is, in general, dependent upon the existence of the storage cells referred to or accessed. Therefore, the use of names and routines is subject to some restrictions related to their "scope". However, the syntax of Algol 68 is such that in many cases the check on scope can be made at compile time, including all cases where no use is made of features whose expressive power transcends that of Algol 60.

### 14.2.2 Declarations in Algol 68

- a) Whereas Algol 60 has type declarations, array declarations, switch declarations and procedure declarations, Algol 68 features the **identity-declaration** whose expressive power includes all of these, and more. The **identity-declaration**, although theoretically sufficient in itself, is augmented by the **variable-declaration** for the convenience of the user.
- b) Moreover, in Algol 68, a **mode-declaration** permits the construction of a new mode from already existing ones. In particular, the modes of multiple values and of structured values may be defined in this way; in addition, a union of modes may be defined, allowing each value referred to by a given name to be of any one of the uniting modes.
- c) Finally, in Algol 68, a **priority-declaration** and an **operation-declaration** permit the introduction of new **operators**, the definition of their operation and the extension of the

class of operands of, and the revision of the meaning of, already established operators.

### 14.2.3 Dynamic storage allocation in Algol 68

Whereas Algol 60 (apart from "own dynamic arrays") implies a "stack"-oriented storage-allocation regime, sufficient to cope with objects having nested lifetimes (an object created before another object being guaranteed not to become inaccessible before that second one), Algol 68 provides, in addition, the ability to create and manipulate objects whose lifetimes are not so restricted. This ability implies the use of an additional area of storage, the "heap", in which garbage-collection techniques must be used.

### 14.2.4 Collateral elaboration in Algol 68

Whereas, in Algol 60, statements are "executed consecutively", in Algol 68, **phrases** are "elaborated serially" or "collaterally". This latter facility is conducive to more efficient object programs under many circumstances, since it allows discretion to the implementer to choose, in many cases, the order of elaboration of certain constructs or even, in some cases, whether they are to be elaborated at all. Thus the user who expects his "side effects" to take place in any well determined manner will receive no support from this Report. Facilities for parallel programming, though restricted to the essentials in view of the none-too-advanced state of the art, have been introduced.

### 14.2.5 Standard declarations in Algol 68

The Algol 60 standard functions are all included in Algol 68 along with many other standard declarations. Amongst these are "environment enquiries", which make it possible to determine certain properties of an implementation, and "transput" declarations, which make it possible, at run time, to obtain data from and to deliver results to external media.

### 14.2.6 Some particular constructions in Algol 68

a) The Algol 60 concepts of block, compound statement and parenthesized expression are unified in Algol 68 into the **serial-clause**. A **serial-clause** may be an **expression** and yield a value. Similarly, the Algol 68 **assignation**, which is a generalization of the Algol 60 assignment statement, may be an **expression** and, as such, also yield a value.

b) The Algol 60 concept of subscripting is generalized to the Algol 68 concept of "indexing", which allows the selection not only of a single element of an array but also of subarrays with the same or any smaller dimensionality and with possibly altered bounds.

- c) Algol 68 provides **row-displays** and **structure-displays**, which serve to compose the multiple and structured values mentioned in 14<sub>0</sub>.2.1.c from other, simpler, values.
- d) The Algol 60 for statement is modified into a more concise and efficient **loop-clause**.
- e) The Algol 60 conditional expression and conditional statement, unified into a **conditional-clause**, are improved by requiring them to end with a closing symbol whereby the two alternative clauses admit the same syntactic possibilities. Moreover, the **conditional-clause** is generalized into a **case-clause**, which allows the efficient selection from an arbitrary number of clauses depending on the value of an **integral-expression**, and a **conformity-clause**, which allows a selection depending upon the actual mode of a value.
- f) Some less successful Algol 60 concepts, such as own quantities and integer labels, have not been included in Algol 68, and some concepts, like designational expressions and switches, do not appear as such in Algol 68 but their expressive power is included in other, more general, constructions.

## 14.2.7 Comparison with the language defined in 1968

The more significant changes to the language are indicated in the sections which follow. The revised language will be described in a new edition of the "Informal Introduction to Algol 68" by C.H. Lindsey and S.G. van der Meulen, which accompanied the original Report.

## 14.2.8 Casts and routine texts

Routines without parameters used to be constructed out of a **cast** in which the **cast-of-symbol** (:) appeared. This construct is now one of the forms of the new **routine-text**, which provides for procedures both with and without parameters. A new form of the **cast** has been provided which may be used in contexts previously not possible. Moreover, both **void-casts** and **procedure-PARAMETY-yielding-void-routine-texts** must now contain an explicit **void-symbol**.

## 14.2.9 Extended ranges

The new **range** which is established by the **enquiry-clause** of a **choice-clause** (which encompasses the old **conditional-** and **case-clauses**) or of a **while-part** now extends into the controlled **serial-clause** or **do-part**.



### 14.2.10 Conformity clauses

The **conformity-relation** and the **case-conformity** which was obtained by extension from it are now replaced by a new **conformity-clause**, which is a further example of the **choice-clause** mentioned above.

### 14.2.11 Modes of multiple values

A new class of modes is introduced, for multiple values whose elements are themselves multiple values. Thus one may now write the **declarer [ ] string**.

Moreover, multiple values no longer have "states" to distinguish their flexibility. Instead, flexibility is now a property of those names which refer to multiple values whose size may change, such names having distinctive modes of the form '**reference to flexible ROWS of MODE**'.

### 14.2.12 Identification of operators

Not only may two **operators**, related to each other by the modes of their operands, not be declared in the same **range**, as before, but now, if two such **operators** be declared in different reaches, any attempt to identify from the inner reach the one in the outer reach will fail. This gives some benefit to the implementer and removes a source of possible confusion to the user.

### 14.2.13 Representations

The manner in which **symbols** for newly defined **mode-indications** and **operators** are to be represented is now more closely defined. Thus it is clear that the implementer is to provide a special alphabet of bold-faced, or "stropped", marks from which **symbols** such as **person** may be made, and it is also clear that **operators** such as » are to be allowed.

### 14.2.14 Standard prelude

In order to ease the problems of implementers who might wish to provide variants of the language suitable for environments where English is not spoken, there are no longer any **field-selectors** known to the user in the **standard-prelude**, with the exception of **re** and **im** of the mode **COMPL**. The **identifiers** and other **indicators** declared in the **standard-prelude** could, of course, easily be defined again in some **library-prelude**, but this would not have been possible in the case of **field-selectors**.

### 14.2.15 Line length in transput

The lines (and the pages also) of the "book" used during transput may now, at the discretion of the implementer, be of varying lengths. This models more closely the actual behaviour of most operating systems and of devices such as teleprinters and paper-tape readers.

### 14.2.16 Internal transput

The transput routines, in addition to sending data to or from external media, may now be associated with **row-of-character-variables** declared by the user.

### 14.2.17 Elaboration of formats

The dynamic **replicators** contained in **format-texts** are now elaborated as and when they are encountered during the formatted transput process. This should give an effect more natural to the user, and is easier to implement.

### 14.2.18 Features removed

Certain features, such as proceduring, gommars and formal bounds, have not been included in the revision.

## 14.3 Changes in the method of description

In response to the directive from the Working Group "to make its study easier for the uninitiated reader", the Editors of this revision have rewritten the original Report almost entirely, using the same basic descriptive technique, but applying it in new ways. It is their hope that less "initiation" will now be necessary.

The more significant changes in the descriptive technique are indicated below.

### 14.3.1 Two-level grammar

a) While the syntax is still described by a two-level grammar of the type now widely known by the name "Van Wijngaarden", new techniques for using such grammars have been applied. In particular, the entire identification process is now described in the syntax using

the metanotion "**NEST**", whose terminal metaproducts are capable of describing, and of passing on to the descendent constructs, all the declared information which is available at any particular node of the production tree.

b) In addition, extensive use is made of "predicates". These are notions which are deliberately made to yield blind alleys when certain conditions are not met, and which yield empty terminal productions otherwise. They have enabled the number of syntax rules to be reduced in many cases, while at the same time making the grammar easier to follow by reducing the number of places where a continuation of a given rule might be found.

It has thus been possible to remove all the "context conditions" contained in the original Report.

### 14.3.2 Modes

a) In the original Report, modes were protonotions of possibly infinite length. It was assumed that, knowing how an infinite mode had been obtained, it was decidable whether or not it was the same as some other infinite mode. However, counterexamples have come to light where this was not so. Therefore, it has been decided to remove all infinities from the process of producing a finite **program** and, indeed, this can now be done in a finite number of moves.

b) A mode, essentially, has to represent a potentially infinite tree. To describe it as a protonotion of finite length requires the use of markers '**MU definition**'s and pointers back to those markers '**MU application**'s within the protonotion. However, a given infinite tree can be "spelled" in many ways by this method, and therefore a mode becomes an equivalence class comprised of all those equivalent spellings of that tree. The equivalence is defined in the syntax using the predicates mentioned earlier.

### 14.3.3 Extensions

The need for many of the extensions given in the original Report had been removed by language changes. Some of the remainder had been a considerable source of confusion and surprises. The opportunity has therefore been taken to remove the extension as a descriptive mechanism, all the former extensions now being specified directly in the syntax.

### 14.3.4 Semantics

a) In order to remove some rather repetitious phrases from the semantics, certain technical terms have been revised and others introduced. The grammar, instead of producing a terminal production directly, now does so by way of a production tree. The semantics

is explained in terms of production trees. Paranotions, which designate constructs, may now contain metanotions and "hypernotations" have been introduced in order to designate protonotions.

b) A model of the hypothetical computer much more closely related to a real one has been introduced. The elaboration of each construct is now presumed to take place in an "environ" and, when a new **range** is entered (and, in particular, when a routine is called), a new "locale" is added to the environ. The locale corresponds to the new **range** and, if recursive procedure calls arise, then there exist many locales corresponding to one same routine. This supersedes the method of "textual substitution" used before, and one consequence of this is that the concept of "protection" is no longer required.

c) The concept of an "instance" of a value is no longer used. This simplifies certain portions of the semantics where, formerly, a "new instance" had to be taken, the effects of which were not always clear to see.

### 14.3.5 Translations

The original Report has been translated into various natural languages. The translators were not always able to adhere strictly to the descriptive method, and so the opportunity has been taken to define more clearly and more liberally certain descriptive features which caused difficulties (see 15<sub>1</sub>.1.5).

{True wisdom knows it must comprise  
some nonsense as a compromise,  
lest fools should fail to find it wise.  
Grooks, Piet Hein. }

## **PART I. Preliminary definitions**

# ALGOL 68 REVISED REPORT

# Chapter 15

## Language and metalanguage

### 15.1 The method of description

### 15.2 Introduction

a) Algol 68 is a language in which algorithms may be formulated for computers, i.e., for automata or for human beings. It is defined by this Report in four stages, the "syntax" {b}, the "semantics" {c}, the "representations" {d} and the "standard environment" {e}.

b) The syntax is a mechanism whereby all the constructs of the language may be produced. This mechanism proceeds as follows:

(i) A set of "hyper-rules" and a set of "metaproduction rules" are given {15<sub>1</sub>.1.3.4, 15<sub>1</sub>.1.3.3}, from which "production rules" may be derived:

(ii) A "construct in the strict language" is a "production tree" {15<sub>1</sub>.1.3.2.f} which may be produced by the application of a subset of those production rules; this production tree contains static information {i.e., information known at "compile time"} concerning that construct: it is composed of a hierarchy of descendent production trees, terminating at the lowest level in the "**symbols**"; with each production tree is associated a "nest" of properties, declared in the levels above, which is passed on to the nests of its descendents;

(iii) A "**program** in the strict language" is a production tree for the notion '**program**' {16<sub>2</sub>.2.1.a}. It must also satisfy the "environment condition" {25<sub>10</sub>.1.2}.

c) The semantics ascribes a "meaning" {16<sub>2</sub>.1.4.1.a} to each construct {i.e. to each production tree} by defining the effect (which may, however, be "undefined") of its "elaboration"

{16<sub>2</sub>.1.4.1}. This proceeds as follows:

- (i) A dynamic {i.e., run-time} tree of active "actions" is set up {16<sub>2</sub>.1.4}; typically, an action is the elaboration of some production tree  $T$  in an "environ" consistent with the nest of  $T$ , and it may bring about the elaboration of descendents of  $T$  in suitable newly created descendent environs;
  - (ii) The meaning of a **program** in the strict language is the effect of its elaboration in the empty "primal environ".
- d) A **program** in the strict language must be represented in some "representation language" {24<sub>9</sub>.3.a} chosen by the implementer. In most cases this will be the official "reference language".
- (i) A **program** in a representation language is obtained by replacing the **symbols** of a **program** in the strict language by certain typographical marks {24<sub>9</sub>.3}.
  - (ii) Even the reference language allows considerable discretion to the implementer {24<sub>9</sub>.4.a, b, c}. A restricted form of the reference language in which such freedom has not been exercised may be termed the "canonical form" of the language, and it is expected that this form will be used in algorithms intended for publication.
  - (iii) The meaning of a **program** in a representation language is the same as that of the **program** {in the strict language} from which it was obtained.
- e) An algorithm is expressed by means of a **particular-program**, which is considered to be embedded, together with the standard environment, in a **program-text** {25<sub>10</sub>.1.1.a}. The meaning of a **particular-program** {in the strict or a representation language} is the meaning of the **program** "akin" to that **program-text** {25<sub>10</sub>.1.2.a}.

## 15.3 Pragmatics

{Merely corroborative detail, intended to give artistic verisimilitude to an otherwise bald and unconvincing narrative.  
Mikado, W.S. Gilbert. }

Scattered throughout this Report are "pragmatic" remarks included between the braces "{" and "}". These are not part of the definition of the language but serve to help the reader to understand the intentions and implications of the definitions and to find corresponding sections or rules.



{Some of the pragmatic remarks contain examples written in the reference language. In these examples, **applied-indicators** occur out of context from their **defining-indicators**. Unless otherwise specified, these occurrences identify those in the **standard-** or **particular-preludes** and the **particular-postlude** (25<sub>10</sub>.2, 25<sub>10</sub>.3, 25<sub>10</sub>.5) (e.g. see 25<sub>10</sub>.2.3.12.a for pi, 25<sub>10</sub>.5.1.b for random and 25<sub>10</sub>.5.2.a for stop), or those in:

```
INT i, j, k, m, n; REAL a, b, x, y; BOOL p, q, overflow; CHAR c; FORMAT f;
  BYTES r; STRING s; BITS t; COMPL w, z; REF REAL xx, yy;
  UNION (INT, REAL) uir;
PROC VOID task1, task2;
```

```
[1 : n] REAL x1, y1; FLEX [1 : n] REAL a1; [1 : m, 1 : n] REAL x2;
  [1 : n, 1 : n] REAL y2; [1 : n] INT i1; [1 : m, 1 : n] INT i2;
  [1 : n] COMPL z1;
```

```
PROC x or y = REF REAL: IF random < .5 THEN x ELSE y FI;
PROC ncos = (INT i) REAL: cos (2 × pi × i / n);
PROC nsin = (INT i) REAL: sin (2 × pi × i / n);
PROC finish = VOID: GO TO stop;
```

```
MODE BOOK = STRUCT (STRING text, REF BOOK next);
BOOK draft;
```

```
princeton: grenoble: st pierre de chartreuse: kootwijk: warsaw:
  zandvoort: amsterdam: tirrenia: north berwick: munich:
  finish.
}
```

## 15.3.1 The syntax of the strict language

### 15.3.1.1 Protonotions

a) In the definition of the syntax of the strict language, a formal grammar is used in which certain syntactic marks appear. These may be classified as follows:

- (i) "small syntactic marks", written, in this Report, as  
**"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z", "(", ")";**
- (ii) "large syntactic marks", written, in this Report, as  
**"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9";**

## ALGOL 68 REVISED REPORT

- (iii) "other syntactic marks", written, in this Report, as  
"." ("point"), "," ("comma"), ":" ("colon"), ";" ("semicolon"), "'" ("apostrophe"), "-" ("hyphen") and "\*" ("asterisk").

{Note that these marks are in another type font than that of the marks in this sentence.}

- b) A "protonotion" is a possibly empty sequence of small syntactic marks.
- c) A "notion" is a {nonempty} protonotion for which a production rule can be derived {15<sub>1</sub>.1.3.2.a, 15<sub>1</sub>.1.3.4.d}.
- d) A "metanotion" is a {nonempty} sequence of large syntactic marks for which a metaproduction rule is given or created {15<sub>1</sub>.1.3.3.a}.
- e) A "hypernotation" is a possibly empty sequence each of whose elements either is a small syntactic mark or is a metanotion.

{Thus the class of protonotions {b} is a subclass of the class of hypernotations. Hypernotations are used in metaproduction rules (15<sub>1</sub>.1.3.3), in hyper-rules (15<sub>1</sub>.1.3.4), as paranotions (15<sub>1</sub>.1.4.2) and, in their own right, to "designate" certain classes of protonotions {15<sub>1</sub>.1.4.1}.}

{A "paranotion" is a hypernotation to which certain special conventions and interpretations apply, as detailed in 15<sub>1</sub>.1.4.2.}

- f) A "symbol" is a protonotion ending with '**symbol**'. {Note that the paranotion **symbol** {24<sub>9</sub>.1.1.h} designates a particular occurrence of such a protonotion.}

{Examples:

- b) '**variable point**'
- c) '**variable point numeral**' {23<sub>8</sub>.1.2.1.b}
- d) "**INTREAL**" {15<sub>1</sub>.2.1.C}
- e) '**reference to INTREAL**'
- f) '**letter a symbol**'.

}

Note that the protonotion '**twas brillig and the slithy toves**' is neither a symbol nor a notion, in that it does not end with '**symbol**' and no production rule can be derived for it. Likewise, "**LEWIS**" and "**CAROLL**" are not metanotions in that no metaproduction rules are given for them.

- g) In order to distinguish the various usages in the text of this Report of the terms defined above, the following conventions are adopted:

- (i) No distinguishing marks {quotes, apostrophes or hyphens} are used in production rules, metaproduction rules or hyper-rules;
- (ii) Metanotions, and hypernotions which stand for themselves {i.e., which do not designate protonotions}, are enclosed in quotes;
- (iii) Paranotions are not enclosed in anything {but, as an aid to the reader, are provided with hyphens where, otherwise, they would have been provided with blanks};
- (iv) All other hypernotions {including protonotions} not covered above are enclosed in apostrophes {in order to indicate that they designate some protonotion, as defined in 15<sub>1</sub>.1.4.1.a};
- (v) Typographical display features, such as blank space, hyphen, and change to a new line or new page, are of no significance (but see also 24<sub>9</sub>.4.d).

{Examples:

- (i) **LEAP :: local ; heap ; primal.** is a metaproduction rule ;
- (ii) "**INTREAL**" is a metanotion and designates nothing but itself;
- (iii) **reference-to-INTREAL-identifier**, which is not enclosed in apostrophes but is provided with hyphens, is a paranotion designating a construct {15<sub>1</sub>.1.4.2.a};
- (iv) '**variable point**' is both a hypernotation and a protonotion; regarded as a hypernotation, it designates itself regarded as a protonotion;
- (v) '**reference to real**' means the same as '**referencetoreal**'.

}

### 15.3.1.2 Production rules and production trees

a) The {derived} "production rules" {b} of the language are those production rules which can be derived from the "hyper-rules" {15<sub>1</sub>.1.3.4}, together with those specified informally in 23<sub>8</sub>.1.4.1.d and 24<sub>9</sub>.2.1.d.

b) A "production rule" consists of the following items, in order:

- an optional asterisk;
- a nonempty protonotion  $N$ ;
- a colon;

## ALGOL 68 REVISED REPORT

- a nonempty sequence of "alternatives" separated by semicolons;
- a point.

It is said to be a production rule "for" {the notion {15<sub>1</sub>.1.3.1.c}} *N*. {The optional asterisk, if present, signifies that the notion is not used in other production rules, but is provided to facilitate discussion in the semantics. It also signifies that that notion may be used as an "abstraction" {15<sub>1</sub>.1.4.2.b} of one of its alternatives.}

c) An "alternative" is a nonempty sequence of "members" separated by commas.

d) A "member" is either

- (i) a notion {and may then be said to be productive, or nonterminal},
- (ii) a symbol {which is terminal},
- (iii) empty, or
- (iv) some other notion {for which no production rule can be derived}, which is then said to be a "blind alley". {For example, the member '**reference to real denotation**' (derived from the hyper-rule 23<sub>8</sub>.0.1.a ) is a blind alley.}

{Examples:

b) **exponent part : times ten to the power choice, power of ten.** {23<sub>8</sub>.1.2.1.g} •

**times ten to the power choice :**  
**times ten to the power symbol ;**  
**letter e symbol** {23<sub>8</sub>.1.2.1.h}

c) **times ten to the power choice, power of ten •**

**times ten to the power symbol •**  
**letter e symbol**

d) **times ten to the power choice •**

**power of ten •**  
**times ten to the power symbol •**  
**letter e symbol }**

e) A "construct in the strict language" is any "production tree" {f} that may be "produced" from a production rule of the language.

f) A "production tree" *T* for a notion *N*, which is termed the "original" of *T*, is "produced" as follows:

- let *P* be some {derived} production rule for *N*;
- a copy is taken of *N*;

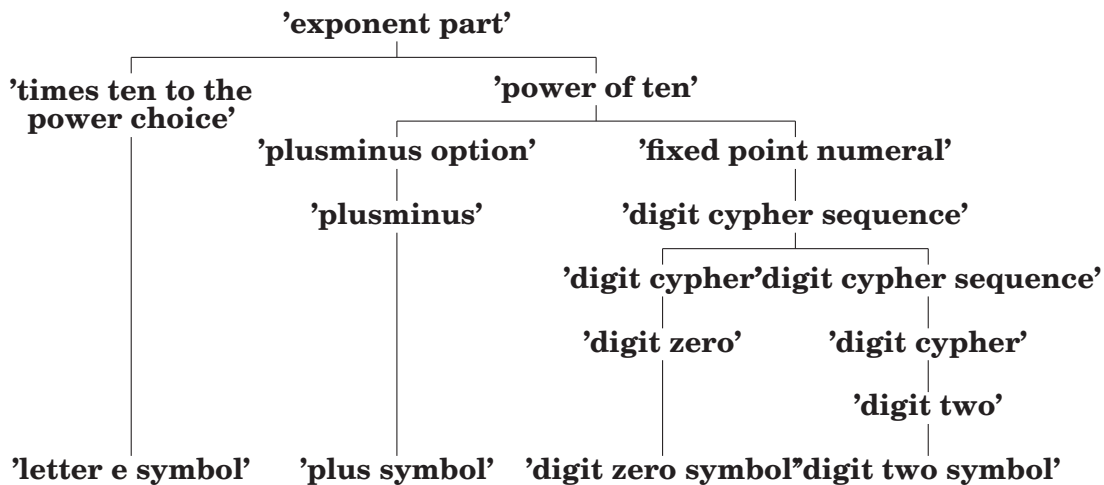
- a sequence of production trees, the "direct descendents" of  $T$ , one produced for each nonempty member of some alternative  $A$  of  $P$ , is attached to the copy; the order of the sequence is the order of those members within  $A$ ;
- the copy of the original, together with the attached direct descendents, comprise the production tree  $T$ .

A "production tree" for a symbol consists of a copy of that symbol {i.e., it consists of a **symbol**}.

The "terminal production" of a production tree  $T$  is a sequence consisting of the terminal productions of the direct descendents of  $T$ , taken in order.

The "terminal production" of a production tree consisting only of a **symbol** is that **symbol**.

{Example:



}

{The terminal production of this tree is the sequence of **symbols** at the bottom of the tree. In the reference language, its representation would be  $e+02$ .}

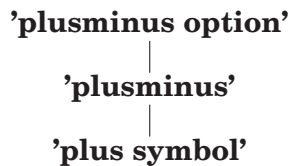
A "terminal production" of a notion is the terminal production of some production tree for that notion {thus there are many other terminal productions of '**exponent part**' besides the one shown}.

{The syntax of the strict language has been chosen in such a way that a given sequence of **symbols** which is a terminal production of some notion is so by virtue of a unique production tree, or by a set of production trees which differ only in such a way that the result of their elaboration is the same (e.g., production trees derived from rules 17<sub>3</sub>.2.1.e (balanc

ing), 15<sub>1</sub>.3.1.d,e (predicates) and 21<sub>6</sub>.7.1.a,b (choice of spelling of the mode of a **coercend** to be voided); see also 16<sub>2</sub>.2.2.a ).

Therefore, in practice, terminal productions (or representations thereof) are used, in this Report and elsewhere, in place of production trees. Nevertheless, it is really the production trees in terms of which the elaboration of programs is defined by the semantics of this Report, which is concerned with explaining the meaning of those constructs whose originals are the notion **'program'**.)

g) A production tree  $P$  is a "descendent" of a production tree  $Q$  if it is a direct descendent {f} either of  $Q$  or of a descendent of  $Q$ .  $Q$  is said to "contain" its descendents and those descendents are said to be "smaller" than  $Q$ . {For example, the production tree



occurs as a descendent in (and is contained within and is smaller than) the production tree for **'exponent part'** given above.)

h) A "visible" ("invisible") production tree is one whose terminal production is not (is) empty.

i) A descendent {g}  $U$  of a production tree  $T$  is "before" ("after") a descendent  $V$  of  $T$  if the terminal production {f} of  $U$  is before (after) that of  $V$  in the terminal production of  $T$ . The {partial} ordering of the descendents of  $T$  thus defined is termed the "textual order". {In the example production tree for **'exponent part'** {f}, the production tree whose original is **'plusminus'** is before that whose original is **'digit two'**.)

j) A descendent  $A$  of a production tree "follows" ("precedes") another descendent  $B$  in some textual order if  $A$  is after (before)  $B$  in that textual order, and there exists no visible {h} descendent  $C$  which comes between  $A$  and  $B$ . {Thus "immediately" following (preceding) is implied.)

k) A production tree  $A$  is "akin" to a production tree  $B$  if the terminal production {f} of  $A$  is identical to the terminal production of  $B$ .

### 15.3.1.3 Metaproduction rules and simple substitution

{The metaproduction rules of the language form a set of context-free grammars defining a "metalanguage".}

a) The "metaproduction rules" {b} of the language are those given in the sections of this

Report whose heading begins with "Syntax", "Metasyntax" or "Metaproduction rules", together with those obtained as follows:

- for each given metaproduction rule, whose metanotation is  $M$  say, additional rules are created each of which consists of a copy of  $M$  followed by one of the large syntactic marks "0", "1", "2", "3", "4", "5", "6", "7", "8" or "9", followed by two colons, another copy of that  $M$  and a point.

{Thus, the metaproduction rule "**MODE1 :: MODE.**" is to be added.}

b) A "metaproduction rule" consists of the following items, in order:

- an optional asterisk;
- a nonempty sequence  $M$  of large syntactic marks;
- two colons;
- a nonempty sequence of hypernotations {15<sub>1</sub>.1.3.1.e} separated by semicolons;
- a point.

It is said to be a metaproduction rule "for" {the metanotation {15<sub>1</sub>.1.3.1.d}}  $M$ .

{The asterisk, if present, signifies that the metanotation is not used in other metaproduction rules or in hyper-rules, but is provided to facilitate discussion in the semantics.}

{Examples:

**INTREAL :: SIZETY integral ; SIZETY real. (1.2.1.C) •**  
**SIZETY :: long LONGSETY ; short SHORTSETY ; EMPTY. (1.2.1.D) }**

c) A "terminal metaproduction" of a metanotation  $M$  is any protonotation which is a "simple substitute" {d} for one of the hypernotations {on the right hand side} of the metaproduction rule for  $M$ .

d) A protonotation  $P$  is a "simple substitute" for a hypernotation  $H$  if a copy of  $H$  can be transformed into a copy of  $P$  by replacing each metanotation  $M$  in the copy by some terminal metaproduction of  $M$ .

{Thus two possible terminal metaproductions {c} of "INTREAL" are '**integral**' and '**long long real**'. This is because the hypernotations '**SIZETY integral**' and '**SIZETY real**' (the hypernotations of the metaproduction rule for "INTREAL") may, upon simple substitution {d}, give rise to '**integral**' and '**long long real**', which, in turn, is because " (the empty protonotation) and '**long long**' are terminal metaproductions of "SIZETY".}

{The metanotations used in this Report have been so chosen that no concatenation of one or more of them gives the same sequence of large syntactic marks as any other such concatenation. Thus a source of possible ambiguity has been avoided.

Although the recursive nature of some of the metaproduction rules makes it possible to produce terminal metaproducts of arbitrary length, the length of the terminal metaproducts necessarily involved in the production of any given **program** is finite.)

#### 15.3.1.4 Hyper-rules and consistent substitution

a) The hyper-rules {b} of the language are those given in the sections of this Report whose heading begins with "Syntax".

b) A "hyper-rule" consists of the following items, in order:

- an optional asterisk ;
- a nonempty hypernotation  $H$  ;
- a colon ;
- a nonempty sequence of "hyperlalternatives" separated by semicolons ;
- a point.

It is said to be a hyper-rule "for" {the hypernotation {15<sub>1</sub>.1.3.1.e}}  $H$ .

c) A "hyperlalternative" is a nonempty sequence of hypernotations separated by commas.

{Examples:

b) **NOTION sequence : NOTION ; NOTION, NOTION sequence.** {15<sub>1</sub>.3.3.b}

c) **NOTION, NOTION sequence }**

d) A production rule  $PR$  {15<sub>1</sub>.1.3.2.b} is derived from a hyper-rule  $HR$  if a copy of  $HR$  can be transformed into a copy of  $PR$  by replacing the set of all the hypernotations in the copy by a "consistent substitute" {e} for that set.

e) A set of {one or more} hypernotations  $PP$  is a "consistent substitute" for a corresponding set of hypernotations  $HH$  if a copy of  $HH$  can be transformed into a copy of  $PP$  by means of the following step:

Step : If the copy contains one or more hypernotations then, for some terminal metaproduction  $T$  of one such hypernotation  $M$ , each occurrence of  $M$  in the copy is replaced by a copy of  $T$  and the Step is taken again.

{See 15<sub>1</sub>.1.4.1.a for another application of consistent substitution.}

{Applying this derivation process to the hyper-rule given above {c} may give rise to



- **digit cypher sequence :**  
**digit cypher ; digit cypher, digit cypher sequence.**

which is therefore a production rule of the language. Note that

- **digit cypher sequence :**  
**digit cypher ; digit cypher, letter b sequence.**

is not a production rule of the language, since the replacement of the metanotion "**NOTION**" by one of its terminal metaproducts must be consistent throughout.)

{Since some metanotions have an infinite number of terminal metaproducts, the number of production rules which may be derived is infinite. The language is, however, so designed that, for the production of any **program** of finite length, only a finite number of those production rules is needed.}

{f) The rules under Syntax are provided with "cross-references" to be interpreted as follows.

Each hypernotation  $H$  of a hyperalternative of a hyper-rule  $A$  is followed by a reference to those hyper-rules  $B$  whose derived production rules are for notions which could be substituted for that  $H$ . Likewise, the hypernotations of each hyper-rule  $B$  are followed by a reference back to  $A$ . Alternatively, if  $H$  is to be replaced by a symbol, then it is followed by a reference to its representation in section 24<sub>0</sub>.4.1 . Moreover, in some cases, it is more convenient to give a **cross-reference** to one metaproduct rule rather than to many hyper-rules, and in these cases the missing cross-references will be found in the metaproduct rule.

Such a reference is, in principle, the section number followed by a letter indicating the line where the rule or representation appears, with the following conventions:

- (i) the references whose section number is that of the section in which they appear are given first and their section number is omitted; e.g., "23<sub>8</sub>.2.1.a" appears as "a" in section "23<sub>8</sub>.2.1";
- (ii) a final 1 is omitted; e.g., "23<sub>8</sub>.2.1.a" appears as "23<sub>8</sub>.2.a" elsewhere and "25<sub>10</sub>.3.4.1.1.i" appears as "25<sub>10</sub>.3.4.1.i";<sup>1</sup>
- (iii) a section number which is the same as that of the preceding reference is omitted; e.g., "23<sub>8</sub>.2.a, 23<sub>8</sub>.2.b, 23<sub>8</sub>.2.c" appears as "23<sub>8</sub>.2.a,b,c";

---

<sup>1</sup>in the original Revised Report this read: *all points and a final 1 are omitted, and 10 appears as A; e.g., "8.2.1.a" appears as "82a" elsewhere and "10.3.4.1.1.i" appears as "A341i"*; L<sup>A</sup>T<sub>E</sub>X does not support 10 to appear as  $A$  in references, thus the substitution of 10 for  $A$  is not maintained in this translation. Also, points are not omitted in this translation as to improve legibility of cross-references

- (iv) the presence of a blind alley derived from that hypernotation is indicated by "-"; e.g., in 23<sub>8</sub>.0.1.a after "**MOID denotation**", since "**MOID**" may be replaced by, for example, '**reference to real**', but '**reference to real denotation**' is not a notion.}

## 15.3.2 The semantics

The "meaning" of **programs** {16<sub>2</sub>.2.1.a} in the strict language is defined in the semantics by means of sentences {in somewhat formalized natural language} which specify the "actions" to be carried out during the "elaboration" {16<sub>2</sub>.1.4.1} of those **programs**. The meaning of a **program** in a representation language is the same as the meaning of the **program** in the strict language it represents {24<sub>9</sub>.3}.

{The semantics makes extensive use of hypernotations and paranotions in order to "designate", respectively, protonotions and constructs. The word "designate" should be understood in the sense that the word "flamingo" may "designate" any animal of the family *Phoenicopteridae*.}

### 15.3.2.1 Hypernotations, designation and envelopment

{Hypernotations, when enclosed between apostrophes, are used to "designate" protonotions belonging to certain classes; e.g., '**LEAP**' designates any of the protonotions '**local**', '**primal**' and '**heap**'.}

a) Hypernotations standing in the text of this Report, except those in hyper-rules {15<sub>1</sub>.1.3.4.b} or metaproduction rules {15<sub>1</sub>.1.3.3.b} "designate" any protonotions which may be consistently substituted {15<sub>1</sub>.1.3.4.e} for them, the consistent substitution being applied over all the hypernotations contained in each complete sub-section of the text (a sub-section being one of the lettered sub-divisions, if any, or else the whole, of a numbered section).

{Thus '**QUALITY TAX**' is a hypernotation designating protonotions such as '**integral letter i**', '**real letter x**', etc. If, in some particular discussion, it in fact designates '**integral letter i**', then all occurrences of "**QUALITY**" in that subsection must, over the span of that discussion, designate '**integral**' and all occurrences of "**TAX**" must designate '**letter i**'. It may then be deduced from subsection 18<sub>4</sub>.8.2.a that in order, for example, to "ascribe to an **integral-defining-indicator-with-letter-i**", it is '**integral letter i**' that must be "made to access *V* inside the locale".}

Occasionally, where the context clearly so demands, consistent substitution may be applied over less than a section. {For example, in the introduction to section 16<sub>2</sub>.1.1.2, there are several occurrences of "**MOID**", of which two are stated to designate specific (and different) protonotions spelled out in full, and of which others occur in the plural form "**MOID**'s", which is clearly intended to designate a set of different members of the class of terminal metaproducts of "**MOID**".}

b) If a protonotion (a hypernotation)  $P$  consists of the concatenation of the protonotions (hypernotations)  $A$ ,  $B$  and  $C$ , where  $A$  and  $C$  are possibly empty, then  $P$  "contains"  $B$  at the position {in  $P$ } determined by the length of  $A$ . {Thus, '**abcdefcdgh**' contains '**cd**' at its third and seventh positions.}

c) A protonotion  $P1$  "envelops" a protonotion  $P2$  as specifically designated by a hypernotation  $H2$  if  $P2$ , or some equivalent {16<sub>2</sub>.1.1.2.a} of it, is contained {b} at some position within  $P1$  but not, at that position, within any different {intermediate} protonotion  $P3$  also contained in  $P1$  such that  $H2$  could also designate  $P3$ .

{Thus the '**MODE**' enveloped by '**reference to real closed clause**' is '**reference to real**' rather than '**real**'; moreover, the mode {16<sub>2</sub>.1.1.2.b} specified by STRUCT (REAL a, STRUCT (BOOL b, CHAR c) d) envelops '**FIELD**' just twice.}

### 15.3.2.2 Paranotions

{In order to facilitate discussion, in this Report, of constructs with specified originals, the concept of a "paranotion" is introduced. A paranotion is a noun that designates constructs {15<sub>1</sub>.1.3.2.e}; its meaning is not necessarily that found in a dictionary but can be construed from the rules which follow.}

a) A "paranotion"  $P$  is a hypernotation {not between apostrophes} which is used, in the text of this Report, to "designate" any construct whose original  $O$  satisfies the following:

- $P$ , regarded as a hypernotation {i.e., as if it had been enclosed in apostrophes}, designates {15<sub>1</sub>.1.4.1.a} an "abstraction" {b} of  $O$ .

{For example, the paranotion "**fixed-point-numeral**" could designate the construct represented by 02, since, had it been in apostrophes, it would have designated an abstraction of the notion '**fixed point numeral**', which is the original of that construct. However, that same representation could also be described as a **digit-cypher-sequence**, and as such it would be a direct descendent of that **fixed-point-numeral**.}

{As an aid to the reader in distinguishing them from other hypernotations, paranotions are not enclosed between apostrophes and are provided with hyphens where, otherwise, they would have been provided with blanks.}

The meaning of a paranotion to which the small syntactic mark "s" has been appended is the same as if the letter "s" {which is in the same type font as the marks in this sentence} had been appended instead. {Thus the **fixed-point-numeral** 02 may be said to contain two **digit-cyphers**, rather than two **digit-cyphers**.} Moreover, the "s" may be inserted elsewhere than at the end if no ambiguity arises {e.g., "**sources-for-MODINE**" means the same as "**source-for-MODINES**"}.}

An initial small syntactic mark of a paranotion is often replaced by the corresponding

large syntactic mark (in order to improve readability, as at the start of a sentence) without change of meaning {; e.g., "**Identifier**" means the same as "**identifier**"}

b) A protonotion  $P2$  is an "abstraction" of a protonotion  $P1$  if

- (i)  $P2$  is an abstraction of a notion whose production rule begins with an asterisk and of which  $P1$  is an alternative

{e.g., '**trimscript**' {19<sub>5</sub>.3.2.1.h} is an abstraction of any of the notions designated by '**NEST trimmer**', '**NEST subscript**' and '**NEST revised lower bound option**'}, or

- (ii)  $P1$  envelops a protonotion  $P3$  which is designated by one of the "elidable hypernotions" listed in section c below, and  $P2$  is an abstraction of the protonotion consisting of  $P1$  without that enveloped  $P3$

{e.g., '**choice using boolean start**' is an abstraction of the notions '**choice using boolean brief start**' and '**choice using boolean bold start**' (by elision of a '**STYLE**' from 24<sub>9</sub>.1.1.a)}, or

- (iii)  $P2$  is equivalent to {16<sub>2</sub>.1.1.2.a} $P1$

{e.g., '**bold begin symbol**' is an abstraction of '**bold begin symbol**'} .

{For an example invoking all three rules, it may be observed that '**union of real integral mode defining indicator**' is an abstraction of some '**union of integral real mode NEST defining identifier with letter a**' {18<sub>4</sub>.8.1.a}. Note, however, that '**choice using union or integral real mode brief start**' is not an abstraction of the notion '**choice using union of integral real boolean mode brief start**', because the '**boolean**' that has apparently been elided is not an enveloped '**MOID**' of that notion.}

c) The "elidable hypernotions" mentioned in section b above are the following:

"**STYLE**" • "**TALLY**" • "**LEAP**" • "**DEFIED**" • "**VICTAL**" • "**SORT**" • "**MOID**" • "**NEST**" • "**REFETY routine**" • "**label**" • "**with TAX**" • "**with DECSETY LABSETY**" • "**of DECSETY LABSETY**" • "**defining LAYER**".

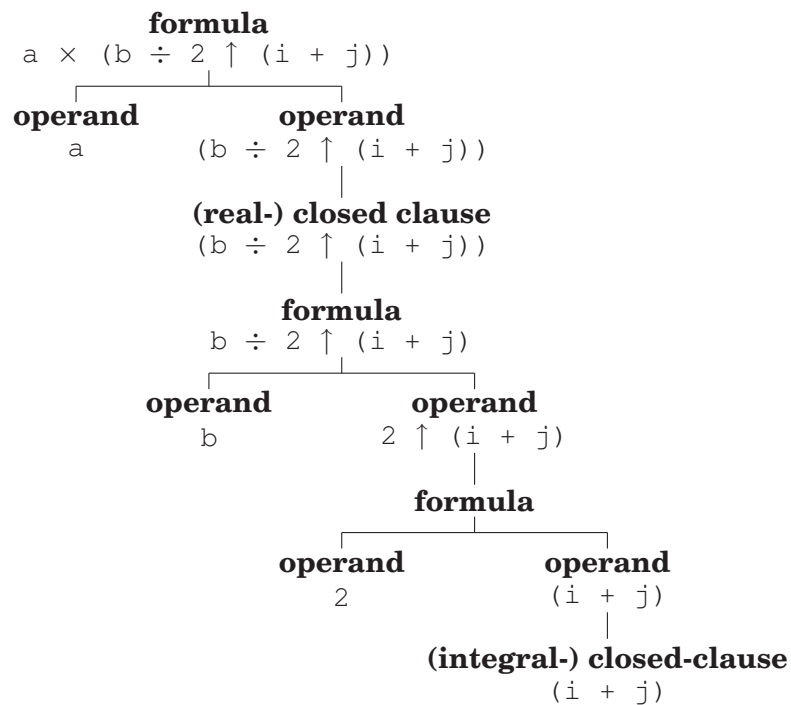
{Which one of several possible notions or symbols is the original of a construct designated by a given paranotion will be apparent from the context in which that paranotion appears. For example, when speaking of the **formal-declarer** of an **identity-declaration**, if the **identity-declaration** is one whose terminal production {15<sub>1</sub>.1.3.2.f} happens to be  $\text{REF REAL } x = \text{LOC REAL}$ , then the original of that **formal-declarer** is some notion designated by '**formal reference to real NEST declarer**'}

{Since a paranotion designates a construct, all technical terms which are defined for constructs can be used with paranotions without formality.}

d) If two paranotions  $P$  and  $Q$  designate, respectively, two constructs  $S$  and  $T$ , and if  $S$  is a descendent of  $T$ , then  $P$  is termed a "constituent" of  $Q$  unless there exists some {intermediate construct}  $U$  such that

- (i)  $S$  is a descendent of  $U$ ,
- (ii)  $U$  is a descendent of  $T$ , and
- (iii) either  $P$  or  $Q$  could {equally well} designate  $U$ .

{Hence  $a$  ( $S1$ ) is a constituent **operand** of the **formula**  $a \times (b \div 2 \uparrow (i + j))$  ( $T$ ), but  $b$  ( $S2$ ) is not, since it is a descendent of an intermediate **formula**  $b \div 2 \uparrow (i + j)$  ( $U$ ), which is itself descended from  $T$ . Likewise,  $b \div 2 \uparrow (i + j)$  is a constituent **closed-clause** of the **formula**  $T$ , but the **closed-clause**  $(i + j)$  is not, because it is descended from an intermediate **closed-clause**. However,  $(i + j)$  is a constituent **integral-closed-clause** of  $T$ , because the intermediate **closed-clause** is, in fact, a **real-closed-clause**.



}

### 15.3.2.3 Undefined

a) If something is left "undefined" or is said to be "undefined", then this means that it is not defined by this Report alone and that, for its definition, information from outside this Report has to be taken into account.

{A distinction must be drawn between the yielding of an undefined value (whereupon elaboration continues with possibly unpredictable results) and the complete undefinedness of the further elaboration. The action to be taken in this latter case is at the discretion of the implementer, and may be some form of continuation (but not necessarily the same as any other implementer's continuation), or some form of interruption {16<sub>2</sub>.1.4.3.h} brought about by some run-time check.}

b) If some condition is "required" to be satisfied during some elaboration then, if it is not so satisfied, the further elaboration is undefined.

c) A "meaningful" **program** is a **program** {16<sub>2</sub>.2.1.a} whose elaboration is defined by this Report.

{Whether all **programs**, only **particular-programs**, only meaningful **programs**, or even only meaningful **particular-programs** are "Algol 68" **programs** is a matter for individual taste.}

## 15.4 Translations and variants

a) The definitive version D of this Report is written in English. A translation T of this Report into some other language is an acceptable translation if:

- T defines the same set of production trees as D, except that
  - (i) the originals contained in each production tree of T may be different protonotions obtained by some uniform translation of the corresponding originals contained in the corresponding production tree of D, and
  - (ii) descendents of those production trees need not be the same if their originals are predicates {15<sub>1</sub>.3.2};
- T defines the meaning {16<sub>2</sub>.1.4.1.a} of each of its **programs** to be the same as that of the corresponding **program** defined by D;
- T defines the same reference language 24<sub>9</sub>.4 and the same standard environment 25<sub>10</sub> as D;
- T preserves, under another mode of expression, the meaning of each section of D except that:
  - (i) different syntactic marks {15<sub>1</sub>.1.3.1.a} may be used {with a correspondingly different metaproduction rule for "**ALPHA**" {15<sub>1</sub>.3.1.B}} ;

(ii) the method of derivation of the production rules {15<sub>1</sub>.1.3.4} and their interpretation {15<sub>1</sub>.1.3.2} may be changed to suit the peculiarities of the particular natural language {; e.g., in a highly inflected natural language, it may be necessary to introduce some inflections into the hypernotations, for which changes such as the following might be required:

- 1) additional means for the creation of extra metaproduction rules {15<sub>1</sub>.1.3.3.a};
- 2) a more elaborate definition of "consistent substitute" {15<sub>1</sub>.1.3.4.e};
- 3) a more elaborate definition of "equivalence" between protonotions {16<sub>2</sub>.1.1.2.a};
- 1) different inflections for paranotions {15<sub>1</sub>.1.4.2.a};

}

(iii) some pragmatic remarks {15<sub>1</sub>.1.2} may be changed.

b) A version of this Report may, additionally, define a "variant of Algol 68" by providing:

- (i) additional or alternative representations in the reference language {24<sub>9</sub>.4},
- (ii) additional or alternative rules for the notion '**character glyph**' {23<sub>8</sub>.1.4.1.c} and for the metanotions "**ABC**" {24<sub>9</sub>.4.2.1.L} and "**STOP**" {25<sub>10</sub>.1.1.B},
- (iii) additional or alternative declarations in the standard environment which must, however, have the same meaning as the ones provided in D;

provided always that such additional or alternative items are delineated in the text in such a way that the original language, as defined in D, is still defined therein.

## 15.5 General metaproduction rules

{The reader may find it helpful to note that a metanotation ending in "**ETY**" always has "**EMPTY**" as one of the hypernotations on its right-hand side.}

### 15.5.1 Metaproduction rules of modes

- A) **MODE :: PLAIN ; STOWED ; REF to MODE ; PROCEDURE ; UNITED ;  
MU definition of MODE ; MU application.**
- B) **PLAIN :: INTREAL ; boolean ; character.**
- C) **INTREAL :: SIZETY integral ; SIZETY real.**

- D) **SIZETY** :: long LONGSETY ; short SHORTSETY ; EMPTY.
- E) **LONGSETY** :: long LONGSETY ; EMPTY.
- F) **SHORTSETY** :: short SHORTSETY ; EMPTY.
- G) **EMPTY** :: .
- H) **STOWED** :: structured with **FIELDS** mode ; **FLEXETY** ROWS of **MODE**.
  - I) **FIELDS** :: **FIELD** ; **FIELDS** **FIELD**.
  - J) **FIELD** :: **MODE** field TAG{24<sub>9</sub>.4.2.A} .
  - K) **FLEXETY** :: flexible ; **EMPTY**.
  - L) **ROWS** :: row ; **ROWS** row.
- M) **REF** :: reference ; transient reference.
- N) **PROCEDURE** :: procedure **PARAMETY** yielding **MOID**.
- O) **PARAMETY** :: with **PARAMETERS** ; **EMPTY**.
- P) **PARAMETERS** :: **PARAMETER** ; **PARAMETERS** **PARAMETER**.
- Q) **PARAMETER** :: **MODE** parameter.
- R) **MOID** :: **MODE** ; void.
- S) **UNITED** :: union of **MOODS** mode.
- T) **MOODS** :: **MOOD** ; **MOODS** **MOOD**.
- U) **MOOD** :: **PLAIN** ; **STOWED** ; reference to **MODE** ; **PROCEDURE** ; void.
- V) **MU** :: mu**TALLY**.
- W) **TALLY** :: i ; **TALLY** i.

{The metaproduction rule for "TAG" is given in section 24<sub>9</sub>.4.2.1. It suffices for the present that it produces an arbitrarily large number of terminal metaproductions.}

### **15.5.2 Metaproduction rules associated with phrases and coercion**

- A) **ENCLOSED** :: closed ; collateral ; parallel ; **CHOICE**{17<sub>3</sub>.4.A} ; loop.
- B) **SOME** :: **SORT** **MOID** **NEST**.
- C) **SORT** :: strong ; firm ; meek ; weak ; soft.



### 15.5.3 Metaproduction rules associated with nests

- A) **NEST :: LAYER ; NEST LAYER.**
- B) **LAYER :: new DECSETY LABSETY.**
- C) **DECSETY :: DECS ; EMPTY.**
- D) **DECS :: DEC ; DECS DEC.**
- E) **DEC ::**  
  - MODE TAG**{24<sub>9</sub>.4.2.A} ;
  - priority PRIO TAD**{24<sub>9</sub>.4.2.F} ;
  - MOID TALLY TAB**{24<sub>9</sub>.4.2.D} ;
  - DUO TAD**{24<sub>9</sub>.4.2.F} ;
  - MONO TAM**{24<sub>9</sub>.4.2.K} .
- F) **PRIO :: i ; ii ; iii ; iii i ; iii ii ; iii iii ; iii iii i ; iii iii ii ; iii iii iii.**
- G) **MONO :: procedure with PARAMETER yielding MOID.**
- H) **DUO :: procedure with PARAMETER1 PARAMETER2 yielding MOID.**
- I) **LABSETY :: LABS ; EMPTY.**
- J) **LABS :: LAB ; LABS LAB.**
- K) **LAB :: label TAG**{24<sub>9</sub>.4.2.A} .

{The metaproduction rules for "TAB", "TAD" and "TAM" are given in section 24<sub>9</sub>.4.2.1. It suffices for the present that each of them produces an arbitrarily large number of terminal metaproducts, none of which is a terminal metaproduction of "TAG".}

{"Well, 'slithy' means 'lithe and slimy'. ...  
 You see it's like a portmanteau - there are  
 two meanings packed up into one word."  
 Through the Looking Glass, Lewis Carroll. }

## 15.6 General hyper-rules

{Predicates are used in the syntax to enforce certain restrictions on the production trees, such as that each **applied-indicator** should identify a uniquely determined **defining-indicator**. A more modest use is to reduce the number of hyper-rules by grouping several similar cases as alternatives in one rule. In these cases predicates are used to test which alternative applies.}

### 15.6.1 Syntax of general predicates

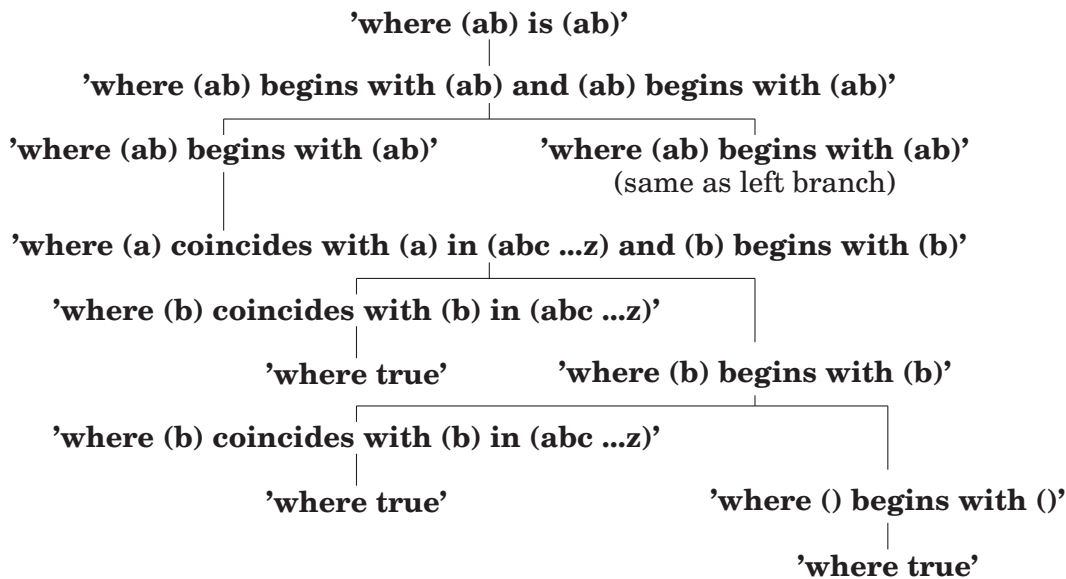
- A) **NOTION** :: ALPHA ; NOTION ALPHA.
- B) **ALPHA** :: a ; b ; c ; d ; e ; f ; g ; h ; i ; j ; k ; l ; m ; n ; o ; p ; q ; r ; s ; t ; u ; v ; w ; x ; y ; z.
- C) **NOTETY** :: NOTION ; EMPTY.
- D) **THING** :: NOTION ; (NOTETY1) NOTETY2 ; THING (NOTETY1) NOTETY2.
- E) **WHETHER** :: where ; unless.
  - a) **where true** : EMPTY.
  - b) **unless false** : EMPTY.
  - c) **where THING1 and THING2** : where THING1, where THING2.
  - d) **where THING1 or THING2** : where THING1 ; where THING2.
  - e) **unless THING1 and THING2** : unless THING1 ; unless THING2.
  - f) **unless THING1 or THING2** : unless THING1, unless THING2.
  - g) **WHETHER (NOTETY1) is (NOTETY2)** :  
**WHETHER (NOTETY1) begins with (NOTETY2){h,i,j}**  
**and (NOTETY2) begins with (NOTETY1){h,i,j}.**
  - h) **WHETHER (EMPTY) begins with (NOTION){g,j}** : **WHETHER false{b,-}.**
  - i) **WHETHER (NOTETY) begins with (EMPTY){g,j}** : **WHETHER true{a,-}.**
  - j) **WHETHER (ALPHA1 NOTETY1) begins with (ALPHA2 NOTETY2){g,j,m}** :  
**WHETHER (ALPHA1) coincides with (ALPHA2) in**  
**(abcdefghijklmnopqrstuvwxy){k,l,-}**  
**and (NOTETY1) begins with (NOTETY2){h,i,j}.**
  - k) **where (ALPHA) coincides with (ALPHA) in (NOTION){j}** : **where true {a}.**
  - l) **unless (ALPHA1) coincides with (ALPHA2) in (NOTION){j}** :  
**where (NOTION) contains (ALPHA1 NOTETY ALPHA2){m}**  
**or (NOTION) contains (ALPHA2 NOTETY ALPHA1){m}.**
  - m) **WHETHER (ALPHA NOTETY) contains (NOTION){l,m}** :  
**WHETHER (ALPHA NOTETY) begins with (NOTION){j}**  
**or (NOTETY) contains (NOTION){m,n}.**
  - n) **WHETHER (EMPTY) contains (NOTION){m}** : **WHETHER false{b,-}.**

{The small syntactic marks "(" and ")" are used to ensure, in a simple way, the unambiguous application of these predicates.}

### 15.6.2 The holding of predicates

A "predicate" is a proposition which begins with **'where'** or **'unless'** {unified into **'WHETHER'**}. For a predicate *P*, either one or more production trees may be produced {15<sub>1</sub>.1.3.2.f} {all of which are then invisible}, in which case *P* "holds", or no production tree may be produced {since each attempt to produce one runs into blind alleys}, and then *P* "does not hold".

{For example, the predicate **'where (ab) is (ab)'** holds. Its production tree may be depicted thus:



}

If a predicate holds, then its production tree always terminates via **'where true'** or **'unless false'**. If it does not hold, then, in general, the blind alleys are **'where false'** and **'unless true'**. Although almost all the hyper-rules concerned are for hypernotions beginning with "WHETHER" and so provide, each time, production rules for pairs of predicates such as **'where THING1'** and **'unless THING1'**, this does not mean that in each such case one of the pair must hold. For example, **'where digit four counts iii'** {18<sub>4</sub>.3.1.c} does not hold, but no care has been taken to make **'unless digit four counts iii'** hold either, since there is no application for it in this Report.

In the semantics, no meaning is ascribed to constructs whose originals are predicates. They serve purely syntactical purposes.}

### 15.6.3 Syntax of general constructions

- A) **STYLE :: brief ; bold ; style TALLY.**
- a) **NOTION option : NOTION ; EMPTY.**
- b) **NOTION sequence{b} : NOTION ; NOTION, NOTION sequence{b}.**
- c) **NOTION list{c} : NOTION ; NOTION, and also{24<sub>9</sub>.4.f} token, NOTION list{c}.**
- d) **NOTETY STYLE pack :**  
**STYLE begin{24<sub>9</sub>.4.f,-} token, NOTETY, STYLE end{24<sub>9</sub>.4.f,-} token.**
- e) **NOTION STYLE bracket :**  
**STYLE sub{24<sub>9</sub>.4.f,-} token, NOTION, STYLE bus{24<sub>9</sub>.4.f,-} token.**
- f) **THING1 or alternatively THING2 : THING1 ; THING2.**

{It follows from this syntax that production rules such as

**digit cypher sequence : digit cypher ;  
digit cypher, digit cypher sequence.**

(which was used in the production of the example in 15<sub>1</sub>.1.3.2.f, but for which no more explicit hyper-rule is given) are immediately available. Thus the number of hyper-rules actually written in this Report has been reduced and those that remain have, hopefully, been made more readable, since these general constructions are so worded as to suggest what their productions should be.

For this reason, cross-references (15<sub>1</sub>.1.3.4.f) to these rules have been replaced by more helpful references; e.g. in 23<sub>8</sub>.1.1.1.b, instead of "**digit cypher sequence** {15<sub>1</sub>.3.3.b}", the more helpful "**digit cypher** {c}**sequence**" is given. Likewise, references within the general constructions themselves have been restricted to a bare minimum.}

# Chapter 16

## The computer and the program

The meaning of a **program** in the strict language is explained in terms of a hypothetical computer which performs the set of actions {16<sub>2</sub>.1.4} which constitute the elaboration {16<sub>2</sub>.1.4.1} of that **program**. The computer deals with a set of "objects" {16<sub>2</sub>.1.1}.

### 16.1 Terminology

{ "When *I* use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean - neither more nor less."  
Through the Looking Glass, Lewis Carroll. }

#### 16.1.1 Objects

An "object" is a construct {15<sub>1</sub>.1.3.2.e}, a "value" {16<sub>2</sub>.1.1.1.a}, a "locale" {16<sub>2</sub>.1.1.1.b}, an "environ" {16<sub>2</sub>.1.1.1.c} or a "scene" {16<sub>2</sub>.1.1.1.d}.

{Constructs may be classified as "external objects", since they correspond to the text of the **program**, which, in a more realistic computer, would be compiled into some internal form in which it could operate upon the "internal objects", namely the values, the locales, the environs and the scenes. However, the hypothetical computer has no need of a compilation phase, it being presumed able to examine the **program** and all of its descendent constructs at the same time as it is manipulating the internal objects.}

##### 16.1.1.1 Values, locales, environs and scenes

a) A "value" is a "plain value" {16<sub>2</sub>.1.3.1}, a "name" {16<sub>2</sub>.1.3.2}, a "stowed value" (i.e., a "structured value" {16<sub>2</sub>.1.3.3} or a "multiple value" {16<sub>2</sub>.1.3.4}) or a "routine" {16<sub>2</sub>.1.3.5}.

{For example, a real number is a plain value. A special font is used for values appearing in the text of this Report, thus: 3.14, true. This is not to be confused with the italic and bold fonts used for constructs. This same special font is also used for letters designating such things as constructs and protonotions.}

b) A "locale" {is an internal object which} corresponds to some '**DECSETY LABSETY**' {15<sub>1</sub>.2.3.C,I}. A "vacant locale" is one for which that '**DECSETY LABSETY**' is '**EMPTY**'.

{Each '**QUALITY TAX**' {18<sub>4</sub>.8.1.F, G} enveloped by that '**DECSETY LABSETY**' corresponds to a **QUALITY-defining-indicator-with-TAX** {i.e., to an **identifier, operator or mode-indication**} declared in the construct whose elaboration caused that locale to be created. Such a '**QUALITY TAX**' may be made to "access" a value or a scene "inside" that locale {16<sub>2</sub>.1.2.c}

A locale may be thought of as a number of storage cells, into which such accessed objects are placed.}

{The terminal metaproductions of the metanotions "**DEC**", "**LAB**" and "**FIELD**" (or of the more frequently used "**PROP**", which includes them all) are all of the form '**QUALITY TAX**'. These "properties" are used in the syntax and semantics concerned with nests and locales in order to associate, in a particular situation, some quality with that '**TAX**'.}

c) An "environ" is either empty, or is composed of an environ and a locale.

{Hence, each environ is derived from a series of other enviros, stemming ultimately from the empty "primal environ" in which the **program** is elaborated {16<sub>2</sub>.2.2.a}.}

d) A "scene" *S* is an object which is composed of a construct *C* {15<sub>1</sub>.1.3.2.e} and an environ *E*. *C* is said to be the construct, and *E* the environ, "of" *S*.

{Scenes may be accessed inside locales {16<sub>2</sub>.1.2.c} by '**LAB**'s or '**DEC**'s arising from **label-identifiers** or from **mode-indications**, and they may also be values {16<sub>2</sub>.1.3.5}.}

### 16.1.1.2 Modes

{Each value has an attribute, termed its "mode", which defines how that value relates to other values and which actions may be applied to it. This attribute is described, or "spelled", by means of some '**MOID**' {15<sub>1</sub>.2.1.R} (thus there is a mode spelled '**real**', and there is a mode spelled '**structured with real field letter r letter e real field letter i letter m mode**'). Since it is intended that the modes specified by the **mode-indications** A and B in

```
MODE A = STRUCT (REF A b),  
MODE B = STRUCT (REF STRUCT (REF B b) b)
```

should in fact be the same mode, it is necessary that both the **'MOID'**

**'mui definition of structured with reference to mui application field letter b mode'**

and the **'MOID'**

**'muu definition of structured with reference to structured with reference to muu application field letter b mode field letter b mode'**

(and indeed many others) should be alternative spellings of that same mode. Similarly, the mode specified by the **declarer** UNION (INT, REAL) may be spelled as either **'union of integral real mode'** or **'union of real integral mode'**. All those **'MOID'**'s which are spellings of one same mode are said to be "equivalent to" one another {a}.

Certain **'MOID'**'s, such as **'reference to muuu application'**, **'reference to muuuu definition of reference to muuuu application'**, **'union of real reference to real mode'**, and **'structured with integral field letter a real field letter a mode'**, are ill formed {22<sub>7</sub>.4, 18<sub>4</sub>.7.1.f, 18<sub>4</sub>.8.1.c} and do not spell any mode.

Although for most practical purposes a "mode" can be regarded as simply a **'MOID'**, its rigorous definition therefore involves the whole class of **'MOID'**'s, equivalent to each other, any of which could describe it.}

a) **'MOID1'** {15<sub>1</sub>.2.1.R} is "equivalent to" **'MOID2'** if the predicate **'where MOID1 equivalent MOID2'** {22<sub>7</sub>.3.1.a} holds {15<sub>1</sub>.3.2}.

{A well formed **'MOID'** is always equivalent to itself: **'union of integral real mode'** is equivalent to **'union of real integral mode'**.}

A protonotion  $P$  is "equivalent to" a protonotion  $Q$  if it is possible to transform a copy  $P_c$  of  $P$  into a copy  $Q_c$  of  $Q$  in the following step:

Step: If  $P_c$  is not identical to  $Q_c$ , then some **'MOID1'** contained in  $P_c$ , but not within any {larger} **'MOID2'** contained in  $P_c$ , is replaced by some equivalent **'MOID'**, and the Step is taken again.

{Thus **'union of integral real mode identifier'** is equivalent to **'union of real integral mode identifier'**.}

b) A "mode" is a class  $C$  of **'MOID'**'s such that each member of  $C$  is equivalent {a} to each other member of  $C$  and also to itself {in order to ensure well formedness}, but not to any **'MOID1'** which is not a member of  $C$ .

{However, it is possible (except when equivalence of modes is specifically under discussion) to discuss a mode as if it were simply a terminal metaproduction of "**MOID**", by virtue of the abbreviation to be given in 16<sub>2</sub>.1.5.f.}

c) Each value is of one specific mode.

{For example, the mode of the value 3.14 is '**real**'. However, there are no values whose mode begins with '**union of**', '**transient reference to**' or '**flexible ROWS of**' (see 16<sub>2</sub>.1.3.6).}

### 16.1.1.3 Scopes

{A value  $V$  may "refer to" {16<sub>2</sub>.1.2.e}, or be composed from {16<sub>2</sub>.1.1.1.d} another internal object  $O$  (e.g., a name may refer to a value; a routine, which is a scene, is composed, in part, from an environ). Now the lifetime of the storage cells containing {16<sub>2</sub>.1.3.2.a} or implied by {16<sub>2</sub>.1.1.1.b}  $O$  may be limited (in order that they may be recovered after a certain time), and therefore it must not be possible to preserve  $V$  beyond that lifetime, for otherwise an attempt to reach some no-longer-existent storage cell via  $V$  might still be made. This restriction is expressed by saying that, if  $V$  is to be "assigned" {19<sub>5</sub>.2.1.2.b} to some name  $W$ , then the "scope" of  $W$  must not be "older" than the scope of  $V$ . Thus, the scope of  $V$  is a measure of the age of those storage cells, and hence of their lifetime.}

a) Each value has one specific "scope" {which depends upon its mode or upon the manner of its creation; the scope of a value is defined to be the same as that of some environ}.

b) Each environ has one specific "scope". {The scope of each environ is "newer" {16<sub>2</sub>.1.2.f} than that of the environ from which it is composed {16<sub>2</sub>.1.1.1.c}.}

{The scope of an environ is not to be confused with the scopes of the values accessed inside its locale. Rather, the scope of an environ is used when defining the scope of scenes for which it is necessary {22<sub>7</sub>.2.2.c} or of the yields of generators for which it is "local" {19<sub>5</sub>.2.3.2.b}. The scope of an environ is defined relative {16<sub>2</sub>.1.2.f} to the scope of some other environ, so that hierarchies of scopes are created depending ultimately upon the scope of the primal environ {16<sub>2</sub>.2.2.a}.}

### 16.1.2 Relationships

a) Relationships either are "permanent", i.e., independent of the **program** and of its elaboration, or actions may cause them to "hold" or to cease to hold. Relationships may also be "transitive"; i.e., if "\*" is such a relationship and  $A*B$  and  $B*C$  hold, then  $A*C$  holds also.

b) "To be the yield of" is a relationship between a value and an action, viz., the elaboration of a scene. This relationship is made to hold upon the completion of that elaboration {16<sub>2</sub>.1.4.1.b}.



- c) "To access" is a relationship between a **'PROP'** {18<sub>4</sub>.8.1.E} and a value or a scene  $V$  which may hold "inside" some specified locale  $L$  {whose **'DECSETY LABSETY'** envelops **'PROP'**} . This relationship is made to hold when **'PROP'** is "made to access"  $V$  inside  $L$  {17<sub>3</sub>.5.2.Step 4, 18<sub>4</sub>.8.2.a} and it then holds also between any **'PROP1'** equivalent to {16<sub>2</sub>.1.1.2.a} **'PROP'** and  $V$  inside  $L$ .
- d) The permanent relationships between values are: "to be of the same mode as" {16<sub>2</sub>.1.1.2.c}, "to be smaller than", "to be widenable to", "to be lengthenable to" {16<sub>2</sub>.1.3.1.e} and "to be equivalent to" {16<sub>2</sub>.1.3.1.g}. If one of these relationships is defined at all for a given pair of values, then it either holds or does not hold permanently. These relationships are all transitive.
- e) "To refer to" is a relationship between a "name" {16<sub>2</sub>.1.3.2.a}  $N$  and some other value. This relationship is made to hold when  $N$  is "made to refer to" that value and ceases to hold when  $N$  is made to refer to some other value.
- f) There are three transitive relationships between scopes, viz., a scope  $A$  {16<sub>2</sub>.1.1.3} may be either "newer than", or "the same as" or "older than" a scope  $B$ . If  $A$  is newer than  $B$ , then  $B$  is older than  $A$  and vice-versa. If  $A$  is the same as  $B$ , then  $A$  is neither newer nor older than  $B$  {but the converse is not necessarily true, since the relationship is not defined at all for some pairs of scopes}.
- g) "To be a subname of" is a relationship between a name and a "stowed name" {16<sub>2</sub>.1.3.2.b}. This relationship is made to hold when that stowed name is "endowed with subnames" {16<sub>2</sub>.1.3.3.e, 16<sub>2</sub>.1.3.4.g} or when it is "generated" {16<sub>2</sub>.1.3.4.j, l}, and it continues to hold until that stowed name is endowed with a different set of subnames.

## 16.1.3 Values

### 16.1.3.1 Plain values

- a) A plain value is either an "arithmetic value", i.e., an "integer" or a "real number", or is a "truth value" {f}, a "character" {g} or a "void value" {h}.
- b) An arithmetic value has a "size", i.e., an integer characterizing the degree of discrimination with which it is kept in the computer.
- c) The mode of an integer or of a real number of size  $n$  is, respectively, some **'SIZETY integral'** or **'SIZETY real'** where, if  $n$  is positive (zero, negative), that **'SIZETY'** is  $n$  times **'long'** (is empty, is  $-n$  times **'short'**).
- d) The number of integers or of real numbers of a given size that can be distinguished increases (decreases) with that size until a certain size is reached, viz., the "number of extra lengths" (minus the "number of extra shorths") of integers or of real numbers, respectively, {25<sub>10</sub>.2.1.a, b, d, e} after which it is constant.

{Taking Three as the subject to reason about -  
A convenient number to state - }

e) For the purpose of explaining the meaning of the widening coercion and of the **operators** declared in the **standard-prelude**, the following properties of arithmetic values are assumed:

- for each pair of integers or of real numbers of the same size, the relationship "to be smaller than" is defined with its usual mathematical meaning {25<sub>10</sub>.2.3.3.a, 25<sub>10</sub>.2.3.4.a};
- for each pair of integers of the same size, a third distinguishable integer of that size may exist, the first integer "minus" the other {25<sub>10</sub>.2.3.3.g};

{We add Seven, and Ten, and then multiply out  
By One Thousand diminished by Eight. }

- for each pair of real numbers of the same size, three distinguishable real numbers of that size may exist, the first real number "minus" ("times", "divided by") the other one {25<sub>10</sub>.2.3.4.g, l, m};

in the foregoing, the terms "minus", "times" and "divided by" have their usual mathematical meaning but, in the case of real numbers, their results are obtained "in the sense of numerical analysis", i.e., by performing those operations on numbers which may deviate slightly from the given ones (; this deviation is left undefined in this Report);

{The result we proceed to divide, as you see,  
By Nine Hundred and Ninety and Two }

- each integer of a given size is "widenable to" a real number close to it and of that same size {21<sub>6</sub>.5};
- each integer (real number) of a given size can be "lengthened to" an integer (real number) close to it whose size is greater by one {25<sub>10</sub>.2.3.3.q, 25<sub>10</sub>.2.3.4.n}.

f) A "truth value" is either "true" or "false". Its mode is **'boolean'**.

{Then subtract Seventeen, and the answer must be  
Exactly and perfectly true.  
The Hunting of the Snark, Lewis Carroll. }

g) Each "character" is "equivalent" to a nonnegative integer of size zero, its "integral equivalent" {25<sub>10</sub>.2.1.n}; this relationship is defined only to the extent that different characters

have different integral equivalents, and that there exists a "largest integral equivalent" {25<sub>10</sub>.2.1.p}. The mode of a character is **'character'**.

h) The only "void value" is "empty". Its mode is **'void'**.

{The elaboration of a construct yields a void value when no more useful result is needed. Since the syntax does not provide for **void-variables**, **void-identity-declarations** or **void-parameters**, the programmer cannot make use of void values, except those arising from uniting {21<sub>6</sub>.4}.}

i) The scope of a plain value is the scope of the primal environ {16<sub>2</sub>.2.2.a}.

### 16.1.3.2 Names

{What's in a name? that which we call a rose  
By any other name would smell as sweet.  
Romeo and Juliet,                      William Shakespeare. }

a) A "name" is a value which can be "made to refer to" {d, 19<sub>5</sub>.2.3.2.a, 19<sub>5</sub>.2.1.2.b} some other value, or which can be "nil" {and then refers to no value}; moreover, for each mode beginning with **'reference to'**, there is exactly one nil name of that mode.

A name may be "newly created" {by the elaboration of a **generator** {19<sub>5</sub>.2.3.2} or a **rowed-to-FORM** {21<sub>6</sub>.6.2}, when a stowed name is endowed with subnames {16<sub>2</sub>.1.3.3.e, 16<sub>2</sub>.1.3.4.g} and, possibly, when a name is "generated" {16<sub>2</sub>.1.3.4.j, l} }. The name so created is different from all names already in existence.

{A name may be thought of as the address of the storage cell or cells, in the computer, used to contain the value referred to. The creation of a name implies the reservation of storage space to hold that value.}

b) The mode of a name *N* is some **'reference to MODE'** and any value which is referred to by *N* must be "acceptable to" {16<sub>2</sub>.1.3.6.d} that **'MODE'**. If **'MODE'** is some **'STOWED'**, then *N* is said to be a "stowed name".

c) The scope of a name is the scope of some specific environ {usually the "local environ" {19<sub>5</sub>.2.3.2.b} of some **generator**}. The scope of a name which is nil is the scope of the primal environ {16<sub>2</sub>.2.2.a}.

d) If *N* is a stowed name referring to a structured (multiple) value *V* {16<sub>2</sub>.1.3.3, 16<sub>2</sub>.1.3.4}, and if a subname {16<sub>2</sub>.1.2.g} of *N* selected {16<sub>2</sub>.1.3.3.e, 16<sub>2</sub>.1.3.4.g} by a **'TAG'** (an index) *I* is made to refer to a {new} value *X*, then *N* is made to refer to a structured (multiple) value which is the same as *V* except for its field (element) selected by *L* which is {now made to be} *X*.

{For the mode of a subname, see 16<sub>2</sub>.1.3.3.d and 16<sub>2</sub>.1.3.4.f.}

**16.1.3.3 Structured values**

a) A "structured value" is composed of a sequence of other values, its "fields", each of which is "selected" {b} by a specific **'TAG'** {24<sub>9</sub>.4.2.1.A}. {For the selection of a field by a **field-selector**, see 16<sub>2</sub>.1.5.g .}

{The ordering of the fields of a structured value is utilized in the semantics of **structure-displays** {17<sub>3</sub>.3.2.b} and **format-texts** {25<sub>10</sub>.3.4}, and in straightening {25<sub>10</sub>.3.2.3.c}.}

b) The mode of a structured value  $V$  is some **'structured with FIELDS mode'**. If the  $n$ -th **'FIELD'** enveloped by that **'FIELDS'** is some **'MODE field TAG'**, then the  $n$ -th field of  $V$  is "selected" by **'TAG'** and is acceptable to {16<sub>2</sub>.1.3.6.d} **'MODE'**.

c) The scope of a structured value is the newest of the scopes of its fields.

d) If the mode of a name  $N$  {referring to a structured value} is some **'reference to structured with FIELDS mode'**, and if the predicate **'where MODE field TAG resides in FIELDS'** holds {22<sub>7</sub>.2.1.b, c}, then the mode of the subname of  $N$  selected {e} by **'TAG'** is **'reference to MODE'**.

e) When a name  $N$  which refers to a structured value  $V$  is "endowed with subnames" {e, 16<sub>2</sub>.1.3.4.g, 18<sub>4</sub>.4.2.b, 19<sub>5</sub>.2.3.2.a}, then,

For each **'TAG'** selecting a field  $F$  in  $V$ ,

- a new subname  $M$  is created of the same scope as  $N$ ;
- $M$  is made to refer to  $F$ ;
- $M$  is said to be the name "selected" by **'TAG'** in  $N$ ;
- if  $M$  is a stowed name {16<sub>2</sub>.1.3.2.b}, then it is itself endowed with subnames {e, 16<sub>2</sub>.1.3.4.g}.

**16.1.3.4 Multiple values**

a) A "multiple value" {of  $n$  dimensions} is composed of a "descriptor" and a sequence of other values, its "elements", each of which may be "selected" by a specific  $n$ -tuple of integers, its "index".

b) The "descriptor" is of the form

$$((l_1, u_1), (l_2, u_2), \dots, (l_n, u_n))$$

where each  $(l_i, u_i), i = 1, \dots, n$ , is a "bound pair" of integers in which  $l_i$  is the  $i^{\text{th}}$  "lower bound" and  $u_i$  is the  $i^{\text{th}}$  "upper bound".

c) If for any  $i, i = 1, \dots, n, u_i < l_i$ , then the descriptor is said to be "flat" and there is one element, termed a "ghost element" {, and not selected by any index; see also {19<sub>5</sub>.2.1.2.b} }

; otherwise, the number of elements is  $(u_1 - l_1 + 1) \times (u_2 - l_2 + 1) \times \dots \times (u_n - l_n + 1)$  and each is selected by a specific index  $(r_1, \dots, r_n)$  where  $l_i \leq r_i \leq u_i, i = 1, \dots, n$ .

d) The mode of a multiple value  $V$  is some **'ROWS of MODE'**, where that **'ROWS'** is composed of as many times **'row'** as there are bound pairs in the descriptor of  $V$  and where each element of  $V$  is acceptable to {16<sub>2</sub>.1.3.6.d} that **'MODE'**.

{For example, given [ ] UNION (INT, REAL) ruir = (1, 2.0), the mode of the yield of ruir is **'row of union of integral real mode'**, the mode of its first element is **'integral'** and that of its second element is **'real'**.}

e) The scope of a multiple value is the newest of the scopes of its elements, if its descriptor is not flat, and, otherwise, is the scope of the primal environ {16<sub>2</sub>.2.2.a}.

f) A multiple value, of mode **'ROWS of MODE'**, may be referred to either by a "flexible" name of mode **'reference to flexible ROWS of MODE1'**, or by a "fixed" name of mode **'reference to ROWS of MODE1'** where {in either case} **'MODE1'** "deflexes" {16<sub>2</sub>.1.3.6.b} to **'MODE'**.

{The difference implies a possible difference in the method whereby the value is stored in the computer. The flexible case must allow a multiple value with different bounds to be assigned {19<sub>5</sub>.2.1.2.b} to that name, whereas the fixed case can rely on the fact that those bounds will remain fixed during the lifetime of that name. Note that the "flexibility" is a property of the name; the underlying multiple value is the same value in both cases.}

If the mode of a name  $N$  {referring to a multiple value} is some **'reference to FLEXETY ROWS of MODE'**, then the mode of each subname of  $N$  is **'reference to MODE'**.

g) When a name  $N$  which refers to a multiple value  $V$  is "endowed with subnames" {g, 16<sub>2</sub>.1.3.3.e, 18<sub>4</sub>.4.2.b, 19<sub>5</sub>.2.1.2.b, 19<sub>5</sub>.2.3.2.a}, then,

For each index selecting an element  $E$  of  $V$ ,

- a new subname  $M$  is created of the same scope as  $N$ ;
- $M$  is made to refer to  $E$ ;
- $M$  is said to be the name "selected" by that index in  $N$ ;
- if  $M$  is a stowed name {16<sub>2</sub>.1.3.2.b}, then it is itself endowed with subnames {g, 16<sub>2</sub>.1.3.3.e}.

{In addition to the selection of an element {a} or a name {g} by means of an index, it is also possible to select a value, or to generate a new name referring to such a value, by means of a trim {h, i, j} or a **'TAG'** {k, l}. Both indexes and trims are used in the elaboration of **slices** {19<sub>5</sub>.3.2.2}.}

h) A "trim" is an  $n$ -tuple, each element of which is either an integer {corresponding to a **subscript**} or a triplet (l, u, d) {corresponding to a **trimmer** or a **revised-lower-bound-**

**option**}, such that at least one of those elements is a triplet {if all the elements are integers, then the  $n$ -tuple is an index {a}}. Each element of such a triplet is either an integer or is "absent".

{A trim (or an index) is yielded by the elaboration of an **indexer** {19<sub>5</sub>.3.2.2.b}.}

i) The multiple value  $W$  {of  $m$  dimensions} "selected" by a trim  $T$  in a multiple value  $V$  {of  $n$  dimensions,  $1 \leq m \leq n$ } is determined as follows:

- Let  $T$  be composed of integers and triplets  $T_i, i = 1, \dots, n$ , of which  $m$  are actually triplets; let the  $j^{\text{th}}$  triplet be  $(l_j, u_j, d_j), j = 1, \dots, m$ ;

- $W$  is composed of

(i) a descriptor  $((l_1 - d_1, u_1 - d_1), (l_2 - d_2, u_2 - d_2), \dots, (l_m - d_m, u_m - d_m))$ ;

(ii) elements of  $V$ , where the element, if any, selected in  $W$  by an index  $(w_1, \dots, w_m)$   $\{l_j - d_j \leq w_j \leq u_j - d_j\}$  is that selected in  $V$  by the index  $(v_1, \dots, v_n)$  determined as follows:

For  $i = 1, \dots, n$ ,

Case A:  $T_i$  is an integer:

- $v_i = T_i$ ;

Case B:  $T_i$  is the  $j^{\text{th}}$  triplet  $(l_j, u_j, d_j)$  of  $T$ :

- $v_i = w_j + d_j$ .

j) The name  $M$  "generated" by a trim  $T$  from a name  $N$  which refers to a multiple value  $V$  is a {fixed} name, of the same scope as  $N$ , {not necessarily newly created} which refers to the multiple value  $W$  selected {i} by  $T$  in  $V$ . Each subname of  $M$ , as selected by an index  $I_W$ , is one of the {already existing} subnames of  $N$ , as selected by an index  $I_V$ , where each  $I_V$  is determined from  $T$  and the corresponding  $I_W$  using the method given in the previous sub-section.

k) The multiple value  $W$  "selected" by a '**TAG**' in a multiple value  $V$  {each of whose elements is a structured value} is composed of

(i) the descriptor of  $V$ , and

(ii) the fields selected by '**TAG**' in the elements of  $V$ , where the element, if any, selected in  $W$  by an index  $I$  is the field selected by '**TAG**' in the element of  $V$  selected by  $I$ .

l) The name  $M$  "generated" by a '**TAG**' from a name  $N$  which refers to a multiple value  $V$  {each of whose elements is a structured value} is a {fixed} name, of the same scope as  $N$ , {not necessarily newly created} which refers to the multiple value selected {k} by '**TAG**' in  $V$ . Each subname of  $M$  selected by an index  $I$  is the {already existing} name selected {16<sub>2</sub>.1.3.3.e} by '**TAG**' in the subname of  $N$  selected {g} by  $I$ .

### 16.1.3.5 Routines

a) A "routine" is a scene {16<sub>2</sub>.1.1.1.d} composed of a **routine-text** {19<sub>5</sub>.4.1.1.a,b} together with an environ {16<sub>2</sub>.1.1.1.c}.

{A routine may be "called" {19<sub>5</sub>.4.3.2.b}, whereupon the **unit** of its **routine-text** is elaborated.}

b) The mode of a routine composed of a **PROCEDURE-routine-text** is **'PROCEDURE'**.

c) The scope of a routine is the scope of its environ.

### 16.1.3.6 Acceptability of values

a) {There are no values whose mode begins with **'union of'**. There exist names whose modes begin with **'reference to union of'**, e.g., u in UNION (INT, REAL) u; . Here, however, u, whose mode is **'reference to union of integral real mode'**, refers either to a value whose mode is **'integral'** or to a value whose mode is **'real'**. It is possible to discover which of these situations obtains, at a given moment, by means of a **conformity-clause** {17<sub>3</sub>.4.1.q}.

The mode **'MOID'** is "united from" the mode **'MOOD'** if **'MOID'** is some **'union of MOOD-SETY1 MOOD MOODSETY2 mode'**.

b) {There are no values whose mode begins with **'flexible'**. There exist flexible names whose modes begin with **'reference to flexible'**, e.g., a1 in FLEX [1: n] REAL a1; . Here a1, whose mode is **'reference to flexible row of real'**, refers to a multiple value whose mode is **'row of real'** (see also 16<sub>2</sub>.1.3.4.f). In general, there exist values only for those modes obtainable by "deflexing".}

The mode **'MOID1'** "deflexes" to the mode **'MOID2'** if the predicate **'where MOID1 deflexes to MOID2'** holds {18<sub>4</sub>.7.1.a, b, c}.

{The deflexing process obtains **'MOID2'** by removing all **'flexible'**s contained at positions in **'MOID1'** where they are not also contained in any **'REF to MOID3'**. Thus

**'structured with flexible row of character field letter a mode'**,  
which is not the mode of any value, deflexes to

**'structured with row of character field letter a mode'**  
which is therefore the mode of a value referable to by a flexible name of mode

**'reference to structured with flexible row of character field letter a mode'**.  
This mode is already the mode of a name and therefore it cannot be deflexed any further.}

c) {There are no names whose mode begins with **'transient reference to'**.

The yield of a **transient-reference-to-MODE-FORM** is a "transient name" of mode '**reference to MODE**', but, there being no **transient-reference-to-MODE-declarators** in the language {18<sub>4</sub>.6.1}, the syntax ensures that transient names can never be assigned, ascribed or yielded by the calling of a routine.

E.g., `xx := a1[i]` is not an **assignment** because `xx` is not a **reference-to-transient-reference-to-real-identifier**. Transient names originate from the slicing, multiple selection or rowing of a flexible name.}

d) A value of mode  $M1$  is "acceptable to" a mode  $M2$  if

- (i)  $M1$  is the same as  $M2$ , or
- (ii)  $M2$  is united {a} from  $M1$  {thus the mode specified by `UNION (REAL, INT)` accepts values whose mode is that specified by either `REAL` or `INT`}, or
- (iii)  $M2$  deflexes {b} to  $M1$  {thus the mode '**flexible row of real**' (a mode of which there are no values) accepts values such as the yield of the **actual-declarer** `FLEX [1 : n] REAL` which is a value of mode '**row of real**'}, or
- (iv)  $M1$  is some '**reference to MODE**' and  $M2$  is '**transient reference to MODE**' {thus the mode '**transient reference to real**' accepts values (such as the yield of `a1 [i]`) whose mode is '**reference to real**'}.

{See 16<sub>2</sub>.1.4.1.b for the acceptability of the yield of a scene.}

## 16.1.4 Actions

{Suit the action to the word,  
the word to the action.  
Hamlet,                      William Shakespeare. }

### 16.1.4.1 Elaboration

a) The "elaboration" of certain scenes {those whose constructs are designated by certain paranotions} is specified in the sections of this Report headed "Semantics", which describe the sequence of "actions" which are to be carried out during the elaboration of each such scene.

{Examples of actions which may be specified are:

- the causing to hold of relationships,
- the creation of new names, and



- the elaboration of other scenes.}

The "meaning" of a scene is the effect of the actions carried out during its elaboration. Any of these actions or any combination thereof may be replaced by any action or combination which causes the same effect.

b) The elaboration of a scene  $S$  may "yield" a value. If the construct of  $S$  is a **MOID-NOTION**, then that value is, unless otherwise specified, {of such a mode that it is} acceptable to {16<sub>2</sub>.1.3.6.d} '**MOID**'.

{This rule makes it possible, in the semantics, to discuss yields without explicitly prescribing their modes.}

c) If the elaboration of some construct  $A$  in some environ  $E$  is not otherwise specified in the semantics of this Report, and if  $B$  is the only direct descendent of  $A$  which needs elaboration {see below}, then the elaboration of  $A$  in  $E$  consists of the elaboration of  $B$  in  $E$  and the yield, if any, of  $A$  is the yield, if any, of  $B$  {; this automatic elaboration is termed the "pre-elaboration" of  $A$  in  $E$ } .

A construct needs no elaboration if it is invisible {15<sub>1</sub>.1.3.2.h}, if it is a **symbol** {24<sub>9</sub>.1.1.h}, or if its elaboration is not otherwise specified in the semantics of this Report and none of its direct descendents needs elaboration.

{Thus the elaboration of the **reference-to-real-closed-clause** {17<sub>3</sub>.1.1.a} ( $x := 3.14$ ) is (and yields the same value as) the elaboration of its constituent **reference-to-real-serial-clause** {17<sub>3</sub>.2.1.a}  $x := 3.14$ .}

#### 16.1.4.2 Serial and collateral actions

a) An action may be "inseparable", "serial" or "collateral". A serial or collateral action consists of one or more other actions, termed its "direct actions". An inseparable action does not consist of other actions {; what actions are inseparable is left undefined by this Report}.

b) A "descendent action" of another action  $B$  is a direct action either of  $B$ , or of a descendent action of  $B$ .

c) An action  $A$  is the "direct parent" of an action  $B$  if  $B$  is a direct action {a} of  $A$ .

d) The direct actions of a serial action  $S$  take place one after the other; i.e., the completion {16<sub>2</sub>.1.4.3.c, d} of a direct action of  $S$  is followed by the initiation {16<sub>2</sub>.1.4.3.b, c} of the next direct action, if any, of  $S$ . {The elaboration of a scene, being in general composed of a sequence of actions, is a serial action.}

e) The direct actions of a collateral action are merged in time; i.e., one of its descendent inseparable actions which, at that moment, is "active" {16<sub>2</sub>.1.4.3.a} is chosen and carried

out, upon the completion {16<sub>2</sub>.1.4.3.c} of which another such action is chosen, and so on {until all are completed}.

If two actions {collateral with each other} have been said to be "incompatible with" {25<sub>10</sub>.2.4} each other, then {they shall not be merged; i.e.,} no descendent inseparable action of the one shall (then the one {if it is already inseparable} shall not) be chosen if, at that moment, the other is active and one or more, but not all, of its descendent inseparable actions have already been completed; otherwise, the method of choice is left undefined in this Report.

f) If one or more scenes are to be "elaborated collaterally", then this elaboration is the collateral action consisting of the {merged} elaboration of those scenes.

### 16.1.4.3 Initiation, completion and termination

a) An action is either "active" or "inactive".

An action becomes active when it is "initiated" {b, c} or "resumed" {g} and it becomes inactive when it is "completed" {c, d}, "terminated" {e}, "halted" {f} or "interrupted" {h}.

b) When a serial action is "initiated", then the first of its direct actions is initiated. When a collateral action is "initiated", then all of its direct actions are initiated.

c) When an inseparable action is "initiated", it may then be carried out {see 16<sub>2</sub>.1.4.2.e}, whereupon it is "completed".

d) A serial action is "completed" when its last direct action has been completed. A collateral action is "completed" when all of its direct actions have been completed.

e) When an action *A* {whether serial or collateral} is "terminated", then all of its direct actions {and hence all of its descendent actions} are terminated {whereupon another action may be initiated in its place}. {Termination of an action is brought about by the elaboration of a **jump** {19<sub>5</sub>.4.4.2}.}

f) When an action is "halted", then all of its active direct actions {and hence all of its active descendent actions} are halted. {An action may be halted during a "calling" of the routine yielded by the **operator** DOWN {25<sub>10</sub>.2.4.d}, whereupon it may subsequently be resumed during a calling of the routine yielded by the **operator** UP {25<sub>10</sub>.2.4.e}.}

If, at any time, some action is halted and it is not descended from a "process" of a "parallel action" {25<sub>10</sub>.2.4} of whose other process (es) there still exist descendent active inseparable actions, then the further elaboration is undefined. {Thus it is not defined that the elaboration of the **collateral-clause** in

```
BEGIN SEMA sergei = LEVEL 0;
  (PAR BEGIN (DOWN sergei; print (pokrovsky)), SKIP END,
    (read (pokrovsky); UP sergei))
END
```

will ever be completed.}

g) When an action  $A$  is "resumed", then those of its direct actions which had been halted consequent upon the halting of  $A$  are resumed.

h) An action may be "interrupted" by an event {e.g., "overflow"} not specified by the semantics of this Report but caused by the computer if its limitations {16<sub>2</sub>.2.2.b} do not permit satisfactory elaboration. When an action is interrupted, then all of its direct actions, and possibly its direct parent also, are interrupted. {Whether, after an interruption, that action is resumed, some other action is initiated or the elaboration of the **program** ends, is left undefined by this Report.}

{The effect of the definitions given above is as follows:

During the elaboration of a **program** {16<sub>2</sub>.2.2.a} the elaboration of its **closed-clause** in the empty primal environ is active. At any given moment, the elaboration of one scene may have called for the elaboration of some other scene or of several other scenes collaterally. If and when the elaboration of that other scene or scenes has been completed, the next step of the elaboration of the original scene is taken, and so on until it, in turn, is completed.

It will be seen that all this is analogous to the calling of one subroutine by another; upon the completion of the execution of the called subroutine, the execution of the calling subroutine is continued; the semantic rules given in this Report for the elaboration of the various paranotions correspond to the texts of the subroutines; the semantic rules may even, in suitable circumstances, invoke themselves recursively (but with a different construct or in a different environ on each occasion).

Thus there exists, at each moment, a tree of active actions descended {16<sub>2</sub>.1.4.2.b} from the elaboration of the **program**.}

#### 16.1.4.4 Abbreviations

{In order to avoid some long and turgid phrases which would otherwise have been necessary in the Semantics, certain abbreviations are used freely throughout the text of this Report.}

a) The phrase "the  $A$  of  $B$ ", where  $A$  and  $B$  are paranotions, stands for "the  $A$  which is a direct descendent {15<sub>1</sub>.1.3.2.f} of  $B$ ".

{This permits the abbreviation of "direct descendent of" to "of" or "its", e.g., in the **assignment** {19<sub>5</sub>.2.1.1.a}  $i := 1$ ,  $i$  is "its" **destination** (or  $i$  is the, or a, **destination** "of" the **assignment**  $i := 1$ ), whereas  $i$  is not a **destination** of the **serial-clause**  $i := 1$ ;  $j := 2$  (although it is a constituent **destination** {15<sub>1</sub>.1.4.2.d} of it).}

b) The phrase " $C$  in  $E$ ", where  $C$  is a construct and  $E$  is an environ, stands for "the scene composed {16<sub>2</sub>.1.1.1.d} of  $C$  and  $E$ ". It is sometimes even further shortened to just " $C$ " when it is clear which environ is meant.

{Since the process of elaboration {16<sub>2</sub>.1.4.1.a} may be applied only to scenes, this abbreviation appears most frequently in forms such as "A **loop-clause**  $C$ , in an environ  $E1$ , is elaborated ... " {17<sub>3</sub>.5.2} and "An **assertion**  $A$  is elaborated ... " (19<sub>5</sub>.2.1.2.a, where it is the elaboration of  $A$  in any appropriate environ that is being discussed).}

c) The phrase "the yield of  $S$ ", where  $S$  is a scene whose elaboration is not explicitly prescribed, stands for "the yield obtained by initiating the elaboration of  $S$  and awaiting its completion".

{Thus the sentence {17<sub>3</sub>.2.2.c} :

" $W$  is the yield of that **unit**;"

(which also makes use of the abbreviation defined in b above) is to be interpreted as meaning:

" $W$  is the yield obtained upon the completion of the elaboration, hereby initiated, of the scene composed of that **unit** and the environ under discussion;" .

}

d) The phrase "the yields of  $S_1, \dots, S_n$ ", where  $S_1, \dots, S_n$  are scenes whose elaboration is not explicitly prescribed, stands for "the yields obtained by initiating the collateral elaboration {16<sub>2</sub>.1.4.2.f} of  $S_1, \dots, S_n$  and awaiting its completion {which implies the completion of the elaboration on them all} ".

If some or all of  $S_1, \dots, S_n$  are described as being, in some environ, certain constituents of some construct, then their yields are to be considered as being taken in the textual order {15<sub>1</sub>.1.3.2.i} of those constituents within that construct.

{Thus the sentence {17<sub>3</sub>.3.2.b} :

"let  $V_1, \dots, V_m$  be the {collateral} yields of the constituent **units** of  $C$ ;"

is to be interpreted as meaning:

"let  $V_1, \dots, V_m$  be the respective yields obtained upon the completion of the collateral elaboration, hereby initiated, of the scenes composed of the constituent **units** of  $C$ , considered in their textual order, together with the environ in which  $C$  was being elaborated;" .

}

e) The phrase "if  $A$  is  $B$ ", where  $A$  and  $B$  are hypernotions, stands for "if  $A$  is equivalent {16<sub>2</sub>.1.1.2.a} to  $B$ ".

{Thus, in "Case C: **'CHOICE'** is some **'choice using UNITED'**" {17<sub>3</sub>.4.2.b}, it matters not whether **'CHOICE'** happens to begin with **'choice using union of'** or with some **'choice using MU definition of union of'**.}

f) The phrase "the mode is *A*", where *A* is a hypernotation, stands for "the mode {is a class of **'MOID'**s which} includes *A*".

{This permits such shortened forms as "the mode is some **'structured with FIELDS mode'**", "the mode begins with **'union of'**", and "the mode envelops a **'FIELD'**"; in general, a mode may be specified by quoting just one of the **'MOID'**s included in it.)

g) The phrase "the value selected (generated) by the **field-selector *F***" stands for "if *F* is a **field-selector-with-TAG** {18<sub>4</sub>.8.1.f}, then the value selected {16<sub>2</sub>.1.3.3.a, e, 16<sub>2</sub>.1.3.4.k} (generated {16<sub>2</sub>.1.3.4.l}) by that **'TAG'**".

## 16.2 The program

### 16.2.1 Syntax

a) **program : strong void new closed clause** {17<sub>3</sub>.1.a} . {See also 25<sub>10</sub>.1 .}

### 16.2.2 Semantics

{"I can explain all the poems that ever were invented -  
and a good many that haven't been invented just yet."  
Through the Looking-glass, Lewis Carroll. }

a) The elaboration of a **program** is the elaboration of its **strong-void-new-closed-clause** in an empty environ {16<sub>2</sub>.1.1.1.c} termed the "primal environ".

{Although the purpose of this Report is to define the meaning of a **particular-program** (25<sub>10</sub>.1.1.g), that meaning is established only by first defining the meaning of a **program** in which that **particular-program** is embedded (25<sub>10</sub>.1.2).}

{In this Report, the syntax says which sequences of symbols are terminal productions of **'program'**, and the semantics which actions are performed by the computer when elaborating a program. Both syntax and semantics are recursive. Though certain sequences of symbols may be terminal productions of **'program'** in more than one way (see also 15<sub>1</sub>.1.3.2.f), this syntactic ambiguity does not lead to a semantic ambiguity.}

b) In Algol 68, a specific syntax for constructs is provided which, together with its recursive definition, makes it possible to describe and to distinguish between arbitrarily large

production trees, to distinguish between arbitrarily many different values of a given mode (except certain modes like **'boolean'** and **'void'**) and to distinguish between arbitrarily many modes, which allows arbitrarily many objects to exist within the computer and which allows the elaboration of a **program** to involve an arbitrarily large, not necessarily finite, number of actions. This is not meant to imply that the notation of the objects in the computer is that used in this Report nor that it has the same possibilities. It is not assumed that these two notations are the same nor even that a one-to-one correspondence exists between them; in fact, the set of different notations of objects of a given category may be finite. It is not assumed that the computer can handle arbitrary amounts of presented information. It is not assumed that the speed of the computer is sufficient to elaborate a given **program** within a prescribed lapse of time, nor that the number of objects and relationships that can be established is sufficient to elaborate it at all.

c) A model of the hypothetical computer, using a physical machine, is said to be an "implementation" of Algol 68 if it does not restrict the use of the language in other respects than those mentioned above. Furthermore, if a language *A* is defined whose **particular-programs** are also **particular-programs** of a language *B*, and if each such **particular-program** for which a meaning is defined in *A* has the same defined meaning in *B*, then *A* is said to be a "sublanguage" of *B*, and *B* a "superlanguage" of *A*.

{Thus a sublanguage of Algol 68 might be defined by omitting some part of the syntax, by omitting some part of the **standard-prelude**, and/or by leaving undefined something which is defined in this Report, so as to enable more efficient solutions to certain classes of problem or to permit implementation on smaller machines.

Likewise, a superlanguage of Algol 68 might be defined by additions to the syntax, semantics or **standard-prelude**, so as to improve efficiency (by allowing the user to provide additional information) or to permit the solution of problems not readily amenable to Algol 68.}

A model is said to be an implementation of a sublanguage if it does not restrict the use of the sublanguage in other respects than those mentioned above.

{See 24<sub>9</sub>.3.c for the term "implementation of the reference language".}

{A sequence of **symbols** which is not a **particular-program** but can be turned into one by deleting or inserting a certain number of **symbols** and not a smaller number could be regarded as a **particular-program** with that number of syntactical errors. Any **particular-program** that can be obtained by deleting or inserting that number of symbols may be termed a "possibly intended" **particular-program**. Whether a **particular-program** or one of the possibly intended **particular-programs** has the effect its author in fact intended it to have is a matter which falls outside this Report.}

{In an implementation, the **particular-program** may be "compiled", i.e., translated into an "object program" in the code of the physical machine. Under certain circumstances, it may be advantageous to compile parts of the **particular-program** independently, e.g.,

parts which are common to several **particular-programs**. If such a part contains **applied-indicators** which identify **defining-indicators** not contained in that part, then compilation into an efficient object program may be assured by preceding the part by a sequence of **declarations** containing those **defining-indicators**.)

{The definition of specific sublanguages and also the specification of actions not definable by any **program** (e.g., compilation or initiation of the elaboration) is not given in this Report. See, however, 24<sub>9</sub>.2 for the suggested use of **pragmats** to control such actions.}

# ALGOL 68 REVISED REPORT



## PART II - Fundamental Constructions

{This part presents the essential structure of **programs**:

- the general rules for constructing them;
- the ways of defining **indicators** and their properties, at each new level of construction;
- the constructs available for programming primitive actions.}

# ALGOL 68 REVISED REPORT

# Chapter 17

## Clauses

{Clauses provide

- a hierarchical structure for **programs**,
- the introduction of new **ranges** of definitions,
- serial or collateral composition, parallelism, choices and loops.}

### 17.0.3 Syntax

- a) **\*phrase : SOME unit** {17<sub>3</sub>.2.d} ;  
    **NEST declaration of DECS** {18<sub>4</sub>.1.a} .
- b) **\*SORT MODE expression : SORT MODE NEST UNIT** {19<sub>5</sub>.A} .
- c) **\*statement : strong void NEST UNIT** {19<sub>5</sub>.A} .
- d) **\*MOID constant :**  
    **MOID NEST DEFIED identifier with TAG** {18<sub>4</sub>.8.a, b} ;  
    **MOID NEST denoter** {23<sub>8</sub>.0.a} .
- e) **\*MODE variable :**  
    **reference to MODE NEST DEFIED identifier with TAG** {18<sub>4</sub>.3.a, b} .
- f) **\*NEST range :**  
    **SOID NEST serial clause defining LAYER** {17<sub>3</sub>.2.a} ;  
    **SOID NEST chooser CHOICE STYLE clause** {17<sub>3</sub>.4.b} ;  
    **SOID NEST case part of choice using UNITED** {17<sub>3</sub>.4.i} ;  
    **NEST STYLE repeating part with DEC** {17<sub>3</sub>.5.e} ;

**NEST STYLE while do part** {17<sub>3</sub>.5.f} ;  
**PROCEDURE NEST routine text** {19<sub>5</sub>.4.1.a, b} .

{**NEST-ranges** arise in the definition of "identification" {22<sub>7</sub>.2.2.b}.}

## 17.0.4 Semantics

A "nest" is a '**NEST**'. The nest "of" a construct is the '**NEST**' enveloped by the original of that construct, but not by any '**defining LAYER**' contained in that original.

{The nest of a construct carries a record of all the **declarations** forming the environment in which that construct is to be interpreted.

Those constructs which are contained in a **range** *R*, but not in any smaller **range** contained within *R*, may be said to comprise a "reach". All constructs in a given reach have the same nest, which is that of the immediately surrounding reach with the addition of one extra '**LAYER**'. The syntax ensures {17<sub>3</sub>.2.1.b, 17<sub>3</sub>.4.1.i, j, k, 17<sub>3</sub>.5.1.e, 19<sub>5</sub>.4.1.1.b} that each '**PROP**' {18<sub>4</sub>.8.1.E} or "property" in the extra '**LAYER**' is matched by a **defining-indicator** {18<sub>4</sub>.8.1.a} contained in a **definition** in that reach.}

## 17.1 Closed clauses

{**Closed-clauses** are usually used to construct **units** from **serial-clauses** as, e.g.,

```
(REAL x; read (x); x) in
(REAL x; read (x); x) + 3.14. }
```

### 17.1.1 Syntax

A) **SOID :: SORT MOID.**

B) **PACK :: STYLE pack.**

- a) **SOID NEST closed clause** {16<sub>2</sub>.2.a, 19<sub>5</sub>.D, 19<sub>5</sub>.5.1.a, 25<sub>10</sub>.3.4.1.h, 25<sub>10</sub>.3.4.9.a} :  
**SOID NEST serial clause defining LAYER** {17<sub>3</sub>.2.a} **PACK.**  
**{LAYER :: new DECSETY LABSETY.}**

{Example

```
a) BEGIN x := 1; y := 2 END }
```

{The yield of a **closed-clause** is that of its constituent **serial-clause**, by way of pre-elaboration {16<sub>2</sub>.1.4.1.c}.}

## 17.2 Serial clauses

{The purposes of **serial-clauses** are

- the construction of new **ranges** of definitions, and
- the serial composition of actions.

A **serial-clause** consists of a possibly empty sequence of unlabelled **phrases**, the last of which, if any, is a **declaration**, followed by a sequence of possibly labelled **units**. The **phrases** and the **units** are separated by **go-on-tokens**, viz., semicolons. Some of the **units** may instead be separated by **completers**, viz., `EXITS`; after a **completer**, the next **unit** must be labelled so that it can be reached. The value of the final **unit**, or of a **unit** preceding an `EXIT`, determines the value of the **serial-clause**.

For example, the following **serial-clause** yields `true` if and only if the vector `a` contains the integer 8:

```
INT n; read (n);
[1 : n] INT a; read (a);
FOR i TO n DO IF a [i] = 8 THEN GOTO success FI OD;
FALSE EXIT
success: TRUE.}
```

### 17.2.1 Syntax

- a) **SOID NEST serial clause defining new PROPSETY** {17<sub>3</sub>.1.a, 17<sub>3</sub>.4.f, l, 17<sub>3</sub>.5.h} :  
**SOID NEST new PROPSETY series with PROPSETY {b}.**  
 {Here **PROPSETY :: DECSETY LABSETY.**}
- b) **SOID NEST series with PROPSETY** {a, b, 17<sub>3</sub>.4.c} :  
**strong void NEST unit {d}, go on {24<sub>9</sub>.4.f} token,**  
**SOID NEST series with PROPSETY {b};**  
**where (PROPSETY) is (DECS DECSETY LABSETY),**  
**NEST declaration of DECS {18<sub>4</sub>.1.a} , go on {24<sub>9</sub>.4.f} token,**  
**SOID NEST series with DECSETY LABSETY {b};**  
**where (PROPSETY) is (LAB LABSETY),**  
**NEST label definition of LAB {c},**

- SOID NEST series with LABSETY {b};**  
**where (PROPSETY) is (LAB LABSETY) and SOID balances SOID1 and SOID2 {e},**  
**SOID1 NEST unit {d}, completion {24<sub>9</sub>.4.f} token,**  
**NEST label definition of LAB {c},**  
**SOID2 NEST series with LABSETY {b};**  
**where (PROPSETY) is (EMPTY),**  
**SOID NEST unit {d}.**
- c) **NEST label definition of label TAG {b} :**  
**label NEST defining identifier with TAG {18<sub>4</sub>.8.a} , label {24<sub>9</sub>.4.f} token.**
- d) **SOME unit {b, 17<sub>3</sub>.3.b, g, 17<sub>3</sub>.4.i, 17<sub>3</sub>.5.d, 18<sub>4</sub>.6.m, n, 19<sub>5</sub>.2.1.c, 19<sub>5</sub>.3.2.e, 19<sub>5</sub>.4.1.a, b, 19<sub>5</sub>.4.3.c, 25<sub>10</sub>.3.4.10.b, c, d} :**  
**SOME UNIT {19<sub>5</sub>A, -}.**
- e) **WHETHER SORT MOID balances SORT1 MOID1 and SORT2 MOID2 {b, 17<sub>3</sub>.3.b, 17<sub>3</sub>.4.d, h} :**  
**WHETHER SORT balances SORT1 and SORT2 {f} and MOID balances MOID1 and MOID2 {g}.**
- f) **WHETHER SORT balances SORT1 and SORT2 {e, 19<sub>5</sub>.2.2.a} :**  
**where (SORT1) is (strong), WHETHER (SORT2) is (SORT);**  
**where (SORT2) is (strong), WHETHER (SORT1) is (SORT).**
- g) **WHETHER MOID balances MOID1 and MOID2 {e} :**  
**where (MOID1) is (MOID2), WHETHER (MOID) is (MOID1) ;**  
**where (MOID1) is (transient MOID2), WHETHER (MOID) is (MOID1) ;**  
**where (MOID2) is (transient MOID1), WHETHER (MOID) is (MOID2).**
- h) **\*SOID unitary clause : SOID NEST unit {d}.**
- i) **\*establishing clause :**  
**SOID NEST serial clause defining LAYER {17<sub>3</sub>.2.a} ;**  
**MODE NEST enquiry clause defining LAYER {17<sub>3</sub>.4.c} .**

{Examples:

```
b) read (x1); REAL s := 0;
   sum: FOR i TO n DO (x1[i] >0 | s += x1[i] | nonpos) OD EXIT
   nonpos: print (s) •
REAL s := 0;
   sum: FOR i TO n DO (x1[i] >0 | s += x1[i] | nonpos) OD EXIT
   nonpos: print (s) •
sum: FOR i TO n DO (x1[i] >0 | s += x1[i] | nonpos) OD EXIT
nonpos: print (s) •
```

```

FOR i TO n DO (x1[i] >0 | s += x1[i] | nonpos) OD EXIT
  nonpos: print (s) •
print (s)

```

c) sum:

d) print (s) }

{Often, a **series** must be "balanced" {17<sub>3</sub>.2.1.e}. For remarks concerning balancing, see 17<sub>3</sub>.4.1 .}

## 17.2.2 Semantics

a) The yield of a **serial-clause**, in an environ  $E$ , is the yield of the elaboration of its **series**, or of any **series** elaborated "in its place" {19<sub>5</sub>.4.4.2}, in the environ "established" {b} around  $E$  according to that **serial-clause**; it is required that the yield be not newer in scope than  $E$ .

b) The environ  $E$  "established"

- upon an environ,  $E_1$ , possibly not specified, {which determines its scope, }
- around an environ  $E_2$  {which determines its composition},
- according to a **NOTION-defining-new-PROPSETY**  $C$ , possibly absent, {which prescribes its locale, }
- with values  $V_1, \dots, V_n$ , possibly absent, {which are possibly to be ascribed, }

is determined as follows:

- if  $E_1$  is not specified, then let  $E_1$  be  $E_2$ ;
- $E$  is newer in scope than  $E_1$  and is composed of  $E_2$  and a new locale corresponding to '**PROPSETY**', if  $C$  is present, and to '**EMPTY**' otherwise;

Case A:  $C$  is an **establishing-clause**:

For each constituent **mode-definition**  $M$ , if any, of  $C$ ,

- the scene composed of
  - (i) the **actual-declarer** of  $M$ , and
  - (ii) the environ necessary for {22<sub>7</sub>.2.2.c} that **actual-declarer** in  $E$ , is ascribed in  $E$  to the **mode-indication** of  $M$ ;

For each constituent **label-definition**  $L$ , if any, of  $C$ ,

- the scene composed of
  - (i) the **series** of which  $L$  is a direct descendent, and
  - (ii) the environ  $E$ ,

is ascribed in  $E$  to the **label-identifier** of  $L$ :

If each '**PROP**' enveloped by '**PROPSETY**' is some '**DYADIC TAD**' or '**label TAG**', then  $E$  is said to be "nonlocal" {see 19<sub>5</sub>.2.3.2.b} ;

Case B:  $C$  is a **declarative**, a **for-part** or a **specification**: For  $i = 1 \dots n$ , where  $n$  is the number of '**DEC**'s enveloped by '**PROPSETY**',

- $V_i$  is ascribed {18<sub>4</sub>.8.2.a} in  $E$  to the  $i^{th}$  constituent **defining-identifier**, if any, of  $C$  and, otherwise {in the case of an invisible **for-part**}, to an **integral-defining-indicator-with-letter-aleph**;

If  $C$  is a **for-part** or a **specification**,

then  $E$  is nonlocal.

{Other cases, i.e., when  $C$  is absent:

- $E$  is local (see 19<sub>5</sub>.2.3.2.b), but not further defined. }

c) The yield  $W$  of a **series**  $C$  is determined as follows:

If  $C$  contains a direct descendent **unit** which is not followed by a **go-on-token**, then

- $W$  is the yield of that **unit**;

otherwise,

- the **declaration** or the **unit**, if any, of  $C$  is elaborated;
- $W$  is the yield of the **series** of  $C$ .

{See also 19<sub>5</sub>.4.4.2.Case A .}

## 17.3 Collateral and parallel clauses

{**Collateral-clauses** allow an arbitrary merging of streams of actions. **Parallel-clauses** provide, moreover, levels of coordination for the synchronization {25<sub>10</sub>.2.4} of that merging.

A **collateral-** or **parallel-clause** consists of a sequence of **units** separated by **and-also-symbols** (viz., ", "), and is enclosed by parentheses or by a **BEGIN-END** pair; a **parallel-clause** begins moreover with **PAR**.



**Collateral-clauses**, but not **parallel-clauses**, may yield stowed values composed from the yields of the constituent **units**.

Examples of **collateral-clauses** yielding stowed values:

```
[] INT q = (1, 4, 9, 16, 25);
STRUCT (INT price, STRING category) bike := (150, "sport").
```

Example of a **parallel-clause** which synchronizes eating and speaking:.

```
PROC VOID eat, speak; SEMA mouth = LEVEL 1;
PAR BEGIN
  DO
    DOWN mouth;
    eat;
    UP mouth
  OD,
  DO
    DOWN mouth;
    speak;
    UP mouth
  OD
END.}
```

### 17.3.1 Syntax

- a) **strong void NEST collateral clause** {19<sub>5</sub>.D, 19<sub>5</sub>.5.1.a} :  
**strong void NEST joined portrait** {b} **PACK**.
- b) **SOID NEST joined portrait** {a, b, c, d, 17<sub>3</sub>.4.g} :  
**where SOID balances SOID1 and SOID2** {17<sub>3</sub>.2.e} ,  
**SOID1 NEST unit** {17<sub>3</sub>.2.d} 1, **and also** {24<sub>9</sub>.4.f} **token**,  
**SOID2 NEST unit** {17<sub>3</sub>.2.d}  
**or alternatively SOID2 NEST joined portrait** {b}.
- c) **strong void NEST parallel clause** {19<sub>5</sub>.D, 19<sub>5</sub>.5.1.a} :  
**parallel** {24<sub>9</sub>.4.f} **token**, **strong void NEST joined portrait** {b} **PACK**.
- d) **strong ROWS of MODE NEST collateral clause** {19<sub>5</sub>.D, 19<sub>5</sub>.5.1.a} :  
**where (ROWS) is (row)**,  
**strong MODE NEST joined portrait** {b} **PACK** ;  
**where (ROWS) is (row ROWS1)**,  
**strong ROWS1 of MODE NEST joined portrait** {b} **PACK** ;  
**EMPTY PACK**.
- e) **strong structured with FIELDS FIELD mode NEST collateral clause** {19<sub>5</sub>.D,  
19<sub>5</sub>.5.1.a} :  
**NEST FIELDS FIELD portrait** {f} **PACK**.

- f) **NEST FIELDS FIELD portrait** {e, f} :  
**NEST FIELDS portrait** {f, g}, and also {24<sub>9</sub>.4.f} token,  
**NEST FIELD portrait** {g}.  
**{FIELD :: MODE field TAG.}**
- g) **NEST MODE field TAG portrait** {f} : **strong MODE NEST unit** {17<sub>3</sub>.2.d} .
- h) **\*structure display** :  
**strong structured with FIELDS FIELD mode NEST collateral clause** {e}.
- i) **\*row display** : **strong ROWS of MODE NEST collateral clause** {d}.
- j) **\*display** : **strong STOWED NEST collateral clause** {d, e}.
- k) **\*vacuum** : **EMPTY PACK.**

{Examples:

- a) (x := 1, y := 2)
- b) x := 1, y := 2
- c) x := 1, y := 2
- d) (1, 2) (in [] REAL (1, 2))
- e) (1, 2) (in COMPL (1, 2))
- f) 1, 2
- g) 1 }

{**Structure-displays** must contain at least two **FIELD-portraits**, for, otherwise, in the reach of

MODE M = STRUCT (REF M m); M nobuo, yoneda;  
the **assignment** nobuo := (yoneda) would be syntactically ambiguous and could produce different effects; however,  $\text{m OF nobuo := yoneda}$  is unambiguous.

**Row-displays** contain zero, two or more constituent **units**. It is also possible to present a single value as a multiple value, e.g., [1 : 1] INT v := 123, but this uses a coercion known as rowing {21<sub>6</sub>.6}.

## 17.3.2 Semantics

- a) The elaboration of a **void-collateral-clause** or **void-parallel-clause** consists of the collateral elaboration of its constituent **units** and yields `empty`.

b) The yield  $W$  of a **STOWED-collateral-clause**  $C$  is determined as follows:  
 If the direct descendent of  $C$  is a **vacuum**,  
 then

{'STOWED' is some 'ROWS of MODE' and} each bound pair in the descriptor of  $W$   
 is  $(1, 0)$  {and it has one ghost element whose value is irrelevant};

otherwise,

- let  $V_1, \dots, V_m$  be the {collateral} yields of the constituent **units** of  $C$ ;

Case A: 'STOWED' is some 'structured with FIELDS mode':

- the fields of  $W$ , taken in order, are  $V_1, \dots, V_m$ ;

Case B: 'STOWED' is some 'row of MODE1':

- $W$  is composed of
  - a descriptor  $((1, m))$ ,
  - $V_1, \dots, V_m$ ;

For  $i = 1 \dots m$ ,

- the element selected by the index (i) in  $W$  is  $V_i$ ;

Case C: 'STOWED' is some 'row ROWS of MODE2':

- it is required that the descriptors of  $V_1, \dots, V_m$  be identical;
- let the descriptor of {say}  $V_1$  be  $((l_1, u_1), \dots, (l_n, u_n))$ ;
- $W$  is composed of
  - a descriptor  $((1, m), (l_1, u_1), \dots, (l_n, u_n))$ ;
  - the elements of  $V_1, \dots, V_m$ ;

For  $i = 1 \dots m$ ,

the element selected by an index  $(i, i_1, \dots, i_n)$  in  $W$  is that selected by  $(i_1, \dots, i_n)$   
 in  $V_i$ .

{Note that in [„ ] CHAR block = ("abc", "def"), the descriptor of the three-dimensional  
 yield  $W$  will be  $((1, 2), (1, 1), (1, 3))$ , since the **units** "abc" and "def" are first rowed {21<sub>6</sub>.6},  
 so that  $V_1$  and  $V_2$  have descriptors  $((1, 1), (1, 3))$ .}

## 17.4 Choice clauses

{**Choice-clauses** enable a dynamic choice to be made among different paths in a computation. The choice among the alternatives (the **in-CHOICE-** and the **out-CHOICE-clause**) is determined by the success or failure of a test on a truth value, on an integer or on a mode. The value under test is computed by an **enquiry-clause** before the choice is made.

A **choice-using-boolean-clause** (or **conditional-clause**) is of the form

```
(x >0 | x | 0) in the "brief" style, or
```

```
IF x >0 THEN x ELSE 0 FI in the "bold" style;
```

$x >0$  is the **enquiry-clause**, THEN  $x$  is the **in-CHOICE-clause** and ELSE 0 is the **out-CHOICE-clause**; all three may have the syntactical structure of a **series**, because all **choice-clauses** are well closed. A **choice-using-boolean-clause** may also be reduced to

```
(x <0 | x := - x) or
```

```
IF x <0 THEN x := - x FI;
```

the omitted **out-CHOICE-clause** is then understood to be an ELSE SKIP. On the other hand, the choice can be reiterated by writing

```
(x >0 | 1 + x | : x <0 | 1 - x | 1) or
```

```
IF x >0 THEN 1 + x ELIF x <0 THEN 1 - x ELSE 1 FI,
```

and so on; this is to be understood as

```
(x >0 | 1 + x | (x <0 | 1 - x | 1)).
```

**CASE-clauses**, which define choices depending on an integer or on a mode, are different in that the **in-CASE-clause** is further decomposed into **units**. The general pattern is

```
(-- | --, ..., -- | --) or
```

```
CASE -- IN --, ..., -- OUT -- ESAC.
```

The choice may also be reiterated by use of OUSE.

In a **choice-using-integral-clause** (or **case-clause**), the parts are simply **units** and there must be at least two of them; the choice among the **units** follows their textual ordering.

Example:

```
PROC VOID work, relax, enjoy;
CASE INT day; read (day); day
IN work, work, work, work, relax, enjoy
OUT print ((day, "is not in the week"))
ESAC.
```

In a **choice-using-UNITED-clause** (or **conformity-clause**), which tests modes, each **case-part-of-CHOICE** is of the form (DECLARER identifier): unit or (DECLARER): unit. The mode specified by the **declarer** is compared with the mode of the value under test; the **identifier**, if present, is available inside the **unit** to access that value, with the full security of syntactical mode checking. The '**UNITED**' mode provides the required freedom for the mode of the value under test; moreover, that '**UNITED**' mode must contain the mode of each **specification** for, otherwise, the corresponding **case-part-of-CHOICE**

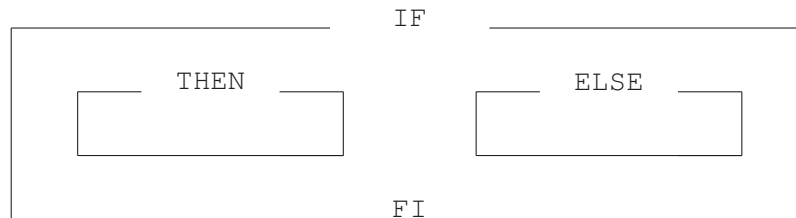
could never be chosen.

Example:

```
MODE BOY = STRUCT (INT age, REAL weight),
MODE GIRL = STRUCT (INT age, REAL beauty);
PROC UNION (BOY, GIRL) newborn;
CASE newborn IN
  (BOY john): print (weight OF john),
  (GIRL mary): print (beauty OF mary)
ESAC. }
```

{The flowers that bloom in the spring,  
Tra la,  
Have nothing to do with the case.  
Mikado, W.S. Gilbert. }

{The hierarchy of **ranges** in **conditional-clauses** is illustrated by



and similarly for the other kinds of choice. Thus the nest and the environ of the **enquiry-clause** remain valid over the **in-CHOICE-clause** and the **out-CHOICE-clause**. However, no transfer back from the **in-** or **out-CHOICE-clause** into the **enquiry-clause** is possible, since the latter can contain no **label-definitions** (except within a **closed-clause** contained within it).}

### 17.4.1 Syntax

A) **CHOICE** :: choice using boolean ; **CASE**.

B) **CASE** :: choice using integral ; choice using **UNITED**.

- a) **SOID NEST1 CHOICE clause** {19<sub>5</sub>.D, 19<sub>5</sub>.5.1.a, 25<sub>10</sub>.3.4.1.h, 25<sub>10</sub>.3.4.9.a} :  
**CHOICE STYLE start** {24<sub>9</sub>.1.a, -},  
**SOID NEST1 chooser CHOICE STYLE clause** {b},  
**CHOICE STYLE finish** {24<sub>9</sub>.1.e, -}.

- b) **SOID NEST1 chooser choice using MODE STYLE clause** {a, l} :

- MODE NEST1 enquiry clause defining LAYER2 {c, -},  
SOID NEST1 LAYER2 alternate choice using MODE STYLE clause {d}.**
- c) **MODE NEST1 enquiry clause defining new DECSETY2 {b, 17<sub>3</sub>.5.g} :  
meek MODE NEST1 new DECSETY2 series with DECSETY2 {17<sub>3</sub>.2.b} .**
  - d) **SOID NEST2 alternate CHOICE STYLE clause {b} :  
SOID NEST2 in CHOICE STYLE clause {e};  
where SOID balances SOID1 and SOID2 {17<sub>3</sub>.2.e} ,  
SOID1 NEST2 in CHOICE STYLE clause {e},  
SOID2 NEST2 out CHOICE STYLE clause {l}.**
  - e) **SOID NEST2 in CHOICE STYLE clause {d} :  
CHOICE STYLE in {24<sub>9</sub>.1.b, -}, SOID NEST2 in part of CHOICE {f, g, h}.**
  - f) **SOID NEST2 in part of choice using boolean {e} :  
SOID NEST2 serial clause defining LAYER3 {17<sub>3</sub>.2.a} .**
  - g) **SOID NEST2 in part of choice using integral {e} :  
SOID NEST2 joined portrait {17<sub>3</sub>.3.b} .**
  - h) **SOID NEST2 in part of choice using UNITED {e, h} :  
SOID NEST2 case part of choice using UNITED {i};  
where SOID balances SOID1 and SOID2 {17<sub>3</sub>.2.e} ,  
SOID1 NEST2 case part of choice using UNITED {i}, and also {24<sub>9</sub>.4.f}  
token,  
SOID2 NEST2 in part of choice using UNITED {h}.**
  - i) **SOID NEST2 case part of choice using UNITED {h} :  
MOID NEST2 LAYER3 specification defining LAYER3 {j, k, -},  
where MOID unites to UNITED {21<sub>6</sub>.4.b} ,  
SOID NEST2 LAYER3 unit {17<sub>3</sub>.2.d} .  
{Here LAYER :: new MODE TAG ; new EMPTY.}**
  - j) **MODE NEST3 specification defining new MODE TAG3 {i} :  
NEST3 declarative defining new MODE TAG3 {19<sub>5</sub>.4.1.e} brief pack,  
colon {24<sub>9</sub>.4.f} token.**
  - k) **MOID NEST3 specification defining new EMPTY {i} :  
formal MOID NEST3 declarer {18<sub>4</sub>.6.b} brief pack,  
colon {24<sub>9</sub>.4.f} token.**
  - l) **SOID NEST2 out CHOICE STYLE clause {d} :  
CHOICE STYLE out {24<sub>9</sub>.1.d, -},  
SOID NEST2 serial clause defining LAYER3 {17<sub>3</sub>.2.a} ;  
CHOICE STYLE again {24<sub>9</sub>.1.c, -},  
SOID NEST2 chooser CHOICE2 STYLE clause {b},  
where CHOICE2 may follow CHOICE {m}.**

- m) **WHETHER choice using MODE2 may follow choice using MODE1 {l} :**  
**where (MODE1) is (MOOD), WHETHER (MODE2) is (MODE1) ;**  
**where (MODE1) begins with (union of),**  
**WHETHER (MODE2) begins with (union of).**
- n) **\*SOME choice clause : SOME CHOICE clause {a}.**
- o) **\*SOME conditional clause : SOME choice using boolean clause {a}.**
- p) **\*SOME case clause : SOME choice using integral clause {a}.**
- q) **\*SOME conformity clause : SOME choice using UNITED clause {a}.**

{Examples:

- a)  $(x > 0 \mid x \mid 0) \bullet$   
CASE i IN princeton, grenoble OUT finish ESAC •  
CASE uir IN (INT i): print (i), (REAL): print ("no") ESAC
- b)  $x > 0 \mid x \mid 0$
- c)  $x > 0 \bullet i \bullet uir$
- d)  $\mid x \bullet \mid x \mid 0$
- e)  $\mid x \bullet$   
IN princeton, grenoble •  
IN (INT i): print (i), (REAL): print ("no")
- f) x
- g) princeton, grenoble
- h) (INT i): print (i), (REAL): print ("no")
- i) (INT i): print (i)
- j) (INT i):
- k) (REAL):
- l) OUT finish • |:  $x < 0 \mid -x \mid 0$

{I would to God they would either conform,  
or be more wise, and not be caught!  
Diary. 7 Aug. 1664, Samuel Pepys. }

{Rule d illustrates why 'SORT MOID's should be "balanced". If an **alternate-CHOICE-clause** is, say, firm, then at least its **in-CHOICE-clause** or its **out-CHOICE-clause** must

be firm, while the other may be strong. For example, in

$(p \mid x \mid \text{SKIP}) + (p \mid \text{SKIP} \mid y)$ ,

the **conditional-clause**  $(p \mid x \mid \text{SKIP})$  is balanced by making  $\mid x$  firm and  $\mid \text{SKIP}$  strong whereas  $(p \mid \text{SKIP} \mid y)$  is balanced by making  $\mid \text{SKIP}$  strong and  $\mid y$  firm. The counterexample

$(p \mid \text{SKIP} \mid \text{SKIP}) + y$

illustrates that not both may be strong, for otherwise the **operator**  $+$  could not be identified.)

## 17.4.2 Semantics

a) The yield  $W$  of a **chooser-CHOICE-clause**  $C$ , in an environ  $E1$ , is determined as follows:

- let  $E2$  be the environ established {17<sub>3</sub>.2.2.b} around  $E1$  according to the **enquiry-clause** of  $C$ ;
- let  $V$  be the yield, in  $E2$ , of that **enquiry-clause**;
- $W$  is the yield of the scene "chosen" {b} by  $V$  from  $C$  in  $E2$ ; it is required that  $W$  be not newer in scope than  $E1$ .

b) The scene  $S$  "chosen" by a value  $V$  from a **MOID-chooser-CHOICE-clause**  $C$ , in an environ  $E2$ , is determined as follows: Case A: '**CHOICE**' is '**choice using boolean**' and  $V$  is true:

- $S$  is the constituent **in-CHOICE-clause** of  $C$ , in  $E2$ ;

Case B: '**CHOICE**' is '**choice using integral**' and  $1 \leq V \leq n$ , where  $n$  is the number of constituent **units** of the constituent **in-part-of-CHOICE** of  $C$ :

- $S$  is the  $V^{\text{th}}$  such **unit**, in  $E2$ ;

Case C: '**CHOICE**' is some '**choice using UNITED**' and  $V$  is acceptable to {16<sub>2</sub>.1.3.6.d} the '**MOID2**' of some constituent **MOID2-specification**  $D$  of  $C$  {; if there exists more than one such constituent **specification**, it is not defined which one is chosen as  $D$ };

- $S$  is the **unit** following that  $D$ , in an environ established {nonlocally {17<sub>3</sub>.2.2.b} } around  $E2$ , according to  $D$ , with  $V$ ;

Other Cases {when the **enquiry-clause** has been unsuccessful} :

If  $C$  contains a constituent **out-CHOICE-clause**  $O$ ,

then  $S$  is  $O$  in  $E2$ ;

otherwise,  $S$  is a **MOID-skip** in  $E2$ .



## 17.5 Loop clauses

{**Loop-clauses** are used for repeating dynamically one same sequence of instructions. The number of repetitions is controlled by a finite sequence of equidistant integers, by a condition to be tested each time, or by both.

Example 1:

```
INT fac := 1;
FOR i FROM n BY -1 TO 1
DO fac × := i OD.
```

Example 2:

```
INT a, b; read ((a, b)) PR ASSERT a ≥ 0 ∧ b > 0 PR;
INT q := 0, r := a;
WHILE r ≥ b PR ASSERT a = b × q + r ∧ 0 ≤ r PR
DO (q += 1, r -= b) OD
PR ASSERT a = b × q + r ∧ 0 ≤ r ∧ r < b PR
```

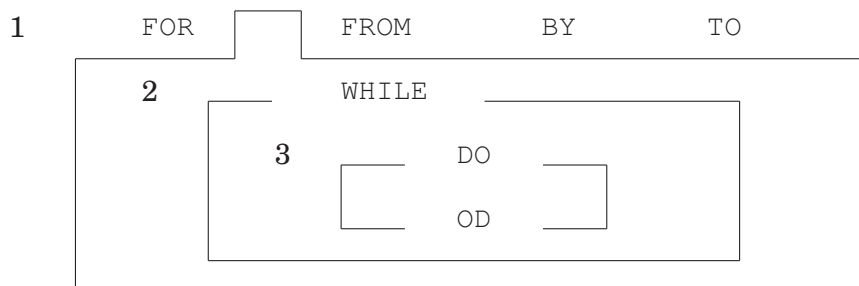
(see 24<sub>9</sub>.2 for an explanation of the **pragmats**).

The controlled **identifier**, e.g., *i* in Example 1, is defined over the **repeating-part**. Definitions introduced in the **while-part** are also valid over the **do-part**.

If the controlled **identifier** is not applied in the **repeating-part**, then the **for-part** may be omitted. A **from-part** FROM 1 may be omitted; similarly, BY 1 may be omitted. The **to-part** may be omitted if no test on the final value of the control-integer is required. A **while-part** WHILE TRUE may be omitted. For example,

```
FOR i FROM 1 BY 1 TO n WHILE TRUE DO print ("a") OD
may be written
TO n DO print ("a") OD.
```

The hierarchy of **ranges** is illustrated by:



}

## 17.5.1 Syntax

- A) **FROBYT :: from ; by ; to.**
- a) **strong void NEST1 loop clause** {19<sub>5</sub>.D, 19<sub>5</sub>.5.1.a} :  
**NEST1 STYLE for part defining new integral TAG2** {b},  
**NEST1 STYLE intervals** {c},  
**NEST1 STYLE repeating part with integral TAG2** {e}.
- b) **NEST1 STYLE for part defining new integral TAG2** {a} :  
**STYLE for** {24<sub>9</sub>.4.g, -} **token,**  
**integral NEST1 new integral TAG2 defining identifier with TAG2** {18<sub>4</sub>.8.a}  
**;**  
**where (TAG2) is (letter aleph), EMPTY.**
- c) **NEST1 STYLE intervals** {a} :  
**NEST1 STYLE from part** {d} **option,**  
**NEST1 STYLE by part** {d} **option,**  
**NEST1 STYLE to part** {d} **option.**
- d) **NEST1 STYLE FROBYT part** {c} :  
**STYLE FROBYT** {24<sub>9</sub>.4.g, -} **token, meek integral NEST1 unit** {17<sub>3</sub>.2.d} .
- e) **NEST1 STYLE repeating part with DEC2** {a} :  
**NEST1 new DEC2 STYLE while do part** {f};  
**NEST1 new DEC2 STYLE do part** {h}.
- f) **NEST2 STYLE while do part** {e} :  
**NEST2 STYLE while part defining LAYER3** {g},  
**NEST2 LAYER3 STYLE do part** {h}.
- g) **NEST2 STYLE while part defining LAYER3** {f} :  
**STYLE while** {24<sub>9</sub>.4.g, -} **token, boolean NEST2 enquiry clause defining**  
**LAYER3** {17<sub>3</sub>.4.c, -}.
- h) **NEST3 STYLE do part** {e, f} :  
**STYLE do** {24<sub>9</sub>.4.g, -} **token,**  
**strong void NEST3 serial clause defining LAYER4** {17<sub>3</sub>.2.a} ,  
**STYLE od** {24<sub>9</sub>4g, -} **token.**

{Examples:

- a) FOR i WHILE i <n DO task1 OD •  
 TO n DO task1; task2 OD
- b) FOR i

- c) FROM -5 TO +5
- d) FROM -5
- e) WHILE i <n DO task1 OD • DO task1; task2 OD
- f) WHILE i <n DO task 1; task2 OD
- g) WHILE i <n
- h) DO task1; task2 OD }

## 17.5.2 Semantics

A **loop-clause**  $C$ , in an environ  $E1$ , is elaborated in the following Steps:

Step 1: All the constituent **FROBYT-parts**, if any, of  $C$  are elaborated collaterally in  $E1$ ;

- let  $f$  be the yield of the constituent **from-part**, if any, of  $C$ , and be 1 otherwise;
- let  $b$  be the yield of the constituent **by-part**, if any, of  $C$ , and be 1 otherwise;
- let  $t$  be the yield of the constituent **to-part**, if any, of  $C$ , and be *absent* otherwise;
- let  $E2$  be the environ established {nonlocally {17<sub>3</sub>.2.2.b} } around  $E1$ , according to the **for-part-defining-new-integral-TAG2** of  $C$ , and with the integer  $f$ ;

Step 2: Let  $i$  be the integer accessed {16<sub>2</sub>.1.2.c} by '**integral TAG2**' inside the locale of  $E2$ ;

If  $t$  is not *absent*,

then

If  $b > 0$  and  $i > t$  or if  $b < 0$  and  $i < t$ ,

then  $C$  in  $E1$  {is completed and} yields empty;

{otherwise, Step 3 is taken; }

Step 3: Let an environ  $E3$  and a truth value  $w$  be determined as follows:

Case A:  $C$  does not contain a constituent **while-part**:

- $E3$  is  $E2$ ;
- $w$  is true;

Case B:  $C$  contains a constituent **while-part**  $P$ :

## ALGOL 68 REVISED REPORT

- $E3$  is established {perhaps nonlocally {17<sub>3</sub>.2.2.b} } around  $E2$  according to the **enquiry-clause** of  $P$ ;
- $w$  is the yield in  $E3$  of that **enquiry-clause**;

Step 4: If  $w$  is true, then

- the constituent **do-part** of  $C$  is elaborated in  $E3$ ;
- '**integral TAG2**' is made to access  $i + b$  inside the locale of  $E2$ ;<sup>1</sup>
- Step 2 is taken again; otherwise,
- $C$  in  $E1$  {is completed and} yields empty.

### {The **loop-clause**

FOR  $i$  FROM  $u1$  BY  $u2$  TO  $u3$  WHILE condition DO action OD  
is thus equivalent to the following **void-closed-clause**:

```
BEGIN INT f:= u1, INT b = u2, t = u3;
  step2:
    IF (b > 0 ∧ f ≤ t) ∨ (b < 0 ∧ f ≥ t) ∨ b = 0
      THEN INT i = f;
        IF condition
          THEN action; f += b; GO TO step2
        FI
      FI
  END.
```

This equivalence might not hold, of course, if the **loop-clause** contains **local-generators**, or if some of the **operators** above do not identify those in the standard environment {25<sub>10</sub>}.

---

<sup>1</sup>Note the possibility of overflow even in case TAG2 and intervals are empty.

# Chapter 18

## Declarations, declarers and indicators

{Declarations serve

- to announce new **indicators**, e.g., **identifiers**,
- to define their modes or priorities, and
- to ascribe values to those **indicators** and to initialize **variables**.}

### 18.1 Declarations

#### 18.1.1 Syntax

- A) **COMMON :: mode ; priority ; MODINE identity ;  
reference to MODINE variable ; MODINE operation ; PARAMETER ; MODE  
FIELDS.  
{MODINE :: MODE ; routine.}**
- a) **NEST declaration of DECS {a, 17<sub>3</sub>.2.b} :**  
**NEST COMMON declaration of DECS {18<sub>4</sub>.2.a, 18<sub>4</sub>.3.a, 18<sub>4</sub>.4.a, e, 18<sub>4</sub>.5.a, -};**  
**where (DECS) is (DECS1 DECS2),**  
**NEST COMMON declaration of DECS1 {18<sub>4</sub>.2.a, 18<sub>4</sub>.3.a, 18<sub>4</sub>.4.a, e, 18<sub>4</sub>.5.a,**  
**-},**  
**and also {94f} token, NEST declaration of DECS2 {a}.**

- b) **NEST COMMON joined definition of PROPS PROP**  
 {b, 18<sub>4</sub>.2.a, 18<sub>4</sub>.3.a, 18<sub>4</sub>.4.a, e, 18<sub>4</sub>.5.a, 18<sub>4</sub>.6.e, 19<sub>5</sub>.4.1.e} :  
**NEST COMMON joined definition of PROPS {b, c}, and also {94f} token,**  
**NEST COMMON joined definition of PROP {c}.**
- c) **NEST COMMON joined definition of PROP**  
 {b, 18<sub>4</sub>.2.a, 18<sub>4</sub>.3.a, 18<sub>4</sub>.4.a, e, 18<sub>4</sub>.5.a, 18<sub>4</sub>.6.e, 19<sub>5</sub>.4.1.e} :  
**NEST COMMON definition of PROP**  
 {18<sub>4</sub>.2.b, 18<sub>4</sub>.3.b, 18<sub>4</sub>.4.c, f, 18<sub>4</sub>.5.c, 18<sub>4</sub>.6.f, 19<sub>5</sub>.4.1.f, -}.
- d) **\*definition of PROP : NEST COMMON definition of PROP**  
 {18<sub>4</sub>.2.b, 18<sub>4</sub>.3.b, 18<sub>4</sub>.4.c, f, 18<sub>4</sub>.5.c, 18<sub>4</sub>.6.f, 19<sub>5</sub>.4.1.f} ;  
**NEST label definition of PROP {17<sub>3</sub>.2.c} .**

{Examples:

- a) `MODE R = REF REAL, S = CHAR • Prio V = 2, ^ = 3 • INT m = 4096 •`  
`REAL x, y •`  
`OP V = (BOOL a, b) BOOL : (a | TRUE | b)`
- b) `R = REF REAL, S = CHAR • V = 2, ^ = 3 • m = 4096 • x, y •`  
`V = (BOOL a, b) BOOL : (a | TRUE | b)`
- c) `R = REF REAL • V = 2 • m = 4096 • x •`  
`V = (BOOL a, b) BOOL : (a | TRUE | b) }`

## 18.1.2 Semantics

The elaboration of a **declaration** consists of the collateral elaboration of its **COMMON-declaration** and of its **declaration**, if any. {Thus, all the **COMMON-declarations** separated by **and-also-tokens** are elaborated collaterally.}

## 18.2 Mode declarations

{**Mode-declarations** provide the **defining-mode-indications**, which act as abbreviations for **declarers** constructed from the more primitive ones, or from other **declarers**, or even from themselves.

For example,

```
MODE ARRAY = [m, n] REAL, and
MODE BOOK = STRUCT (STRING text, REF BOOK next)
```

In the latter example, the **applied-mode-indication** BOOK is not only a convenient abbreviation, but is essential to the **declaration** .}

### 18.2.1 Syntax

- a) **NEST mode declaration of DECS** {18<sub>4</sub>.1.a} :  
**mode** {24<sub>9</sub>.4.d} token, **NEST mode joined definition of DECS** {18<sub>4</sub>.1.b, c} .
- b) **NEST mode definition of MOID TALLY TAB** {41c} :  
**where (TAB) is (bold TAG) or (NEST) is (new LAYER),**  
**MOID TALLY NEST defining mode indication with TAB** {18<sub>4</sub>.8.a} ,  
**is defined as** {24<sub>9</sub>.4.d} token,  
**actual MOID TALLY NEST declarer** {c}.
- c) **actual MOID TALLY1 NEST declarer** {b} :  
**where (TALLY1) is (i),**  
**actual MOID NEST declarator** {18<sub>4</sub>.6.c, d, g, h, o, s, -};  
**where (TALLY1) is (TALLY2 i),**  
**MOID TALLY2 NEST applied mode indication with TAB2** {18<sub>4</sub>.8.b} .

{Examples:

- a) MODE R = REF REAL, S = CHAR
- b) R = REF REAL
- c) REF REAL • CHAR }

{The use of "TALLY" excludes circular chains of **mode-definitions** such as MODE A = B,  
 B = A.

**Defining-mode-indications-with-SIZETY-STANDARD** may be declared only in the **standard-prelude**, where the nest is of the form "new LAYER" {25<sub>10</sub>.1.1.b}.

### 18.2.2 Semantics

The elaboration of a **mode-declaration** {involves no action, yields no value and} is completed.

## 18.3 Priority declarations

{**Priority-declarations** are used to specify the priority of **operators**. Priorities from 1 to 9 are available.

Since **monadic-operators** have effectively only one priority-level, which is higher than that of all **dyadic-operators**, **monadic-operators** do not require **priority-declarations**.)

### 18.3.1 Syntax

- a) **NEST priority declaration of DECS** {18<sub>4</sub>.1.a} :  
     **priority** {24<sub>9</sub>.4.d} **token**, **NEST priority joined definition of DECS** {18<sub>4</sub>.1.b, c}  
     .
- b) **NEST priority definition of priority PRIO TAD** {18<sub>4</sub>.1.c} :  
     **priority PRIO NEST defining operator with TAD** {18<sub>4</sub>.8.a} ,  
     **is defined as** {24<sub>9</sub>.4.d} **token**, **DIGIT** {24<sub>9</sub>.4.b} **token**,  
     **where DIGIT counts PRIO** {c, d}.  
     **DIGIT :: digit zero; digit one; digit two; digit three; digit four;**  
     **digit five; digit six; digit seven; digit eight; digit nine.**
- c) **WHETHER DIGIT1 counts PRIO i** {b, c} :  
     **WHETHER DIGIT2 counts PRIO** {c, d},  
     **where (digit one digit two digit three digit four digit five digit six**  
     **digit seven digit eight digit nine) contains (DIGIT2 DIGIT1).**
- d) **WHETHER digit one counts i** {b, c} : **WHETHER true.**

{Examples:

- a)  $\text{PRIO } \vee 2, \wedge = 3$
- b)  $\vee = 2$  }

### 18.3.2 Semantics

The elaboration of a **priority-declaration** {involves no action, yields no value and} is completed.

## 18.4 Identifier declarations

{**Identifier-declarations** provide **MODE-defining-identifiers**, by means of either **identity-definitions** or **variable-definitions**.



Examples:

```
REAL pi = 3.1416 • REAL scan := 0.05.
```

The latter example, which is a **variable-declaration**, may be considered as an equivalent form of the **identity-declaration**

```
REF REAL scan = LOC REAL := 0.05.
```

The elaboration of **identifier-declarations** causes values to be ascribed to their **identifiers**; in the examples given above, 3.1416 is ascribed to `pi` and a new local name which refers to 0.05 is ascribed to `scan`.)

## 18.4.1 Syntax

- A) **MODINE :: MODE ; routine.**
- B) **LEAP :: local ; heap ; primal.**
- a) **NEST MODINE identity declaration of DECS** {18<sub>4</sub>.1.a}  
**: formal MODINE NEST declarer** {b, 18<sub>4</sub>.6.b} ,  
**NEST MODINE identity joined definition of DECS** {18<sub>4</sub>.1.b, c} .
- b) **VICTAL routine NEST declarer** {a, 19<sub>5</sub>.2.3.b} :  
**procedure** {24<sub>9</sub>.4.d} **token.**
- c) **NEST MODINE identity definition of MODE TAG** {18<sub>4</sub>.1.c} :  
**MODE NEST defining identifier with TAG** {18<sub>4</sub>.8.a} ,  
**is defined as** {24<sub>9</sub>.4.d} **token,**  
**MODE NEST source for MODINE** {d}.
- d) **MODE NEST source for MODINE** {c, f, 18<sub>4</sub>.5.c} :  
**where (MODINE) is (MODE),**  
**MODE NEST source** {19<sub>5</sub>.2.1.c} ;  
**where (MODINE) is (routine),**  
**MODE NEST routine text** {19<sub>5</sub>.4.1.a, b, -}.
- e) **NEST reference to MODINE variable declaration of DECS** {18<sub>4</sub>.1.a} :  
**reference to MODINE NEST LEAP sample generator** {19<sub>5</sub>.2.3.b} ,  
**NEST reference to MODINE variable joined definition of DECS** {18<sub>4</sub>.1.b, c}  
 .
- f) **NEST reference to MODINE variable definition of reference to MODE TAG**  
 {18<sub>4</sub>.1.c} :  
**reference to MODE NEST defining identifier with TAG** {18<sub>4</sub>.8.a} ,  
**becomes** {24<sub>9</sub>.4.c} **token,**  
**MODE NEST source for MODINE** {d};

**where (MODINE) is (MODE),  
reference to MODE NEST defining identifier with TAG {18<sub>4</sub>.8.a} .**

- g) **\*identifier declaration :**  
**NEST MODINE identity declaration of DECS {a} ;**  
**NEST reference to MODINE variable declaration of DECS {e}.**

{Examples:

- a) INT m = 4096 • PROC r10 = REAL: random × 10  
b) PROC  
c) m = 4096  
d) 4096 • REAL: random × 10  
e) REAL x, y • PROC pp := REAL: random × 10  
f) pp := REAL: random × 10 • x }

## 18.4.2 Semantics

a) An **identity-declaration** *D* is elaborated as follows:

- the constituent **sources-for-MODINE** of *D* are elaborated collaterally;  
For each constituent **identity-definition** *D1* of *D*,

- the yield *V* of the **source-for-MODINE** of *D1* is ascribed {18<sub>4</sub>.8.2.a} to the **defining-identifier** of *D1*.

b) A **variable-declaration** *D* is elaborated as follows:

- the **sample-generator** {19<sub>5</sub>.2.3.1.b} *G* of *D* and all the **sources-for-MODINE**, if any, of the constituent **variable-definitions** of *D* are elaborated collaterally;

For each constituent **variable-definition-of-reference-to-MODE-TAG** *D1* of *D*,

- let *W1* be a "variant" {c}, for 'MODE', of the value referred to by the yield *N* of *G*;
- let *N1* be a newly created name equal in scope to *N* and referring to *W1*;
- if *N1* is a stowed name {16<sub>2</sub>.1.3.2.b}, then *N1* is endowed with subnames {16<sub>2</sub>.1.3.3.e, 16<sub>2</sub>.1.3.4.g};
- *N1* is ascribed {18<sub>4</sub>.8.2.a} to the **defining-identifier** of *D1*;
- the yield of the **source-for-MODINE**, if any, of *D1* is assigned {19<sub>5</sub>.2.1.2.b} to *N1*.

{An **actual-declarer** which is common to a number of **variable-definitions** is elaborated only once. For example, the elaboration of

```
INT m := 10; [1 : m += 1] INT p, q; print (m)
```

causes 11 to be printed, and not 12; moreover, two new local names referring to multiple values with descriptor ((1, 11)), and undefined elements, are ascribed to *p* and to *q*.)

c) A "variant" of a value *V*, for a mode *M*, is a value *W* acceptable to {16<sub>2</sub>.1.3.6.d} *M*, and determined as follows:

Case A: *M* is some '**structured with FIELDS mode**':

For each '**MODE field TAG**' enveloped by '**FIELDS**',

- the field selected by '**TAG**' in *W* is a variant, for '**MODE**', of the field selected by '**TAG**' in *V*;

Case B: *M* is some '**FLEXETY ROWS of MODE1**':

- the descriptor of *W* is that of *V*;
- each element of *W* is a variant, for '**MODE1**', of some element of *V*;

Other Cases:

- *W* is any value acceptable to *M*.

d) The yield of an **actual-routine-declarer** is some routine {whose mode is of no relevance}.

## 18.5 Operation declarations

{**Operation-declarations** provide **defining-operators**.

Example:

```
OP MC = (REAL a, b) REAL: (3 × a < b | a | b).
```

Unlike the case with, e.g., **identifier-declarations**, more than one **operation-declaration** involving the same **TAO-token** may occur in the same reach; e.g., the previous example may very well be in the same reach as

```
OP MC = (COMPL carthy, john) COMPL: (random < .5 | carthy | john);
```

the **operator** MC is then said to be "overloaded".}

## 18.5.1 Syntax

- A) PRAM :: DUO ; MONO.
- B) TAO :: TAD ; TAM.
- a) NEST MODINE operation declaration of DECS {18<sub>4</sub>.1.a} :  
     operator {24<sub>9</sub>.4.d} token, formal MODINE NEST plan {b, 18<sub>4</sub>.6.p, -},  
     NEST MODINE operation joined definition of DECS {18<sub>4</sub>.1.b, c} .
- b) formal routine NEST plan {a} : EMPTY.
- c) NEST MODINE operation definition of PRAM TAO {41c} :  
     PRAM NEST defining operator with TAO {18<sub>4</sub>.8.a} ,  
     is defined as {24<sub>9</sub>.4.d} token,  
     PRAM NEST source for MODINE {18<sub>4</sub>.4.d} .

{Examples:

- a) OP V = (BOOL a, b) BOOL: (a | TRUE | b)
- c) V = (BOOL a, b) BOOL: (a | TRUE | b) }

## 18.5.2 Semantics

- a) The elaboration of an **operation-declaration** consists of the collateral elaboration of its constituent **operation-definitions**.
- b) An **operation-definition** is elaborated by ascribing {18<sub>4</sub>.8.2.a} the routine yielded by its **source-for-MODINE** to its **defining-operator**.

## 18.6 Declarers

{**Declarers** specify modes. A **declarer** is either a **declarator**, which explicitly constructs a mode, or an **applied-mode-indication**, which stands for some **declarator** by way of a **mode-declaration**. Declarators are built from VOID, INT, REAL, BOOL and CHAR {25<sub>10</sub>.2.2}, with the assistance of other **symbols** such as REF, STRUCT, [ ], PROC, and UNION. For example, PROC (REAL)BOOL specifies the mode '**procedure with real parameter yielding boolean**'.

**Actual-declarers**, used typically in **generators**, require the presence of bounds. **Formal-declarers**, used typically in **formal-parameters** and **casts**, are without bounds. The

**declarer** following a `ref` is always **'virtual'**; it may then specify a **'flexible ROWS of MODE'**, because flexibility is a property of names. Since **actual-declarers** follow an implicit **'reference to'** in **generators**, they may also specify **'flexible ROWS of MODE'**.)

### 18.6.1 Syntax

- A) **VICTAL :: VIRACT ; formal.**
- B) **VIRACT :: virtual ; actual.**
- C) **MOIDS :: MOID ; MOIDS MOID.**
  - a) **VIRACT MOID NEST declarer** {c, e, g, h, 19<sub>5</sub>.2.3.a, b} :  
**VIRACT MOID NEST declarator** {c, d, g, h, o, s, -};  
**MOID TALLY NEST applied mode indication with TAB** {18<sub>4</sub>.8.b, -}.
  - b) **formal MOID NEST declarer** {e, h, p, r, u, 17<sub>3</sub>.4.k, 18<sub>4</sub>.4.a, 19<sub>5</sub>.4.1.a, b, e, 19<sub>5</sub>.5.1.a} :  
:
    - where MOID deflexes to MOID** {18<sub>4</sub>.7.a, b, c, -},
    - formal MOID NEST declarator** {c, d, h, o, s, -};
    - MOID1 TALLY NEST applied mode indication with TAB** {18<sub>4</sub>.8.b, -},
    - where MOID1 deflexes to MOID** {18<sub>4</sub>.7.a, b, c, -}.
  - c) **VICTAL reference to MODE NEST declarator** {a, b, 18<sub>4</sub>.2.c} :  
**reference to {94d} token, virtual MODE NEST declarer {a}.**
  - d) **VICTAL structured with FIELDS mode NEST declarator** {a, b, 18<sub>4</sub>.2.c} :  
**structure {24<sub>9</sub>.4.d} token, VICTAL FIELDS NEST portrayer of FIELDS {e}**  
**brief pack.**
  - e) **VICTAL FIELDS NEST portrayer of FIELDS1** {d, e} :  
**VICTAL MODE NEST declarer** {a, b} ,  
**NEST MODE FIELDS joined definition of FIELDS1** {18<sub>4</sub>.1.b, c} ;  
**where (FIELDS1) is (FIELDS2 FIELDS3),**  
**VICTAL MODE NEST declarer** {a, b} ,  
**NEST MODE FIELDS joined definition of FIELDS2** {18<sub>4</sub>.1.b, c} ,  
**and also {24<sub>9</sub>.4.f} token,**  
**VICTAL FIELDS NEST portrayer of FIELDS3** {e}.
  - f) **NEST MODE FIELDS definition of MODE field TAG** {18<sub>4</sub>.1.c} :  
**MODE field FIELDS defining field selector with TAG** {18<sub>4</sub>.8.c} .
  - g) **VIRACT flexible ROWS of MODE NEST declarator** {a, 42c} :  
**flexible {24<sub>9</sub>.4.d} token, VIRACT ROWS of MODE NEST declarer {a}.**

- h) **VICTAL ROWS of MODE NEST declarator** {a, b, 18<sub>4</sub>.2.c} :  
**VICTAL ROWS NEST rower** {i, j, k, l} **STYLE bracket,**  
**VICTAL MODE NEST declarer** {a, b}.
- i) **VICTAL row ROWS NEST rower** {h, i} :  
**VICTAL row NEST rower** {j, k, l}, **and also** {94f} **token,**  
**VICTAL ROWS NEST rower** {i, j, k, l}.
- j) **actual row NEST rower** {h, i} :  
**NEST lower bound** {m}, **up to** {24<sub>9</sub>.4.f} **token, NEST upper bound** {n};  
**NEST upper bound** {n}.
- k) **virtual row NEST rower** {h, i}: **up to** {24<sub>9</sub>.4.f} **token option.**
- l) **formal row NEST rower** {h, i}: **up to** {24<sub>9</sub>.4.f} **token option.**
- m) **NEST lower bound** {j, 19<sub>5</sub>.3.2.f, g} : **meek integral NEST unit** {17<sub>3</sub>.2.d} .
- n) **NEST upper bound** {j, 19<sub>5</sub>.3.2.f} : **meek integral NEST unit** {17<sub>3</sub>.2.d} .
- o) **VICTAL PROCEDURE NEST declarator** {a, b, 18<sub>4</sub>.2.c} :  
**procedure** {24<sub>9</sub>.4.d} **token, formal PROCEDURE NEST plan** {p}.
- p) **formal procedure PARAMETY yielding MOID NEST plan** {o, 18<sub>4</sub>.5.a} :  
**where (PARAMETY) is (EMPTY),**  
**formal MOID NEST declarer** {b};  
**where (PARAMETY) is (with PARAMETERS),**  
**PARAMETERS NEST joined declarer** {q, r}  
**brief pack, formal MOID NEST declarer** {b}.
- q) **PARAMETERS PARAMETER NEST joined declarer** {p, q} :  
**PARAMETERS NEST joined declarer** {q, r}, **and also** {94f} **token,**  
**PARAMETER NEST joined declarer** {r}.
- r) **MODE parameter NEST joined declarer** {p, q} : **formal MODE NEST declarer**  
{b}.
- s) **VICTAL union of MOODS1 MOOD1 mode NEST declarator** {a, b, 18<sub>4</sub>.2.c} :  
**unless EMPTY with MOODS1 MOOD1 incestuous** {18<sub>4</sub>.7.f} ,  
**union of** {24<sub>9</sub>.4.d} **token,**  
**MOIDS NEST joined declarer** {t, u} **brief pack,**  
**where MOIDS ravel to MOODS2** {18<sub>4</sub>.7.g}  
**and safe MOODS1 MOOD1 subset of safe MOODS2** {22<sub>7</sub>.3.1}  
**and safe MOODS2 subset of safe MOODS1 MOOD1** {22<sub>7</sub>.3.1, m} .
- t) **MOIDS MOID NEST joined declarer** {s, t} :  
**MOIDS NEST joined declarer** {t, u}, **and also** {24<sub>9</sub>.4.f} **token,**  
**MOID NEST joined declarer** {u}.

## DECLARATIONS, DECLARERS AND INDICATORS

u) **MOID NEST joined declarer** {s, t} : **formal MOID NEST declarer** {b}.

{Examples:

- a) [1 : n] REAL • PERSON
- b) [] REAL • STRING
- c) REF REAL
- d) STRUCT (INT age, REF PERSON father, son)
- e) REF PERSON father, son • INT age, REF PERSON father, son
- f) age
- g) FLEX [1 : n] REAL
- h) [1 : m, 1 : n] REAL
- i) 1 : m, 1 : n
- j) 1 : n
- k) :
- l) :
- m) 1
- n) n
- o) PROC (BOOL, BOOL) BOOL
- p) (BOOL, BOOL) BOOL q) BOOL, BOOL
- r) BOOL s) UNION (INT, CHAR)
- t) INT, CHAR
- u) INT }

{For **actual-MOID-TALLY-declarers**, see 18<sub>4</sub>.2.1.c ; for **actual-routine-declarers**, see 18<sub>4</sub>.4.1.b . There are no **declarers** specifying modes such as '**union of integral union of integral real mode mode**' or '**union of integral real integral mode**'. The **declarers** UNION (INT, UNION (INT, REAL)) and UNION (INT, REAL, INT) may indeed be written, but in both cases the mode specified is '**union of integral real mode**' (which can as well be spelled '**union of real integral mode**').}

## 18.6.2 Semantics

a) The yield  $W$  of an **actual-MODE-declarer**  $D$ , in an environ  $E$ , is determined as follows:  
If **'MODE'** is some **'STOWED'**, then

- let  $D1$  in  $E1$  be "developed" {c} from  $D$  in  $E$ ;
- $W$  is the yield of {the **declarator**}  $D1$  in an environ established {locally, see 17<sub>3</sub>.2.2.b} upon  $E$  and around  $E1$ ;

otherwise,

- $W$  is any value {acceptable to **'MODE'**} .

b) The yield  $W$  of an **actual-STOWED-declarator**  $D$  is determined as follows:

Case A: **'STOWED'** is some **'structured with FIELDS mode'**:

- the constituent **declarers** of  $D$  are elaborated collaterally;
  - each field of  $W$  is a variant {18<sub>4</sub>.4.2.c}
- (i) of the yield of the last constituent **MODE-declarer** of  $D$  occurring before the constituent **defining-field-selector** of  $D$  selecting {16<sub>2</sub>.1.5.g} that field,
- (ii) for that **'MODE'**;

Case B: **'STOWED'** is some **'ROWS of MODE'**:

- all the constituent **lower-bounds** and **upper-bounds** of  $D$  and the **declarer**  $D1$  of  $D$  are elaborated collaterally;  
For  $i = 1 \dots n$ , where  $n$  is the number of **'row'**'s contained in **'ROWS'**,
  - let  $l_i$  be the yield of the **lower-bound**, if any, of the  $i^{th}$  constituent **row-rower** of  $D$ , and be 1 otherwise;
  - let  $u_i$  be the yield of the **upper-bound** of that **row-rower**;
  - $W$  is composed of
- (i) a descriptor  $((l_1, u_1), \dots, (l_n, u_n))$ ,
- (ii) variants of the yield of  $D1$ , for **'MODE'**;

Case C: **'STOWED'** is some **'flexible ROWS of MODE'**:



- $W$  is the yield of the **declarer** of  $D$ .

c) The scene  $S$  "developed from" an **actual-STOWED-declarer**  $D$  in an environ  $E$  is determined as follows:

If the visible direct descendent  $D1$  of  $D$  is a **mode-indication**, then

- $S$  is the scene developed from that yielded by  $D1$  in  $E$ ;

otherwise { $D1$  is a **declarator**},

- $S$  is composed of  $D1$  and  $E$ .

d) A given **MOID-declarer** "specifies" the mode '**MOID**'.

## 18.7 Relationships between modes

{Some modes must be deflexed because the mode of a value may not be flexible {16<sub>2</sub>.1.3.6.b}. Incestuous unions must be prevented in order to avoid ambiguities. A set of '**UNITED**'s and '**MOODS**'s may be unravelled by replacing all those '**UNITED**'s by their component '**MOODS**'s.}

### 18.7.1 Syntax

A) **NONSTOWED :: PLAIN ; REF to MODE ; PROCEDURE ; UNITED ; void.**

B) **MOODSETY :: MOODS ; EMPTY.**

C) **MOIDSETY :: MOIDS ; EMPTY.**

a) **WHETHER NONSTOWED deflexes to NONSTOWED**

{b, e, 18<sub>4</sub>.6.b, 19<sub>5</sub>.2.1.c, 21<sub>6</sub>.2.a, 22<sub>7</sub>.1.n} :

**WHETHER true.**

b) **WHETHER FLEXETY ROWS of MODE1 deflexes to ROWS of MODE2**

{b, e, 18<sub>4</sub>.6.b, 19<sub>5</sub>.2.1.c, 21<sub>6</sub>.2.a, 22<sub>7</sub>.1.n} :

**WHETHER MODE1 deflexes to MODE2** {a, b, c, -}.

c) **WHETHER structured with FIELDS1 mode deflexes to structured with FIELDS2 mode**

{b, e, 18<sub>4</sub>.6.b, 19<sub>5</sub>.2.1.c, 21<sub>6</sub>.2.a, 22<sub>7</sub>.1.n} :

**WHETHER FIELDS1 deflexes to FIELDS2** {d, e, -}.

- d) **WHETHER FIELDS1 FIELD1 deflexes to FIELDS2 FIELD2 {c, d} :**  
**WHETHER FIELDS1 deflexes to FIELDS2 {d, e, -}**  
**and FIELD1 deflexes to FIELD2 {e, -}.**
- e) **WHETHER MODE1 field TAG deflexes to MODE2 field TAG {c, d} :**  
**WHETHER MODE1 deflexes to MODE2 {a, b, c, -}.**
- f) **WHETHER MOODSETY1 with MOODSETY2 incestuous {f, 18<sub>4</sub>.6.s} :**  
**where (MOODSETY2) is (MOOD MOODSETY3),**  
**WHETHER MOODSETY1 MOOD with MOODSETY3 incestuous {f}**  
**or MOOD is firm union of MOODSETY1 MOODSETY3 mode {22<sub>7</sub>.1.m} ;**  
**where (MOODSETY2) is (EMPTY),**  
**WHETHER false.**
- g) **WHETHER MOIDS ravels to MOODS {g, 18<sub>4</sub>.6.s} :**  
**where (MOIDS) is (MOODS),**  
**WHETHER true ;**  
**where (MOIDS) is (MOODSETY union of MOODS1 mode MOIDSETY),**  
**WHETHER MOODSETY MOODS1 MOIDSETY ravels to MOODS {g}.**

{A component mode of a union may not be firmly coerced to one of the other component modes or to the union of those others (rule f) for, otherwise, ambiguities could arise. For example,

```
UNION (REF INT, INT) (LOC INT)
```

is ambiguous in that dereferencing may or may not occur before the uniting. Similarly,

```
MODE SZP = UNION (SZEREDI, PETER) ;
```

```
UNION (REF SZP, SZP) (LOC SZP)
```

is ambiguous. Note that, because of ravelling (rule g), the mode specified by the **declarer** of the **cast** is more closely suggested by UNION (REF SZP, SZEREDI, PETER).}

## 18.8 Indicators and field selectors

### 18.8.1 Syntax

- A) **INDICATOR :: identifier ; mode indication ; operator.**
- B) **DEFIED :: defining ; applied.**
- C) **PROPSETY :: PROPS ; EMPTY.**
- D) **PROPS :: PROP ; PROPS PROP.**
- E) **PROP :: DEC ; LAB ; FIELD.**

- F) **QUALITY :: MODE ; MOID TALLY ; DYADIC ; label ; MODE field.**
- G) **TAX :: TAG ; TAB ; TAD ; TAM.**
- a) **QUALITY NEST new PROPSETY1 QUALITY TAX PROPSETY2 defining INDICATOR with TAX**  
 {17<sub>3</sub>.2.c, 17<sub>3</sub>.5.b, 18<sub>4</sub>.2.b, 18<sub>4</sub>.3.b, 18<sub>4</sub>.4.c, f, 18<sub>4</sub>.5.c, 19<sub>5</sub>.4.1.f} :  
**where QUALITY TAX independent PROPSETY1 PROPSETY2** {22<sub>7</sub>.1.a, b, c},  
**TAX** {24<sub>9</sub>.4.2.A, D, F, K} **token.**
- b) **QUALITY NEST applied INDICATOR with TAX**  
 {18<sub>4</sub>.2.c, 18<sub>4</sub>.6.a, b, 19<sub>5</sub>.D, 19<sub>5</sub>.4.2.a, b, 19<sub>5</sub>.4.4.a} :  
**where QUALITY TAX identified in NEST** {72a},  
**TAX** {24<sub>9</sub>.4.2.A, D, F, K} **token.**
- c) **MODE field PROPSETY1 MODE field TAG PROPSETY2 defining field selector with TAG** {18<sub>4</sub>.6.f} :  
**where MODE field TAG independent PROPSETY1 PROPSETY2** {22<sub>7</sub>.1.a, b, c},  
**TAG** {24<sub>9</sub>.4.2.A} **token.**
- d) **MODE field FIELDS applied field selector with TAG** {19<sub>5</sub>.3.1.a} :  
**where MODE field TAG resides in FIELDS** {22<sub>7</sub>.2.b, c, -},  
**TAG** {24<sub>9</sub>.4.2.A} **token.**
- e) **\*QUALITY NEST DEFIED indicator with TAX :**  
**QUALITY NEST DEFIED INDICATOR with TAX** {a, b}.
- f) **\*MODE DEFIED field selector with TAG :**  
**MODE field FIELDS DEFIED field selector with TAG** {c, d}.

{Examples:

- a) x (in REAL x, y)
- b) x (in x + y)
- c) next (see 15<sub>1</sub>.1.2)
- d) next (in next OF draft)}

## 18.8.2 Semantics

- a) When a value or a scene *V* is "ascribed" to a **QUALITY-defining-indicator-with-TAX**, in an environ *E*, then '**QUALITY TAX**' is made to access *V* inside the locale of *E* {16<sub>2</sub>.1.2.c}.

b) The yield  $W$  of a **QUALITY-applied-indicator-with-TAX**  $I$  in an environ  $E$  composed of an environ  $E1$  and a locale  $L$  is determined as follows:

If  $L$  corresponds to a '**DECSETY LABSETY**' which envelops {15<sub>1</sub>.1.4.1.c} that '**QUALITY TAX**',  
then

$W$  is the value or scene, if any, accessed inside  $L$  by '**QUALITY TAX**' and, otherwise, is undefined;

otherwise,  $W$  is the yield of  $I$  in  $E1$ .

{Consider the following **closed-clause**, which contains another one:

```
BEGIN CO range 1 CO
  INT i = 421, INT a := 5, PROC p = VOID: print (a);
  BEGIN CO range 2 CO
    REAL a; a := i; p
  END
END.
```

By the time  $a := i$  is encountered during the elaboration, two new environs have been created, one for each **range**. The **defining-identifier**  $i$  is first sought in the newer one,  $E2$ , is not found there, and then is sought and found in the older one,  $E1$ . The locale of  $E1$  corresponds to '**integral letter i reference to integral letter a procedure yielding void letter p**'. The yield of the **applied-identifier**  $i$  is therefore the value 421 which has been ascribed {a} to '**integral letter i**' inside the locale of  $E1$ . The yield of  $a$ , in  $a := i$ , however, is found from the locale of  $E2$ .

When  $p$  is called {19<sub>5</sub>.4.3.2.b}, its **unit** is elaborated in an environ  $E3$  established around  $E1$  but upon  $E2$  {17<sub>3</sub>.2.2.b}. This means that, for scope purposes,  $E3$  is newer than  $E2$ , but the component environ of  $E3$  is  $E1$ . When  $a$  comes to be printed, it is the yield 5 of the **reference-to-integral-identifier**  $a$  declared in the outer **range** that is obtained.

Thus, the meaning of an **indicator** applied but not defined within a routine is determined by the context in which the routine was created, rather than that in which it is called.)

# Chapter 19

## Units

{**Units** are used to program the more primitive actions or to put into one single piece the larger constructs of Chapter 17<sub>3</sub>.

**NOTION-coercees** are the results of coercion (Chapter 21<sub>6</sub>), but **hips** are not; in the case of **ENCLOSED-clauses**, any coercions needed are performed inside them.

The syntax below implies, for example, that text OF draft + "the\_end" is parsed as (text OF draft) + "the\_end" since a **selection** is a '**SECONDARY**' whereas a **formula** is a '**TERTIARY**'.

### 19.1 Syntax

- A) **UNIT** {17<sub>3</sub>.2.d} ::
  - assignation** {19<sub>5</sub>.2.1.a} **coercee** ;
  - identity relation** {19<sub>5</sub>.2.2.a} **coercee** ;
  - routine text** {19<sub>5</sub>.4.1.a, b} **coercee** ;
  - jump** {19<sub>5</sub>.4.4.a} ;
  - skip** {19<sub>5</sub>.5.2.a} ;
  - TERTIARY** {B}.
  
- B) **TERTIARY** {A, 19<sub>5</sub>.2.1.b, 19<sub>5</sub>.2.2.a} ::
  - ADIC formula** {19<sub>5</sub>.4.2.a, b} **coercee** ;
  - nihil** {19<sub>5</sub>.2.4.a} ;
  - SECONDARY** {C}.
  
- C) **SECONDARY** {B, 19<sub>5</sub>.3.1.a, 19<sub>5</sub>.4.2.c} ::
  - LEAP generator** {19<sub>5</sub>.2.3.a} **coercee** ;

**selection** {19<sub>5</sub>.3.1.a} **coercee** ;  
**PRIMARY** {D}.

D) **PRIMARY** {C, 19<sub>5</sub>.3.2.a, 19<sub>5</sub>.4.3.a} ::

**slice** {19<sub>5</sub>.3.2.a} **coercee** ;  
**call** {19<sub>5</sub>.4.3.a} **coercee** ;  
**cast** {19<sub>5</sub>.5.1.a} **coercee** ;  
**denoter** {23<sub>8</sub>.0.a} **coercee** ;  
**format text** {25<sub>10</sub>.3.4.1.a} **coercee** ;  
**applied identifier with TAG** {18<sub>4</sub>.8.b} **coercee** ;  
**ENCLOSED clause** {17<sub>3</sub>.1.a, 17<sub>3</sub>.3.a, c, d, e, 17<sub>3</sub>.4.a, 17<sub>3</sub>.5.a} .

{The hyper-rules for '**SORT MOID FORM coercee**' are given in 21<sub>6</sub>.1.1.a, b, c, d and e, the entry rules of the coercion syntax. When the coercion syntax is invoked for some '**SORT MOID FORM coercee**', it will eventually return to a rule in this chapter for some '**MOID1 FORM**' (blind alleys apart). It is the cross-reference to that rule that is given in the metaproduction rules above. No other visible descendent has been produced in the meantime; the coercion syntax merely transforms '**MOID**' into '**MOID1**' for semantical purposes.}

a) \***SOME hip** : **SOME jump** {19<sub>5</sub>.4.4.a} ; **SOME skip** {19<sub>5</sub>.5.2.a} ; **SOME nihil** {19<sub>5</sub>.2.4.a} .

{The mode of a **hip** is always that required, a posteriori, by its context, and its yield is acceptable to that mode. Since any mode is so easily accommodated, no coercion is permitted.}

## 19.2 Units associated with names

{Names may be assigned to {19<sub>5</sub>.2.1}, compared with other names {19<sub>5</sub>.2.2} and created {19<sub>5</sub>.2.3}.}

### 19.2.1 Assignations

{In **assignations**, a value is "assigned", to a name. E.g., in  $x := 3.14$ , the real number yielded by the **source** 3.14 is assigned to the name yielded by the **destination**  $x$ .}

#### 19.2.1.1 Syntax

- a) **REF to MODE NEST assignment** {19<sub>5</sub>.A} :  
**REF to MODE NEST destination** {b},  
**becomes** {24<sub>9</sub>.4.c} **token**,  
**MODE NEST source** {c}.
- b) **REF to MODE NEST destination** {a} :  
**soft REF to MODE NEST TERTIARY** {19<sub>5</sub>.B} .
- c) **MODE1 NEST source** {a, 18<sub>4</sub>.4.d} :  
**strong MODE2 NEST unit** {17<sub>3</sub>.2.d} ,  
**where MODE1 deflexes to MODE2** {18<sub>4</sub>.7.a, b, c, -}.

{Examples:

- a)  $x := 3.14$
- b)  $x$
- c)  $3.14$  }

### 19.2.1.2 Semantics

a) An **assignment**  $A$  is elaborated as follows:

- let  $N$  and  $W$  be the {collateral} yields {a name and another value} of the **destination** and **source** of  $A$ ;
- $W$  is assigned to {b} $N$ ;
- the yield of  $A$  is  $N$ .

b) A value  $W$  is "assigned to" a name  $N$ , whose mode is some '**REF to MODE**', as follows:  
It is required that

- $N$  be not nil, and that
- $W$  be not newer in scope than  $N$ ;

Case A: '**MODE**' is some '**structured with FIELDS mode**':

- For each '**TAG**' selecting a field in  $W$ ,
- that field is assigned to the subname selected by '**TAG**' in  $N$ ;

Case B: **'MODE'** is some **'ROWS of MODE1'**:

- let  $V$  be the {old} value referred to by  $N$ ;
- it is required that the descriptors of  $W$  and  $V$  be identical;
- For each index  $I$  selecting an element in  $W$ ,
- that element is assigned to the subname selected by  $I$  in  $N$ ;

Case C: **'MODE'** is some **'flexible ROWS of MODE1'**:

- let  $V$  be the {old} value referred to by  $N$ ;
- $N$  is made to refer to a multiple value composed of
  - (i) the descriptor of  $W$ ,
  - (ii) variants {18<sub>4</sub>.4.2.c} of some element {possibly a ghost element} of  $V$ ;
- $N$  is endowed with subnames {16<sub>2</sub>.1.3.4.g};
  - For each index  $I$  selecting an element in  $W$ ,
  - that element is assigned to the subname selected by  $I$  in  $N$ ;

Other Cases (e.g., where **'MODE'** is some **'PLAIN'** or some **'UNITED'**) :

- $N$  is made to refer {16<sub>2</sub>.1.3.2.a} to  $W$ .

{Observe how, given

```
FLEX [1: 0] [1: 3] INT flexfix,
the presence of the ghost element {162.1.3.4.c} ensures that the meaning of
flexfix := LOC [1 : 1] [1 : 3] INT
is well defined, but that of
flexfix := LOC [1 : 1] [1 : 4] INT
is not, since the bound pairs of the second dimension are different.}
```

## 19.2.2 Identity relations

{**Identity-relations** may be used to ask whether two names of the same mode are the same.

E.g., after the **assignment** `draft := ("abc", NIL)`, the **identity-relation** `next OF draft :=: REF BOOK (NIL)` yields `true`. However, `next OF draft :=: NIL` yields `false` because it is equivalent to `next OF draft :=: REF REF BOOK (NIL)`; the yield of `next OF draft`, without any coercion, is the name referring to the second field of the structured value referred to by the value of `draft` and, hence, is not `nil`.)



### 19.2.2.1 Syntax

- a) **boolean NEST identity relation** {19<sub>5</sub>.A} :  
     **where** soft balances **SORT1** and **SORT2** {17<sub>3</sub>.2.f} ,  
     **SORT1** reference to **MODE NEST TERTIARY1** {19<sub>5</sub>.B} ,  
     **identity relator** {b},  
     **SORT2** reference to **MODE NEST TERTIARY2** {5B} .
- b) **identity relator {a} : is** {24<sub>9</sub>.4.f} **token ; is not** {24<sub>9</sub>.4.f} **token.**

{Examples:

- a) next OF draft ::= REF BOOK (NIL)
- b) ::= • :≠: }

{Observe that `a1[i] ::= a1[j]` is not produced by this syntax. The comparison, by an **identity-relation**, of transient names {16<sub>2</sub>.1.3.6.c} is thus prevented.}

### 19.2.2.2 Semantics

The yield  $W$  of an **identity-relation**  $I$  is determined as follows:

- let  $N1$  and  $N2$  be the {collateral} yields of the **"TERTIARY"**s of  $I$ ;

Case A: The **token** of the **identity-relator** of  $I$  is an **is-token**:

- $W$  is true if {the name}  $N1$  is the same as  $N2$ , and is false otherwise;

Case B: The **token** of the **identity-relator** of  $I$  is an **is-not-token**:

- $W$  is true if  $N1$  is not the same as  $N2$ , and is false, otherwise.

### 19.2.3 Generators

{And as imagination bodies forth  
 The forms of things unknown, the poet's pen  
 Turns them to shapes, and gives to airy nothing  
 A local habitation and a name.  
 A Midsummer-night's Dream,                      William Shakespeare. }

{The elaboration of a **generator**, e.g., `LOC REAL in xx := LOC REAL := 3.14`, or of a **sample-generator**, e.g., `[1 : n] CHAR in [1 : n] CHAR u, v;`, involves the creation of a name, i.e., the reservation of storage.

The use of a **local-generator** implies (with most implementations) the reservation of storage on a run-time stack, whereas **heap-generators** imply the reservation of storage in another region, termed the "heap", in which garbage-collection techniques may be used for storage retrieval. Since this is less efficient, **local-generators** are preferable; this is why only LOC may be omitted from **sample-generators** of **variable-declarations**.)

### 19.2.3.1 Syntax

{LEAP :: local ; heap primal.}

- a) **reference to MODE NEST LEAP generator** {19<sub>5</sub>.C} :  
     **LEAP** {24<sub>9</sub>.4.d, -} **token, actual MODE NEST declarer** {18<sub>4</sub>.6.a} .
- b) **reference to MODINE NEST LEAP sample generator** {18<sub>4</sub>.4.e} :  
     **LEAP** {24<sub>9</sub>.4.d, -} **token, actual MODINE NEST declarer** {18<sub>4</sub>.4.b, 18<sub>4</sub>.6.a} ;  
     **where (LEAP) is (local),**  
     **actual MODINE NEST declarer** {18<sub>4</sub>.4.b, 18<sub>4</sub>.6.a} .

{Examples:

- a) LOC REAL
- b) LOC REAL • REAL }

{There is no representation for the **primal-symbol** (see 24<sub>9</sub>.4.a).}

### 19.2.3.2 Semantics

a) The yield  $W$  of a **LEAP-generator** or **LEAP-sample-generator**  $G$ , in an environ  $E$ , is determined as follows:

- $W$  is a newly created name which is made to refer {16<sub>2</sub>.1.3.2.a} to the yield in  $E$  of the **actual-declarer** {18<sub>4</sub>.4.2.d, 18<sub>4</sub>.6.2.a} of  $G$ ;
- $W$  is equal in scope to the environ  $E_1$  determined as follows:

Case A: '**LEAP**' is '**local**':

- $E_1$  is the "local environ" {b} accessible from  $E$ ;

Case B: '**LEAP**' is '**heap**':

- $E_1$  is {the first environ created during the elaboration of the **particular-program**, which is} such that

- (i) the primal environ {16<sub>2</sub>.2.2.a} is the environ of the environ of the environ of the environ of  $E1$  {sic}, and
- (ii)  $E1$  is, or is older than,  $E$ ;

Case C: '**LEAP**' is '**primal**':

- $E1$  is the primal environ;
- if  $W$  is a stowed name {16<sub>2</sub>.1.3.2.b}, then  $W$  is endowed with subnames {16<sub>2</sub>.1.3.3.e, 16<sub>2</sub>.1.3.4.g}.

{The only examples of **primal-generators** occur in the **standard-** and **system-preludes** {25<sub>10</sub>.3.1.1.h, 25<sub>10</sub>.3.1.4.b, n, o, 25<sub>10</sub>.4.1.a}.

When  $G$  is a **reference-to-routine-sample-generator**, the mode of  $W$  is of no relevance.}

b) The "local environ" accessible from an environ  $E$  is an environ  $E1$  determined as follows:

If  $E$  is "nonlocal" {17<sub>3</sub>.2.2.b},

then  $E1$  is the local environ accessible from the environ of  $E$ ;

otherwise,  $E1$  is  $E$ .

{An environ is nonlocal if it has been established according to a **serial-clause** or **enquiry-clause** which contains no constituent **mode-**, **identifier-**, or **operation-declaration**, or according to a **for-part** {17<sub>3</sub>.5.1.b} or a **specification** {17<sub>3</sub>.4.1.j, k}.}

## 19.2.4 Nihils

### 19.2.4.1 Syntax

- a) **strong reference to MODE NEST nihil** {5B} : **nil** {94f} **token**.

{Example:

- a) NIL }

### 19.2.4.2 Semantics

The yield of a **nihil** is a nil name.

## 19.3 Units associated with stowed values

{In Flanders fields the poppies blow  
Between the crosses, row on row, ...  
In Flanders Fields,                    John McCrae. }

{The fields of structured values may be obtained by **selections** {19<sub>5</sub>.3.1} and the elements of multiple values by **slices** {19<sub>5</sub>.3.2}; the corresponding effects on stowed names are defined also.}

### 19.3.1 Selections

{A **selection** selects a field from a structured value or (if it is a "multiple selection") it selects a multiple value from a multiple value whose elements are structured values. For example, **re OF z** selects the first real field (usually termed the real part) of the yield of *z*. If *z* yields a name, then **re OF z** also yields a name, but if *g* yields a complex value, then **re OF g** yields a real value, not a name referring to one.}

#### 19.3.1.1 Syntax

- A) **REFETY :: REF to ; EMPTY.**
- B) **REFLEXETY :: REF to ; REF to flexible ; EMPTY.**  
{REF :: reference ; transient reference.}
- a) **REFETY MODE1 NEST selection** {19<sub>5</sub>.C} :  
  - MODE1 field FIELDS applied field selector with TAG** {18<sub>4</sub>.8.d} ,
  - of** {24<sub>9</sub>.4.f} **token,**
  - weak REFETY structured with FIELDS mode NEST SECONDARY** {19<sub>5</sub>.C}
- ;
- where (MODE1) is (ROWS of MODE2),**  
  - MODE2 field FIELDS applied field selector with TAG** {18<sub>4</sub>.8.d} ,
  - of** {94f} **token,**
  - weak REFLEXETY ROWS of structured with FIELDS mode**  
**NEST SECONDARY** {19<sub>5</sub>.C} ,
  - where (REFETY) is derived from (REFLEXETY) {b, c, -}.**
- b) **WHETHER (transient reference to) is derived from (REF to flexible)**  
{a, 19<sub>5</sub>.3.2.a, 21<sub>6</sub>.6.a} :  
**WHETHER true.**
- c) **WHETHER (REFETY) is derived from (REFETY) {a, 19<sub>5</sub>.3.2.a, 21<sub>6</sub>.6.a} :**  
**WHETHER true.**

{Examples:

a) re OF z • re OF z1 }

{The mode of re OF z begins with '**reference to**' because that of z does.

Example:

```
INT age := 7; STRUCT (BOOL sex, INT age) jill;
age OF jill := age;
```

Note that the **destination** age OF jill yields a name because jill yields one. After the **identity-declaration**

```
STRUCT (BOOL sex, INT age) jack = (TRUE, 9),
age OF jack cannot be assigned to since jack is not a variable.)
```

### 19.3.1.2 Semantics

The yield  $W$  of a **selection**  $S$  is determined as follows:

- let  $V$  be the yield of the **SECONDARY** of  $S$ ;
- it is required that  $V$  (if it is a name) be not nil;
- $W$  is the value selected in {16<sub>2</sub>.1.3.3.a, e, 16<sub>2</sub>.1.3.4.k} or the name generated from {16<sub>2</sub>.1.3.4.l}  $V$  by the **field-selector** of  $S$ .

{A **selection** in a name referring to a structured value yields an existing subname {16<sub>2</sub>.1.3.3.e} of that name. The name generated from a name referring to a multiple value, by way of a selection with a **ROWS-of-MODE-SECONDARY** (as in re OF z1), is a name which may or may not be newly created for the purpose.}

## 19.3.2 Slices

{**Slices** are obtained by subscripting, e.g.,  $x1[i]$ , by trimming, e.g.,  $x1[2 : n]$  or by both, e.g.,  $x2[j : n, j]$  or  $x2[, k]$ . Subscripting and trimming may be done only to **PRIMARYs**, e.g.,  $x1$  or  $(p | x1 | y1)$  but not re OF z1. The value of a **slice** may be either one element of the yield of its **PRIMARY** or a subset of the elements; e.g.,  $x1[i]$  is a real number from the row of real numbers  $x1$ ,  $x2[i, ]$  is the  $i$ -th row of the matrix  $x2$  and  $x2[, k]$  is its  $k$ -th column.}

### 19.3.2.1 Syntax

A) ROWSETY :: ROWS ; EMPTY.

a) REFETY MODE1 NEST slice {19<sub>5</sub>.D} :  
 weak REFLEXETY ROWS1 of MODE1 NEST PRIMARY {19<sub>5</sub>.D} ,

**ROWS1 leaving EMPTY NEST indexer {b, c, -} STYLE bracket,**  
**where (REFETY) is derived from (REFLEXETY) {19<sub>5</sub>.3.1.b, c, -};**  
**where (MODE1) is (ROWS2 of MODE2),**  
**weak REFLEXETY ROWS1 of MODE2 NEST PRIMARY {19<sub>5</sub>.D} ,**  
**ROWS1 leaving ROWS2 NEST indexer {b, d, -} STYLE bracket,**  
**where (REFETY) is derived from (REFLEXETY) {19<sub>5</sub>.3.1.b, c, -}.**  
**{ROWS :: row ; ROWS row.} item[b] row ROWS leaving ROWSETY1 ROWSETY2**  
**NEST indexer {a, b} :**  
**row leaving ROWSETY1 NEST indexer {c, d, -},**  
**and also {24<sub>9</sub>.4.f} token,**  
**ROWS leaving ROWSETY2 NEST indexer {b, c, d, -}.**

c) **row leaving EMPTY NEST indexer {a, b} : NEST subscript {e}.**

d) **row leaving row NEST indexer {a, b} :**  
**NEST trimmer {f};**  
**NEST revised lower bound {g} option.**

e) **NEST subscript {c} : meek integral NEST unit {32d} .**

f) **NEST trimmer {d} :**  
**NEST lower bound {18<sub>4</sub>.6.m} option,**  
**up to {94f} token,**  
**NEST upper bound {18<sub>4</sub>.6.n} option,**  
**NEST revised lower bound {g} option.**

g) **NEST revised lower bound {d, f} :**  
**at {24<sub>9</sub>.4.f} token,**  
**NEST lower bound {18<sub>4</sub>.6.m} .**

h) **\*trimscript :**  
**NEST subscript {e};**  
**NEST trimmer {f};**  
**NEST revised lower bound {g} option.**

i) **\*indexer : ROWS leaving ROWSETY NEST indexer {b, c, d}.**

j) **\*boundscript :**  
**NEST subscript {e};**  
**NEST lower bound {18<sub>4</sub>.6.m} ;**  
**NEST upper bound {18<sub>4</sub>.6.n} ;**  
**NEST revised lower bound {g}.**

{Examples:

a)  $x2[i, j] \bullet x2[, j]$

- b)  $1 : 2, j$  (in  $x2[1 : 2, j]$  •  $i, j$  (in  $x2[i, j]$ )
- c)  $j$  (in  $x2[1 : 2, j]$ )
- d)  $1 : 2$  •  $@0$  (in  $x1[@0]$ )
- e)  $j$
- f)  $1 : 2 @0$
- g)  $@0 \}$

{A **subscript** decreases the number of dimensions by one, but a **trimmer** leaves it unchanged. In rule a, '**ROWS1**' reflects the number of **trimscrip**ts in the **slice**, and '**ROWS2**' the number of these which are **trimmers** or **revised-lower-bound-options**. If the value to be sliced is a name, then the yield of the **slice** is also a name. Moreover, if the mode of the former name is '**reference to flexible ROWS1 of MODE**', then that yield is a transient name (see 16<sub>2</sub>.1.3.6.c).}

### 19.3.2.2 Semantics

a) The yield  $W$  of a **slice**  $S$  is determined as follows:

- let  $V$  and  $(I_1, \dots, I_n)$  be the {collateral} yields of the **PRIMARY** of  $S$  and of the **indexer** {b} of  $S$ ;
  - it is required that  $V$  {if it is a name} be not nil;
  - let  $((r_1, s_1), \dots, (r_n, s_n))$  be the descriptor of  $V$  or of the value referred to by  $V$ ;
- For  $i = 1, \dots, n$ ,

Case A:  $I_i$  is an integer:

- it is required that  $r_i \leq I_i \leq s_i$ ;

Case B:  $I_i$  is some triplet  $(l, u, l')$ :

- let  $L$  be  $r_i$ , if  $l$  is *absent*, and be  $l$  otherwise;
- let  $U$  be  $s_i$ , if  $u$  is *absent*, and be  $u$  otherwise;
- it is required that  $r_i \leq L$  and  $U \leq s_i$ ;
- let  $D$  be 0 if  $l'$  is *absent*, and be  $L - l'$  otherwise; { $D$  is the amount to be subtracted from  $L$  in order to get the revised lower bound;}
- $I_i$  is replaced by  $(L, U, D)$ ;

- $W$  is the value selected in {16<sub>2</sub>.1.3.4.a, g, i} or the name generated from {16<sub>2</sub>.1.3.4.j}  $V$  by  $I_1, \dots, I_n$ .

b) The yield of an **indexer**  $I$  of a **slice**  $S$  is a trim {16<sub>2</sub>.1.3.4.h} or an index {16<sub>2</sub>.1.3.4.a}  $I_1, \dots, I_n$  determined as follows:

- the constituent **boundscripts** of  $S$  are elaborated collaterally; For  $i = 1 \dots n$ , where  $n$  is the number of constituent **trimscripts** of  $S$ ,

Case A: the  $i$ -th **trimscript** is a **subscript**:

- $I_i$  is {the integer which is} the yield of that **subscript**;

Case B: the  $i$ -th **trimscript** is a **trimmer**  $T$ :

- $I_i$  is the triplet  $(l, u, l')$ , where
  - $l$  is the yield of the constituent **lower-bound**, if any, of  $T$ , and is *absent*, otherwise,
  - $u$  is the yield of the constituent **upper-bound**, if any, of  $T$ , and is *absent*, otherwise,
  - $l'$  is the yield of the constituent **revised-lower-bound**, if any, of  $T$ , and is 1, otherwise;

Case C: the  $i$ -th **trimscript** is a **revised-lower-bound-option**  $N$ :

- $I_i$  is the triplet (absent, absent,  $l'$ ), where
  - $l'$  is the yield of the **revised-lower-bound**, if any, of  $N$ , and is absent otherwise.

{Observe that, if  $(I_1, \dots, I_n)$  contains no triplets, it is an index, and selects one element; otherwise, it is a trim, and selects a subset of the elements.}

{A **slice** from a name referring to a multiple value yields an existing subname {16<sub>2</sub>.1.3.4.j} of that name if all the constituent **trimscripts** of that **slice** are **subscripts**. Otherwise, it yields a generated name which may or may not be newly created for the purpose. Hence, the yield of `x1[1 : 2] := x1[1 : 2]` is not defined, although `x1[1] := x1[1]` must always yield `true`.}

{The various possible bounds in the yield of a **slice** are illustrated by the following examples, for each of which the descriptor of the value referred to by the yield is shown:

```
[0 : 9, 2 : 11] INT i3;
i3[1, 3 : 10 @3] ⚡((3,10))⚡;
i3[1, 3 : 10] ⚡((1,8))⚡;
i3[1, 3 : ] ⚡((1,9))⚡;
i3[1, : ] ⚡((1,10))⚡;
i3[1, ] ⚡((2,11))⚡;
i3[, 2] ⚡((0,9))⚡
```



## 19.4 Units associated with routines

{Routines are created from **routine-texts** {19<sub>5</sub>.4.1} or from **jumps** {19<sub>5</sub>.4.4}, and they may be "called" by **calls** {19<sub>5</sub>.4.3}, **formulas** {19<sub>5</sub>.4.2} or by deproceduring {21<sub>6</sub>.3}.

### 19.4.1 Routine texts

{A **routine-text** always has a **formal-declarer**, specifying the mode of the result, and a **routine-token**, viz., a colon. To the right of this colon stands a **unit**, which prescribes the computations to be performed when the routine is called. If there are parameters, then to the left of the **formal-declarer** stands a **declarative** containing the various **formal-parameters** required.

Examples:

```
VOID: print (x);
(REF REAL a, REAL b) BOOL: (a <b | a := b; TRUE | FALSE).}
```

#### 19.4.1.1 Syntax

- a) **procedure yielding MOID NEST1 routine text** {18<sub>4</sub>.4.d, 19<sub>5</sub>.A} :  
**formal MOID NEST1 declarer** {18<sub>4</sub>.6.b} ,  
**routine** {24<sub>9</sub>.4.f} **token**,  
**strong MOID NEST1 unit** {17<sub>3</sub>.2.d} .
- b) **procedure with PARAMETERS yielding MOID NEST1 routine text** {18<sub>4</sub>.4.d, 19<sub>5</sub>.A} :  
**NEST1 new DECS2 declarative defining new DECS2** {e} **brief pack**,  
**where DECS2 like PARAMETERS** {c, d, -},  
**formal MOID NEST1 declarer** {18<sub>4</sub>.6.b} ,  
**routine** {24<sub>9</sub>.4.f} **token**,  
**strong MOID NEST1 new DECS2 unit** {17<sub>3</sub>.2.d} .
- c) **WHETHER DECS DEC like PARAMETERS PARAMETER** {b, c} :  
**WHETHER DECS like PARAMETERS** {c, d, -}  
**and DEC like PARAMETER** {d, -}.  
**{PARAMETER :: MODE parameter.}**
- d) **WHETHER MODE TAG like MODE parameter** {b, c} : **WHETHER true.**
- e) **NEST2 declarative defining new DECS2** {b, e, 17<sub>3</sub>.4.j} :  
**formal MODE NEST2 declarer** {18<sub>4</sub>.6.b} ,  
**NEST2 MODE parameter joined definition of DECS2** {18<sub>4</sub>.1.b, c} ;  
**where (DECS2) is (DECS3 DECS4),**

**formal MODE NEST2 declarer** {18<sub>4</sub>.6.b} ,  
**NEST2 MODE parameter joined definition of DECS3** {18<sub>4</sub>.1.b, c} ,  
**and also** {24<sub>9</sub>.4.f} **token**,  
**NEST2 declarative defining new DECS4** {e}.

- f) **NEST2 MODE parameter definition of MODE TAG2** {18<sub>4</sub>.1.c} :  
**MODE NEST2 defining identifier with TAG2** {18<sub>4</sub>.8.a} .
- g) **\*formal MODE parameter** :  
**NEST MODE parameter definition of MODE TAG** {f}.

{Examples:

- a) REAL: random × 10
- b) (BOOL a, b) BOOL: (a | b | FALSE)
- e) BOOL a, b • BOOL a, BOOL b
- f) a }

### 19.4.1.2 Semantics

The yield of a **routine-text**  $T$ , in an environ  $E$ , is the routine composed of

- (i)  $T$ , and
- (ii) the environ necessary for {22<sub>7</sub>.2.2.c}  $T$  in  $E$ .

## 19.4.2 Formulas

{**Formulas** are either dyadic or monadic: e.g.,  $x + i$  or **ABS**  $x$ . The order of elaboration of a **formula** is determined by the priority of its **operators**; monadic **formulas** are elaborated first and then the dyadic ones from the highest to the lowest priority.}

### 19.4.2.1 Syntax

- A) **DYADIC :: priority PRIO.**
- B) **MONADIC :: priority iii iii iii i.**
- C) **ADIC :: DYADIC ; MONADIC.**

D) **TALLETY :: TALLY ; EMPTY.**

- a) **MOID NEST DYADIC formula** {c, 5B} :  
**MODE1 NEST DYADIC TALLETY operand** {c, -},  
**procedure with MODE1 parameter MODE2 parameter yielding**  
**MOID NEST applied operator with TAD** {18<sub>4</sub>.8.b} ,  
**where DYADIC TAD identified in NEST** {22<sub>7</sub>.2.a} ,  
**MODE2 NEST DYADIC TALLY operand** {c, -}.
- b) **MOID NEST MONADIC formula** {c, 19<sub>5</sub>.B} :  
**procedure with MODE parameter yielding**  
**MOID NEST applied operator with TAM** {18<sub>4</sub>.8b} ,  
**MODE NEST MONADIC operand** {c}.
- c) **MODE NEST ADIC operand** {a, b} :  
**firm MODE NEST ADIC formula** {a, b} **coercee** {21<sub>6</sub>.1.b} ;  
**where (ADIC) is (MONADIC),**  
**firm MODE NEST SECONDARY** {5C} .
- d) **\*MOID formula** : **MOID NEST ADIC formula** {a, b}.
- e) **\*DUO dyadic operator with TAD** :  
**DUO NEST DEFIED operator with TAD** {18<sub>4</sub>.8.a, b} .
- f) **\*MONO monadic operator with TAM** :  
**MONO NEST DEFIED operator with TAM** {18<sub>4</sub>.8.a, b} .
- g) **\*MODE operand** : **MODE NEST ADIC operand** {c}.

{Examples:

- a)  $-x + 1$   
b)  $-x$   
c)  $-x \bullet 1$  }

### 19.4.2.2 Semantics

The yield  $W$  of a **formula**  $F$ , in an environ  $E$ , is determined as follows:

- let  $R$  be the routine yielded in  $E$  by the **operator** of  $F$ ;
- let  $V_1, \dots, V_n$  { $n$  is 1 or 2} be the {collateral} yields of the **operands** of  $F$ , in an environ  $E_1$  established {locally, see 17<sub>3</sub>.2.2.b} around  $E$ ;

- $W$  is the yield of the calling {19<sub>5</sub>.4.3.2.b} of  $R$  in  $E1$ , with  $V_1, \dots, V_n$ ;
- it is required that  $W$  be not newer in scope than  $E$ .

{Observe that  $a \uparrow b$  is not precisely the same as  $a^b$  in the usual notation; indeed, the value of  $(-1 \uparrow 2 + 4 = 5)$  and that of  $(4 - 1 \uparrow 2 = 3)$  both are true, since the first **minus-symbol** is a **monadic-operator**, whereas the second is a **dyadic-operator**.}

### 19.4.3 Calls

{**Calls** are used to command the elaboration of routines parametrized with **actual-parameters**.  
Examples:

`sin (x) • (p | sin | cos) (x).`

#### 19.4.3.1 Syntax

- MOID NEST call** {5D} :  
**meek procedure with PARAMETERS yielding MOID NEST PRIMARY** {19<sub>5</sub>.D}  
 ,  
**actual NEST PARAMETERS** {b, c} **brief pack**.
- actual NEST PARAMETERS PARAMETER** {a, b} :  
**actual NEST PARAMETERS** {b, c},  
**and also** {94f} **token**,  
**actual NEST PARAMETER** {c}.
- actual NEST MODE parameter** {a, b} : **strong MODE NEST unit** {17<sub>3</sub>.2.d} .

{Examples:

- `put (stand out, x)` (see 25<sub>10</sub>.3.3.1.a)
- `stand out, x`
- `x }`

#### 19.4.3.2 Semantics

- The yield  $W$  of a **call**  $C$ , in an environ  $E$ , is determined as follows:

- let  $R$  {a routine} and  $V_1, \dots, V_n$  be the {collateral} yields of the **PRIMARY** of  $C$ , in  $E$ , and of the constituent **actual-parameters** of  $C$ , in an environ  $E1$  established {locally, see 17<sub>3</sub>.2.2.b} around  $E$ ;
- $W$  is the yield of the calling {b} of  $R$  in  $E1$  with  $V_1, \dots, V_n$ ;
- it is required that  $W$  be not newer in scope than  $E$ .

b) The yield  $W$  of the "calling", of a routine  $R$  in an environ  $E1$ , possibly with {parameter} values  $V_1, \dots, V_n$ , is determined as follows:

- let  $E2$  be the environ established {17<sub>3</sub>.2.2.b} upon  $E1$ , around the environ of  $R$ , according to the **declarative** of the **declarative-pack**, if any, of the **routine-text** of  $R$ , with the values  $V_1, \dots, V_n$ , if any;
- $W$  is the yield in  $E2$  of the **unit** of the **routine-text** of  $R$ .

{Consider the following **serial-clause**:

```
PROC samelson = (INT n, PROC (INT) REAL f) REAL:
  BEGIN LONG REAL s := LONG 0;
    FOR i TO n DO s += LENG f (i) ↑ 2 OD;
    SHORTEN long sqrt (s)
  END;
samelson (m, (INT j) REAL: x1[j]).
```

In that context, the last **call** has the same effect as the following **cast**:

```
REAL (
  INT n = m, PROC (INT) REAL f = (INT j) REAL: x1[j];
  BEGIN LONG REAL s := LONG 0;
    FOR i TO n DO s += LENG f (i) ↑ 2 OD;
    SHORTEN long sqrt (s)
  END).
```

The transmission of **actual-parameters** is thus similar to the elaboration of **identity-declarations** {18<sub>4</sub>.4.2.a}; see also establishment {17<sub>3</sub>.2.2.b} and ascription {18<sub>4</sub>.8.2.a}.

## 19.4.4 Jumps

{A **jump** may terminate the elaboration of a **series** and cause some other labelled **series** to be elaborated in its place.

Examples:

```
y = IF x ≥ 0 THEN sqrt (x) ELSE GOTO princeton FI •
GOTO st pierre de chartreuse.
```

Alternatively, if the context expects the mode '**procedure yielding MOID**', then a routine whose **unit** is that **jump** is yielded instead, as in

```
PROC VOID m := GOTO north berwick.}
```

#### 19.4.4.1 Syntax

- a) **strong MOID NEST jump** {19<sub>5</sub>.A} :  
     **go to** {b} **option**,  
     **label NEST applied identifier with TAG** {18<sub>4</sub>.8.b} .
- b) **go to** {a} :  
     **STYLE go to** {24<sub>9</sub>.4.f, -} **token** ;  
     **STYLE go** {24<sub>9</sub>.4.f, -} **token**, **STYLE to symbol** {24<sub>9</sub>.4.g, -}.

{Examples:

- a) GOTO kootwijk • GO TO warsaw • zandvoort
- b) GOTO • GO TO }

#### 19.4.4.2 Semantics

A **MOID-NEST-jump** *J*, in an environ *E*, is elaborated as follows:

- let the scene yielded in *E* by the **label-identifier** of *J* be composed of a **series** *S2* and an environ *E1*;

Case A: '**MOID**' is not any '**procedure yielding MOID1**':

- let *S1* be the **series** of the smallest {15<sub>1</sub>.1.3.2.g} **serial-clause** containing *S2*;
- the elaboration of *S1* in *E1*, or of any **series** in *E1* elaborated in its place, is terminated {16<sub>2</sub>.1.4.3.e};
- *S2* in *E1* is elaborated "in place of" *S1* in *E1*;

Case B: '**MOID**' is some '**procedure yielding MOID1**':

- *J* in *E* {is completed and} yields the routine composed of
- a new **MOID-NEST-routine-text** whose **unit** is akin {15<sub>1</sub>.1.3.2.k} to *J*,
- *E1*.

## 19.5 Units associated with values of any mode

### 19.5.1 Casts

{**Casts** may be used to provide a strong position. For example,

```
REF REAL (xx) in REF REAL (xx) := 1,
REF BOOK (NIL) in next OF draft :=: REF BOOK (NIL) and
STRING (p | c | r) in s +:= STRING (p | c | r).}
```

#### 19.5.1.1 Syntax

- a) **MOID NEST cast** {19<sub>5</sub>.D} :  
     **formal MOID NEST declarer** {18<sub>4</sub>.6.b} ,  
     **strong MOID NEST ENCLOSED clause** {17<sub>3</sub>.1.a, 17<sub>3</sub>.3.a, c, d, e, 17<sub>3</sub>.4.a,  
     17<sub>3</sub>.5.a, -}.

{Example:

- a) REF BOOK (NIL) }

{The yield of a **cast** is that of its **ENCLOSED-clause**, by way of pre-elaboration {16<sub>2</sub>.1.4.1.c}.}

### 19.5.2 Skips

#### 19.5.2.1 Syntax

- a) **strong MOID NEST skip** {19<sub>5</sub>.A} : **skip** {24<sub>9</sub>.4.f} **token**.

#### 19.5.2.2 Semantics

The yield of a **skip** is some {undefined} value equal in scope to the primal environ.

{The mode of the yield of a **MOID-skip** is 'MOID'. A **void-skip** serves as a dummy statement and may be used, for example, after a **label** which marks the end of a **serial-clause**.}

# ALGOL 68 REVISED REPORT



# Chapter 20

## Specification of partial parametrization proposal

by C.H. Lindsey (University of Manchester)

The following specification<sup>1</sup> has been released by the IFIP Working Group 2.1 Standing subcommittee on Algol 68 Support, with the authorization of the Working Group. This proposal has been scrutinized to ensure that

- a. it is strictly upwards-compatible with Algol 68,
- b. it is consistent with the philosophy and orthogonal framework of that language, and
- c. it fills a clearly discernible gap in the expressive power of that language.

In releasing this extension, the intention is to encourage implementers experimenting with features similar to those described below to use the formulation here given, so as to avoid proliferation of dialects.

{{ Although routines are values in Algol 68, and can therefore be yielded by other routines, the practical usefulness of this facility is limited by scope restrictions. Consider:

```
PROC f = (REAL x) PROC (REAL) REAL: (REAL y) REAL: x + y;  
PROC (REAL) REAL g := f (3);  
x := g (4)
```

This attempts to assign to `g` the routine "add 3". It does not work because the body of the routine is still fundamentally the **routine-text** `(REAL y) REAL : x + y` which expects

---

<sup>1</sup>This chapter is not a part of the Algol 68 Revised Report, and is reproduced here with kind permission of C.H. Lindsey. The original publication is ALGOL BULLETIN 39.3.1 pages 6-9.

## ALGOL 68 REVISED REPORT

to find the value  $x$  (i.e. 3) on the stack in the form of an actual-parameter of  $f$ , and by this time  $f$  is finished with and its stack level has disappeared. The problem arises whenever a **routine-text** uses identifiers declared globally to itself and the limitation is expressed in the Report by making the scope of a routine dependent on its necessary environ (22<sub>7</sub>.2.2.c). Here is an attempt at functional composition which fails to work for the same reason:

```
PROC compose = (PROC (REAL) REAL f, g) PROC (REAL) REAL: (REAL x) REAL:
f (g (x));
PROC (REAL) REAL sex = compose (sqrt, exp)
```

Clearly, if the restriction is to be lifted, a routine value has to have associated with it copies of these global values. Unfortunately, their number is in general indeterminable at compile time, and so the implementation of such values must be similar to that of multiple values referred to by flexible names (16<sub>2</sub>.1.3.4) requiring, in most implementations, the use of the heap.

In this variant, all the intended global values appear to the **routine-text** as its own formal-parameters. At each call, some or all of these parameters are provided with actual values, resulting in a routine with that number of parameters fewer. Ultimately (possibly after several calls) a routine without parameters is obtained and, if the context so demands, deproceduring can now take place. Thus, all calls in the original language turn out to be parametrizations followed by immediate deproceduring, but their effect is the same. Here are some examples:

- 1) PROC f = (REAL x, y) x + y;  
PROC (REAL) REAL g := f (3, );  
x := g (4)  $\dot{c}$  or x := f (3, ) (4)  $\dot{c}$
- 2) PROC compose = (PROC (REAL) REAL f, g, REAL x) REAL; f (g (x));  
PROC (REAL) REAL sex = compose (sqrt, exp, )
- 3) OP  $\uparrow$  = (PROC (REAL) REAL a, INT b) PROC (REAL) REAL:  
( (PROC (REAL) REAL a, INT b, REAL p) REAL:  
(REAL x := 1; TO b DO x  $\times$  := a (p) OD; x )) (a, b, );  
REAL theta; print ((cos  $\uparrow$  2) (theta) + (sin  $\uparrow$  2) (theta))

}}

{{A routine now includes an extra locale.}}

### 16<sub>2</sub>.1.3.5. Routines

- a) A "routine" is a value composed of a **routine-text** {19<sub>5</sub>.4.1.1.a,b}, an environ {16<sub>2</sub>.1.1.1.c} and a locale {16<sub>2</sub>.1.1.1.b}. {The locale corresponds to a 'DECSETY' reflecting the formal-parameters, if any, of the **routine-text**.}

- b) The mode of a routine is some **'PROCEDURE'**.
- c) The scope of a routine is the newest of the scopes of its environ and of the values, if any, accessed {16<sub>2</sub>.1.2.c} inside its locale.

{{A **routine-text** yields the new style of routine.}}

### 19<sub>5</sub>.4.1.2. Semantics

The yield of a **routine-text** *T*, in an environ *E*, is the routine composed of

- (i) *T*,
- (ii) the environ necessary for {22<sub>7</sub>.2.2.c} *T* in *E*, and
- (iii) a locale corresponding to **'DECS2'** if *T* has a **declarative-defining-new-DECS2-brief-pack**, and to **'EMPTY'** otherwise.

{{Most of the remaining changes to the Report needed to incorporate this facility are in section 19<sub>5</sub>.4.3 (calls).}}

### 19<sub>5</sub>.4.3. Calls (with partial parametrization)

A **call** is used to provide **actual-parameters** to match some or all of the **formal-parameters** of a routine. It yields a routine with correspondingly fewer **formal-parameters** or with none at all, in which case the yield is usually subject to deproceduring (21<sub>6</sub>.3).

{Examples:

```
y := sin (x) •
PROC REAL nccossini = (p | ncos | nsin) (i) •
print ((set char number (, 5), x)).}
```

#### 19<sub>5</sub>.4.3.1 Syntax

A) **PARAMSETY :: PARAMETERS ; EMPTY.**

- a) **procedure yielding MOID NEST call {19<sub>5</sub>.D}:**  
**meek procedure with PARAMETERS1 yielding MOID NEST PRIMARY {19<sub>5</sub>.D} ,**  
**actual NEST PARAMETERS1 leaving EMPTY {c,d,e} brief pack.**
- b) **procedure with PARAMETERS2 yielding MOID NEST call {19<sub>5</sub>.D} :**  
**meek procedure with PARAMETERS1 yielding MOID NEST PRIMARY {19<sub>5</sub>.D} ,**  
**actual NEST PARAMETERS1 leaving PARAMETERS2 {c,d,e,f} brief pack.**

- c) **actual NEST PARAMETER PARAMETERS leaving PARAMSETY1 PARAM-SETY2** {a,b,c}:  
     **actual NEST PARAMETER leaving PARAMSETY1** {d,e}, **and also** {24<sub>9</sub>.4.f} **token,**  
     **actual NEST PARAMETERS leaving PARAMSETY2** {c,d,e}.
- d) **actual NEST MODE parameter leaving EMPTY** {a,b,c}: **strong MODE NEST unit** {17<sub>3</sub>.2.d} .
- e) **actual NEST PARAMETER leaving PARAMETER** {a,b,c}: **EMPTY.**
- f) \* **actual MODE parameter: actual NEST MODE parameter leaving EMPTY** {d}.
- g) \* **dummy parameter: actual NEST PARAMETER leaving PARAMETER** {e}.

{Examples:

- a) set char number (stand out, 5)
- b) set char number (, 5)
- c) , 5
- d) 5

}

### 19<sub>5</sub>.4.3.2. Semantics

a1) The yield  $W$  of a **call**  $C$ , in an environ  $E$ , is determined as follows:

- let  $R$  {a routine} and  $V_1, \dots, V_n$  be the {collateral} yields of the **PRIMARY** of  $C$ , in  $E$ , and of the constituent **actual-** and **dummy-parameters** of  $C$ , in an environ  $E_1$  established {locally, see 17<sub>3</sub>.2.2.b} around  $E$ , where the yield of a **dummy-parameter** is "*absent*";
- $W$  is {the routine which is} the yield of the "parametrization" a2 of  $R$  with  $V_1, \dots, V_n$ ;
- except where  $C$  is the constituent **call** of a **deprocedured-to-MOID-call** 21<sub>6</sub>.3.1.a, it is required that  $W$  be not newer in scope than  $E$  {; thus, PROC (CHAR, STRING) BOOL cs = char in string (, LOC INT,) is undefined but q := char in string ("A", LOC INT, s) is not}.

a2) The yield  $W$  of the "parametrization" of a routine  $R_0$  with values  $V_1, \dots, V_n$  is determined as follows:

## SPECIFICATION OF PARTIAL PARAMETRIZATION PROPOSAL

- let  $T_0$ ,  $E_0$  and  $L_0$  be, respectively, the **routine-text**, the environ and the locale of  $R_0$ , and let  $L_0$  correspond {16<sub>2</sub>.1.1.1.b} to some '**DECS0**';
- let  $L_1$  be a new locale corresponding to '**DECS0**', and let the value, if any, accessed by any '**DEC0**' inside  $L_0$  be accessed also by that '**DEC0**' inside  $L_1$ ;
- let '**DECS1**' be a sequence composed of all those '**DEC0**'s enveloped by '**DECS0**' which have not {yet} been made to access values inside  $L_1$ , taken in their order within '**DECS0**';

For  $i = 1, \dots, n$ , If  $V_i$  is not *absent*, {see a1}, then the  $i$ -th '**DEC1**' enveloped by '**DECS1**' is made to access  $V_i$  inside  $L_1$ ; otherwise, the  $i$ -th '**DEC1**' still does not access anything;

- $W$  is the routine composed of  $T_0$ ,  $E_0$  and  $L_1$ .

{A routine may be parametrized in several stages. Upon each occasion the yields of the new **actual-parameters** are made to be accessed inside its locale and the scope of the routine becomes the newest of its original scope and the scopes of those yields.}

- b) The yield  $W$  of the "calling" of a routine  $R_0$  in an environ  $E_1$  {see 19<sub>5</sub>.4.2.2 and 21<sub>6</sub>.3.2} is determined as follows:
- let  $T_0$ ,  $E_0$  and  $L_0$  be, respectively, the **routine-text**, the environ and the locale of  $R_0$ ;
  - let  $E_2$  be a {newly established} environ, newer in scope than  $E_1$ , composed of  $E_0$  and  $L_0$  { $E_2$  is local};
  - $W$  is the yield, in  $E_2$ , of the unit of  $T_0$ .

Consider the following **serial-clause**:

```
PROC samelson = (INT n, PROC (INT) REAL f) REAL:
  BEGIN LONG REAL s := LONG 0;
    FOR i TO n DO s += LENG f (i) ↑ 2 OD;
    SHORTEN long sqrt (s)
  END;
y := samelson (m, (INT j) REAL: x1[j])
```

In that context, the last **deprocedured-to-real-call** has the same effect as the **deprocedured-to-real-routine-text** in:

```
y := REAL: (
  INT n = m, PROC (INT) REAL f = (INT j) REAL: x1[j];
  BEGIN LONG REAL s := LONG 0;
```

## ALGOL 68 REVISED REPORT

```
FOR i TO n DO s +:= LENG f (i) ↑ 2 OD;  
SHORTEN long sqrt (s)  
END) .
```

The transmission of the **actual-parameters** is thus similar to the elaboration of **identity-declarations** (18<sub>4</sub>.4.2.a); see also establishment (17<sub>3</sub>.2.2.b) and ascription (18<sub>4</sub>.8.2.a).

{{Minor changes are required at other places in the Report.}}

{{The third bullet of 19<sub>5</sub>.4.2.2 (semantics of formulas) is replaced by}}

- let  $R_1$  be {the routine which is} the yield of the parametrization {19<sub>5</sub>.4.3.2.a2} of  $R$  with  $V_1, \dots, V_n$ ;
- $W$  is the yield of the calling {19<sub>5</sub>.4.3.2.b} of  $R_1$  in  $E_1$ ;

{{19<sub>5</sub>.4.4.2.Case B, 25<sub>10</sub>.3.4.1.2.c and 25<sub>10</sub>.3.4.9.2 must be modified to show that the routines there created are composed, additionally, from a vacant locale {16<sub>2</sub>.1.1.1.b}.}}

## PART III - Context Dependence

{This Part deals with those rules which do not alter the underlying syntactical structure:

- the transformations of modes implicitly defined by the context, with their accompanying actions;
- the syntax needed for the equivalence of modes and for the safe application of the properties kept in the nests.}

# ALGOL 68 REVISED REPORT



# Chapter 21

## Coercion

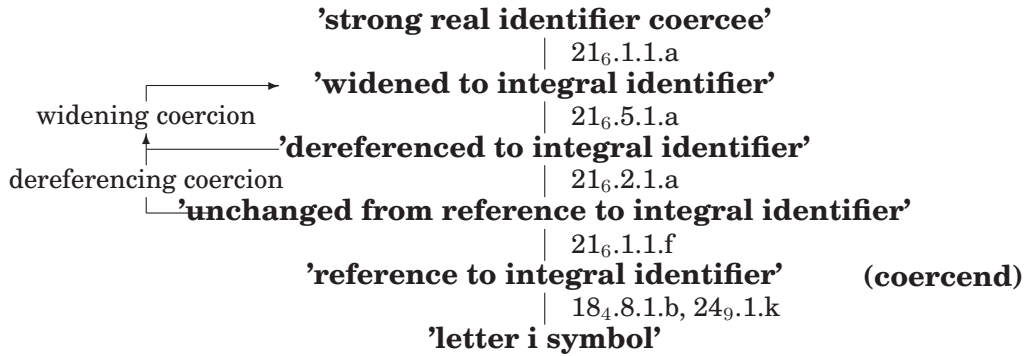
{The coercions produce a **coercend** from a **coercee** according to three criteria: the a priori mode of the **coercend** before the application of any coercion, the a posteriori mode of the **coercee** required after those coercions, and the syntactic position or "sort" of the **coercee**. Coercions may be cascaded.

There are six possible coercions, termed "deproceduring", "dereferencing", "uniting", "widening", "rowing" and "voiding". Each coercion, except "uniting", prescribes a corresponding dynamic effect on the associated values. Hence, a number of primitive actions can be programmed implicitly by coercions.}

### 21.1 Coercees

{A **coercee** is a construct whose production tree may begin a sequence of coercions ending in a **coercend**. The order of (completion of) the elaboration of the coercions is therefore from the **coercend** to the **coercee** (hence the choice of these paranotions). For example, *i* in *REAL (i)* is a **coercee** whose production tree involves '**widened to**' and '**dereferenced to**', in that order, in passing from the **coercee** to the **coercend**. Note that the dereferencing must be completed before the widening takes place.

The relevant production tree (with elision of '**NEST**', '**applied**' and '**with TAG**', and with invisible subtrees omitted) is:



}

### 21.1.1 Syntax

- A) **STRONG** {a, 21<sub>6</sub>.6.a } :: **FIRM** {B}; **widened to** {21<sub>6</sub>.5.a, b, c, d } ;  
**rowed to** {21<sub>6</sub>.6.a } ; **voided to** {21<sub>6</sub>.7.a, b } .
- B) **FIRM** {A, b } :: **MEEK** {c}; **united to** {21<sub>6</sub>.4.a } .
- C) **MEEK** {B, c, d, 21<sub>6</sub>.2.a, 21<sub>6</sub>.3.a, 21<sub>6</sub>.4.a, 21<sub>6</sub>.5.a, b, c, d } ::  
**unchanged from** {f}; **dereferenced to** {21<sub>6</sub>.2.a } ; **deprocedured to** {21<sub>6</sub>.3.a } .
- D) **SOFT** {e, 21<sub>6</sub>.3.b } :: **unchanged from** {f}; **softly deprocedured to** {21<sub>6</sub>.3.b } .
- E) **FORM** :: **MORF** ; **COMORF**.
- F) **MORF** :: **NEST selection** ; **NEST slice** ; **NEST routine text** ; **NEST ADIC formula** ;  
**NEST call** ; **NEST applied identifier with TAG**.
- G) **COMORF** :: **NEST assignation** ; **NEST identity relation** ; **NEST LEAP generator** ;  
**NEST cast** ; **NEST denoter** ; **NEST format text**.
- a) **strong MOID FORM coercee** {19<sub>5</sub>.A, B, C, D, 25<sub>10</sub>.3.4.1.i } :  
**where (FORM) is (MORF), STRONG {A} MOID MORF** ;  
**where (FORM) is (COMORF), STRONG {A} MOID COMORF,**  
**unless (STRONG MOID) is (deprocedured to void).**
- b) **firm MODE FORM coercee** {19<sub>5</sub>.A, B, C, D, 19<sub>5</sub>.4.2.c } : **FIRM {B} MODE FORM**.
- c) **meek MOID FORM coercee** {19<sub>5</sub>.A, B, C, D } : **MEEK {C} MOID FORM**.
- d) **weak REFETY STOWED FORM coercee** {19<sub>5</sub>.A, B, C, D } :  
**MEEK {C} REFETY STOWED FORM,**  
**unless (MEEK) is (dereferenced to) and (REFETY) is (EMPTY).**

- e) **soft MODE FORM coercee** {19<sub>5</sub>.A, B, C, D} : **SOFT {D} MODE FORM.**
- f) **unchanged from MOID FORM** {C, D, 21<sub>6</sub>.7.a, b } : **MOID FORM.**
- g) **\*SORT MOID coercee** : **SORT MOID FORM coercee** {a, b, c, d, e}.
- h) **\*MOID coerced** : **MOID FORM.**

{Examples:

- a) 3.14 (in x := 3.14)
- b) 3.14 (in x + 3.14)
- c) sin (in sin (x))
- d) x1 (in x1[2] := 3.14)
- e) x (in x := 3.14)}

{For '**MOID FORM**' (rule f), see the cross-references inserted in sections 19<sub>5</sub>.1.A, B, C, D before "**coercee**". Note, however, that a '**MOID FORM**' may be a blind alley. Blind alleys within this chapter are not indicated.}

{There are five sorts of syntactic position. They are:

- "strong" positions, i.e., **actual-parameters**, e.g., x in sin (x), **sources**, e.g., x in y := x, the **ENCLOSED-clause** of a **cast**, e.g., (NIL) in REF BOOK (NIL), and **statements**, e.g., y := x in (y := x; x := 0);
- "firm" positions, i.e., **operands**, e.g., x in x + y;
- "meek" positions, i.e., **enquiry-clauses**, e.g., x > 0 in (x > 0 | x | 0), **bound-scripts**, e.g., i in x1 [i], and the **PRIMARY** of a **call**, e.g., sin in sin (x);
- "weak" positions, i.e., the **SECONDARY** of a **selection** and the **PRIMARY** of a **slice**, e.g., x1 in x1 [i];
- "soft" positions, i.e., **destinations**, e.g., x in x := y and one of the **TERTIARYs** of an **identity-relation**, e.g., x in xx :=: x.

Strong positions also arise in balancing {17<sub>3</sub>.2.1.e}.

In strong positions, all six coercions may occur; in firm positions, rowing, widening and voiding are forbidden; in meek and weak positions, uniting is forbidden also, and in soft positions only deproceduring is allowed. However, a **dereferenced-to-STOWED-FORM**

may not be directly descended from a **weak-STOWED-FORM-coercee** (rule d) for, otherwise,  $x := x1[i]$  would be syntactically ambiguous (although, in this case, not semantically). Also, a **deprocedured-to-void-COMORF** may not be directly descended from a **strong-void-COMORF-coercee** (rule a) for, otherwise,

```
(PROC VOID engelfriet; PROC VOID rijpens = SKIP;
  engelfriet := rijpens; SKIP)
would be ambiguous.)
```

## 21.2 Dereferencing

{Dereferencing serves to obtain the value referred to by a name, as in  $x := y$ , where  $y$  yields a name referring to a real number and it is this number which is assigned to the name yielded by  $x$ . The a priori mode of  $y$ , regarded as a **coercend**, is 'reference to real' and its a posteriori mode, when  $y$  is regarded as a **coercee**, is 'real'.}

### 21.2.1 Syntax

- a) **dereferenced to** {21<sub>6</sub>.1.C} **MODE1 FORM** :  
**MEEK** {21<sub>6</sub>.1.C} **REF to MODE2 FORM**,  
 where **MODE2** deflexes to **MODE1** {18<sub>4</sub>.7.a,b,c, -}.

{Example:

- a)  $x$  (in REAL ( $x$ )) }

### 21.2.2 Semantics

The yield  $W$  of a **dereferenced-to-MODE-FORM**  $F$  is determined as follows:

- let {the name}  $N$  be the yield of the **MEEK-FORM** of  $F$ ;
- it is required that  $N$  be not nil;
- $W$  is the value referred to by  $N$ .

## 21.3 Deproceduring

{Deproceduring is used when a routine without parameters is to be called. E.g., in `x := random`, the routine yielded by `random` is called and the real number yielded is assigned; the a posteriori mode of `random` is **'real'**. Syntactically, an initial **'procedure yielding'** is removed from the a priori mode.}

### 21.3.1 Syntax

- a) **deprocedured to** {21<sub>6</sub>.1.C, 21<sub>6</sub>.7.a} **MOID FORM :**  
**MEEK** {21<sub>6</sub>.1.C } **procedure yielding MOID FORM.**
- b) **softly deprocedured to** {21<sub>6</sub>.1.D } **MODE FORM :**  
**SOFT** {21<sub>6</sub>.1.D } **procedure yielding MODE FORM.**

{Examples:

- a) `random (in REAL (random))`
- b) `x or y (in x or y := 3.14, see 151.1.2) }`

### 21.3.2 Semantics

The yield  $W$  of a **deprocedured-to-MOID-FORM** or **softly-deprocedured-to-MOID-FORM**  $F$ , in an environ  $E$ , is determined as follows:

- let {the routine}  $R$  be the yield in  $E$  of the direct descendent of  $F$ ;
- $W$  is the yield of the calling {19<sub>5</sub>.4.3.2.b} of  $R$  in  $E$ ;
- it is required that  $W$  be not newer in scope than  $E$ .

## 21.4 Uniting

{Uniting does not change the mode of the run-time value yielded by a construct, but simply gives more freedom to it. That value must be acceptable to not just that one mode, but rather to the whole of a given set of modes. However, after uniting, that value may be subject to a primitive action only after being dynamically tested in a **conformity-clause** {17<sub>3</sub>.4.1.q}; indeed, no primitive action can be programmed with a construct of a **'UNITED'** mode (except to assign it to a **UNITED-variable**, of course).

Example:

```
UNION (BOOL, CHAR) t, v;
t := "a"; t := TRUE; v := t. }
```

## 21.4.1 Syntax

- a) **united to** {21<sub>6</sub>.1.B} **UNITED FORM :**  
**MEEK** {21<sub>6</sub>.1.C} **MOID FORM,**  
 where **MOID unites to UNITED** {b}.
- b) **WHETHER MOID1 unites to MOID2** {a, 17<sub>3</sub>.4.i, 22<sub>7</sub>.1.m} :  
 where **MOID1 equivalent MOID2** {22<sub>7</sub>.3.a} , **WHETHER false ;**  
**unless MOID1 equivalent MOID2** {22<sub>7</sub>.3.a} ,  
**WHETHER safe MOODS1 subset of safe MOODS2** {22<sub>7</sub>.3.1, m, n} ,  
 where **(MOODS1) is (MOID1)**  
**or (union of MOODS1 mode) is (MOID1),**  
 where **(MOODS2) is (MOID2)**  
**or (union of MOODS2 mode) is (MOID2).**

{Examples:

- a)  $x \text{ (in uir := x) } \bullet$   
 $u \text{ (in UNION (CHAR, INT, VOID) (u), in a reach containing}$   
 $\text{UNION (INT, VOID) } u := \text{EMPTY}) \}$

## 21.5 Widening

{Widening transforms integers to real numbers, real numbers to complex numbers (in both cases, with the same size), a value of mode 'BITS' to an unpacked vector of truth values, or a value of mode 'BYTES' to an unpacked vector of characters.

For example, in  $z := 1$ , the yield of 1 is widened to the real number 1.0 and then to the complex number (1.0, 0.0); syntactically, the a priori mode specified by INT is changed to that specified by REAL and then to that specified by COMPL.}

### 21.5.1 Syntax

- A) **BITS :: structured with row of boolean field SITHETY letter aleph mode.**
- B) **BYTES :: structured with row of character field SITHETY letter aleph mode.**
- C) **SITHETY :: LENGTH LENGTHEY ; SHORTH SHORTHETY ; EMPTY.**
- D) **LENGTH :: letter l letter o letter n letter g.**
- E) **SHORTH :: letter s letter h letter o letter r letter t.**

F) **LENGTHETY :: LENGTH LENGTHETY ; EMPTY.**

G) **SHORTHETY :: SHORTH SHORTHETY ; EMPTY.**

a) **widened to {b, 21<sub>6</sub>.1.A} SIZETY real FORM :**

**MEEK {21<sub>6</sub>.1.C} SIZETY integral FORM. {SIZETY :: long LONGSETY ; short SHORTSETY ; EMPTY.}**

b) **widened to {21<sub>6</sub>.1.A} structured with SIZETY real field letter r letter e SIZETY real field letter i letter m mode FORM :**

**MEEK {61C} SIZETY real FORM ; widened to {a} SIZETY real FORM.**

c) **widened to {21<sub>6</sub>.1.A} row of boolean FORM : MEEK {61C} BITS FORM.**

d) **widened to {21<sub>6</sub>.1.A} row of character FORM : MEEK {61C} BYTES FORM.**

{Examples:

a) 1 (in x := 1)

b) 1.0 (in z := 1.0) • 1 (in z := 1)

c) 2r101 (in [] BOOL (2r101))

d) r (in [] CHAR (r), see 15<sub>1</sub>.1.2)

## 21.5.2 Semantics

The yield  $W$  of a **widened-to-MODE-FORM**  $F$  is determined as follows:

• let  $V$  be the yield of the direct descendent of  $F$ ; Case A: **'MODE'** is some **'SIZETY real'**:

- $W$  is the real number widenable from {16<sub>2</sub>.1.3.1.e}  $V$ ;

Case B: **'MODE'** is some **'structured with SIZETY real letter r letter e SIZETY real letter i letter m mode'**:

- $W$  is {the complex number which is} a structured value whose fields are respectively  $V$  and the real number 0 of the same size {16<sub>2</sub>.1.3.1.b} as  $V$ ;

Case C: **'MODE'** is **'row of boolean'** or **'row of character'**:

- $W$  is the {only} field of  $V$ .

## 21.6 Rowing

{Rowing permits the building of a multiple value from a single element. If the latter is a name then the result of rowing may also be a name referring to that multiple value.

Example:

```
[1 : 1] REAL b1 := 4.13 }
```

### 21.6.1 Syntax

- a) **rowed to** {21<sub>6</sub>.1.A} **REFETY ROWS1 of MODE FORM :**  
**where (ROWS1) is (row),**  
**STRONG** {21<sub>6</sub>.1.A} **REFLEXETY MODE FORM,**  
**where (REFETY) is derived from (REFLEXETY) {19<sub>5</sub>.3.1.b,c, -};**  
**where (ROWS1) is (row ROWS2),**  
**STRONG** {21<sub>6</sub>.1.A} **REFLEXETY ROWS2 of MODE FORM,**  
**where (REFETY) is derived from (REFLEXETY) {19<sub>5</sub>.3.1.b, c, -}.**

{Examples:

- a) 4.13 (in [1 : 1] REAL b1 := 4.13) •  
 x1 (in [1 : 1, 1 : n] REAL b2 := x1) }

### 21.6.2 Semantics

a) The yield  $W$  of a **rowed-to-REFETY-ROWS1-of-MODE-FORM**  $F$  is determined as follows:

•let  $V$  be the yield of the **STRONG-FORM** of  $F$ ;

Case A: **'REFETY'** is **'EMPTY'**:

- $W$  is the multiple value "built" {b} from  $V$  for **'ROWS1'**;

Case B: **'REFETY'** is **'REF to'**:

- If  $V$  is nil,
- then  $W$  is a nil name;
- otherwise,  $W$  is the name "built" {c} from  $V$  for **'ROWS1'**.



b) The multiple value  $W$  "built" from a value  $V$ , for some **'ROWS1'**, is determined as follows:

Case A: **'ROWS1'** is **'row'**:

- $W$  is composed of
- a descriptor  $((1, 1))$ ,
- {one element}  $V$ ;

Case B: **'ROWS1'** is some **'row ROWS2'**:

- let the descriptor of  $V$  be  $((l_1, u_1), \dots, (l_n, u_n))$ ;
- $W$  is composed of
  - a descriptor  $((1, 1), (l_1, u_1), \dots, (l_n, u_n))$ ,
  - the elements of  $V$ :
- the element selected by an index  $(i_1, \dots, i_n)$  in  $V$  is that selected by  $(1, i_1, \dots, i_n)$  in  $W$ .

c) The name  $N1$  "built" from a name  $N$ , for some **'ROWS1'**, is determined as follows:

•  $N1$  is a name {not necessarily newly created}, equal in scope to  $N$  and referring to the multiple value built {b}, for **'ROWS1'**, from the value referred to by  $N$ ;

Case A: **'ROWS1'** is **'row'**:

the {only} subname of  $N1$  is  $N$ ;

Case B: **'ROWS1'** is some **'row ROWS2'**:

the subname of  $N1$  selected by  $(1, i_1, \dots, i_n)$  is the subname of  $N$  selected by  $((i_1, \dots, i_n))$ .

## 21.7 Voiding

{Voiding is used to discard the yield of some **unit** whose primary purpose is to cause its side-effects; the a posteriori mode is then simply **'void'**. For example, in `x := 1; y := 1;` the **assignment** `y := 1` is voided, and in `PROC t = INT: ENTIER (random × 100); t;`, the **applied-identifier** `t` is voided after a deproceduring, which prescribes the calling of a routine.

**Assignations** and other **COMORFs** are voided without any deproceduring so that, in `PROC VOID p: p := finish,` the **assignment** `p := finish` does not prescribe an unexpected calling of the routine `finish`.)

## 21.7.1 Syntax

- A) **NONPROC :: PLAIN ; STOWED ; REF to NONPROC ;  
procedure with PARAMETERS yielding MOID ; UNITED.**
- a) **voided to {21<sub>6</sub>.1.A} void MORF :  
deprocedured to {21<sub>6</sub>.3.a} NONPROC MORF ;  
unchanged from {21<sub>6</sub>.1.f} NONPROC MORF.**
- b) **voided to {21<sub>6</sub>.1.A} void COMORF :  
unchanged from {21<sub>6</sub>.1.f} MODE COMORF.**

{Examples:

- a) random (in SKIP; random;) •  
next random (last random) (in SKIP; next random (last random);)
- b) PROC VOID (pp) (in PROC PROC VOID pp = PROC VOID: (print (1);  
VOID: print (2)); PROC VOID (pp);)

## 21.7.2 Semantics

The elaboration of a **voided-to-void-FORM** consists of that of its direct descendent, and yields empty.

# Chapter 22

## Modes and nests

{The identification of a property in a nest is the static counterpart of the dynamic determination (4.8.2.b) of a value in an environ: the search is conducted from the newest (youngest) level towards the previous (older) ones.

Modes are composed from the primitive modes, such as **'boolean'**, with the aid of **'HEAD'**s, such as **'structured with'**, and they may be recursive. Recursive modes spelled in different ways may nevertheless be equivalent. The syntax tests the equivalence of such modes by proving that it is impossible to find any discrepancy between their respective structures or component modes.

A number of unsafe uses of properties are prevented. An **identifier** or **mode-indication** is not declared more than once in each reach. The modes of the **operands** of a **formula** do not determine more than one operation. Recursions in modes do not cause the creation of dynamic objects of unlimited size and do not allow ambiguous coercions.}

### 22.1 Independence of properties

{The following syntax determines whether two properties (i.e., two **'PROP'**s), such as those corresponding to `REAL x` and `INT x`, may or may not be enveloped by the same **'LAYER'**.}

#### 22.1.1 Syntax

- A) **PREF :: procedure yielding ; REF to.**
- B) **NONPREF :: PLAIN ; STOWED ;  
    procedure with PARAMETERS yielding MOID ; UNITED ; void.**

C) **\*PREFSETY :: PREF PREFSETY ; EMPTY.**

**{PROP :: DEC ; LAB ; FIELD.**

**QUALITY :: MODE ; MOID TALLY ; DYADIC ; label ; MODE field.**

**TAX :: TAG ; TAB ; TAD ; TAM.**

**TAO :: TAD ; TAM.}**

- a) **WHETHER PROP1 independent PROPS2 PROP2 {a, 18<sub>4</sub>.8.a,c, 22<sub>7</sub>.2.a} :**  
**WHETHER PROP1 independent PROPS2 {a, c} and PROP1 independent PROP2 {c}.**
- b) **WHETHER PROP independent EMPTY {18<sub>4</sub>.8.a,c, 22<sub>7</sub>.2.a} : WHETHER true.**
- c) **WHETHER QUALITY1 TAX1 independent QUALITY2 TAX2 {a, 18<sub>4</sub>.8.a,c, 22<sub>7</sub>.2.a} :**  
**unless (TAX1) is (TAX2), WHETHER true ;**  
**where (TAX1) is (TAX2) and (TAX1) is (TAO),**  
**WHETHER QUALITY1 independent QUALITY2 {d}.**
- d) **WHETHER QUALITY1 independent QUALITY2 {c} :**  
**where QUALITY1 related QUALITY2 {e, f, g, h, i, j, -}, WHETHER false ;**  
**unless QUALITY1 related QUALITY2 {e, f, g, h, i, j, -}, WHETHER true.**
- e) **WHETHER MONO related DUO {d} : WHETHER false.**
- f) **WHETHER DUO related MONO {d} : WHETHER false.**
- g) **WHETHER PRAM related DYADIC {d} : WHETHER false.**
- h) **WHETHER DYADIC related PRAM {d} : WHETHER false.**
- i) **WHETHER procedure with MODE1 parameter MODE2 parameter yielding MOID1 related procedure with MODE3 parameter MODE4 parameter yielding MOID2 {d} :**  
**WHETHER MODE1 firmly related MODE3 {k}**  
**and MODE2 firmly related MODE4 {k}.**
- j) **WHETHER procedure with MODE1 parameter yielding MOID1 related procedure with MODE2 parameter yielding MOID2 {d} :**  
**WHETHER MODE1 firmly related MODE2 {k}.**
- k) **WHETHER MOID1 firmly related MOID2 {i, j} :**  
**WHETHER MOODS1 is firm MOID2 {l, m} or MOODS2 is firm MOID1 {l, m},**  
**where (MOODS1) is (MOID1) or (union of MOODS1 mode) is (MOID1),**  
**where (MOODS2) is (MOID2) or (union of MOODS2 mode) is (MOID2).**
- l) **WHETHER MOODS MOOD is firm MOID {k, l} :**  
**WHETHER MOODS is firm MOID {l, m} or MOOD is firm MOID {m}.**

- m) **WHETHER MOID1 is firm MOID2** {k, l, n, 18<sub>4.7.f</sub>} :  
**WHETHER MOID1 equivalent MOID2** {22<sub>7.3.a</sub>}  
**or MOID1 unites to MOID2** {21<sub>6.4.b</sub>} **or MOID1 deprefs to firm MOID2**  
{n}.
- n) **WHETHER MOID1 deprefs to firm MOID2** {m} :  
**where (MOID1) is (PREF MOID3),**  
**WHETHER MOID5 is firm MOID2** {m},  
**where MOID3 deflexes to MOID5** {18<sub>4.7.a, b, c</sub>};  
**where (MOID1) is (NONPREF), WHETHER false.**

{To prevent the ambiguous application of **indicators**, as in `REAL x, INT x; x := 0`, certain restrictions are imposed on **defining-indicators** contained in a given reach. These are enforced by the syntactic test for "independence" of properties enveloped by a given **'LAYER'** (rules a, b, c). A sufficient condition, not satisfied in the example above, for the independence of a pair of properties, each being some **'QUALITY TAX'**, is that the **'TAX's** differ (rule c). For **'TAX's** which are not some **'TAO'**, this condition is also necessary, so that even `REAL x, INT x; SKIP` is not a **serial-clause**.

For two properties **'QUALITY1 TAO'** and **'QUALITY2 TAO'** the test for independence is more complicated, as is exemplified by the **serial-clause**

```
OP + = (INT i) BOOL: TRUE,
  OP + = (INT i, j) INT: 1,
  OP + = (INT i, BOOL j) INT: 2,
  PRIO + = 6;
  0 + + 0 ç = 2 ç.
```

Ambiguities would be present in

```
PRIO + = 6, + = 7; 1 + 2 * 3 ç 7 or 9 ? ç,
```

in

```
OP Z = (INT i) INT: 1, MODE Z = INT;
  Z i ç formula or declaration? ç; SKIP,
```

and in

```
OP ? = (UNION (REF REAL, CHAR) a) INT: 1, OP ? = (REAL a) INT: 2;
  ? LOC REAL ç 1 or 2 ? ç.
```

In such cases a test is made that the two **'QUALITY's** are independent (rules c, d). A **'MOID TALLY'** is never independent of any **'QUALITY'** (rule d). A **'MONO'** is always independent of a **'DUO'** (rules d, e, f) and both are independent of a **'DYADIC'** (i.e., of a **'priority PRIO'**) (rules d, g, h). In the case of two **'PRAM's** which are both **'MONO'** or both **'DUO'**, ambiguities could arise if the corresponding parameter modes were "firmly related", i.e., if some (pair of) operand mode (s) could be firmly coerced to the (pair of) parameter mode (s) of either **'PRAM'** (rules i, j). In the example with the two definitions of `?`, the two **'PRAM's** are related since the modes specified by `UNION (REF REAL, CHAR)` and by `REAL` are firmly related, the mode specified by `REF REAL` being firmly coercible to

either one.

It may be shown that two modes are firmly related if one of them, or some component **'MOOD'** of one of them, may be firmly coerced to the other (rules k, l), which requires a sequence of zero or more meek coercions followed by at most one uniting {21<sub>6</sub>.4.1.a}. The possibility or otherwise of such a sequence of coercions between two modes is determined by the predicate **'is firm'** (rules m, n).

A **'PROP1'** also renders inaccessible a **'PROP2'** in an outer **'LAYER'** if that **'PROP2'** is not independent of **'PROP1'**; e.g.,

```
BEGIN INT x;
  BEGIN REAL x; ⚡ here the 'PROP1' is 'reference to real letter x' ⚡
    SKIP
  END
END
END
```

and likewise

```
BEGIN OP ? = (INT i) INT: 1, INT k := 2;
  BEGIN OP ? = (REF INT i) INT: 3;
    ? k ⚡ delivers 3, but ? 4 could not occur here because its
      operator is inaccessible ⚡
  END
END. }
```

## 22.2 Identification in nests

{This section ensures that for each **applied-indicator** there is a corresponding property in some suitable **'LAYER'** of the nest.}

### 22.2.1 Syntax

**{PROPSETY :: PROPS ; EMPTY.**  
**PROPS :: PROP ; PROPS PROP.**  
**PROP :: DEC ; LAB ; FIELD.**  
**QUALITY :: MODE ; MOID TALLY ; DYADIC ; label ; MODE field.**  
**TAX :: TAG ; TAB ; TAD ; TAM.}**

- a) **WHETHER PROP identified in NEST new PROPSETY {a, 18<sub>4</sub>.8.b, 19<sub>5</sub>.4.2.a} :**  
**where PROP resides in PROPSETY {b, c, -}, WHETHER true ;**  
**where PROP independent PROPSETY {22<sub>7</sub>.1.a, b, c} ,**  
**WHETHER PROP identified in NEST {a, -}.**

- b) **WHETHER PROP1 resides in PROPS2 PROP2** {a, b, 18<sub>4</sub>.8.d } :  
**WHETHER PROP1 resides in PROP2** {c, -} or **PROP1 resides in PROPS2** {b, c, -}.
- c) **WHETHER QUALITY1 TAX resides in QUALITY2 TAX** {a, b, 18<sub>4</sub>.8.d } :  
**where (QUALITY1) is (label) or (QUALITY1) is (DYADIC)**  
**or (QUALITY1) is (MODE field),**  
**WHETHER (QUALITY1) is (QUALITY2) ;**  
**where (QUALITY1) is (MOID1 TALLETY) and (QUALITY2) is (MOID2 TALLETY),**  
**WHETHER MOID1 equivalent MOID2** {22<sub>7</sub>.3.a } .

{A nest, except the primal one (which is just 'new'), is some 'NEST LAYER' (i.e., some 'NEST new PROPSETY'). A 'PROP' is identified by first looking for it in that 'LAYER' (rule a). If the 'PROP' is some 'label TAX' or 'DYADIC TAX', then a simple match of the 'PROP's is a sufficient test (rule c). If the 'PROP' is some 'MOID TALLETY TAX', then the mode equivalencing mechanism must be invoked (rule c). If it is not found in the 'LAYER', then the search continues with the 'NEST' (without that 'LAYER'), provided that it is independent of all 'PROP's in that 'LAYER'; otherwise the search is abandoned (rule a). Note that rules b and c do double duty in that they are also used to check the validity of **applied-field-selectors** {18<sub>4</sub>.8.1.d}.

## 22.2.2 Semantics

- a) If some **NEST-range** *R* {17<sub>3</sub>.0.1.f} contains an **applied-indicator** *I* {18<sub>4</sub>.8.1.b} of which there is a descendent **where-PROP-identified-in-NEST-LAYER**, but no descendent **where-PROP-identified-in-NEST**, then *R* is the "defining **range**" of that *I*. {Note that 'NEST' is always the nest in force just outside the **range**.}
- b) A **QUALITY-applied-indicator-with-TAX** *I* whose defining **NEST-range** {a} is *R* "identifies" the **QUALITY-NEST-LAYER-defining-indicator-with-TAX** contained in *R*.

{For example, in

```
(ç 1 ç REAL i = 2.0; (ç 2 ç INT i = 1;
(ç 3 ç REAL x; print (i) ç 3 ç) ç 2 ç) ç 1 ç)
```

there are three **ranges**. The **applied-identifier** *i* in `print (i)` is forced, by the syntax, to be an **integral-NEST-new-real-letter-i- new-integral-letter-i- new-reference-to-real-letter-x- applied- identifier- with- letter-i** {18<sub>4</sub>.8.1.b}. Its defining **range** is the **NEST-new-real-letter-i-serial-clause-defining-new-integral-letter-i** {17<sub>3</sub>.2.1.a} numbered ç 2 ç, it identifies the **defining- identifier** *i* contained in `INT i` (not the one in `REAL i`), and its mode is 'integral'.

{By a similar mechanism, a **DYADIC-formula** {19<sub>5</sub>.4.2.1.a} may be said to "identify" that **DYADIC-defining-operator** {18<sub>4</sub>.8.1.a} which determines its priority.}

c) The environ  $E$  "necessary for" a construct  $C$  in an environ  $E_1$  is determined as follows:

If  $E_1$  is the primal environ {16<sub>2</sub>.2.2.a },

then  $E$  is  $E_1$ ;

otherwise, letting  $E_1$  be composed of a locale  $L$  corresponding to some '**PROPSETY**' and another environ  $E_2$ ,

If  $C$  contains any **QUALITY-applied-indicator-with-TAX**

- which does not identify {b} a **defining-indicator** contained in  $C$ ,
  - which is not a **mode-indication** directly descended from a **formal-** or **virtual-declarer**, and
  - which is such that the predicate '**where QUALITY TAX resides in PROPSETY**' {22<sub>7</sub>.2.1.b} holds,
- then  $E$  is  $E_1$ ;
  - otherwise, { $L$  is not necessary for  $C$  and}  $E$  is the environ necessary for  $C$  in  $E_2$ .

{The environ necessary for a construct is used in the semantics of **routine-texts** {19<sub>5</sub>.4.1.2} and in "establishing" {17<sub>3</sub>.2.2.b}. For example, in

```

ç 2 ç PROC VOID pp; INT n;
  (ç 1 ç PROC p = VOID: print (n); pp := p)

```

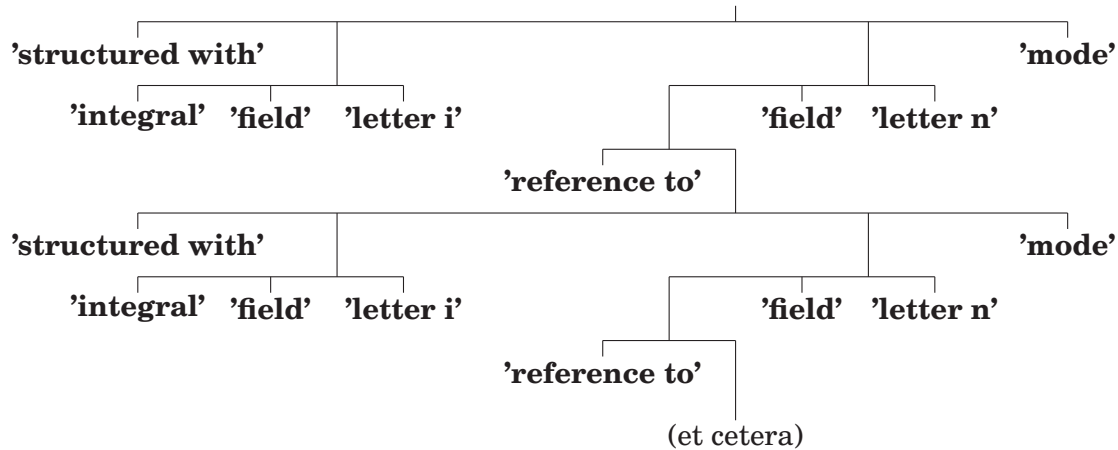
if  $E_1$  and  $E_2$  are the enviros established by the elaboration of the **serial-clauses** marked by the **comments** ç 1 ç and ç 2 ç, then  $E_2$  is the environ necessary in  $E_1$  for the **routine-text** VOID: print (n), and so the routine yielded by p in  $E_1$  is composed of that **routine-text** together with  $E_2$  {19<sub>5</sub>.4.1.2}. Therefore, the scope of that routine is the scope of  $E_2$  {16<sub>2</sub>.1.3.5.c} and hence the assignment {19<sub>5</sub>.2.1.2.b} invoked by pp := p is well defined.}

## 22.3 Equivalence of modes

{The equivalence or nonequivalence of '**MOID**'s is determined in this section. For a discussion of equivalent '**MOID**'s see 16<sub>2</sub>.1.1.2.}

{One way of viewing recursive modes is to consider them as infinite trees. Such a "mode tree" is obtained by repeatedly substituting in some spelling, for each '**MU application**', the '**MODE**' of the corresponding '**MU definition of MODE**'. Thus, the spelling '**mui definition of structured with integral field letter i reference to mui application field letter n mode**' would give rise to the following mode tree:





Two spellings are equivalent if and only if they give rise to identical mode trees. The equivalence syntax tests the equivalence of two spellings by, as it were, simultaneously developing the two trees until a difference is found (resulting in a blind alley) or until it becomes apparent that no difference can be found. The growing production tree reflects to some extent the structure of the mode trees.}

### 22.3.1 Syntax

- A) **SAFE :: safe ; MU has MODE SAFE ; yin SAFE ; yang SAFE ; remember MOID1 MOID2 SAFE.**
- B) **HEAD :: PLAIN ; PREF {227.1.A } ; structured with ; FLEXETY ROWS of ; procedure with ; union of ; void.**
- C) **TAILETY :: MOID ; FIELDS mode ; PARAMETERS yielding MOID ; MOODS mode ; EMPTY.**
- D) **PARTS :: PART ; PARTS PART.**
- E) **PART :: FIELD ; PARAMETER.**
  - a) **WHETHER MOID1 equivalent MOID2 {216.4.b, 227.1.m, 227.2.c} : WHETHER safe MOID1 equivalent safe MOID2 {b}.**
  - b) **WHETHER SAFE1 MOID1 equivalent SAFE2 MOID2 {a, b, e, i, j, n} : where (SAFE1) contains (remember MOID1 MOID2) or (SAFE2) contains (remember MOID2 MOID1), WHETHER true ; unless (SAFE1) contains (remember MOID1 MOID2) or (SAFE2) contains (remember MOID2 MOID1),**

**WHETHER (HEAD3) is (HEAD4)  
and remember MOID1 MOID2 SAFE3 TALETY3  
equivalent SAFE4 TALETY4 {b, d, e, k, q, -},  
where SAFE3 HEAD3 TALETY3 develops from SAFE1 MOID1 {c} and  
SAFE4 HEAD4 TALETY4 develops from SAFE2 MOID2 {c}.**

- c) **WHETHER SAFE2 HEAD TALETY develops from SAFE1 MOID {b, c} :**  
**where (MOID) is (HEAD TALETY),  
WHETHER (HEAD) shields SAFE1 to SAFE2 {227.4.a, b, c, d, -};  
where (MOID) is (MU definition of MODE),  
unless (SAFE1) contains (MU has),  
WHETHER SAFE2 HEAD TALETY develops from MU  
has MODE SAFE1 MODE {c};  
where (MOID) is (MU application)  
and (SAFE1) is (NOTION MU has MODE SAFE3)  
and (NOTION) contains (yin) and (NOTION) contains (yang),  
WHETHER SAFE2 HEAD TALETY develops from SAFE1 MODE {c}.**
- [d] **WHETHER SAFE1 FIELDS1 mode equivalent SAFE2 FIELDS2 mode {b} :**  
**WHETHER SAFE1 FIELDS1 equivalent SAFE2 FIELDS2 {f, g, h, i}.**
- e) **WHETHER SAFE1 PARAMETERS1 yielding MOID1  
equivalent SAFE2 PARAMETERS2 yielding MOID2 {b} :**  
**WHETHER SAFE1 PARAMETERS1 equivalent SAFE2 PARAMETERS2 {f,  
g, h, j}  
and SAFE1 MOID1 equivalent SAFE2 MOID2 {b}.**
- f) **WHETHER SAFE1 PARTS1 PART1 equivalent SAFE2 PARTS2 PART2 {d, e,  
f} :**  
**WHETHER SAFE1 PARTS1 equivalent SAFE2 PARTS2 {f, g, h, i, j}  
and SAFE1 PART1 equivalent SAFE2 PART2 {i, j}.**
- g) **WHETHER SAFE1 PARTS1 PART1 equivalent SAFE2 PART2 {d, e, f} :**  
**WHETHER false.**
- h) **WHETHER SAFE1 PART1 equivalent SAFE2 PARTS2 PART2 {d, e, f} :**  
**WHETHER false.**
- i) **WHETHER SAFE1 MODE1 field TAG1 equivalent SAFE2 MODE2 field TAG2  
{d, f} :**  
**WHETHER (TAG1) is (TAG2) and SAFE1 MODE1 equivalent SAFE2 MODE2  
{b}.**
- j) **WHETHER SAFE1 MODE1 parameter equivalent SAFE2 MODE2 parameter  
{e, f} :**  
**WHETHER SAFE1 MODE1 equivalent SAFE2 MODE2 {b}.**

- k) **WHETHER SAFE1 MOODS1 mode equivalent SAFE2 MOODS2 mode {b} :**  
**WHETHER SAFE1 MOODS1 subset of SAFE2 MOODS2 {l, m, n}**  
**and SAFE2 MOODS2 subset of SAFE1 MOODS1 {l, m, n}**  
**and MOODS1 number equals MOODS2 number {o, p}.**
- l) **WHETHER SAFE1 MOODS1 MOOD1 subset of SAFE2 MOODS2 {k, l, 18<sub>4</sub>.6.s, 21<sub>6</sub>.4.b} :**  
**WHETHER SAFE1 MOODS1 subset of SAFE2 MOODS2 {l, m, n}**  
**and SAFE1 MOOD1 subset of SAFE2 MOODS2 {m, n}.**
- m) **WHETHER SAFE1 MOOD1 subset of SAFE2 MOODS2 MOOD2 {k, l, m, 18<sub>4</sub>.6.s, 21<sub>6</sub>.4.b} :**  
**WHETHER SAFE1 MOOD1 subset of SAFE2 MOODS2 {m, n}**  
**or SAFE1 MOOD1 subset of SAFE2 MOOD2 {n}.**
- n) **WHETHER SAFE1 MOOD1 subset of SAFE2 MOOD2 {k, l, m, 21<sub>6</sub>.4.b} :**  
**WHETHER SAFE1 MOOD1 equivalent SAFE2 MOOD2 {b}.**
- o) **WHETHER MOODS1 MOOD1 number equals MOODS2 MOOD2 number {k, o} :**  
**WHETHER MOODS1 number equals MOODS2 number {o, p, -}.**
- p) **WHETHER MOOD1 number equals MOOD2 number {k, o} : WHETHER true.**
- q) **WHETHER SAFE1 EMPTY equivalent SAFE2 EMPTY {b} : WHETHER true.**

{Rule a introduces the 'SAFE's which are used as associative memories during the determination of equivalence. There are two of them, one belonging to each mode. Rule b draws an immediate conclusion if the 'MOID's under consideration are already remembered (see below) in an appropriate 'SAFE' in the form 'remember MOID1 MOID2'. If this is not the case, then the two 'MOID's are first remembered in a 'SAFE' (the one on the left) and then each 'MOID' is developed (rule c) and split into its 'HEAD' and its 'TAILETY', e.g., 'reference to real' is split into 'reference to' and 'real'.

If the 'HEAD's differ, then the matter is settled (rule b); otherwise the 'TAILETY's are analyzed according to their structure (which must be the same if the 'HEAD's are identical). In each case, except where the 'HEAD's were 'union of', the equivalence is determined by examining the corresponding components, according to the following scheme:

| rule | 'TAILETY'                  | components                   |
|------|----------------------------|------------------------------|
| d    | 'FIELDS mode'              | 'FIELDS'                     |
| e    | 'PARAMETERS yielding MOID' | 'PARAMETERS' and 'MOID'      |
| f    | 'FIELDS FIELD'             | 'FIELDS' and 'FIELD'         |
| f    | 'PARAMETERS PARAMETER'     | 'PARAMETERS' and 'PARAMETER' |
| i    | 'MODE field TAG'           | 'MODE' and 'TAG'             |
| j    | 'MODE parameter'           | 'MODE'                       |

In the case of unions, the **'TAILETY's** are of the form **'MOODS1 mode'** and **'MOODS2 mode'**. Since **'MOOD's** within equivalent unions may commute, as in the modes specified by UNION (REAL, INT) and UNION (INT, REAL), the equivalence is determined by checking that **'MOODS1'** is a subset of **'MOODS2'** and that **'MOODS2'** is a subset of **'MOODS1'**, where the subset test, of course, invokes the equivalence test recursively (rules k, l, m, n, o, p).

A **'MOID'** is developed (rule c) into the form **'HEAD TAILETY'** by determining that

it is already of that form: in which case markers (**'yin'** and **'yang'**) may be placed in its **'SAFE'** for the later determination of well-formedness (see 22<sub>7.4</sub>);

it is some **'MU definition of MODE'**: in which case **'MU has MODE'** is stored in its **'SAFE'** (provided that this particular **'MU'** is not there already) and the **'MODE'** is developed;

it is some **'MU application'**: in which case there must be some **'MU has MODE'** in its **'SAFE'** already. That **'MODE'** is then developed after a well-formedness check (see 22<sub>7.4</sub>) consisting of the determination that there is at least one **'yin'** and at least one **'yang'** in the **'SAFE'** which is more recent than the **'MU has MODE'**.

}

{Before a pair of **'TAILETY's** is tested for equivalence, it is remembered in the **'SAFE'** that the original pair of **'MOID's** is being tested. This is used to force a shortcut to **'WHETHER true'** if these **'MOID's** should ever be tested again for equivalence lower down the production tree. Since the number of pairs of component **'MOID's** that can be derived from any two given **'MOID's** is finite, it follows that the testing process terminates.

It remains to be shown that the process is correct. Consider the unrestricted (possibly infinite) production tree that would be obtained if there were no shortcut in the syntax (by omitting the first alternative together with the first member of the other alternative of rule b). If two **'MOID's** are not equivalent, then there exists in their mode trees a shortest path from the top node to some node exhibiting a difference. Obviously, the reflection of this shortest path in the unrestricted production tree cannot contain a repeated test for the equivalence of any pair of **'MOID's**, and therefore none of the shortcuts to **'WHETHER true'** in the restricted production tree can occur on this shortest path. Consequently, the path to the difference must be present also in the (restricted) production tree produced by the syntax. If the testing process does not exhibit a difference in the restricted tree, then no difference can be found in any number of steps: i.e., the **'MOID's** are equivalent.}

## 22.4 Well-formedness

{A mode is well formed if

- (i) the elaboration of an **actual-declarer** specifying that mode is a finite action (i.e., any value of that mode can be stored in a finite memory)  
and
- (ii) it is not strongly coercible from itself (since this would lead to ambiguities in coercion).

}

### 22.4.1 Syntax

- a) **WHETHER (NOTION) shields SAFE to SAFE {73c} :**  
  - where (NOTION) is (PLAIN)
  - or (NOTION) is (FLEXETY ROWS of)
  - or (NOTION) is (union of)
  - or (NOTION) is (void),**WHETHER true.**
- b) **WHETHER (PREF) shields SAFE to yin SAFE {73c} : WHETHER true.**
- c) **WHETHER (structured with) shields SAFE to yang SAFE {73c} :**  
**WHETHER true.**
- d) **WHETHER (procedure with) shields SAFE to yin yang SAFE {73c} :**  
**WHETHER true.**

{As a by-product of mode equivalencing, modes are tested for well-formedness {22<sub>7</sub>.3.1.c}. All nonrecursive modes are well formed. For recursive modes, it is necessary that each cycle in each spelling of that mode (from 'MU definition of MODE' to 'MU application') passes through at least one 'HEAD' which is yin, ensuring condition (i) and one (possibly the same) 'HEAD' which is yang, ensuring condition (ii). Yin 'HEAD's are 'PREF' and 'procedure with'. Yang 'HEAD's are 'structured with' and 'procedure with'. The other 'HEAD's, including 'FLEXETY ROWS of' and 'union of', are neither yin nor yang. This means that the modes specified by A, B and C in

```
MODE A = STRUCT (INT n, REF A next),
B = STRUCT (PROC B next),
C = PROC (C) C
```

are all well formed. However,

```
MODE D = [1 : 10] D,
E = UNION (INT, E)
```

is not a **mode-declaration**.)

## ALGOL 68 REVISED REPORT

{Tao produced the one.  
The one produced the two.  
The two produced the three.  
And the three produced the ten thousand  
things.  
The ten thousand things carry the yin and  
embrace the yang, and through the  
blending of the material force they achieve  
harmony.  
Tao-te Ching, 42, Lao Tzu. }

## **PART IV. Elaboration-independent constructions**

# ALGOL 68 REVISED REPORT



# Chapter 23

## Denotations

{**Denotations**, e.g., 3.14 or "abc", are constructs whose yields are independent of any action. In other languages, they are sometimes termed "literals" or "constants".}

### 23.0.1. Syntax

- a) **MOID NEST denoter** { $19_5.D$ ,  $25_{10}.3.4.1.i$ } :  
    **pragment** {24<sub>9</sub>.2.a } **sequence option**,  
    **MOID denotation** {23<sub>8</sub>.1.0.a, 23<sub>8</sub>.1.1.a, 23<sub>8</sub>.1.2.a, 23<sub>8</sub>.1.3.a,  
    23<sub>8</sub>.1.4.a, 23<sub>8</sub>.1.5.a, 23<sub>8</sub>.2.a, b, c, 17<sub>3</sub>.3.a, -}.

{The meaning of a **denotation** is independent of any nest.}

## 23.1 Plain denotations

{**Plain-denotations** are those of arithmetic values, truth values, characters and the void value, e.g., 1, 3.14, true, "a" and EMPTY.}

### 23.1.0.1. Syntax

- A) **SIZE :: long ; short.**
- B) **\*NUMERAL ::**  
    **fixed point numeral ; variable point numeral ; floating point numeral.**
- a) **SIZE INTREAL denotation** {a, 23<sub>8</sub>.0.a } :  
    **SIZE symbol** {24<sub>9</sub>.4.d } , **INTREAL denotation** {a, 811a, 812a} . b) **\*plain denotation :**  
    **PLAIN denotation** {a, 811a, 812a, 813a, 814a} ; **void denotation** {815a} .

{Example:

a) LONG 0 }

### 23.1.0.2. Semantics

The yield  $W$  of an **INTREAL-denotation** is the "intrinsic value" {23<sub>8</sub>.1.1.2, 23<sub>8</sub>.1.2.2.a, b} of its constituent **NUMERAL**;

- it is required that  $W$  be not greater than the largest value of mode '**INTREAL**' that can be distinguished {16<sub>2</sub>.1.3.1.d}.

{An **INTREAL-denotation** yields an arithmetic value {16<sub>2</sub>.1.3.1.a}, but arithmetic values yielded by different **INTREAL-denotations** are not necessarily different (e.g., 123 . 4 and 1 . 234<sub>10</sub>+2). }

## 23.1.1 Integral denotations

### 23.1.1.1 Syntax

- a) **integral denotation** {23<sub>8</sub>.0.a, 810a} : **fixed point numeral** {b}.
- b) **fixed point numeral** {a, 23<sub>8</sub>.1.2.c, d, f, i, 25<sub>10</sub>.3.4.1.h} : **digit cypher** {c} **sequence**.
- c) **digit cypher** {b} : **DIGIT symbol** {24<sub>9</sub>.4.b} .

{Example:

a) 4096

b) 4096

c) 4 }

### 23.1.1.2 Semantics

The intrinsic value of a **fixed-point-numeral**  $N$  is the integer of which the reference-language form of  $N$  {24<sub>9</sub>.3.b} is a decimal representation.

## 23.1.2 Real denotations

### 23.1.2.1 Syntax

- a) **real denotation** {23<sub>8</sub>.0.a, 810a} : **variable point numeral** {b}; **floating point numeral** {e}.
- b) **variable point numeral** {a, f} :  
**integral part** {c} **option**, **fractional part** {d}.
- c) **integral part** {b} : **fixed point numeral** {23<sub>8</sub>.1.1.b } .
- d) **fractional part** {b} : **point symbol** {24<sub>9</sub>.4.b } , **fixed point numeral** {23<sub>8</sub>.1.1.b } .
- e) **floating point numeral** {a} : **stagnant part** {f}, **exponent part** {g}.
- f) **stagnant part** {e} :  
**fixed point numeral** {23<sub>8</sub>.1.1.b } ; **variable point numeral** {b}.
- g) **exponent part** {e} : **times ten to the power choice** {h}, **power of ten** {i}.
- h) **times ten to the power choice** {g} :  
**times ten to the power symbol** {24<sub>9</sub>.4.b } ; **letter e symbol** {24<sub>9</sub>.4.a } .
- i) **power of ten** {g} : **plusminus** {j} **option**, **fixed point numeral** {811b } .
- j) **plusminus** {i} : **plus symbol** {24<sub>9</sub>.4.c } ; **minus symbol** {24<sub>9</sub>.4.c } .

{Example:

- a) 0.00123 • 1.23e-3
- b) 0.00123
- c) 0
- d) .00123
- e) 1.23e-3
- f) 123 • 1.23
- g) e-3
- h) <sub>10</sub> • e
- i) -3
- j) + • - }

### 23.1.2.2 Semantics

a) The intrinsic value  $V$  of a **variable-point-numeral**  $N$  is determined as follows:

- let  $I$  be the intrinsic value of the **fixed-point-numeral** of its constituent **integral-part**, if any, and be 0 otherwise;
- let  $F$  be the intrinsic value of the **fixed-point-numeral** of its **fractional-part**  $P$  divided by 10 as many times as there are **digit-cyphers** contained in  $P$ ;
- $V$  is the sum in the sense of numerical analysis of  $I$  and  $F$ .

b) The intrinsic value  $V$  of a **floating-point-numeral**  $N$  is determined as follows:

- let  $S$  be the intrinsic value of the **NUMERAL** of its **stagnant-part**;
- let  $E$  be the intrinsic value of the constituent **fixed-point-numeral** of its **exponent-part**;

Case A: The constituent **plusminus-option** of  $N$  contains a **minus-symbol**:

- $V$  is the product in the sense of numerical analysis of  $S$  and  $1/10$  raised to the power  $E$ ;

Case B: The direct descendent of that **plusminus-option** contains a **plus-symbol** or is empty:

- $V$  is the product in the sense of numerical analysis of  $S$  and  $10$  raised to the power  $E$ .

## 23.1.3 Boolean denotations

### 23.1.3.1 Syntax

a) **boolean denotation** {23<sub>8</sub>.0.a } : **true** {24<sub>9</sub>.4.b} **symbol** ; **false** {24<sub>9</sub>.4.b } **symbol**.

{Example:

a) TRUE • FALSE }

### 23.1.3.2 Semantics

The yield of a **boolean-denotation** is **true** (**false**) if its direct descendent is a **true-symbol** (**false-symbol**).

### 23.1.4 Character denotations

{**Character-denotations** consist of a **string-item** between two **quote-symbols**, e.g., "a". To indicate a quote, a **quote-image-symbol** (represented by " ") is used, e.g., "" ". Since the syntax nowhere allows **character-** or **string-denotations** to follow one another, this causes no ambiguity.}

#### 23.1.4.1 Syntax

- a) **character denotation** {23<sub>8</sub>.0.a } :  
**quote** {24<sub>9</sub>.4.b } **symbol**, **string item** {b}, **quote symbol** {24<sub>9</sub>.4.b } .
- b) **string item** {a, 23<sub>8</sub>.3.b } :  
**character glyph** {c};  
**quote image symbol** {24<sub>9</sub>.4.b } ;  
**other string item** {d}.
- c) **character glyph** {b, 24<sub>9</sub>.16<sub>2</sub>.c } :  
**LETTER symbol** {24<sub>9</sub>.4.a } ; **DIGIT symbol** {24<sub>9</sub>.4.b } ; **point symbol** {24<sub>9</sub>.4.b } ;  
**open symbol** {24<sub>9</sub>.4.f } ; **close symbol** {24<sub>9</sub>.4.f } ; **comma symbol** {24<sub>9</sub>.4.b } ;  
**space symbol** {24<sub>9</sub>.4.b } ; **plus symbol** {24<sub>9</sub>.4.c } ; **minus symbol** {24<sub>9</sub>.4.c } .
- d) A production rule may be added for the notion '**other string item**' {b, for which no hyper-rule is given in this Report} each of whose alternatives is a symbol {15<sub>1</sub>.1.3.1.f} which is different from any terminal production of '**character glyph**' {c} and which is not the '**quote symbol**'.

{Example:

- a) "a"
- b) a • " " • ?
- c) a • 1 • . • ( • ) • , • \_ • + • - }

#### 23.1.4.2 Semantics

- a) The yield of a **character-denotation** is the intrinsic value of the **symbol** descended from its **string-item**.
- b) The intrinsic value of each distinct **symbol** descended from a **string-item** is a unique character. {Characters have no inherent meaning, except insofar as some of them are

interpreted in particular ways by the transport declarations {25<sub>10</sub>.3 }. The **character-glyphs**, which include all the characters needed for transport, form a minimum set which all implementations {16<sub>2</sub>.2.2.c} are expected to provide.}

## 23.1.5 Void denotation

{A **void-denotation** may be used to assign a void value to a **UNITED-variable**, e.g.,  
UNION ([ ]REAL, VOID) u := EMPTY.}

### 23.1.5.1 Syntax

- a) **void denotation** {23<sub>8</sub>.0.a } : **empty** {24<sub>9</sub>.4.b} **symbol**.

{Example:

- a) EMPTY }

### 23.1.5.2 Semantics

The yield of a **void-denotation** is empty.

## 23.2 Bits denotations

### 23.2.1 Syntax

- A) **RADIX :: radix two ; radix four ; radix eight ; radix sixteen.**

- a) **structured with row of boolean field**

**LENGTH LENGTHEY letter aleph mode denotation** {a, 23<sub>8</sub>.0.a } :  
**long** {24<sub>9</sub>.4.d} **symbol, structured with row of boolean field**  
**LENGTHEY letter aleph mode denotation** {a, c}.

- b) **structured with row of boolean field**

**SHORTH SHORTHETY letter aleph mode denotation** {b, 23<sub>8</sub>.0.a } :  
**short** {24<sub>9</sub>.4.d} **symbol, structured with row of boolean field**  
**SHORTHETY letter aleph mode denotation** {b, c}.

- c) **structured with row of boolean field letter aleph mode denotation**  $\{a, b, 23_8.0.a\}$  :  
**RADIX**  $\{d, e, f, g\}$ , **letter r symbol**  $\{24_9.4.a\}$  , **RADIX digit**  $\{h, i, j, k\}$  **sequence.**
- d) **radix two**  $\{c, 25_{10}.3.4.7.b\}$  : **digit two**  $\{24_9.4.b\}$  **symbol.**
- e) **radix four**  $\{c, 25_{10}.3.4.7.b\}$  : **digit four**  $\{24_9.4.b\}$  **symbol.**
- f) **radix eight**  $\{c, 25_{10}.3.4.7.b\}$  : **digit eight**  $\{24_9.4.b\}$  **symbol.**
- g) **radix sixteen**  $\{c, 25_{10}.3.4.7.b\}$  : **digit one symbol**  $\{24_9.4.b\}$  , **digit six symbol**  $\{24_9.4.b\}$  .
- h) **radix two digit**  $\{c, i\}$  : **digit zero symbol**  $\{24_9.4.b\}$  ; **digit one symbol**  $\{24_9.4.b\}$  .
- i) **radix four digit**  $\{c, j\}$  : **radix two digit**  $\{h\}$  ;  
**digit two symbol**  $\{24_9.4.b\}$  ; **digit three symbol**  $\{24_9.4.b\}$  .
- j) **radix eight digit**  $\{c, k\}$  : **radix four digit**  $\{i\}$  ;  
**digit four symbol**  $\{24_9.4.b\}$  ; **digit five symbol**  $\{24_9.4.b\}$  ;  
**digit six symbol**  $\{24_9.4.b\}$  ; **digit seven symbol**  $\{24_9.4.b\}$  .
- k) **radix sixteen digit**  $\{c\}$  : **radix eight digit**  $\{j\}$  ;  
**digit eight symbol**  $\{24_9.4.b\}$  ; **digit nine symbol**  $\{24_9.4.b\}$  ;  
**letter a symbol**  $\{24_9.4.a\}$  ; **letter b symbol**  $\{24_9.4.a\}$  ;  
**letter c symbol**  $\{24_9.4.a\}$  ; **letter d symbol**  $\{24_9.4.a\}$  ;  
**letter e symbol**  $\{24_9.4.a\}$  ; **letter f symbol**  $\{24_9.4.a\}$  .
- l) **\*bits denotation : BITS denotation**  $\{a, b, c\}$ .  
**{BITS :: structured with row of boolean field SITHETY letter aleph mode.}**
- m) **\*radix digit : RADIX digit**  $\{h, i, j, k\}$ .

{Example:

- a) LONG 2r101  
b) SHORT 16rffff  
c) 8r231 }

## 23.2.2 Semantics

a) The yield  $V$  of a **bits-denotation**  $D$  is determined as follows:

- let  $W$  be the intrinsic boolean value  $\{b\}$  of its constituent **RADIX-digit-sequence**:

- let  $m$  be the length of  $W$ ;
- let  $n$  be the value of  $\{l\}$  bits width  $\{25_{10}.2.1.j\}$ , where  $\{l\}$  stands for as many times long (short) as there are **long-symbols (short-symbols)** contained in  $D$ ;
- it is required that  $m$  be not greater than  $n$ ;
- $V$  is a structured value {whose mode is some 'BITS'} whose only field is a multiple value having
  - (i) a descriptor  $((1, n))$  and
  - (ii)  $n$  elements, that selected by (i) being false if  $1 \leq i \leq n - m$ , and being the  $(i + m - n)^{th}$  truth value of {the sequencer}  $W$  otherwise.

b) The intrinsic boolean value of a **RADIX-digit-sequence**  $S$  is the shortest sequence of truth values which, regarded as a binary number (true corresponding to 1 and false to 0), is the same as the intrinsic integral value  $\{c\}$  of  $S$ .

c) The intrinsic integral value of a **radix-two- (radix-four-, radix-eight-, radix-sixteen-) -digit-sequence**  $S$  is the integer of which the reference-language form of  $S$   $\{24_9.3.b\}$  is a binary, (quaternary, octal, hexadecimal) representation, where the representations a, b, c, d, e and f, considered as digits, have values 10, 11, 12, 13, 14 and 15 respectively.

## 23.3 String denotations

{**String-denotations** are a convenient way of specifying "strings", i.e., multiple values of mode 'row of character'.

{Example:

```
STRING message := "all is well" }
```

### 23.3.1 Syntax

- a) **row of character denotation**  $\{23_8.0.a\}$  :  
**quote**  $\{24_9.4.b\}$  **symbol, string**  $\{b\}$  **option, quote symbol**  $\{24_9.4.b\}$  .
- b) **string**  $\{a\}$  : **string item**  $\{23_8.1.4.b\}$  , **string item**  $\{23_8.1.4.b\}$  **sequence**.
- c) **\*string denotation : row of character denotation**  $\{a\}$ .

{Example:



- a) "abc"
- b) abc }

### 23.3.2 Semantics

The yield of a **string-denotation**  $D$  is a multiple value  $V$  determined as follows:

- let  $n$  be the number of **string-items** contained in  $D$ ;
- the descriptor of  $V$  is  $((1, n))$ ;
- for  $i = 1, \dots, n$ , the element of  $V$  with index  $(i)$  is the intrinsic value {23<sub>8</sub>.1.4.2.b} of the  $i^{\text{th}}$  constituent **symbol** of the string of  $D$ .

{"a" is a **character-denotation**, not a **string-denotation**. However, in all strong positions, e.g., `STRING s := "a"`, it can be rowed to a multiple value {21<sub>6</sub>.6}. Elsewhere, where a multiple value is required, a **cast** {19<sub>5</sub>.5.1.1.a} may be used, e.g., `UNION (CHAR, STRING) cs := STRING ("a").`}

# ALGOL 68 REVISED REPORT

# Chapter 24

## Tokens and symbols

### 24.1 Tokens

{**Tokens** {24<sub>9</sub>.1.1.f} are **symbols** {24<sub>9</sub>.1.1.h} possibly preceded by **pragments** {24<sub>9</sub>.2.1.a}. Therefore, **pragments** may appear between **symbols** wherever the syntax produces a succession of **tokens**. However, in a few places, the syntax specifically produces **symbols** rather than **tokens**, notably within **denotations** {8}, **format-texts** {25<sub>10</sub>.3.4.1.1.a} and, of course, within **pragments**. Therefore, **pragments** may not occur in these places.}

#### 24.1.1 Syntax

- a) **CHOICE STYLE start** {17<sub>3</sub>.4.a } :  
    **where (CHOICE) is (choice using boolean),**  
    **STYLE if** {24<sub>9</sub>.4.f, -} **token ;**  
    **where (CHOICE) is (CASE),**  
    **STYLE case** {24<sub>9</sub>.4.f, -} **token.**
- b) **CHOICE STYLE in** {17<sub>3</sub>.4.e } :  
    **where (CHOICE) is (choice using boolean),**  
    **STYLE then** {24<sub>9</sub>.4.f, -} **token ;**  
    **where (CHOICE) is (CASE),**  
    **STYLE in** {24<sub>9</sub>.4.f, -} **token.**
- c) **CHOICE STYLE again** {17<sub>3</sub>.4.l } :  
    **where (CHOICE) is (choice using boolean),**  
    **STYLE else if** {24<sub>9</sub>.4.f, -} **token ;**  
    **where (CHOICE) is (CASE),**  
    **STYLE ouse** {24<sub>9</sub>.4.f, -} **token.**

- d) **CHOICE STYLE out** {17<sub>3</sub>.4.1 } :  
     **where (CHOICE) is (choice using boolean),**  
     **STYLE else** {24<sub>9</sub>.4.f, -} **token ;**  
     **where (CHOICE) is (CASE),**  
     **STYLE out** {24<sub>9</sub>.4.f, -} **token.**
- e) **CHOICE STYLE finish** {17<sub>3</sub>.4.a } :  
     **where (CHOICE) is (choice using boolean),**  
     **STYLE fi** {24<sub>9</sub>.4.f, -} **token ;**  
     **where (CHOICE) is (CASE),**  
     **STYLE esac** {24<sub>9</sub>.4.f, -} **token.**
- f) **NOTION token** :  
     **pragmat** {24<sub>9</sub>.2.a } **sequence option,**  
     **NOTION symbol** {24<sub>9</sub>.4.a, b, c, d, e, f, g, h} .
- g) **\*token : NOTION token** {f}.
- h) **\*symbol : NOTION symbol** {24<sub>9</sub>.4.a, b, c, d, e, f, g, h} .

## 24.2 Comments and pragmats

{A source of innocent merriment.  
Mikado, W.S. Gilbert. }

{A **pragmat** is a **comment** or a **pragmat**. No semantics of **pragmats** is given and therefore the meaning {16<sub>2</sub>.1.4.1.a} of any **program** is quite unaffected by their presence. It is indeed the intention that **comments** should be entirely ignored by the implementation, their sole purpose being the enlightenment of the human interpreter of the **program**.

**Pragmats** may, on the other hand, convey to the implementation some piece of information affecting some aspect of the meaning of the **program** which is not defined by this Report, for example:

- the action to be taken upon overflow {16<sub>2</sub>.1.4.3.h} or if the scope rule is violated (as in 19<sub>5</sub>.2.1.2.b ), e.g., PR overflow check on PR, PR overflow check off PR, PR scope check on PR or PR scope check off PR;
- the action to be taken upon completion of the compilation process, e.g., PR compile only PR, PR dump PR or PR run PR;
- that the language to be implemented is some sublanguage or superlanguage of Algol 68, e.g., PR nonrec PR (for a **routine-text** which may be presumed to be non-recursive);

- that the compilation may check for the truth, or attempt to prove the correctness, of some assertion, e.g.:

```
INT a, b; read ((a, b)) PR ASSERT a ≥ 0 ∧ b > 0 PR
INT q := 0, r := a;
WHILE r ≥ b PR ASSERT a = b × q + r ∧ 0 < r PR
DO (q +:= 1, r -:= b) OD
PR ASSERT a = b × q + r ∧ 0 ≤ r ∧ r < b PR.
```

They may also be used to convey to the implementation that the source text is to be augmented with some other text, or edited in some way, for example:

- some previously compiled portion of the **particular-program** is to be invoked, e.g.,  
PR WITH segment FROM album PR;
- the source text is continued on some other document, e.g., PR READ FROM another file PR;
- the end of the source text has been reached, e.g., PR FINISH PR.

The interpretation of **pragmats** is not defined in this Report, but is left to the discretion of the implementer, who ought, at least, to provide some means whereby all further **pragmats** may be ignored, for example:

```
PR pragmats off PR. }

PR algol 68 PR
BEGIN
  PROC PR nonrec PR pr = VOID: pr;
  pr
END PR run PR PR ? PR
Revised Report on the Algorithmic Language
Algol 68. }
```

## 24.2.1 Syntax

- A) **PRAGMENT :: pragmat ; comment.**
- a) **pragment** {80a, 91f, A341b, h, A348a, b, c, A349a, A34Ab} : **PRAGMENT** {b}.
- b) **PRAGMENT {a} :**  
**STYLE PRAGMENT symbol** {24<sub>9</sub>.4.b, -},  
**STYLE PRAGMENT item {c} sequence option,**  
**STYLE PRAGMENT symbol** {24<sub>9</sub>.4.b, -}.  
**{STYLE :: brief ; bold ; style TALLY.}**

- c) **STYLE PRAGMENT item** {b} : **character glyph** {23<sub>8</sub>.1.4.c } ; **STYLE other PRAGMENT item** {d}.
- d) A production rule may be added for each notion designated by '**STYLE other PRAGMENT item**' {c, for which no hyper-rule is given in this Report} each of whose alternatives is a symbol {15<sub>1</sub>.1.3.1.f}, different from any terminal production of '**character glyph**' {23<sub>8</sub>.1.4.1.c}, and such that no terminal production of any '**STYLE other PRAGMENT item**' is the corresponding '**STYLE PRAGMENT symbol**'. {Thus COMMENT # COMMENT might be a **comment**, but #; #; #; could not.}

{Examples:

- a) PR list PR • ç source program to be listed ç
- c) 1 • ? }

## 24.3 Representations

a) A construct in the strict language must be represented in some "representation language" such as the "reference language", which is used in this Report. Other representation languages specially suited to the supposed preference of some human or mechanical interpreter of the language may be termed "publication" or "hardware" languages. {The reference language is intended to be used for the representation of **particular-programs** and of their descendents. It is, however, also used in Chapter 25<sub>10</sub> for the definition of the standard environment.}

b) A "construct in a representation language" is obtained from the terminal production  $T$  {15<sub>1</sub>.1.3.2.f} of the corresponding construct in the strict language {15<sub>1</sub>.1.3.2.e} by replacing the **symbols** in  $T$  by their representations, as specified in 24<sub>9</sub>.4 below in the case of the reference language.

{Thus, the strict-language **particular-program** whose terminal production is '**bold begin symbol**' '**skip symbol**' '**bold end symbol**' gives rise to the reference language **particular-program** BEGIN SKIP END.}

c) An implementation {16<sub>2</sub>.2.2.c} of Algol 68 which uses representations which are sufficiently close to those of the reference language to be recognized without further elucidation, and which does not augment or restrict the available representations other than as provided for below {24<sub>9</sub>.4.a, b, c }, is an "implementation of the reference language".

{E.g., BEGIN, begin, BEGIN, 'begin and 'begin' could all be representations of the **bold-begin-symbol** in an implementation of the reference language; some combination of holes in a punched card might be a representation of it in some hardware language.}

## 24.4 The reference language

a) The reference language provides representations for various **symbols**, including an arbitrarily large number of **TAX-symbols** {where **TAX** :: **TAG** ; **TAB** ; **TAD** ; **TAM**.} . The representations of some of them are specified below {24<sub>9</sub>.4.1 }, and to these may be added suitable representations for **style-TALLY-letter-ABC-symbols** and **style-TALLY-monad-symbols** and any terminal productions of '**STYLE other PRAGMENT item**' {24<sub>9</sub>.2.1.d} and of '**other string item**' {23<sub>8</sub>.1.4.1.d}. Representations are not provided for any of these {but they enable individual implementations to make available their full character sets for use as characters, to provide additional or extended alphabets for the construction of **TAG-** and **TAB-symbols**, and to provide additional **symbols** for use as **operators**} . There is not, however, {and there must not be, } except in representations of the **standard-**, and other, **preludes** {25<sub>10</sub>.1.3.Step 6 }, any representation of the **letter-aleph-symbol** or the **primal-symbol**. {For the remaining **TAX-symbols**, see 24<sub>9</sub>.4.2 . There are, however, some **symbols** produced by the syntax, e.g., the **brief-pragmat-symbol**, for which no representation is provided at all. This does not preclude the representation of such **symbols** in other representation languages.}

b) Where more than one representation of a **symbol** is given, any of them may be chosen. Moreover, it is sufficient for an implementation of the reference language to provide only one. Also, it is not necessary to provide a representation of any particular **MONAD-symbol** or **NOMAD-symbol** so long as those that are provided are sufficient to represent at least one version {25<sub>10</sub>.1.3.Step 3 } of each **operator** declared in the **standard-prelude**.

{For certain different **symbols**, one same or nearly the same representation is given; e.g., the representation ":" is given for the **routine-symbol**, the **colon-symbol** and the **up-to-symbol** and "." for the **label-symbol**. It follows uniquely from the syntax which of these four **symbols** is represented by an occurrence, outside **comments**, **pragmats** and **string-denotations**, of any mark similar to either of those representations. It is also the case that ". ." could be used, without ambiguity, for any of them, and such might indeed be necessary in implementations with limited character sets. It may be noted that, for such implementations, no ambiguity would be introduced were " (/ " and " / )" to be used as representations of the **style-ii-sub-symbol** and the **style-ii-bus-symbol**, respectively.

Also, some of the given representations appear to be composite; e.g., the representation " : =" of the **becomes-symbol** appears to consist of " : ", the representation of the **routine-symbol**, etc., and " =", the representation of the **equals-symbol** and of the **is-defined-as-symbol**. It follows from the syntax that " : =" can occur, outside **comments**, **pragmats** and **string-denotations**, as a representation of the **becomes-symbol** only (since " =" cannot occur as the representation of a **monadic-operator**). Similarly, the other given composite representations do not cause ambiguity.}

c) The fact that the representations of the **letter-ABC-symbols** given {24<sub>9</sub>.4.1.a} are usually spoken of as small letters is not meant to imply that the corresponding capital letters could not serve equally well. {On the other hand, if both a small letter and the corre-

sponding capital letter occur, then one of them is presumably the representation of some **style-TALLY-letter-ABC-symbol** or of a **bold-letter-ABC-symbol**. See also 15<sub>1</sub>.1.5.b for the possibility of additional 'ABC's in a variant of the language.)

d) A "typographical display feature" is a blank, or a change to a new line or a new page. Such features, when they appear between the **symbols** of a construct in the reference language, are of no significance and do not affect the meaning of that construct. However, a blank contained within a **string-** or **character-denotation** is one of the representations of the **space-symbol** {24<sub>9</sub>.4.1.b } rather than a typographical display feature. Where the representation of a **symbol** in the reference language is composed of several marks {e.g., TO, :=}, those marks form one {indivisible} **symbol** and, unless the contrary is explicitly stated {24<sub>9</sub>.4.2.2.a, c }, typographical display features may not separate them.

## 24.4.1 Representations of symbols

### a) Letter symbols

| symbol   | representation |
|--|----------------|
| <b>letter a symbol</b> {23 <sub>8</sub> .1.4.c, 23 <sub>8</sub> .2.k, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.6.b}                         | a              |
| <b>letter b symbol</b> {23 <sub>8</sub> .1.4.c, 23 <sub>8</sub> .2.k, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.4.b}                         | b              |
| <b>letter c symbol</b> {23 <sub>8</sub> .1.4.c, 23 <sub>8</sub> .2.k, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.8.a}                         | c              |
| <b>letter d symbol</b> {23 <sub>8</sub> .1.4.c, 23 <sub>8</sub> .2.k, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.2.f}                         | d              |
| <b>letter e symbol</b> {23 <sub>8</sub> .1.2.h, 23 <sub>8</sub> .1.4.c, 23 <sub>8</sub> .2.k, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.3.e} | e              |
| <b>letter f symbol</b> {23 <sub>8</sub> .1.4.c, 23 <sub>8</sub> .2.k, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.9.a}                         | f              |
| <b>letter g symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.25 <sub>10</sub> .a}                               | g              |
| <b>letter h symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B}  | h              |
| <b>letter i symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.5.b}   | i              |
| <b>letter j symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B}  | j              |
| <b>letter k symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.1.f}   | k              |
| <b>letter l symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.1.f}   | l              |
| <b>letter m symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B}  | m              |
| <b>letter n symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.1.h}   | n              |
| <b>letter o symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B}  | o              |
| <b>letter p symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.1.f}   | p              |
| <b>letter q symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.1.f}   | q              |
| <b>letter r symbol</b> {23 <sub>8</sub> .1.4.c, 23 <sub>8</sub> .2.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.7.c}                         | r              |
| <b>letter s symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.1.l}   | s              |
| <b>letter t symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B}  | t              |



|  |   |
|--|---|
| <b>letter u symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B}                            | u |
| <b>letter v symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B}                            | v |
| <b>letter w symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B}                            | w |
| <b>letter x symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.1.f} | x |
| <b>letter y symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.1.f} | y |
| <b>letter z symbol</b> {23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.B, 25 <sub>10</sub> .3.4.2.d} | z |

b) Denotation symbols

| symbol  | representation |
|---|----------------|
| <b>digit zero symbol</b> {811c, 814c, 82h, 24 <sub>9</sub> .4.2.C}          | 0              |
| <b>digit one symbol</b> {43b, 811c, 814c, 82g, h, 24 <sub>9</sub> .4.2.C}   | 1              |
| <b>digit two symbol</b> {43b, 811c, 814c, 82d, i, 24 <sub>9</sub> .4.2.C}   | 2              |
| <b>digit three symbol</b> {43b, 811c, 814c, 82i, 24 <sub>9</sub> .4.2.C}    | 3              |
| <b>digit four symbol</b> {43b, 811c, 814c, 82e, j, 24 <sub>9</sub> .4.2.C}  | 4              |
| <b>digit five symbol</b> {43b, 811c, 814c, 82j, 24 <sub>9</sub> .4.2.C}     | 5              |
| <b>digit six symbol</b> {43b, 811c, 814c, 82g, j, 24 <sub>9</sub> .4.2.C}   | 6              |
| <b>digit seven symbol</b> {43b, 811c, 814c, 82j, 24 <sub>9</sub> .4.2.C}    | 7              |
| <b>digit eight symbol</b> {43b, 811c, 814c, 82f, k, 24 <sub>9</sub> .4.2.C} | 8              |
| <b>digit nine symbol</b> {43b, 811c, 814c, 82k, 24 <sub>9</sub> .4.2.C}     | 9              |
| <b>point symbol</b> {812d, 814c, A343d}                                     | .              |
| <b>times ten to the power symbol</b> {812h}                                 | 10             |
| <b>true symbol</b> {813a}   | TRUE           |
| <b>false symbol</b> {813a}  | FALSE          |
| <b>quote symbol</b> {814a, 83a}   | "              |
| <b>quote image symbol</b> {814b}  | ""             |
| <b>space symbol</b> {23 <sub>8</sub> .1.4.c}                                | ␣              |
| <b>comma symbol</b> {23 <sub>8</sub> .1.4.c}                                | ,              |
| <b>empty symbol</b> {815a}  | EMPTY          |

c) Operator symbols

| symbol   | representation |
|--|----------------|
| <b>or symbol</b> {24 <sub>9</sub> .4.2.H}              | ∨              |
| <b>and symbol</b> {24 <sub>9</sub> .4.2.H}             | ∧              |
| <b>ampersand symbol</b> {24 <sub>9</sub> .4.2.H}       | &              |
| <b>differs from symbol</b> {24 <sub>9</sub> .4.2.H}    | ≠              |
| <b>is less than symbol</b> {24 <sub>9</sub> .4.2.I}    | <              |
| <b>is at most symbol</b> {24 <sub>9</sub> .4.2.H}      | ≤              |
| <b>is at least symbol</b> {24 <sub>9</sub> .4.2.H}     | ≥              |
| <b>is greater than symbol</b> {24 <sub>9</sub> .4.2.I} | >              |
| <b>divided by symbol</b> {24 <sub>9</sub> .4.2.I}      | /              |
| <b>over symbol</b> {24 <sub>9</sub> .4.2.H}            | ÷              |

## ALGOL 68 REVISED REPORT

|   |    |
|---|----|
| <b>percent symbol</b> {24 <sub>9</sub> .4.2.H}  | %  |
| <b>window symbol</b> {24 <sub>9</sub> .4.2.H}   | □  |
| <b>floor symbol</b> {24 <sub>9</sub> .4.2.H}  | ⌊  |
| <b>ceiling symbol</b> {24 <sub>9</sub> .4.2.H}  | ⌈  |
| <b>plus i times symbol</b> {24 <sub>9</sub> .4.2.H}   | ⊥  |
| <b>not symbol</b> {24 <sub>9</sub> .4.2.H}  | ¬  |
| <b>tilde symbol</b> {24 <sub>9</sub> .4.2.H}  | ~  |
| <b>down symbol</b> {24 <sub>9</sub> .4.2.H}   | ↓  |
| <b>up symbol</b> {24 <sub>9</sub> .4.2.H}   | ↑  |
| <b>plus symbol</b> {23 <sub>8</sub> .1.2.j, 23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.H, 25 <sub>10</sub> .3.4.2.e}  | +  |
| <b>minus symbol</b> {23 <sub>8</sub> .1.2.j, 23 <sub>8</sub> .1.4.c, 24 <sub>9</sub> .4.2.H, 25 <sub>10</sub> .3.4.2.e} | -  |
| <b>equals symbol</b> {24 <sub>9</sub> .4.2.I}   | =  |
| <b>times symbol</b> {24 <sub>9</sub> .4.2.I}  | ×  |
| <b>asterisk symbol</b> {24 <sub>9</sub> .4.2.I}   | *  |
| <b>assigns to symbol</b> {24 <sub>9</sub> .4.2.J}   | := |
| <b>becomes symbol</b> {18 <sub>4</sub> .4.f, 19 <sub>5</sub> .2.1.a, 24 <sub>9</sub> .4.2.J}                            | := |

### d) Declaration symbols

| symbol   | representation |
|--|----------------|
| <b>is defined as symbol</b> {18 <sub>4</sub> .2.b, 18 <sub>4</sub> .3.b, 18 <sub>4</sub> .4.c, 18 <sub>4</sub> .5.c} | =              |
| <b>long symbol</b> {23 <sub>8</sub> .1.0.a, 23 <sub>8</sub> .2.a}  | LONG           |
| <b>short symbol</b> {23 <sub>8</sub> .1.0.a, 23 <sub>8</sub> .2.b}   | SHORT          |
| <b>reference to symbol</b> {18 <sub>4</sub> .6.c}  | REF            |
| <b>local symbol</b> {19 <sub>5</sub> .2.3.a, b}  | LOC            |
| <b>heap symbol</b> {19 <sub>5</sub> .2.3.a, b}   | HEAP           |
| <b>structure symbol</b> {18 <sub>4</sub> .6.d}   | STRUCT         |
| <b>flexible symbol</b> {18 <sub>4</sub> .6.g}  | FLEX           |
| <b>procedure symbol</b> {18 <sub>4</sub> .18 <sub>4</sub> .b, 18 <sub>4</sub> .6.o}                                  | PROC           |
| <b>union of symbol</b> {18 <sub>4</sub> .6.s}  | UNION          |
| <b>operator symbol</b> {18 <sub>4</sub> .5.a}  | OP             |
| <b>priority symbol</b> {18 <sub>4</sub> .3.a}  | PRIO           |
| <b>mode symbol</b> {18 <sub>4</sub> .2.a}  | MODE           |

### e) Mode standards

| symbol   | representation |
|--|----------------|
| <b>integral symbol</b> {24 <sub>9</sub> .4.2.E}  | INT            |
| <b>real symbol</b> {24 <sub>9</sub> .4.2.E}      | REAL           |
| <b>boolean symbol</b> {24 <sub>9</sub> .4.2.E}   | BOOL           |
| <b>character symbol</b> {24 <sub>9</sub> .4.2.E} | CHAR           |
| <b>format symbol</b> {24 <sub>9</sub> .4.2.E}    | FORMAT         |

|  |         |
|--|---------|
| <b>void symbol</b> {24 <sub>9</sub> .4.2.E}    | VOID    |
| <b>complex symbol</b> {24 <sub>9</sub> .4.2.E} | COMPL   |
| <b>bits symbol</b> {24 <sub>9</sub> .4.2.E}    | BITS    |
| <b>bytes symbol</b> {24 <sub>9</sub> .4.2.E}   | BYTES   |
| <b>string symbol</b> {24 <sub>9</sub> .4.2.E}  | STRING  |
| <b>sema symbol</b> {24 <sub>9</sub> .4.2.E}    | SEMA    |
| <b>file symbol</b> {24 <sub>9</sub> .4.2.E}    | FILE    |
| <b>channel symbol</b> {24 <sub>9</sub> .4.2.E} | CHANNEL |

f) Syntactic symbols

| symbol   | representation |
|--|----------------|
| <b>bold begin symbol</b> {15 <sub>1</sub> .3.3.d}  | BEGIN          |
| <b>bold end symbol</b> {15 <sub>1</sub> .3.3.d}  | END            |
| <b>brief begin symbol</b> {15 <sub>1</sub> .3.3.d, 25 <sub>10</sub> .3.4.8.b, 25 <sub>10</sub> .3.4.A.b}   | (              |
| <b>brief end symbol</b> {15 <sub>1</sub> .3.3.d, 25 <sub>10</sub> .3.4.8.b, 25 <sub>10</sub> .3.4.A.b}   | )              |
| <b>and also symbol</b> {15 <sub>1</sub> .3.3.c, 17 <sub>3</sub> .3.b, f, 17 <sub>3</sub> .4.h, 18 <sub>4</sub> .1.a, b, 18 <sub>4</sub> .6.e,i,q,t, 19 <sub>5</sub> .3.2.b, 19 <sub>5</sub> .4.1.e, 19 <sub>5</sub> .4.3.b, 25 <sub>10</sub> .3.4.8.b, 25 <sub>10</sub> .3.4.A.c, d} | ,              |
| <b>go on symbol</b> {17 <sub>3</sub> .2.b}   | ;              |
| <b>completion symbol</b> {17 <sub>3</sub> .2.b}  | EXIT           |
| <b>label symbol</b> {17 <sub>3</sub> .2.c}   | :              |
| <b>parallel symbol</b> {17 <sub>3</sub> .3.c}  | PAR            |
| <b>open symbol</b> {23 <sub>8</sub> .1.4.c}  | (              |
| <b>close symbol</b> {23 <sub>8</sub> .1.4.c}   | )              |
| <b>bold if symbol</b> {24 <sub>9</sub> .1.a}   | IF             |
| <b>bold then symbol</b> {24 <sub>9</sub> .1.b}   | THEN           |
| <b>bold else if symbol</b> {24 <sub>9</sub> .1.c}  | ELIF           |
| <b>bold else symbol</b> {24 <sub>9</sub> .1.d}   | ELSE           |
| <b>bold fi symbol</b> {24 <sub>9</sub> .1.e}   | FI             |
| <b>bold case symbol</b> {24 <sub>9</sub> .1.a}   | CASE           |
| <b>bold in symbol</b> {24 <sub>9</sub> .1.b}   | IN             |
| <b>bold ouse symbol</b> {24 <sub>9</sub> .1.c}   | OUSE           |
| <b>bold out symbol</b> {24 <sub>9</sub> .1.d}  | OUT            |

# ALGOL 68 REVISED REPORT

|  |               |
|--|---------------|
| <b>bold esac symbol</b> {24 <sub>9</sub> .1.e}                         | ESAC          |
| <b>brief if symbol</b> {24 <sub>9</sub> .1.a}                          | (             |
| <b>brief then symbol</b> {24 <sub>9</sub> .1.b}                        |               |
| <b>brief else if symbol</b> {24 <sub>9</sub> .1.c}                     | :             |
| <b>brief else symbol</b> {24 <sub>9</sub> .1.d}                        |               |
| <b>brief fi symbol</b> {24 <sub>9</sub> .1.e}                          | )             |
| <b>brief case symbol</b> {24 <sub>9</sub> .1.a}                        | (             |
| <b>brief in symbol</b> {24 <sub>9</sub> .1.b}                          |               |
| <b>brief ouse symbol</b> {24 <sub>9</sub> .1.c}                        | :             |
| <b>brief out symbol</b> {24 <sub>9</sub> .1.d}                         |               |
| <b>brief esac symbol</b> {24 <sub>9</sub> .1.e}                        | )             |
| <b>colon symbol</b> {17 <sub>3</sub> .4.j, k}                          | :             |
| <b>brief sub symbol</b> {15 <sub>1</sub> .3.3.e}                       | [             |
| <b>brief bus symbol</b> {15 <sub>1</sub> .3.3.e}                       | ]             |
| <b>style i sub symbol</b> {15 <sub>1</sub> .3.3.e}                     | (             |
| <b>style i bus symbol</b> {15 <sub>1</sub> .3.3.e}                     | )             |
| <b>up to symbol</b> {18 <sub>4</sub> .6.j,k,l, 19 <sub>5</sub> .3.2.f} | :             |
| <b>at symbol</b> {19 <sub>5</sub> .3.2.g}                              | @ AT          |
| <b>is symbol</b> {19 <sub>5</sub> .2.2.b}                              | :=: IS        |
| <b>is not symbol</b> {19 <sub>5</sub> .2.2.b}                          | :≠: :/=: ISNT |
| <b>nil symbol</b> {19 <sub>5</sub> .2.4.a}                             | o NIL         |
| <b>of symbol</b> {19 <sub>5</sub> .3.1.a}                              | OF            |
| <b>routine symbol</b> {19 <sub>5</sub> .4.1.a, b}                      | :             |
| <b>bold go to symbol</b> {19 <sub>5</sub> .4.4.b}                      | GOTO          |
| <b>bold go symbol</b> {19 <sub>5</sub> .4.4.b}                         | GO            |
| <b>skip symbol</b> {19 <sub>5</sub> .5.2.a}                            | ~ SKIP        |
| <b>formatter symbol</b> {25 <sub>10</sub> .3.4.1.a}                    | \$            |

## g) Loop symbols

| symbol   | representation |
|--|----------------|
| <b>bold for symbol</b> {17 <sub>3</sub> .5.b}      | FOR            |
| <b>bold from symbol</b> {17 <sub>3</sub> .5.d}     | FROM           |
| <b>bold by symbol</b> {17 <sub>3</sub> .5.d}       | BY             |
| <b>bold to symbol</b> {17 <sub>3</sub> .5.d, 544b} | TO             |
| <b>bold while symbol</b> {17 <sub>3</sub> .5.g}    | WHILE          |
| <b>bold do symbol</b> {17 <sub>3</sub> .5.h}       | DO             |
| <b>bold od symbol</b> {17 <sub>3</sub> .5.h}       | OD             |

## h) Pragment symbols

symbol

representation

**brief comment symbol** {24<sub>9</sub>.2.b}  
**bold comment symbol** {24<sub>9</sub>.2.b}  
**style i comment symbol** {24<sub>9</sub>.2.b}  
**style ii comment symbol** {24<sub>9</sub>.2.b}  
**bold pragmat symbol** {24<sub>9</sub>.2.b}  
**style i pragmat symbol** {24<sub>9</sub>.2.b}

ç  
COMMENT  
CO  
#  
PRAGMAT  
PR

## 24.4.2 Other TAX symbols

### 24.4.2.1 Metasyntax

- A) **TAG** {D, F, K, 18<sub>4</sub>.8.a, b, c, d} ::  
**LETTER** {B}; **TAG LETTER** {B}; **TAG DIGIT** {C}.
- B) **LETTER** {A} ::  
**letter ABC** {24<sub>9</sub>.4.a } ; **letter aleph** {-}; **style TALLY letter ABC** {-}.
- C) **DIGIT** {A} ::  
**digit zero** {24<sub>9</sub>.4.b } ; **digit one** {24<sub>9</sub>.4.b } ; **digit two** {24<sub>9</sub>.4.b } ;  
**digit three** {24<sub>9</sub>.4.b } ; **digit four** {24<sub>9</sub>.4.b } ; **digit five** {24<sub>9</sub>.4.b } ;  
**digit six** {24<sub>9</sub>.4.b } ; **digit seven** {24<sub>9</sub>.4.b } ; **digit eight** {24<sub>9</sub>.4.b } ;  
**digit nine** {24<sub>9</sub>.4.b } .
- D) **TAB** {18<sub>4</sub>.8.a,b } ::  
**bold TAG** {A, -}; **SIZETY STANDARD** {E}.
- E) **STANDARD** {D} ::  
**integral** {24<sub>9</sub>.4.e } ; **real** {24<sub>9</sub>.4.e } ; **boolean** {24<sub>9</sub>.4.e } ;  
**character** {24<sub>9</sub>.4.e } ; **format** {24<sub>9</sub>.4.e } ; **void** {24<sub>9</sub>.4.e } ;  
**complex** {24<sub>9</sub>.4.e } ; **bits** {24<sub>9</sub>.4.e } ; **bytes** {24<sub>9</sub>.4.e } ;  
**string** {24<sub>9</sub>.4.e } ; **sema** {24<sub>9</sub>.4.e } ; **file** {24<sub>9</sub>.4.e } ;  
**channel** {24<sub>9</sub>.4.e } .
- F) **TAD** {18<sub>4</sub>.8.a, b } ::  
**bold TAG** {A, -};  
**DYAD** {G} **BECOMESETY** {J};  
**DYAD** {G} **cum NOMAD** {I} **BECOMESETY** {J}.
- G) **DYAD** {F} :: **MONAD** {H}; **NOMAD** {I} .
- H) **MONAD** {G, K} ::  
**or** {24<sub>9</sub>.4.c } ; **and** {24<sub>9</sub>.4.c } ; **ampersand** {24<sub>9</sub>.4.c } ;  
**differs from** {24<sub>9</sub>.4.c } ; **is at most** {24<sub>9</sub>.4.c } ; **is at least** {24<sub>9</sub>.4.c } ;

**over** {24<sub>9</sub>.4.c } ; **percent** {24<sub>9</sub>.4.c } ; **window** {24<sub>9</sub>.4.c } ;  
**floor** {24<sub>9</sub>.4.c } ; **ceiling** {24<sub>9</sub>.4.c } ; **plus i times** {24<sub>9</sub>.4.c } ;  
**not** {24<sub>9</sub>.4.c } ; **tilde** {24<sub>9</sub>.4.c } ; **down** {24<sub>9</sub>.4.c } ;  
**up** {24<sub>9</sub>.4.c } ; **plus** {24<sub>9</sub>.4.c } ; **minus** {24<sub>9</sub>.4.c } ;  
**style TALLY monad** {-}.

I) **NOMAD** {F, G, K} ::

**is less than** {24<sub>9</sub>.4.c } ; **is greater than** {24<sub>9</sub>.4.c } ; **divided by** {24<sub>9</sub>.4.c } ;  
**equals** {24<sub>9</sub>.4.c } ; **times** {24<sub>9</sub>.4.c } ; **asterisk** {24<sub>9</sub>.4.c } .

J) **BECOMESETY** {F, K} :: **cum becomes** {24<sub>9</sub>.4.c } ; **cum assigns to** {24<sub>9</sub>.4.c } ;  
**EMPTY**.

K) **TAM** {18<sub>4</sub>.8.a, b } ::

**bold TAG** {A, -};  
**MONAD** {H} **BECOMESETY** {J};  
**MONAD** {H} **cum NOMAD** {I} **BECOMESETY** {J}.

L) **ABC** {B} ::

**a ; b ; c ; d ; e ; f ; g ; h ; i ; j ; k ; l ; m ; n ; o ; p ; q ; r ; s ; t ; u ; v ; w ; x ; y ; z.**

M) **\*DOP** :: **DYAD** {G}; **DYAD** {G} **cum NOMAD** {I}.

{The metanotation "**ABC**" is provided, in addition to the metanotation "**ALPHA**", in order to facilitate the definition of variants of Algol 68 {15<sub>1</sub>.1.5.b).}

### 24.4.2.2 Representation

a) The representation of each **TAG-symbol** not given above {24<sub>9</sub>.4.1 } is composed of marks corresponding, in order, to the '**LETTER**'s or '**DIGIT**'s contained in that '**TAG**'. These marks may be separated by typographical display features {24<sub>9</sub>.4.d }. The mark corresponding to each '**LETTER**' ('**DIGIT**') is the representation of that **LETTER-symbol** (**DIGIT-symbol**). {For example, the representation of a **letter-x-digit-one-symbol** is  $\times 1$ , which may be written  $\times 1$ . **TAG-symbols** are used for **identifiers** and **field-selectors**.}

b) The representation, if any, of each **bold-TAG-symbol** is composed of marks corresponding, in order, to the '**LETTER**'s or '**DIGIT**'s contained in that '**TAG**' {but with no typographical display features in between}. The mark corresponding to each '**LETTER**' ('**DIGIT**') is similar to the mark representing the corresponding **LETTER-symbol** (**DIGIT-symbol**), being, in this Report, the corresponding bold faced letter (digit). {Other methods of indicating the similarity which are recognizable without further elucidation are also acceptable, e.g., **person**, person, PERSON, 'person and 'person' could all be representations of the **bold-letter-p-letter-e-letter-r-letter-s-letter-o-letter-n-symbol**.}

However, the representation of a **bold-TAG-symbol** may not be the same as any representation of any other **symbol** {; thus there may be a finite number of **bold-TAG-symbols**

which have no representation; e.g., there is no representation for the **bold-letter-r-letter-e-letter-a-letter-l-symbol** because `REAL` is a representation of the **real-symbol**; note that the number of **bold-TAG-symbols** available is still arbitrarily large}. If, according to the convention used, a given sequence of marks could be either the representation of one **bold-TAG-symbol** or the concatenation of the representations of two or more other **symbols**, then it is always to be construed as that one **symbol** {; the inclusion of a blank can always force the other interpretation; e.g., `REFREAL` is one **symbol**, whereas `REF REAL` must always be two}. **{Bold-TAG-symbols are used for mode-indications and for operators.}**

c) The representation of each **SIZE-SIZETY-STANDARD-symbol** is composed of the representation of the corresponding **SIZE-symbol**, possibly followed by typographical display features, followed by the representation of the corresponding **SIZETY-STANDARD-symbol**. {For example, the representation of a **long-real-symbol** is `LONG REAL`, or perhaps `'long"real'` (but not, according to section b above, `LONGREAL` or `'longreal'`, for those would be representations of the **bold-letter-l-letter-o-letter-n-letter-g-letter-r-letter-e-letter-a-letter-l-symbol**). **SIZETY-STANDARD-symbols are used for mode-indications.}**

d) The representation of each **DOP-cum-becomes-symbol (DOP-cum-assigns-to-symbol)** is composed of the mark or marks representing the corresponding **DOP-symbol** followed (without intervening typographical display features) by the marks representing the **becomes-symbol (the assigns-to-symbol)**. {For example, the representation of a **plus-cum-becomes-symbol** is `+:=`. **DOP-cum-becomes-symbols are used for operators.}**

e) The representation of each **DYAD-cum-NOMAD-symbol** is composed of the mark representing the corresponding **DYAD-symbol** followed {without intervening typographical display features} by the mark representing the corresponding **NOMAD-symbol**. {For example, the representation of an **over-cum-times-symbol** is `÷×`. **DYAD-cum-NOMAD-symbols are used for operators**, but note that **NOMAD1-cum-NOMAD2-symbols** may be only **dyadic-operators**.}

# ALGOL 68 REVISED REPORT



## **PART V. Environment and examples**

# ALGOL 68 REVISED REPORT

# Chapter 25

## Standard environment

{The "standard environment" encompasses the constituent **EXTERNAL-preludes**, **system-tasks** and **particular-postludes** of a **program-text**.}

### 25.1 Program texts

{The programmer is concerned with **particular-programs** {25<sub>10</sub>.1.1.g}. These are always included in a **program-text** {25<sub>10</sub>.1.1.a} which also contains the **standard-prelude**, a **library-prelude**, which depends upon the implementation, a **system-prelude** and **system-tasks**, which correspond to the operating environment, possibly some other **particular-programs**, one or more **particular-preludes** (one for each **particular-program**) and one or more **particular-postludes**.}

#### 25.1.1 Syntax

- A) **EXTERNAL** :: **standard ; library ; system ; particular**.
- B) **STOP** :: **label letter s letter t letter o letter p**.
- a) **program text** :
  - STYLE begin** {24<sub>9</sub>.4.f, -} **token**,
  - new LAYER1 preludes** {b},
  - parallel** {24<sub>9</sub>.4.f} **token**, **new LAYER1 tasks** {d} **PACK**,
  - STYLE end** {24<sub>9</sub>.4.f, -} **token**.
- b) **NEST1 preludes** {a} :
  - NEST1 standard prelude with DECS1** {c},

**NEST1 library prelude with DECSETY2 {c},  
 NEST1 system prelude with DECSETY3 {c},  
 where (NEST1) is (new EMPTY new DECS1 DECSETY2 DECSETY3) .**

- c) **NEST1 EXTERNAL prelude with DECSETY {b,f} :  
 strong void NEST1 series with DECSETY1 {17<sub>3</sub>.2.b} ,  
 go on {24<sub>9</sub>.4.f} token where (DECSETY1) is (EMPTY), EMPTY.**
- d) **NEST1 tasks {a} :  
 NEST1 system task {e} list,  
 and also {24<sub>9</sub>.4.f} token,  
 NEST1 user task {f} PACK list.**
- e) **NEST1 system task {d} : strong void NEST1 unit {17<sub>3</sub>.2.d} .**
- f) **NEST1 user task {d} :  
 NEST2 particular prelude with DECS {c},  
 NEST2 particular program {g} PACK, go on {24<sub>9</sub>.4.f} token,  
 NEST2 particular prelude {i},  
 where (NEST2) is (NEST1 new DECS STOP) .**
- g) **NEST2 particular program {f} :  
 NEST2 new LABSETY3 joined label definition of LABSETY3 {h},  
 strong void NEST2 new LABSETY3 ENCLOSED clause  
 {17<sub>3</sub>.1.a, 17<sub>3</sub>.3.a,c, 17<sub>3</sub>.4.a, 17<sub>3</sub>.5.a} .**
- h) **NEST joined label definition of LABSETY {g,h} :  
 where (LABSETY) is (EMPTY),  
 EMPTY;  
 where (LABSETY) is (LAB1 LABSETY1),  
 NEST label definition of LAB1 {17<sub>3</sub>.2.c} ,  
 NEST joined label definition of LABSETY1 {h}.**
- i) **NEST2 particular postlude {f} : strong void NEST2 series with STOP {17<sub>3</sub>.2.b}**  
 .

{Examples:

- a) ( **C standard-prelude C ; C library-prelude C; C system-prelude C ;  
 PAR BEGIN C system-task-1 C , C system-task-2 C ,  
 ( C particular-prelude C ;  
 (start: commence: BEGIN SKIP END);  
 C particular-postlude C ) ,  
 ( C another user-task C )  
 END)**

- b) **C** standard-prelude (25<sub>10</sub>.2, 25<sub>10</sub>.3) **C**; **C** library-prelude **C** ;  
**C** system-prelude (25<sub>10</sub>.4.1) **C**;
- d) **C** system-task-1 (25<sub>10</sub>.4.2.a) **C**, **C** system-task-2 **C**,  
( **C** particular-prelude **C** ;  
(start: commence: BEGIN SKIP END);  
**C** particular-postlude **C** ),  
( **C** another user-task **C** )
- f) **C** particular-prelude (25<sub>10</sub>.5.1) **C** ;  
(start: commence: BEGIN SKIP END);  
**C** particular-postlude **C** ),
- g) start: commence: BEGIN SKIP END
- h) start: commence:
- i) stop: lock (stand in); lock (stand out); lock (stand back) }

## 25.1.2 The environment condition

- a) A **program** in the strict language must be akin {15<sub>1</sub>.1.3.2.k} to some **program-text** whose constituent **EXTERNAL-preludes** and **particular-postludes** are as specified in the remainder of this section. {It is convenient to speak of the **standard-prelude**, the **library-prelude**, the **particular-programs**, etc. of a **program** when discussing those parts of that **program** which correspond to the constituent **standard-prelude**, etc. of the corresponding **program-text**.}
- b) The constituent **standard-prelude** of all **program-texts** is that **standard-prelude** whose representation is obtained {25<sub>10</sub>.1.3} from the forms given in sections 25<sub>10</sub>.2 and 25<sub>10</sub>.3.
- c) The constituent **library-prelude** of a **program-text** is not specified in this Report {but must be specified for each implementation; the syntax of '**program text**' ensures that a **declaration** contained in a **library-prelude** may not contradict any **declaration** contained in the **standard-prelude**} .
- d) The constituent **system-prelude (system-task-list)** of all **program-texts** is that **system-prelude (system-task-list)** whose representation is obtained from the forms given in section 25<sub>10</sub>.4, with the possible addition of other forms not specified in this Report {but to be specified to suit the operating environment of each implementation} .
- e) Each constituent **particular-prelude (particular-postlude)** of all **program-texts** is that **particular-prelude (particular-postlude)** whose representation is obtained from

the forms given in section 25<sub>10</sub>.5, with the possible addition of other forms not specified in this Report {but to be specified for each implementation}.

### 25.1.3 The method of description of the standard environment

A representation of an **EXTERNAL-prelude**, **system-task** or **particular-postlude** is obtained by altering each form in the relevant sections of this chapter in the following steps:

Step 1: If a given form  $F$  begins with OP {the **operator-symbol**} followed by one of the marks  $P$ ,  $Q$ ,  $R$  or  $E$ , then  $F$  is replaced by a number of new forms each of which is a copy of  $F$  in which that mark {following the OP} is (all other occurrences in  $F$  of that mark are) replaced, in each respective new form, by:

Case A: The mark is  $P$ :

- $-$ ,  $+$ ,  $\ll \times$ ,  $*$   $\gg$  or  $/$   
( $-$ ,  $+$ ,  $*$  or  $/$ );

Case B: The mark is  $Q$ :

- $\ll \text{MINUSAB}, - := \gg$ ,  $\ll \text{PLUSAB}, + := \gg$ ,  
 $\ll \text{TIMESAB}, \times :=, * := \gg$  or  $\ll \text{DIVAB}, / := \gg$   
( $- :=$ ,  $+ :=$ ,  $\times :=$ ,  $* :=$  or  $/ :=$ );

Case C: The mark is  $R$ :

- $\ll <, \text{LT} \gg$ ,  $\ll \leq, \text{LE} \gg$ ,  $\ll =, \text{EQ} \gg$ ,  
 $\ll \neq, \text{NE} \gg$ ,  $\ll \geq, \text{GE} \gg$  or  $\ll >, \text{GT} \gg$   
( $<$ ,  $\leq$ ,  $=$ ,  $\neq$ ,  $\geq$  or  $>$ );

Case D: The mark is  $E$ :

- $\ll =, \text{EQ} \gg$  or  $\ll \neq, \text{NE} \gg$   
( $=$  or  $\neq$ );

Step 2: If, in some form, as possibly made in the step above,  $\aleph_0$  occurs followed by an **INDICATOR** (a **field-selector**)  $I$ , then that occurrence of  $\aleph_0$  is deleted and each **INDICATOR** (**field-selector**) akin {15<sub>1</sub>.1.3.2.k} to  $I$  contained in any form is replaced by a copy of one same **INDICATOR** (**field-selector**) which does not occur elsewhere in the **program** and Step 2 is taken again<sup>1</sup>

---

<sup>1</sup>This relates to the problem of unspeakable names. In the standard environment a number of definitions occur that cannot be used by the programmer, and therefore need an unspeakable name. In earlier versions of the report these names were started by a sequence of  $\aleph_0$  f's, generated by an infinite production **ALEPH**. Later, infinite productions were removed from the report and **ALEPH**

- Step 3: If a given form  $F$ , as possibly modified or made in the steps above, begins with `OP` {the **operator-symbol**} followed by a chain of **TAO-symbols** separated by **and-also-symbols**, the chain being enclosed between `<<` and `>>`, then  $F$  is replaced by a number of different "versions" of that form each of which is a copy of  $F$  in which that chain, together with its enclosing `<<` and `>>`, has been replaced by one of those **TAO-symbols** {; however, an implementation is not obliged to provide more than one such version {24<sub>9</sub>.4.b} } ;
- Step 4: If, in a given form, as possibly modified or made in the steps above, there occurs a sequence  $S$  of **symbols** enclosed between `<<` and `>>` and if, in that  $S$ ,  $L$  `INT`,  $L$  `REAL`,  $L$  `COMPL`,  $L$  `BITS` or  $L$  `BYTES` occurs, then  $S$  is replaced by a chain of a sufficient number of sequences separated by **and-also-symbols**, the  $n^{\text{th}}$  of which is a copy of  $S$  in which copy each occurrence of  $L$  ( $L$ ,  $K$ ,  $S$ ) is replaced by  $(n - 1)$  times `long` (`LONG`, `LENG`, `SHORTEN`), followed by an **and-also-symbol** and a further chain of a sufficient number of sequences separated by **and-also-symbols**, the  $m^{\text{th}}$  of which is a copy of  $S$  in which copy each occurrence of  $L$  ( $L$ ,  $K$ ,  $S$ ) has been replaced by  $m$  times `short` (`SHORT`, `SHORTEN`, `LENG`); the `<<` and `>>` enclosing that  $S$  are then deleted;
- Step 5: If, in a given form  $F$ , as possibly modified or made in the steps above,  $L$  `INT`,  $L$  `REAL`,  $L$  `COMPL`,  $L$  `BITS` or  $L$  `BYTES`) occurs, then  $F$  is replaced by a sequence of a sufficient number of new forms, the  $n^{\text{th}}$  of which is a copy of  $F$  in which copy each occurrence of  $L$  ( $L$ ,  $K$ ,  $S$ ) is replaced by  $(n - 1)$  times `long` (`LONG`, `LENG`, `SHORTEN`), and each occurrence of `long`  $L$  (`LONG`  $L$ ) by  $n$  times `long` (`LONG`), followed by a further sequence of a sufficient number of new forms, the  $m^{\text{th}}$  of which is a copy of  $F$  in which copy each occurrence of  $L$  ( $L$ ,  $K$ ,  $S$ ) is replaced by  $m$  times `short` (`SHORT`, `SHORTEN`, `LENG`), and each occurrence of `LONG`  $L$  (`long`  $L$ ) by  $(m - 1)$  times `SHORT` (`short`);
- Step 6: Each occurrence of  $F$  (`PRIM`) in any form, as possibly modified or made in the steps above, is replaced by a representation of a **letter-aleph-symbol (primal-symbol)** {24<sub>9</sub>.4.a};
- Step 7: If a sequence of representations beginning with and ending with **C** occurs in any form, as possibly modified or made in the steps above, then this sequence, which is termed a "pseudo-comment", is replaced by a representation of a **declarer or closed-clause** suggested by the sequence;
- Step 8: If, in any form, as possibly modified or made in the steps above, a **routine-text** occurs whose calling involves the manipulation of real numbers, then this **routine-text** may be replaced by any other **routine-text** whose calling has approximately the same effect {; the degree of approximation is left undefined in this Report (see also 16<sub>2</sub>.1.3.1.e) };

---

would have been replaced by  $\aleph$  if the authors would have had a font containing that symbol. They did not, and replaced it with a question-mark-cum-superimposed-tilde in the Revised Report. Since  $\LaTeX$  supports  $\aleph_0$  (but not the mark used in the Revised Report), the former mark is used here.

Step 9: In the case of an **EXTERNAL-prelude**, a form consisting of a **skip-symbol** followed by a **go-on-symbol** {SKIP;} is added at the end.

{The term "sufficient number", as used in Steps 4 and 5 above, implies that no intended **particular-program** should have a different meaning or fail to be produced by the syntax solely on account of an insufficiency of that number.}

Wherever {in the transput declarations} the representation  $_{10}(\backslash, \perp)$  occurs within a **character-denotation** or **string-denotation**, it is to be interpreted as the representation of the **string-item** {23<sub>8</sub>.1.4.1.b} used to indicate "times ten to the power" (an alternative form {, if any,} of "times ten to the power", "plus i times") on external media. {Clearly, these representations have been chosen because of their similarity to those of the **times-ten-to-the-power-symbol** {24<sub>9</sub>.4.1.b} and the **plus-i-times-symbol** {24<sub>9</sub>.4.1.c}, but, on media on which these characters are not available, other **string-items** must be chosen (and the **letter-e-symbol** and the **letter-i-symbol** are obvious candidates) .}

{The declarations in this chapter are intended to describe their effect clearly. The effect may very well be obtained by a more efficient method.}

## 25.2 The standard prelude

{The **declarations** of the **standard-prelude** comprise "environment enquiries", which supply information concerning a specific property of the implementation {16<sub>2</sub>.2.2.c}, "standard modes", "standard operators and functions", "synchronization operations" and "transput declarations" (which are given in section 25<sub>10</sub>.3) .}

### 25.2.1 Environment enquiries

- a) INT int lengths = C 1 plus the number of extra lengths of integers {16<sub>2</sub>.1.3.1.d} C ;
- b) INT int shorths = C 1 plus the number of extra shorths of integers {16<sub>2</sub>.1.3.1.d} C ;
- c) L INT L max int = C the largest L integral value {16<sub>2</sub>.2.2.b} C ;
- d) INT real lengths = C 1 plus the number of extra lengths of real numbers {16<sub>2</sub>.1.3.1.d} C ;
- e) INT real shorths = C 1 plus the number of extra shorths of real numbers {16<sub>2</sub>.1.3.1.d} C ;
- f) L REAL L max real = C the largest L real value {16<sub>2</sub>.2.2.b} C ;
- g) L REAL L small real = C the smallest L real value such that both  $L \ 1 + L \ \text{small real} > L \ 1$  and  $L \ 1 + L \ \text{small real} < L \ 1$  {16<sub>2</sub>.2.2.b} C ;
- h) INT bits lengths = C 1 plus the number of extra widths {j} of bits C ;



- i) INT bits shorths = C 1 plus the number of extra shorths {j} of bits C ;
- j) INT L bits width = C the number of elements in L bits; see L BITS {25<sub>10</sub>.2.2.g} ; this number increases (decreases) with the "size", i.e., the number of 'long's (minus the number of 'short's) of which 'L' is composed, until a certain size is reached, viz., "the number of extra widths" (minus "the number of extra shorths") of bits, after which it is constant C ;
- k) INT bytes lengths = C 1 plus the number of extra widths {m} of bytes C ;
- l) INT bytes shorths = C 1 plus the number of extra shorths {m} of bytes C ;
- m) INT L bytes width = C the number of elements in L bytes; see L BYTES {25<sub>10</sub>.2.2.h} ; this number increases (decreases) with the "size", i.e., the number of 'long's (minus the number of 'short's) of which 'L' is composed, until a certain size is reached, viz., "the number of extra widths" (minus "the number of extra shorths") of bits, after which it is constant C ;
- n) OP ABS = (CHAR a) INT: C the integral equivalent {16<sub>2</sub>.1.3.1.g} of the character 'a' C ;
- o) OP REPR = (INT a) CHAR: C that character 'x', if it exists, for which ABS x = a C ;
- p) INT max abs char = C the largest integral equivalent {16<sub>2</sub>.1.3.1.g} of a character C ;
- q) CHAR null character = C some character C ;
- r) CHAR flip = C the character used to represent 'true' during transput {25<sub>10</sub>.3.3.1.a, 25<sub>10</sub>.3.3.2.a} C ;
- s) CHAR flop = the character used to represent 'false' during transput C ;
- t) CHAR error char = C the character used to represent unconvertible arithmetic values {25<sub>10</sub>.3.2.1.b, c, d, e, f} during transput C ;
- u) CHAR blank = "\_";

## 25.2.2 Standard modes

- a) MODE VOID = C an actual-declarer specifying the mode 'void' C ;
- b) MODE BOOL = C an actual-declarer specifying the mode 'boolean' C ;
- c) MODE L INT = C an actual-declarer specifying the mode 'L integral' C ;
- d) MODE L REAL = C an actual-declarer specifying the mode 'L real' C ;
- e) MODE CHAR = C an actual-declarer specifying the mode 'character' C ;
- f) MODE L COMPL = STRUCT (L REAL re, im);
- g) MODE L BITS = STRUCT ([1 : L bits width] BOOL L F ); {See 25<sub>10</sub>.2.1.j}  
 {The **field-selector** is hidden from the user in order that he may not break open the structure; in particular, he may not subscript the field.}
- h) MODE L BYTES = STRUCT ([1 : L bytes width] CHAR L F ); {See 25<sub>10</sub>.2.1.m}
- i) MODE STRING = FLEX [1 : 0] CHAR;

## 25.2.3 Standard operators and routines

### 25.2.3.1 Standard priorities

a) PRIO MINUSAB = 1, PLUSAB = 1, TIMESAB = 1, DIVAB = 1, OVERAB = 1,  
 MODAB = 1, PLUSTO = 1,  
 -:= = 1, +:= = 1, ×:= = 1, \*:= = 1,  
 /:= = 1, ÷:= = 1, %:= = 1, ÷×:= = 1,  
 ÷\*:= = 1, %×:= = 1, %\*:= = 1, +=: = 1,

∨ = 2, OR = 2,

∧ = 3, AND = 3,

= = 4, EQ = 4, ≠ = 4, /= = 4, NE = 4,

< = 5, LT = 5, ≤ = 5, <= = 5, LE = 5,  
 >= = 5, ≥ = 5, GE = 5, > = 5, GT = 5,

- = 6, + = 6,

× = 7, \* = 7, / = 7, % = 7, OVER = 7,  
 ÷× = 7, ÷\* = 7, %× = 7, %\* = 7, MOD = 7,  
 □ = 7, ELEM = 7,

↑ = 8, \*\* = 8, ↓ = 8, UP = 8,  
 DOWN = 8, SHL = 8, SHR = 8,  
 LWB = 8, UPB = 8, [ = 8, [ = 8

⊥ = 9, +× = 9, +\* = 9, I = 9;

### 25.2.3.2 Rows and associated operations

- a) MODE  $\aleph_0$  ROWS = **C** an actual-declarer specifying a mode united from {16<sub>2</sub>.1.3.6.a} a sufficient set of modes each of which begins with 'row' **C** ;
- b) OP  $\ll$  LWB, [  $\gg$  = (INT n, ROWS a) INT: **C** the lower bound in the n-th bound pair of the descriptor of the value of 'a', if that bound pair exists **C** ;
- c) OP  $\ll$  UPB, [  $\gg$  = (INT n, ROWS a) INT: **C** the upper bound in the n-th bound pair of the descriptor of the value of 'a', if that bound pair exists **C** ;
- d) OP  $\ll$  LWB, [  $\gg$  = (ROWS a) INT: 1 [ a ;
- e) OP  $\ll$  UPB, [  $\gg$  = (ROWS a) INT: 1 [ a ;

{The term "sufficient set", as used in a above and also in 25<sub>10</sub>.3.2.2.b and d, implies that no intended **particular-program** should fail to be produced (nor any unintended **particular-program** be produced) by the syntax solely on account of an insufficiency of modes in that set.}

### 25.2.3.3 Operations on boolean operands

- a) OP  $\ll \vee, \text{OR} \gg = (\text{BOOL } b) \text{ BOOL: } (a \mid \text{TRUE} \mid b);$
- b) OP  $\ll \wedge, \text{AND} \gg = (\text{BOOL } a, b) \text{ BOOL: } (a \mid b \mid \text{FALSE});$
- c) OP  $\ll \neg, \sim, \text{NOT} \gg = (\text{BOOL } a) \text{ BOOL: } (a \mid \text{FALSE} \mid \text{TRUE});$
- d) OP  $\ll =, \text{EQ} \gg = (\text{BOOL } a, b) \text{ BOOL: } (a \wedge b) \vee (\neg a \wedge \neg b);$
- e) OP  $\ll \neq, / =, \text{NE} \gg = (\text{BOOL } a, b) \text{ BOOL: } \neg (a = b);$
- f) OP ABS = (BOOL a) INT: (a | 1 | 0);

### 25.2.3.4 Operations on integral operands

- a) OP  $\ll <, \text{LT} \gg = (L \text{ INT } a, b) \text{ BOOL:}$   
*C true if the value of 'a' is smaller than {16<sub>2</sub>.1.3.1.e} that of 'b' and false otherwise C ;*
- b) OP  $\ll \leq, < =, \text{LE} \gg = (L \text{ INT } a, b) \text{ BOOL: } \neg (b < a);$
- c) OP  $\ll =, \text{EQ} \gg = (L \text{ INT } a, b) \text{ BOOL: } a \leq b \wedge b \leq a;$
- d) OP  $\ll \neq, / =, \text{NE} \gg = (L \text{ INT } a, b) \text{ BOOL: } \neg (a = b);$
- e) OP  $\ll \geq, > =, \text{GE} \gg = (L \text{ INT } a, b) \text{ BOOL: } b \leq a;$
- f) OP  $\ll >, \text{GT} \gg = (L \text{ INT } a, b) \text{ BOOL: } b < a;$
- g) OP - = (L INT a, b) L INT:  
*C the value of 'a' minus {16<sub>2</sub>.1.3.1.e} that of 'b' C ;*
- h) OP - = (L INT a) L INT: L 0 - a;
- i) OP + = (L INT a, b) L INT: a - -b;
- j) OP + = (L INT a) L INT: a;
- k) OP ABS = (L INT a) L INT: (a < L 0 | -a | a);
- l) OP  $\ll \times, * \gg = (L \text{ INT } a) L \text{ INT:}$   

```

BEGIN L INT s := L 0, i := ABS b;
  WHILE i  $\geq$  L 1
    DO s := s + a, i := i - L 1 OD;
  (b < L 0 | -s | s)
END;
```

## ALGOL 68 REVISED REPORT

- m) OP  $\ll \div, \%, \text{ OVER} \gg = (L \text{ INT } a, b) L \text{ INT}:$   
 IF  $b \neq L 0$   
 THEN  $L \text{ INT } q := L 0, r := \text{ABS } a;$   
 WHILE  $(r := r - \text{ABS } b) \geq L 0$  DO  $q := q + L 1$  OD;  
 ( $a < L 0 \wedge b > L 0 \vee a \geq L 0 \wedge b < L 0 \mid -q \mid q$ )  
 FI;
- n) OP  $\ll \div \times, \div *, \% *, \text{ MOD} \gg = (L \text{ INT } a, b) L \text{ INT}:$   
 ( $L \text{ INT } r = a - a \text{ div } b \times b; r < 0 \mid r + \text{ABS } b \mid r$ );<sup>2</sup>
- o) OP  $/ = (L \text{ INT } a, b) L \text{ REAL}:$   $(a) / L \text{ REAL } (b);$
- p) OP  $\ll \uparrow, **, \text{ UP} \gg = (L \text{ INT } a, b) L \text{ INT}:$   
 ( $b \geq 0 \mid L \text{ INT } p := L 1; \text{ TO } b \text{ DO } p := p \times a \text{ OD}; p$ );
- q) OP **LENG** =  $(L \text{ INT } a) \text{ LONG } L \text{ INT}:$   
**C the long  $L$  integral value lengthened from {16<sub>2</sub>.1.3.1.e} the value of 'a' C ;**
- r) OP **SHORTEN** =  $(\text{LONG } L \text{ INT } a) L \text{ INT}:$   
**C the  $L$  integral value, if it exists, which can be lengthened to {16<sub>2</sub>.1.3.1.e} the value of 'a' C ;**
- s) OP **ODD** =  $(L \text{ INT } a) \text{ BOOL}:$   $\text{ABS } a \div \times L 2 = L 1;$
- t) OP **SIGN** =  $(L \text{ INT } a) \text{ INT}:$   $(a > L 0 \mid 1 \mid : a < L 0 \mid -1 \mid 0);$
- u) OP  $\ll \perp, +\times, +*, \text{ I} \gg = (L \text{ INT } a, b) L \text{ COMPL} : (a, b);$

### 25.2.3.5 Operations on real operands

- a) OP  $\ll <, \text{ LT} \gg = (L \text{ REAL } a, b) \text{ BOOL}:$   
**C true if the value of 'a' is smaller than {16<sub>2</sub>.1.3.1.e} that of 'b' and false otherwise C ;**
- b) OP  $\ll \leq, \leq, \text{ LE} \gg = (L \text{ REAL } a, b) \text{ BOOL}:$   $\neg (b < a);$
- c) OP  $\ll =, \text{ EQ} \gg = (L \text{ REAL } a, b) \text{ BOOL}:$   $a \leq b \wedge b \leq a;$
- d) OP  $\ll \neq, \neq, \text{ NE} \gg = (L \text{ REAL } a, b) \text{ BOOL}:$   $\neg (a = b);$
- e) OP  $\ll \geq, \geq, \text{ GE} \gg = (L \text{ REAL } a, b) \text{ BOOL}:$   $b \leq a;$
- f) OP  $\ll >, \text{ GT} \gg = (L \text{ REAL } a, b) \text{ BOOL}:$   $b < a;$
- g) OP  $- = (L \text{ REAL } a, b) L \text{ REAL}:$   
**C the value of 'a' minus {16<sub>2</sub>.1.3.1.e} that of 'b' C ;**
- h) OP  $- = (L \text{ REAL } a) L \text{ REAL}:$   $L 0 - a;$
- i) OP  $+ = (L \text{ REAL } a, b) L \text{ REAL}:$   $a - -b;$
- j) OP  $+ = (L \text{ REAL } a) L \text{ REAL}:$   $a;$
- k) OP **ABS** =  $(L \text{ REAL } a) L \text{ REAL}:$   $(a < L 0 \mid -a \mid a);$
- l) OP  $\ll \times, * \gg = (L \text{ REAL } a) L \text{ REAL}:$   
**C the value of 'a' times {16<sub>2</sub>.1.3.1.e} that of 'b' C ;**

---

<sup>2</sup>MOD always yields a positive result.

- m) OP / = (L REAL a, b) L REAL:  
**C** the value of 'a' divided by {16<sub>2</sub>.1.3.1.e} that of 'b' **C** ;
- n) OP LENG = (L REAL a) LONG L REAL:  
**C** the long L real value lengthened from {16<sub>2</sub>.1.3.1.e} the value of 'a' **C** ;
- o) OP SHORTEN = (LONG L REAL a) L REAL:  
**C** if ABS a ≤ S LENG max real , then a L REAL value 'v' such that, for any L REAL value 'w',  
 ABS (LENG v - a) ≤ ABS (LENG w - a) **C** ;
- p) OP ROUND = (L REAL a) L INT:  
**C** a L integral value, if one exists, which is widenable to {16<sub>2</sub>.1.3.1.e} a L REAL value differing by not more than one-half from the value of 'a' **C** ;
- q) OP SIGN = (L REAL a) INT: (a > L 0 | 1 | : a < L 0 | -1 | 0);
- r) OP ENTIER = (L REAL a) L REAL:  
 BEGIN L INT j := L 0;  
 WHILE j < a DO j := j + L 1 OD;  
 WHILE j > a DO j := j - L 1 OD;  
 j  
 END;
- s) OP ≪ ⊥, +×, +\*, I ≫ = (L REAL a, b) L COMPL : (a, b);

### 25.2.3.6 Operations on arithmetic operands

- a) OP P = (L REAL a, L INT b) L REAL: a P L REAL (b);
- b) OP P = (L INT a, L REAL b) L REAL: L REAL (a) P b;
- c) OP R = (L REAL a, L INT b) BOOL: a R L REAL (b);
- d) OP R = (L INT a, L REAL b) BOOL: L REAL (a) R b;
- e) OP ≪ ⊥, +×, +\*, I ≫ = (L REAL a, L INT b) L COMPL: (a, b);
- f) OP ≪ ⊥, +×, +\*, I ≫ = (L INT a, L REAL b) L COMPL: (a, b);
- g) OP ≪ ↑, \*\*, UP ≫ = (L REAL a, INT b) L REAL:  
 (L REAL p := L 1; TO ABS b DO p := p × a OD; (b ≥ 0 | p | L 1 / p));

### 25.2.3.7 Operations on character operands

- a) OP R (CHAR a, b) BOOL: ABS a R ABS b; {25<sub>10</sub>.2.1.n}
- b) OP + = (CHAR a, b) STRING: (a, b);

**25.2.3.8 Operations on complex operands**

- a) OP RE = (L COMPL a) L REAL: re OF a;
- b) OP IM = (L COMPL a) L REAL: im OF a;
- c) OP ABS = (L COMPL a) L REAL: L sqrt (RE a ↑ 2 + IM a ↑ 2);
- d) OP ARG = (L COMPL a) L REAL:
  - IF L REAL re = RE a, im = IM a;
  - re ≠ L 0 ∨ im ≠ L 0
  - THEN IF ABS re > ABS im
  - THEN L arctan (im / re) + L pi / L 2 \*  
(im < L 0 | SIGN re - 1 | 1 - SIGN re)
  - ELSE - L arctan (re / im) + L pi / L 2 × SIGN im
  - FI
  - FI;
- e) OP CONJ = (L COMPL a) L COMPL: RE a ⊥ - IM a;
- f) OP ≪ =, EQ ≫ = (L COMPL a, b) BOOL: RE a = RE b ∧ IM a = IM b;
- g) OP ≪ ≠, /=, NE ≫ = (L COMPL a, b) BOOL: ¬ (a = b);
- h) OP - = (L COMPL a, b) L COMPL: (RE a - RE b) ⊥ (IM a - IM b);
- i) OP - = (L COMPL a) L COMPL: -RE a ⊥ -IM a;
- j) OP + = (L COMPL a, b) L COMPL: (RE a + RE b) ⊥ (IM a + IM b);
- k) OP + = (L COMPL a) L COMPL: a;
- l) OP ≪ ×, \* ≫ = (L COMPL a, b) L COMPL:
  - (RE a × RE b - IM a × IM b) ⊥ (RE a × IM b + IM a × RE b);
- m) OP / = (L COMPL a, b) L COMPL:
  - (L REAL d = RE (b × CONJ b); L COMPL n = a\* CONJ b;
  - (RE n / d) ⊥ (IM n / d));
- n) OP LENG = (L COMPL a) LONG L COMPL: LENG RE a ⊥ LENG IM a;
- o) OP SHORTEN = (LONG L COMPL a) L COMPL: SHORTEN RE a ⊥ SHORTEN IM a;
- p) OP P = (L COMPL a, L INT b) L COMPL: a P L COMPL (b);
- q) OP P = (L COMPL a, L REAL b) L COMPL: a P L COMPL (b);
- r) OP P = (L INT a, L COMPL b) L COMPL: L COMPL (a) P b;
- s) OP P = (L REAL a, L COMPL b) L COMPL: L COMPL (a) P b;
- t) OP ≪ ↑, \*\*, UP ≫ = (L COMPL a, INT b) L COMPL:
  - (L COMPL p := L 1; TO ABS b DO p := p × a OD; (b ≥ 0 | p | L 1 / p));
- u) OP E = (L COMPL a, L INT b) BOOL: a E L COMPL (b);
- v) OP E = (L COMPL a, L REAL b) BOOL: a E L COMPL (b);
- w) OP E = (L INT a, L COMPL b) BOOL: b E a;
- x) OP E = (L REAL a, L COMPL b) BOOL: b E a;

**25.2.3.9 Bits and associated operations**

- a) OP  $\ll =, EQ \gg = (L \text{ BITS } a, b) \text{ BOOL} :$   
 BEGIN BOOL c;  
 FOR i TO  $L$  bits width  
 WHILE c := (L F OF a) [i] = (L F OF b) [i]  
 DO SKIP OD;  
 c  
 END;
- b) OP  $\ll \neq, /=, NE \gg = (L \text{ BITS } a, b) \text{ BOOL} : \neg (a = b);$
- c) OP  $\ll \vee, OR \gg = (L \text{ BITS } a, b) L \text{ BITS} :$   
 BEGIN  $L$  BITS c;  
 FOR i TO  $L$  bits width  
 DO (L F OF c) [i] = (L F OF a) [i]  $\vee$  (L F OF b) [i] OD;  
 c  
 END;
- d) OP  $\ll \wedge, \&, AND \gg = (L \text{ BITS } a, b) L \text{ BITS} :$   
 BEGIN  $L$  BITS c;  
 FOR i TO  $L$  bits width  
 DO (L F OF c) [i] := (L F OF a) [i]  $\wedge$  (L F OF b) [i] OD;  
 c  
 END;
- e) OP  $\ll \leq, <=, LE \gg = (L \text{ BITS } a, b) \text{ BOOL} : (a \vee b) = b;$
- f) OP  $\ll \geq, >=, GE \gg = (L \text{ BITS } a, b) \text{ BOOL} : b \leq a;$
- g) OP  $\ll \uparrow, UP, SHL \gg = (L \text{ BITS } a, \text{INT } b) L \text{ BITS} :$   
 IF ABS b  $\leq L$  bits width  
 THEN  $L$  BITS c := a;  
 TO ABS b  
 DO IF b > 0 THEN  
 FOR i FROM 2 TO  $L$  bits width  
 DO (L F OF e) [i - 1] := (L F OF c) [i] OD;  
 (L F OF c) [ $L$  bits width] := FALSE  
 ELSE  
 FOR i FROM  $L$  bits width BY -1 TO 2  
 DO (L F OF c) [i] := (L F OF c) [i - 1] OD;  
 (L F OF c) [1] := FALSE  
 FI OD;  
 c  
 FI;
- h) OP  $\ll \downarrow, DOWN, SHR \gg = (L \text{ BITS } x, \text{INT } n) L \text{ BITS} : x \uparrow -n;$
- i) OP ABS = ( $L$  BITS a)  $L$  INT :  
 BEGIN  $L$  INT c :=  $L$  0;  
 FOR i TO  $L$  bits width  
 DO c :=  $L2 \times c + K$  ABS (L F OF a) [i] OD;  
 c  
 END;

## ALGOL 68 REVISED REPORT

- j) OP BIN = (*L* INT *a*) *L* BITS :  
IF  $a \geq L \ 0$   
THEN *L* INT *b* := *a*; *L* BITS *c*;  
FOR *i* FROM *L* bits width BY - 1 TO 1  
DO (*L F* OF *c*) [*i*] := ODD *b*; *b* := *b* % *L* 2 OD;  
*c*  
FI;
- k) OP  $\ll$  ELEM,  $\square$   $\gg$  = (INT *a*, *L* BITS *b*) BOOL: (*L F* OF *b*) [*a*];
- l) PROC *L* bits pack = ([ ] BOOL *a*) *L* BITS :  
IF INT *n* =  $\lceil a[@1]$ ;  
*n*  $\leq L$  bits width  
THEN *L* BITS *c*;  
FOR *i* TO *L* bits width  
DO (*L F* OF *c*) [*i*] :=  
(*i*  $\leq L$  bits width - *n* | FALSE | *a*[@1][*i* - *L* bits width + *n*])  
OD;  
*c*  
FI;
- m) OP  $\ll$   $\neg$ ,  $\sim$ , NOT  $\gg$  = (*L* BITS *a*) *L* BITS :  
BEGIN *L* BITS *c*;  
FOR *i* TO *L* bits width DO (*L F* OF *c*) [*i*] :=  $\neg$  (*L F* OF *a*) [*i*] OD;  
*c*  
END;
- n) OP LENG = (*L* BITS *a*) LONG *L* BITS : long *L* bits pack (*a*);
- o) OP SHORTEN = (LONG *L* BITS *a*) *L* BITS: *L* bits pack ([ ] BOOL (*a*)  
[long *L* bits width - *L* bits width + 1 : ]);

### 25.2.3.10 Bytes and associated operations

- a) OP *R* = (*L* BYTES *a*, *b*) BOOL: STRING (*a*) *R* STRING (*b*);
- b) OP  $\ll$  ELEM,  $\square$   $\gg$  = (INT *a*, *L* BYTES *b*) CHAR: (*L F* OF *b*) [*a*];
- c) PROC *L* bytes pack = (STRING *a*) *L* BYTES:  
IF INT *n* =  $\lceil a$  [*@1*];  
*n*  $\leq L$  bytes width  
THEN *L* BYTES *c*;  
FOR *i* TO *L* bytes width  
DO (*L F* OF *c*) [*i*] := (*i*  $\leq n$  | *a*[@1][*i*] | null character) OD;  
*c*  
FI;
- d) OP LENG = (*L* BYTES *a*) LONG *L* BYTES: long *L* bytes pack (*a*);
- e) OP SHORTEN = (LONG *L* BYTES *a*) *L* BYTES:  
*L* bytes pack (STRING (*a*) [ : *L* bytes width]);



**25.2.3.11 Strings and associated operations**

- a) OP  $\lllt, LT \gggt = (\text{STRING } a, b) \text{ BOOL:}$   
 BEGIN INT m = [ a[@1], n = [ b[@1]; INT c := 0;  
 FOR i TO (m < n | m | n)  
 WHILE (c := ABS a [@1][i] - ABS b[@1][i]) = 0  
 DO SKIP OD;  
 (c = 0 | m < n  $\wedge$  n > 0 | c < 0)  
 END;
- b) OP  $\lllt, \leq, \leq, LE \gggt = (\text{STRING } a, b) \text{ BOOL: } \neg (b < a);$
- c) OP  $\lllt, =, EQ \gggt = (\text{STRING } a, b) \text{ BOOL: } a \leq b \wedge b \geq a;$
- d) OP  $\lllt, \neq, \neq, NE \gggt = (\text{STRING } a, b) \text{ BOOL: } \neg (a = b);$
- e) OP  $\lllt, \geq, \geq, GE \gggt = (\text{STRING } a, b) \text{ BOOL: } b \leq a;$
- f) OP  $\lllt, >, GT \gggt = (\text{STRING } a, b) \text{ BOOL: } b < a;$
- g) OP  $R = (\text{STRING } a, \text{CHAR } b) \text{ BOOL: } a R \text{ STRING } (b);$
- h) OP  $R = (\text{CHAR } a, \text{STRING } b) \text{ BOOL: } \text{STRING } (a) R b;$
- i) OP  $+ = (\text{STRING } a, b) \text{ STRING:}$   
 (INT m = (INT la = [ a[@1]; la < 0 | 0 | la),  
 n = (INT lb = [ b[@1]; lb < 0 | 0 | lb);  
 [1 : m + n] CHAR c;  
 c[1 : m] := a[@1]; c[m + 1 : m + n] := b[@1]; c);<sup>3</sup>
- j) OP  $+ = (\text{STRING } a, \text{CHAR } b) \text{ STRING: } a + \text{STRING } (b);$
- k) OP  $+ = (\text{CHAR } a, \text{STRING } b) \text{ STRING: } \text{STRING } (a) + b;$
- l) OP  $\lllt, \times, * \gggt = (\text{STRING } a, \text{INT } b) \text{ STRING: } (\text{STRING } c; \text{TO } b \text{ DO } c := c + a \text{ OD}; c);$
- m) OP  $\lllt, \times, * \gggt = (\text{INT } a, \text{STRING } b) \text{ STRING: } b \times a;$
- n) OP  $\lllt, \times, * \gggt = (\text{CHAR } a, \text{INT } b) \text{ STRING: } \text{STRING } (a) \times b;$
- o) OP  $\lllt, \times, * \gggt = (\text{INT } a, \text{CHAR } b) \text{ STRING: } b \times a;$

{The operations defined in a, g and h imply that if ABS "a" < ABS "b", then "<" "a"; "a" < "b"; "aa" < "ab"; "aa" < "ba"; "ab" < "b" and "ab" < "ba".}

---

<sup>3</sup>The assignments in this line have potential bounds errors when [ a[@1] < 0 or [ b[@1] < 0 and should read

(m > 0 | c[1 : m] := a[@1]); (n > 0 | c[m + 1 : m + n] := b[@1]); c

**25.2.3.12 Operations combined with assignments**

- a) OP  $\ll$  MINUSAB,  $-=$   $\gg$  = (REF  $L$  INT  $a$ ,  $L$  INT  $b$ ) REF  $L$  INT:  
 $a := a - b$ ;
- b) OP  $\ll$  MINUSAB,  $-=$   $\gg$  = (REF  $L$  REAL  $a$ ,  $L$  REAL  $b$ ) REF  $L$  REAL:  
 $a := a - b$ ;
- c) OP  $\ll$  MINUSAB,  $-=$   $\gg$  = (REF  $L$  COMPL  $a$ ,  $L$  COMPL  $b$ ) REF  $L$  COMPL:  
 $a := a - b$ ;
- d) OP  $\ll$  PLUSAB,  $+=$   $\gg$  = (REF  $L$  INT  $a$ ,  $L$  INT  $b$ ) REF  $L$  INT:  
 $a := a + b$ ;
- e) OP  $\ll$  PLUSAB,  $+=$   $\gg$  = (REF  $L$  REAL  $a$ ,  $L$  REAL  $b$ ) REF  $L$  REAL:  
 $a := a + b$ ;
- f) OP  $\ll$  PLUSAB,  $+=$   $\gg$  = (REF  $L$  COMPL  $a$ ,  $L$  COMPL  $b$ ) REF  $L$  COMPL:  
 $a := a + b$ ;
- g) OP  $\ll$  TIMESAB,  $\times:=$ ,  $*:=$   $\gg$  = (REF  $L$  INT  $a$ ,  $L$  INT  $b$ ) REF  $L$  INT :  
 $a := a \times b$ ;
- h) OP  $\ll$  TIMESAB,  $\times:=$ ,  $*:=$   $\gg$  = (REF  $L$  REAL  $a$ ,  $L$  REAL  $b$ ) REF  $L$  REAL:  
 $a := a \times b$ ;
- i) OP  $\ll$  TIMESAB,  $\times:=$ ,  $*:=$   $\gg$  = (REF  $L$  COMPL  $a$ ,  $L$  COMPL  $b$ ) REF  $L$  COMPL:  
 $a := a \times b$ ;
- j) OP  $\ll$  OVERAB,  $\div:=$ ,  $\%:=$   $\gg$  = (REF  $L$  INT  $a$ ,  $L$  INT  $b$ ) REF  $L$  INT:  
 $a := a \div b$ ;
- k) OP  $\ll$  MODAB,  $\div\times:=$ ,  $\div*:=$ ,  $\%*:=$   $\gg$  = (REF  $L$  INT  $a$ ,  $L$  INT  $b$ ) REF  $L$  INT:  
 $a := a \div \times b$ ;
- l) OP  $\ll$  DIVAB,  $/:=$   $\gg$  = (REF  $L$  REAL  $a$ ,  $L$  REAL  $b$ ) REF  $L$  REAL:  
 $a := a / b$ ;
- m) OP  $\ll$  DIVAB,  $/:=$   $\gg$  = (REF  $L$  COMPL  $a$ ,  $L$  COMPL  $b$ ) REF  $L$  COMPL:  
 $a := a / b$ ;
- n) OP  $Q$  = (REF  $L$  REAL  $a$ ,  $L$  INT  $b$ ) REF  $L$  REAL:  
 $a \ Q \ L \ \text{REAL} \ (b)$ ;
- o) OP  $Q$  = (REF  $L$  COMPL  $a$ ,  $L$  INT  $b$ ) REF  $L$  COMPL:  
 $a \ Q \ L \ \text{COMPL} \ (b)$ ;
- p) OP  $Q$  = (REF  $L$  COMPL  $a$ ,  $L$  REAL  $b$ ) REF  $L$  COMPL:  
 $a \ Q \ L \ \text{COMPL} \ (b)$ ;
- q) OP  $\ll$  PLUSAB,  $+=$   $\gg$  = (REF STRING  $a$ , STRING  $b$ ) REF STRING:  
 $a := a + b$ ;
- r) OP  $\ll$  PLUSTO,  $+=$   $\gg$  = (STRING  $a$ , REF STRING  $b$ ) REF STRING:  
 $b := a + b$ ;
- s) OP  $\ll$  PLUSAB,  $+=$   $\gg$  = (REF STRING  $a$ , CHAR  $b$ ) REF STRING:  
 $a += \ \text{STRING} \ (b)$ ;

- t) OP  $\ll$  PLUSTO, +=:  $\gg$  = (CHAR a, REF STRING b) REF STRING:  
 STRING (a) +=: b;
- u) OP  $\ll$  TIMESAB, x:=, \*:=  $\gg$  = (REF STRING a, INT b) REF STRING:  
 a := a  $\times$  b;

### 25.2.3.13 Standard mathematical constants and functions

- a)  $L$  REAL  $L$  pi = **C** a  $L$  real value close to  $\pi$ ; see Math. of Comp. V. 16, 1962, pp. 80-99 **C** ;
- b) PROC  $L$  sqrt = ( $L$  REAL x)  $L$  REAL: **C** if  $x \geq L$  0, a  $L$  real value close to the square root of 'x' **C** ;
- c) PROC  $L$  exp = ( $L$  REAL x)  $L$  REAL: **C** a  $L$  real value, if one exists, close to the exponential function of 'x' **C** ;
- d) PROC  $L$  ln = ( $L$  REAL x)  $L$  REAL: **C** a  $L$  real value, if one exists, close to the natural logarithm of x **C** ;
- e) PROC  $L$  cos = ( $L$  REAL x)  $L$  REAL: **C** a  $L$  real value close to the cosine of 'x' **C** ;
- f) PROC  $L$  arccos = ( $L$  REAL x)  $L$  REAL: **C** if  $ABS\ x \leq L$  1, a  $L$  real value close to the inverse cosine of 'x',  $L$  0  $\leq L$  arccos (x)  $\leq L$  pi **C** ;
- g) PROC  $L$  sin = ( $L$  REAL x)  $L$  REAL: **C** a  $L$  real value close to the sine of 'x' **C** ;
- h) PROC  $L$  arcsin = ( $L$  REAL x)  $L$  REAL: **C** if  $ABS\ x \leq L$  1, a  $L$  real value close to the inverse sine of 'x',  $ABS\ L$  arcsin (x)  $\leq L$  pi /  $L$  2 **C** ;
- i) PROC  $L$  tan = ( $L$  REAL x)  $L$  REAL: **C** a  $L$  real value, if one exists, close to the tangent of 'x' **C** ;
- j) PROC  $L$  arctan = ( $L$  REAL x)  $L$  REAL: **C** a  $L$  real value close to the inverse tangent of 'x',  $ABS\ L$  arctan (x)  $\leq L$  pi /  $L$  2 **C** ;
- k) PROC  $L$  next random = (REF  $L$  INT a)  $L$  REAL:  
 (a :=  
**C** the next pseudo-random  $L$  integral value after 'a' from a uniformly distributed sequence on the interval [ $L$  0, maxint] **C** ;  
**C** the real value corresponding to 'a' according to some mapping of integral values [ $L$  0,  $L$  max int] into real values [ $L$  0,  $L$  1) {i.e., such that  $0 \leq x < 1$ } such that the sequence of real values so produced preserves the properties of pseudo-randomness and uniform distribution of the sequence of integral values **C** ) ;

## 25.2.4 Synchronization operations

The elaboration of a **parallel-clause**  $P$  {17<sub>3</sub>.3.1.c} in an environ  $E$  is termed a "parallel action". The elaboration of a constituent **unit** of  $P$  in  $E$  is termed a "process" of that parallel action.

Any elaboration  $A$  {in some environ} of either of the **ENCLOSED-clauses** delineated by the **pragmats** {24<sub>9</sub>.2.1.b} PR start of incompatible part PR and PR finish of incompatible part PR in the forms 25<sub>10</sub>.2.4.d and 25<sub>10</sub>.2.4.e is incompatible with {16<sub>2</sub>.1.4.2.e} any elaboration  $B$  of either of those **ENCLOSED-clauses** if  $A$  and  $B$  are descendent actions {16<sub>2</sub>.1.4.2.b} of different processes of some same parallel action.

- a) MODE SEMA = STRUCT (REF INT  $F$ );
- b) OP LEVEL = (INT a) SEMA: (SEMA s;  $F$  OF s := HEAP INT := a; s);
- c) OP LEVEL = (SEMA a) INT:  $F$  OF a;
- d) OP DOWN = (SEMA edsger) VOID:
 

```

      BEGIN REF INT dijkstra =  $F$  OF edsger;
      WHILE
        PR start of incompatible part PR
          IF dijkstra ≥ 1 THEN dijkstra -= 1; FALSE
          ELSE
            C let P be the process such that the elaboration of this pseudo-comment
            {2510.1.3.Step 7} is a descendent action of P, but not of any other process descended
            from P; the process P is halted {162.1.4.3.f} C ;
            TRUE
          FI
        PR finish of incompatible part PR
      DO SKIP OD
      END;
```
- e) OP UP = (SEMA edsger) VOID:
 

```

      PR start of incompatible part PR
      IF REF INT dijkstra =  $F$  OF edsger; (dijkstra += 1) ≥ 1
      THEN
        C all processes are resumed {162.1.4.3.g} which are halted because the integer re-
        ferred to by the name yielded by 'dijkstra' was smaller than one C
      FI
      PR finish of incompatible part PR;
```

## 25.3 Transput declarations

```

{"So it does!" said Pooh, "It goes in!"
 "So it does!" said Piglet, "And it comes out!"
 "Doesn't it?" said Eeyore,
 "It goes in and out like anything,"
 Winnie-the-Pooh,                               A.A. Milne. }
```

{Three ways of "transput" (i.e., input and output) are provided by the **standard-prelude**, viz., formatless transput {25<sub>10</sub>.3.3}, formatted transput {25<sub>10</sub>.3.5} and binary transput {25<sub>10</sub>.3.6}.}

## 25.3.1 Books, channels and files

{ "Books", "channels" and "files" model the transput devices of the physical machine used in the implementation. }

### 25.3.1.1 Books and backfiles

{aa) All information within the system is to be found in a number of "books". A book {a} is a structured value including a field `text` of the mode specified by `FLEXTTEXT {b}` which refers to information in the form of characters. The `text` has a variable number of pages, each of which may have a variable number of lines, each of which may have a variable number of characters. Positions within the `text` are indicated by a page number, a line number and a character number. The book includes a field `lpos` which indicates the "logical end" of the book, i.e., the position up to which it has been filled with information, a string `idf`, which identifies the book and which may possibly include other information, e.g., ownership, and fields `putting` and `users` which permit the book to be opened {25<sub>10</sub>.3.1.4.d} on more than one file simultaneously only if `putting` is not possible on any of them.

bb) The books in the system are accessed via a chain of backfiles. The chain of books available for opening {25<sub>10</sub>.3.1.4.dd} is referenced by `chainbfile`. A given book may be referenced by more than one backfile on this chain, thus allowing simultaneous access to a single book by more than one process {25<sub>10</sub>.2.4}. However such access can only be for reading a book, since only one process may access a book such that it may be written to {aa}. The chain of books which have been locked {25<sub>10</sub>.3.1.4.o} is referenced by `lockedbfile`.

cc) Simultaneous access by more than one process to the chain of backfiles is prevented by use of the semaphore `bfileprotect`, which provides mutual exclusion between such processes.

dd) Books may be created (e.g., by input) or destroyed (e.g., after output) by tasks (e.g., the operating system) in the **system-task-list** {25<sub>10</sub>.4.2}, such books being then added to or removed from the chain of backfiles.}

- a) `MODE N0 BOOK =`  
`STRUCT (FLEXTTEXT text,`  
`POS lpos  $\phi$  logical end of book  $\phi$ ,`  
`STRING idf  $\phi$  identification  $\phi$ ,`  
`BOOL putting  $\phi$  TRUE if the book may be written to  $\phi$ ,`  
`INT users  $\phi$  the number of times the book is opened  $\phi$ );`
- b) `MODE N0 TEXT = REF [][] CHAR,`  
`MODE N0 FLEXTTEXT = REF FLEX [] FLEX [] FLEX [] CHAR;`
- c) `MODE N0 POS = STRUCT (INT p, l, c);`
- d) `PRIO N0 BEYOND=5,`  
`OP BEYOND = (POS a, b) BOOL:`

## ALGOL 68 REVISED REPORT

```
IF p OF a < p OF b THEN FALSE
ELIF p OF a > p OF b THEN TRUE
ELIF l OF a < l OF b THEN FALSE
ELIF l OF a > l OF b THEN TRUE
ELSE c OF a > c OF b
FI;

e) MODE N0 BFILE = STRUCT (REF BOOK book, REF BFILE next);
f) REF BFILE N0 chainbfile := NIL;
g) REF BFILE N0 lockedbfile := NIL;
h) SEMA N0 bfileprotect = (SEMA s; F OF s := PRIM INT := 1; s);
```

### 25.3.1.2 Channels

{aa} A "channel" corresponds to one or more physical devices (e.g., a card reader, a card punch or a line printer, or even to a set up in nuclear physics the results of which are collected by the computer), or to a filestore maintained by the operating system. A channel is a structured value whose fields are routines returning truth values which determine the available methods of access to a book linked via that channel. Since the methods of access to a book may well depend on the book as well as on the channel (e.g., a certain book may have been trapped so that it may be read, but not written to), most of these properties depend on both the channel and the book. These properties may be examined by use of the environment enquiries provided for files {25<sub>10</sub>.3.1.3.ff}. Two environment enquiries are provided for channels. These are:

- `estab possible`, which returns `true` if another file may be "established" {25<sub>10</sub>.3.1.4.cc} on the channel;
- `standconv`, which may be used to obtain the default "conversion key" {bb} for the channel,

bb) A "conversion key" is a value of the mode specified by `CONV` which is used to convert characters to and from the values as stored in "internal" form and as stored in "external" form in a book. It is a structured value comprising a row of structures, each of which contains a value in internal form and its corresponding external value. The implementation may provide additional conversion keys in its **library-prelude**.

cc) Three standard channels are provided, with properties as defined below (e,f,g); the implementation may provide additional channels in its **library-prelude**. The channel number field is provided in order that different channels with otherwise identical possibilities may be distinguished.}

```
a) MODE CHANNEL =
    STRUCT (PROC (REF BOOK) BOOL N0 reset, N0 set, N0 get,
```

- ```

    N0 put, N0 bin, N0 compress, N0 reidf,
    PROC BOOL N0 estab, PROC POS N0 max pos,
    PROC (REF BOOK) CONV N0 standconv, INT N0 channel number);

```
- b) MODE N<sub>0</sub> CONV = STRUCT ([1: INT (SKIP)]  
STRUCT (CHAR internal, external) F);
- c) PROC estab possible = (CHANNEL chan) BOOL: estab OF chan;
- d) PROC standconv = (CHANNEL chan) PROC (REF BOOK) CONV: standconv OF chan;
- e) CHANNEL stand in channel = **C** a channel value whose field selected by 'get' is a routine which always returns true, and whose other fields are some suitable values **C** ;
- f) CHANNEL stand out channel = **C** a channel value whose field selected by 'put' is a routine which always returns true, and whose other fields are some suitable values **C** ;
- g) CHANNEL stand back channel = **C** a channel value whose fields selected by 'set', 'reset', 'get', 'put' and 'bin' are routines which always return true, and whose other fields are some suitable values **C** ;

### 25.3.1.3 Files

{aa) A "file" is the means of communication between a **particular-program** and a book which has been opened on that file via some channel. It is a structured value which includes a reference to the book to which it has been linked {25<sub>10</sub>.3.1.4.bb} and a separate reference to the text of the book. The file also contains information necessary for the transput routines to work with the book, including its current position `cpos` in the text, its current "state" {bb}, its current "format" {25<sub>10</sub>.3.4} and the channel on which it has been opened.

bb) The "state" of a file is determined by five fields:

- `read mood`, which is true if the file is being used for input;
- `write mood`, which is true if the file is being used for output;
- `char mood`, which is true if the file is being used for character transput;
- `bin mood`, which is true if the file is being used for binary transput;
- `opened`, which is true if the file has been linked to a book.

cc) A file includes some "event routines", which are called when certain conditions arise during transput. After opening a file, the event routines provided by default return `false` when called, but the programmer may provide other event routines. Since the fields of a file are not directly accessible to the user, the event routines may be changed by use of the "on routines" (l,m,n,o,p,q,r) . The event routines are always given a reference to the file as a parameter. If the elaboration of an event routine is terminated, then the transput routine which called it can take no further action: otherwise, if it returns `true`, then it is assumed

## ALGOL 68 REVISED REPORT

that the condition has been mended in some way, and, if possible, transput continues, but if it returns `false`, then the system continues with its default action. The on routines are:

- on logical file end. The corresponding event routine is called when, during input from a book or as a result of calling `set`, the logical end of the book is reached (see 25<sub>10</sub>.3.1.6.dd).

**Example:** The programmer wishes to count the number of integers on his input tape. The file `intape` was opened in a surrounding **range**. If he writes:

```
BEGIN INT n := 0;
  on logical file end (intape, (REF FILE file) BOOL: GOTO f);
  DO get (intape, LOC INT); n += 1 OD;
  f: print (n)
END,
```

then the assignment to the field of `intape` violates the scope restriction, since the scope of the routine `(REF FILE file) BOOL: GOTO f` is smaller than the scope of `intape`, so he has to write:

```
BEGIN INT n := 0; FILE auxin := intape;
  on logical file end (auxin, (REF FILE file) BOOL: GOTO f);
  DO get (auxin, LOC INT); n += 1 OD;
  f: print (n)
END.
```

- on physical file end. The corresponding event routine is called when the current page number of the file exceeds the number of pages in the book and further transput is attempted (see 25<sub>10</sub>.3.1.6.dd),
- on page end. The corresponding event routine is called when the current line number exceeds the number of lines in the current page and further transput is attempted (see 25<sub>10</sub>.3.1.6.dd).
- on line end. The corresponding event routine is called when the current character number of the file exceeds the number of characters in the current line and further transput is attempted (see 25<sub>10</sub>.3.1.6.dd),

**Example:** The programmer wishes automatically to give a heading at the start of each page on his file `f`:

```
on page end (f, (REF FILE file) BOOL:
  (put (file, (newpage, "page number ", whole (i += 1, 0),
  newline)); TRUE)
ϕ it is assumed that i has been declared elsewhere ϕ)
```



- on `char error`. The corresponding event routine is called when a character conversion was unsuccessful or when, during input, a character is read which was not "expected" (25<sub>10</sub>.3.4.1.11). The event routine is called with a reference to a character suggested as a replacement. The event routine provided by the programmer may assign some character other than the suggested one. If the event routine returns `true`, then that suggested character as possibly modified is used,

Example: The programmer wishes to read sums of money punched as "\$123.45", "\$23.45", "\$3.45", etc.:

```
on char error (stand in, (REF FILE f, REF CHAR sugg) BOOL:
  IF sugg = "0"
  THEN CHAR c; backspace (f); get (f, c);
    (c = "$" | get (f, sugg); TRUE | FALSE)
  ELSE FALSE
  FI);
INT cents; readf (($z"."dd$, cents))
```

- on `value error`. The corresponding event routine is called when:
  - (i) during formatted transput an attempt is made to transput a value under the control of a "picture" with which it is incompatible, or when the number of "frames" is insufficient. If the routine returns `true`, then the current value and picture are skipped and transput continues; if the routine returns `false`, then first, on output, the value is output by `put`, and next `undefined` is called;
  - (ii) during input it is impossible to convert a string to a value of some given mode (this would occur if, for example, an attempt were made to read an integer larger than `max int` (25<sub>10</sub>.2.1.c)).
- on `format end`. The corresponding event routine is called when, during formatted transput, the format is exhausted while some value still remains to be transput. If the routine returns `true`, then `undefined` is called if a new format has not been provided for the file by the routine; otherwise, the current format is repeated.

dd) The `conv` field of a file is its current conversion key (25<sub>10</sub>.3.1.2.bb). After opening a file, a default conversion key is provided. Some other conversion key may be provided by the programmer by means of a call of `make conv {j}`. Note that such a key must have been provided in the **library-prelude**.

ee) The routine `make term` is used to associate a string with a file. This string is used when inputting a variable number of characters, any of its characters serving as a terminator.

ff) The available methods of access to a book which has been opened on a file may be discovered by calls of the following routines (note that the yield of such a call may be a function of both the book and the channel, and of other environmental factors not defined by this Report):

## ALGOL 68 REVISED REPORT

- `get possible`, which returns `true` if the file may be used for input;
- `put possible`, which returns `true` if the file may be used for output;
- `bin possible`, which returns `true` if the file may be used for binary transput;
- `compressible`, which returns `true` if lines and pages will be compressed {25<sub>10</sub>.3.1.6.aa} during output, in which case the book is said to be "compressible";
- `reset possible`, which returns `true` if the file may be reset, i.e., its current position set to (1, 1, 1);
- `set possible`, which returns `true` if the file may be set, i.e., the current position changed to some specified value; the book is then said to be a "random access" book and, otherwise, a "sequential access" book;
- `reidf possible`, which returns `true` if the `idf` field of the book may be changed;
- `chan`, which returns the channel on which the file has been opened {this may be used, for example, by a routine assigned by `on physical file end`, in order to open another file on the same channel}.

gg) On sequential access books, `undefined` (25<sub>10</sub>.3.1.4.a) is called if binary and character transput is alternated, i.e., after opening or resetting (25<sub>10</sub>.3.1.6.j), either is possible but, once one has taken place, the other may not until after another reset.

hh) On sequential access books, output immediately causes the logical end of the book to be moved to the current position (unless both are in the same line); thus input may not follow output without first resetting (25<sub>10</sub>.3.1.6.j).

### Example:

```
BEGIN FILE f1, t2; [1: 10000] INT x; INT n := 0;
  open (f1, "", channel 2);
  f2 := f1;
  ¢ now f1 and f2 can be used interchangeably ¢
  make conv (f1, flexocode); make conv (f2, telexcode);
  ¢ now f1 and f2 use different codes; flexocode and telexcode are
  defined in the library-prelude for this implementation ¢
  reset (f1);
  ¢ consequently, f2 is reset too ¢
  on logical file end (f1, (REF FILE f) BOOL: GOTO done);
  FOR i DO get (f1, x [i]); n := i OD;
  ¢ too bad if there are more than 10000 integers in the input ¢
done:
  reset (f1); FOR i TO n DO put (f2, x [i]) OD;
  close (f2) ¢ f1 is now closed too ¢
END }
```

- a) MODE FILE =  
 STRUCT (REF BOOK  $\aleph_0$  book,  
 UNION (FLEXTEXT, TEXT)  $\aleph_0$  text,  
 CHANNEL  $\aleph_0$  chan,  
 REF FORMAT  $\aleph_0$  format,  
 REF INT  $\aleph_0$  forp,  
 REF BOOL  $\aleph_0$  read mood,  $\aleph_0$  write mood,  $\aleph_0$  char mood,  $\aleph_0$  bin mood,  
 $\aleph_0$  opened,  
 REF POS  $\aleph_0$  cpos  $\zeta$  current position  $\zeta$ ,  
 STRING  $\aleph_0$  term  $\zeta$  string terminator  $\zeta$ ,  
 CONV  $\aleph_0$  conv  $\zeta$  character conversion key  $\zeta$ ,  
 PROC (REF FILE) BOOL  $\aleph_0$  logical file mended,  
 $\aleph_0$  physical file mended,  $\aleph_0$  page mended,  
 $\aleph_0$  line mended,  $\aleph_0$  format mended,  
 $\aleph_0$  value error mended,  
 PROC (REF FILE, REF CHAR) BOOL  $\aleph_0$  char error mended);
- b) PROC get possible = (REF FILE f) BOOL:  
 (opened OF f | (get OF chan OF f) (book OF f) | undefined; SKIP);
- c) PROC put possible = (REF FILE f) BOOL:  
 (opened OF f | (put OF chan OF f) (book OF f) | undefined; SKIP);
- d) PROC bin possible = (REF FILE f) BOOL:  
 (opened OF f | (bin OF chan OF f) (book OF f) | undefined; SKIP);
- e) PROC compressible = (REF FILE f) BOOL:  
 (opened OF f | (compress OF chan OF f) (book OF f) | undefined; SKIP);
- f) PROC reset possible = (REF FILE f) BOOL:  
 (opened OF f | (reset OF chan OF f) (book OF f) | undefined; SKIP);
- g) PROC set possible = (REF FILE f) BOOL:  
 (opened OF f | (set OF chan OF f) (book OF f) | undefined; SKIP);
- h) PROC reidf possible = (REF FILE f) BOOL:  
 (opened OF f | (reidf OF chan OF f) (book OF f) | undefined; SKIP);
- i) PROC chan = (REF FILE f) CHANNEL:  
 (opened OF f | chan OF f | undefined; SKIP);
- j) PROC make conv = (REF FILE f, PROC (REF BOOK) CONV c) VOID:  
 (opened OF f | conv OF f := c (book OF f) | undefined);
- k) PROC make term = (REF FILE f, STRING t) VOID: term OF f := t;
- l) PROC on logical file end = (REF FILE f, PROC (REF FILE) BOOL p) VOID:  
 logical file mended OF f := p;
- m) PROC on physical file end = (REF FILE f, PROC (REF FILE) BOOL p) VOID:  
 physical file mended OF f := p;
- n) PROC on page end = (REF FILE f, PROC (REF FILE) BOOL p) VOID:  
 page mended OF f := p;
- o) PROC on line end = (REF FILE f, PROC (REF FILE) BOOL p) VOID:  
 line mended OF f := p;

## ALGOL 68 REVISED REPORT

- p) PROC on format end = (REF FILE f, PROC (REF FILE) BOOL p) VOID:  
format mended OF f := p;
- q) PROC on value error = (REF FILE f, PROC (REF FILE) BOOL p) VOID:  
value error mended OF f := p;
- r) PROC on char error = (REF FILE f, PROC (REF FILE,  
REF CHAR) BOOL p) VOID:  
char error mended OF f := p;
- s) PROC reidf = (REF FILE f, STRING idf) VOID:  
IF opened OF f  $\wedge$  reidf possible (f)  $\wedge$  idf ok (idf)  
THEN idf OF book OF f := idf  
FI;

### 25.3.1.4 Opening and closing files

{aa) When, during transport, something happens which is left undefined, for example by explicitly calling `undefined` (a), this does not imply that the elaboration is catastrophically and immediately interrupted (2.1.4.3.h), but only that some sensible action is taken which is not or cannot be described by this Report alone and is generally implementation-dependent.

bb) A book is "linked" with a file by means of `establish` (b), `create` (c) or `open` (d). The linkage may be terminated by means of `close` (n), `lock` (o) or `scratch` (p).

cc) When a file is "established" on a channel, then a book is generated {19<sub>5</sub>.2.3} with a text of the given size, the given identification string, with `putting set` to true, and the logical end of the book at (1, 1, 1). An implementation may require (g) that the characters forming the identification string should be taken from a limited set and that the string should be limited in length, it may also prevent two books from having the same string. If the establishing is completed successfully, then the value 0 is returned; otherwise, some nonzero integer is returned (the value of this integer might indicate why the file was not established successfully). When a file is "created" on a channel, then a file is established with a book whose `text` has the default size for the channel and whose identification string is undefined.

dd) When a file is "opened", then the chain of backfiles is searched for the first book which is such that `match` (h) returns true. (The precise method of matching is not defined by this Report and will, in general, be implementation dependent. For example, the string supplied as parameter to `open` may include a password of some form.) If the end of the chain of backfiles is reached or if a book has been selected, but `putting` of the book yields true, or if `putting` to the book via the channel is possible and the book is already open, then the further elaboration is undefined. If the file is already open, an UP gremlins provides an opportunity for an appropriate system action on the book previously linked (in case no other copy of the file remains to preserve that linkage).

ee) The routine `associate` may be used to associate a file with a value of the mode specified by either `REF [] CHAR`, `REF [][] CHAR` or `REF [][] [] CHAR`, thus enabling such variables to be used as the book of a file,

ff) When a file is "closed", its book is attached to the chain of backfiles referenced by `chainbfile`. Some **system-task** is then activated by means of an UP gremlins. (This may reorganize the chain of backfiles, removing this book, or adding further copies of it. It may also cause the book to be output on some external device.)

gg) When a file is "locked", its book is attached to the chain of backfiles referenced by `lockedbfile`. Some **system-task** is then activated by means of an UP gremlins. A book which has been locked cannot be re-opened until some subsequent **system-task** has re-attached the book to the chain of backfiles available for opening.

hh) When a file is "scratched", some **system-task** is activated by means of an UP gremlins. (This may cause the book linked to the file to be disposed of in some manner.) }

a) PROC  $N_0$  undefined = INT:

**C** some sensible system action yielding an integer to indicate what has been done; it is presumed that the system action may depend on a knowledge of any values accessible {16<sub>2</sub>.1.2.c} inside the locale of any environ which is older than that in which this pseudo-comment is being elaborated notwithstanding that no Algol 68 construct written here could access those values **C** ;

b) PROC establish =

```
(REF FILE file, STRING idf, CHANNEL chan, INT p, l, c) INT:
BEGIN
  DOWN bfileprotect;
  PRIM BOOK book :=
    (PRIM FLEX [1 : p] FLEX [1 : l] FLEX [1 : c] CHAR,
     (1, 1, 1), idf, TRUE, 1);
  IF file available (chan) ^ (put OF chan) (book)
  ^ estab OF chan ^ ¬ (POS (p, l, c) BEYOND max pos OF chan)
  ^ ¬ (POS (1, 1, 1) BEYOND POS (p, l, c)) ^ idfok (idf)
  THEN
    (opened OF file | UP gremlins | UP bfileprotect);
    file :=
      (book, text OF book, chan, SKIP, SKIP,
       ¢ state: ¢ HEAP BOOL := FALSE, HEAP BOOL := TRUE,
       HEAP BOOL := FALSE, HEAP BOOL := FALSE, HEAP BOOL := TRUE,
       HEAP POS := (1, 1, 1), "", (standconv OF chan) (book),
       ¢ event routines: ¢ false, false, false, false, false, false,
       (REF FILE f, REF CHAR a) BOOL: FALSE);
    (¬ bin possible (file) | set char mood (file));
  0
  ELSE UP bfileprotect; undefined
  FI
END;
```

c) PROC create = (REF FILE file, CHANNEL chan) INT:

## ALGOL 68 REVISED REPORT

```
BEGIN POS max pos = max pos OF chan;
  establish (file, SKIP, chan,
    p OF max pos, l OF max pos, c OF max pos)
END;

d) PROC open = (REF FILE file, STRING idf, CHANNEL chan) INT:
BEGIN
  DOWN bfileprotect;
  IF file available (chan)
  THEN REF REF BFILE bf := chainbfile; BOOL found := FALSE;
    WHILE (REF BFILE (bf)  $\neq$  NIL)  $\wedge$   $\neg$  found
    DO
      IF match (idf, chan, book OF bf)
      THEN found := TRUE
      ELSE bf := next OF bf
      FI
    OD;
  IF  $\neg$  found
  THEN UP bfileprotect; undefined
  ELSE REF BOOK book := book OF bf;
    IF putting OF book  $\vee$  (put OF chan) (book)  $\wedge$  users OF book > 0
    THEN
      UP bfileprotect; undefined  $\phi$  in this case opening is
        inhibited by other users - the system may either
        wait, or yield nonzero (indicating unsuccessful
        opening) immediately  $\phi$ 
    ELSE
      users OF book += 1;
      ((put OF chan) (book) | putting OF book := TRUE);
      REF REF BFILE (bf) := next OF bf;  $\phi$  remove bfile from chain  $\phi$ 
      (opened OF file | UP gremlins | UP bfileprotect);
      file :=
        (book, text OF book, chan, SKIP, SKIP,
           $\phi$  state:  $\phi$  HEAP BOOL := FALSE, HEAP BOOL := FALSE,
          HEAP BOOL := FALSE, HEAP BOOL := FALSE,
          HEAP BOOL := TRUE,
          HEAP POS := (1, 1, 1), "", (standconv OF chan) (book),
           $\phi$  event routines:  $\phi$  false, false, false, false, false,
          false, (REF FILE f, REF CHAR a) BOOL: FALSE);
      ( $\neg$  bin possible (file) | set char mood (file));
      ( $\neg$  get possible (file) | set write mood (file));
      ( $\neg$  put possible (file) | set read mood (file));
      0
    FI
  FI
  ELSE UP bfileprotect; undefined
  FI
END;

e) PROC associate =
  (REF FILE file, REF [][][][] CHAR sss) VOID:
```

```

IF // ANW - this fails when UPB sss < 1 or UPB sss[1] < 1 gg
  INT p = LWB sss; INT l = LWB sss [p]; INT c = LWB sss[p][1];
  p = 1 ^ l = 1 ^ c = 1
THEN
  PROC t = (REF BOOK a) BOOL: TRUE;
  PROC f = (REF BOOK a) BOOL: FALSE;
  CHANNEL chan = (t, t, t, t, f, f, f, BOOL: FALSE,
    POS: (max int, max int, max int), SKIP, SKIP);
  (opened OF file | DOWN bfileprotect; UP gremlins);
  file :=
    (HEAP BOOK := (SKIP, (UPB sss + 1, 1, 1),
      SKIP, TRUE, 1), sss, chan,
      SKIP, SKIP,
      ¢ state: ¢ HEAP BOOL := FALSE, HEAP BOOL := FALSE,
      HEAP BOOL := TRUE, HEAP BOOL := FALSE, HEAP BOOL := TRUE,
      HEAP POS := (1, 1, 1), "", SKIP,
      ¢ event routines: ¢ false, false, false, false, false, false,
      (REF FILE f, REF CHAR a) BOOL: FALSE)
ELSE undefined
FI;

f) PROC N0 file available = (CHANNEL chan) BOOL:
  ¢ true if another file, at this instant of time, may be opened on
  'chan' and false otherwise ¢;

g) PROC N0 idf ok = (STRING idf) BOOL:
  ¢ true if 'idf' is acceptable to the implementation as the
  identification of a new book and false otherwise ¢;

h) PROC N0 match = (STRING idf, CHANNEL chan, REF BOOK book name) BOOL:
  ¢ true if the book referred to by 'book name' may be identified by
  'idf', and if the book may legitimately be accessed through
  'chan', and false otherwise ¢;

i) PROC N0 false = (REF FILE file) BOOL: FALSE
  ¢ this is included for brevity in 'establish', 'open' and
  'associate' ¢;

j) PROC N0 set write mood = (REF FILE f) VOID:
  IF ¬ put possible (f) OR
    ¬ set possible (f) ^ bin mood OF f ^ read mood OF f
  THEN undefined
  ELSE REF BOOL (read mood OF f) := FALSE;
  REF BOOL (write mood OF f) := TRUE
  FI;

k) PROC N0 set read mood = (REF FILE f) VOID:
  IF ¬ get possible (f) OR
    ¬ set possible (f) ^ bin mood OF f write mood OF f
  THEN undefined
  ELSE REF BOOL (read mood OF f) := TRUE;
  REF BOOL (write mood OF f) := FALSE
  FI;

```

## ALGOL 68 REVISED REPORT

- l) PROC  $\aleph_0$  set char mood = (REF FILE f) VOID:  
IF  $\neg$  set possible (f)  $\wedge$  bin mood OF f  
THEN undefined  
ELSE REF BOOL (char mood OF f) := TRUE;  
REF BOOL (bin mood OF f) := FALSE  
FI;
- m) PROC  $\aleph_0$  set bin mood = (REF FILE f) VOID:  
IF  $\neg$  bin possible (f)  $\vee$   $\neg$  set possible (f)  $\wedge$  char mood OF f  
THEN undefined  
ELSE REF BOOL (char mood OF f) := FALSE;  
REF BOOL (bin mood OF f) := TRUE  
FI;
- n) PROC close = (REF FILE file) VOID:  
IF opened OF file  
THEN  
DOWN bfileprotect;  
REF BOOL (opened OF file) := FALSE;  
REF BOOK book = book OF file;  
putting OF book := FALSE; users OF book -= 1;  
(text OF file | (FLEXTEXT): chainbfile :=  
PRIM BFILE := (book, chainbfile));  
UP gremlins  
FI;
- o) PROC lock (REF FILE file) VOID:  
IF opened OF file  
THEN DOWN bfileprotect;  
REF BOOL (opened OF file) := FALSE;  
REF BOOK book = book OF file;  
putting OF book := FALSE; users OF book -= 1;  
(text OF file | (FLEXTEXT): lockedbfile :=  
PRIM BFILE = (book, lockedbfile));  
UP gremlins  
FI;
- p) PROC scratch = (REF FILE file) VOID:  
IF opened OF file  
THEN  
DOWN bfileprotect;  
REF BOOL (opened OF file) := FALSE;  
putting OF book OF file := FALSE;  
users OF book OF file -= 1;  
UP gremlins  
FI;

### 25.3.1.5 Position enquiries



{aa) The "current position" of a book opened on a given file is the value referred to by the `cpos` field of that file. It is advanced by each transput operation in accordance with the number of characters written or read.

If  $c$  is the current character number and  $lb$  is the length of the current line, then at all times  $1 \leq c \leq lb + 1$ .  $c = 1$  implies that the next transput operation will be to the first character of the line and  $c = lb + 1$  implies that the line has overflowed and that the next transput operation will call an event routine. If  $lb = 0$ , then the line is empty and is therefore always in the overflowed state. Corresponding restrictions apply to the current line and page numbers. Note that, if the page has overflowed, the current line is empty and, if the book has overflowed, the current page and line are both empty (e).

bb) The user may determine the current position by means of the routines `char number`, `line number` and `page number` (a, b, c).

cc) If the current position has overflowed the line, page or book, then it is said to be outside the "physical file" (f, g, h).

dd) If, on reading, the current position is at the logical end, then it is said to be outside the "logical file" (i).}

{Each routine in this section calls `undefined` if the file is not open on entry.}

```
a) PROC char number = (REF FILE f) INT:
    (opened OF f | c OF cpos OF f | undefined);

b) PROC line number = (REF FILE f) INT:
    (opened OF f | l OF cpos OF f | undefined);

c) PROC page number = (REF FILE f) INT:
    (opened OF f | p OF cpos OF f | undefined);

d) PROC  $\aleph_0$  current pos = (REF FILE f) POS:
    (opened OF f | cpos OF f | undefined; SKIP);

e) PROC  $\aleph_0$  book bounds = (REF FILE f) POS:
    BEGIN POS cpos = current pos (f);
    INT p = p OF cpos, l = l OF cpos;
    CASE text OF f IN
        (TEXT t1):
            (INT pb = UPB t1;
             INT lb = (p ≤ 0 ∨ p > pb | 0 | UPB t1 [p]);
             INT cb = (l ≤ 0 ∨ l > lb | 0 | UPB t1[p][l]);
             (pb, lb, cb)),
        (FLEXTTEXT t2):
            (INT pb = UPB t2;
             INT lb = (p ≤ 0 ∨ p > pb | 0 | UPB t2[p]);
             INT cb = (l ≤ 0 ∨ l > lb | 0 | UPB t2[p][l]);
             (pb, lb, cb))
    ESAC
END;
```

## ALGOL 68 REVISED REPORT

- f) PROC  $N_0$  line ended = (REF FILE f) BOOL:  
(INT c = c OF current pos (f); c > c OF book bounds (f));
- g) PROC  $N_0$  page ended = (REF FILE f) BOOL:  
(INT l = l OF current pos (f); l > l OF book bounds (f));
- h) PROC  $N_0$  physical file ended = (REF FILE f) BOOL:  
(INT p = p OF current pos (f); p > p OF book bounds (f));
- i) PROC  $N_0$  logical file ended = (REF FILE f) BOOL:  
(lpos OF book OF f BEYOND current pos (f));

### 25.3.1.6 Layout routines

{aa} A book input from an external medium by some **system-task** may contain lines and pages not all of the same length. Contrariwise, the lines and pages of a book which has been established (25<sub>10</sub>.3.1.4.cc) are all initially of the size specified by the user. However if, during output to a compressible book (25<sub>10</sub>.3.1.3.ff), `newline (newpage)` is called with the current position in the same line (page) as the logical end of the book, then that line (the page containing that line) is shortened to the character number (line number) of the logical end. Thus `print ("abcde", newline)` could cause the current line to be reduced to 5 characters in length. Note that it is perfectly meaningful for a line to contain no characters and for a page to contain no lines.

Although the effect of a channel whose books are both compressible and of random access (25<sub>10</sub>.3.1.3.ff) is well defined, it is not anticipated that such a combination is likely to occur in actual implementations.

bb) The routines `space` (a), `newline` (c) and `newpage` (d) serve to advance the current position to the next character, line or page, respectively. They do not, however, (except as provided in cc below) alter the contents of the positions skipped over. Thus `print ("a", backspace, space)` has a different effect from `print ("a", backspace, blank)`.

The current position may be altered also by calls of `backspace` (b), `set char number` (k) and, on appropriate channels, of `set` (i) and `reset` (j).

cc) The contents of a newly established book are undefined and both its current position and its logical end are at (1, 1, 1). As output proceeds, it is filled with characters and the logical end is moved forward accordingly. If, during character output with the current position at the logical end of the book, `space` is called, then a space character is written (similar action being taken in the case of `newline` and `newpage` if the book is not compressible).

A call of `set` which attempts to leave the current position beyond the logical end results in a call of `undefined` (a sensible system action might then be to advance the logical end to the current position, or even to the physical end of the book). There is thus no defined way

in which the current position can be made to be beyond the logical end, nor in which any character within the logical file can remain in its initial undefined state.

dd) A reading or writing operation, or a call of `space`, `newline`, `newpage`, `set` or `set char number`, may bring the current position outside the physical or logical file (25<sub>10</sub>.3.1.5.cc, dd), but this does not have any immediate consequence. However, before any further transput is attempted, or a further call of `space`, `newline` or `newpage` (but not of `set` or `set char number`) is made, the current position must be brought to a "good" position. The file is "good" if, on writing (reading), the current position is not outside the physical (logical) file (25<sub>10</sub>.3.1.5.cc, dd). The page (line) is "good" if the line number (character number) has not overflowed. The event routine (25<sub>10</sub>.3.1.3.cc) corresponding to `on logical file end`, `on physical file end`, `on page end` or `on line end` is therefore called as appropriate. Except in the case of formatted transput (which uses `check pos`, 25<sub>10</sub>.3.3.2.c), the default action, if the event routine returns `false`, is to call, respectively, `undefined`, `undefined`, `newpage` or `newline`. After this (or if `true` is returned), if the position is still not good, an event routine (not necessarily the same one) is called again.

ee) The state of the file (25<sub>10</sub>.3.1.3.bb) controls some effects of the layout routines. If the read/write mood is reading, the effect of `space`, `newline` and `newpage`, upon attempting to pass the logical end, is to call the event routine corresponding to `on logical file end` with default action `undefined`; if it is writing, the effect is to output spaces (or, in `bin` mood, to write some undefined character) or to compress the current line or page (see cc) . If the read/write mood is not determined on entry to a layout routine, `undefined` is called. On exit, the read/write mood present on entry is restored.)

```
a) PROC space = (REF FILE f) VOID:
  IF ¬ opened OF f THEN undefined
  ELSE
    BOOL reading = (read mood OF f | TRUE |:
      write mood OF f | FALSE | undefined; SKIP);
    (¬ get good line (f, reading) | undefined);
    REF POS cpos = cpos OF f;
    IF reading THEN c OF cpos += 1
    ELSE IF logical file ended (f) THEN
      IF bin mood OF f THEN
        (text OF f | (FLEXTEXT t2):
          t2[p OF cpos][l OF cpos][c OF cpos] := SKIP);
        c OF pos += 1;
        lpos OF book OF f := cpos
      ELSE put char (f, " ")
    FI
    ELSE c OF cpos += 1
  FI
FI;

b) PROC backspace = (REF FILE f) VOID:
  IF ¬ opened OF f THEN undefined
```

## ALGOL 68 REVISED REPORT

```
ELSE REF INT c = c OF cpos OF f;  
  (c>1 | c-:= 1 | undefined)  
FI;
```

```
c) PROC newline = (REF FILE f) VOID:  
  IF ¬ opened OF f THEN undefined  
  ELSE BOOL reading = (read mood OF f | TRUE |:  
    write mood OF f | FALSE | undefined; SKIP);  
  (get good page (f, reading) | undefined);  
  REF POS cpos = cpos OF f, lpos = lpos OF book OF f;  
  IF p OF cpos = p OF lpos ∧ l OF cpos = l OF lpos  
  THEN c OF cpos := c OF lpos;  
  IF reading THEN newline (f)  
  ELSE  
    IF compressible (f)  
    THEN REF INT pl = p OF lpos, ll = l OF lpos;  
      FLEXTTEXT text = (text OF f | (FLEXTTEXT t2): t2);  
      text [pl][ll] := text [pl][ll][ : c OF lpos - 1]  
    ELSE WHILE ¬ line ended (f) DO space (f) OD  
    FI;  
    cpos := lpos := (p OF cpos, l OF cpos + 1, 1)  
  FI  
  ELSE cpos := (p OF cpos, l OF cpos + 1, 1)  
  FI  
  FI;
```

```
d) PROC newpage = (REF FILE f) VOID:  
  IF ¬ opened OF f THEN undefined  
  ELSE  
    BOOL reading = (read mood OF f) | TRUE |:  
      write mood OF f | FALSE | undefined; SKIP);  
    (get good file (f, reading) | undefined);  
    REF POS cpos = cpos OF f, lpos = lpos OF book OF f;  
    IF p OF cpos = p OF lpos  
    THEN cpos := lpos;  
    IF reading THEN newpage (f)  
    ELSE  
      IF compressible (f) ∧ l OF lpos ≤ l OF book bounds (f)  
      THEN REF INT pl = p OF lpos, ll = l OF lpos;  
        FLEXTTEXT text = (text OF f | (FLEXTTEXT t2): t2);  
        text[pl][ll] := text [pl][ll]t: c OF lpos - 1];  
        text[pl] := text [pl][ : (c OF lpos > 1 | ll | ll - 1)]  
      ELSE WHILE ¬ page ended (f) DO newline (f) OD  
      FI;  
      cpos := lpos := (p OF cpos + 1, 1, 1)  
    FI  
    ELSE cpos := (p OF cpos + 1, 1, 1)  
    FI  
  FI;
```

{Each of the following 3 routines either returns `true`, in which case the line, page or file is good (dd), or it returns `false`, in which case the current position may be outside the logical file or the page number may have overflowed, or it loops until the matter is resolved, or it is terminated by a **jump**. On exit, the read/write mood is as determined by its reading parameter.}

- e) PROC  $\aleph_0$  get good line = (REF FILE f, BOOL reading) BOOL:  
 BEGIN BOOL not ended;  
 WHILE not ended := get good page (f, reading);  
 line ended (f)  $\wedge$  not ended  
 DO ( $\neg$  (line mended OF f) (f) |  
 set mood (f, reading); newline (f)) OD;  
 not ended  
 END;
- f) PROC  $\aleph_0$  get good page = (REF FILE f, BOOL reading) BOOL:  
 BEGIN BOOL not ended;  
 WHILE not ended := get good file (f, reading);  
 page ended (f)  $\wedge$  not ended  
 DO ( $\neg$  (page mended OF f) (f) |  
 set mood (f, reading); newpage (f)) OD;  
 not ended  
 END;
- g) PROC  $\aleph_0$  get good file = (REF FILE f, BOOL reading) BOOL:  
 BEGIN BOOL not ended := TRUE;  
 WHILE set mood (f, reading);  
 not ended  $\wedge$   
 (reading | logical file ended | physical file ended) (f)  
 DO not ended := (reading | logical file mended OF f  
 | physical file mended OF f) (f)  
 OD;  
 not ended  
 END;
- h) PROC  $\aleph_0$  set mood = (REF FILE f, BOOL reading) VOID:  
 (reading | set read mood (f) | set write mood (f));
- i) PROC set = (REF FILE f, INT p, l, c) VOID:  
 IF  $\neg$  opened OF f  $\vee$   $\neg$  get possible (f) THEN undefined  
 ELSE BOOL reading = (read mood OF f | TRUE  
 |: write mood OF f | FALSE undefined; SKIP);  
 REF POS cpos = cpos OF f, lpos = lpos OF book OF f;  
 POS ccpos = cpos;  
 IF (cpos := (p, l, c)) BEYOND lpos  
 THEN cpos := lpos;  
 ( $\neg$  (logical file mended OF f) (f) | undefined);  
 set mood (f, reading)  
 ELIF POS bounds = book bounds (f);  
 p < 1  $\vee$  p > p OF bounds + 1  
 $\vee$  l < 1  $\vee$  l > l OF bounds + 1

## ALGOL 68 REVISED REPORT

```
    V c < 1 V c > c OF bounds + 1
    THEN cpos := cpos; undefined
    FI
FI;

j) PROC reset = (REF FILE f) VOID:
    IF ¬ opened OF f V ¬ reset possible (f) THEN undefined
    ELSE
        REF BOOL (read mood OF f) := ¬ put possible (f);
        REF BOOL (write mood OF f) := ¬ get possible (f);
        REF BOOL (char mood OF f) := ¬ bin possible (f);
        REF BOOL (bin mood OF f) := FALSE;
        REF POS (cpos OF f) := (1, 1, 1)4
    FI;

k) PROC set char number = (REF FILE f, INT c) VOID:
    IF ¬ opened OF f THEN undefined
    ELSE REF REF POS cpos = cpos OF f;
        WHILE c OF cpos ≠ c
        DO
            IF c < 1 V c > c OF book bounds (f) + 1
            THEN undefined
            ELIF c > c OF cpos
            THEN space (f)
            ELSE backspace (f)
            FI
        OD
    FI;
FI;
```

### 25.3.2 Transput values

#### 25.3.2.1 Conversion routines

{The routines `whole`, `fixed` and `float` are intended to be used with the formatless output routines `put`, `print` and `write` when it is required to have a little extra control over the layout produced. Each of these routines has a `width` parameter whose absolute value specifies the length of the string to be produced by conversion of the arithmetic value  $V$  provided. Each of `fixed` and `float` has an `after` parameter to specify the number of digits required after the decimal point, and an `exp` parameter in `float` specifies the width allowed for the exponent. If  $V$  cannot be expressed as a string within the given width, even when the value of `after`, if provided, has been reduced, then a string filled with `errorchar` (25<sub>10</sub>.2.1.t) is returned instead. Leading zeroes are replaced by spaces and a sign is normally included. The user can, however, specify that a sign is to be included only for negative values by specifying a negative width. If the width specified is zero, then

---

<sup>4</sup>These lines should be followed by ; DOWN bfileprotect; UP gremlins

the shortest possible string into which  $V$  can be converted, consistently with the other parameters, is returned. The following examples illustrate some of the possibilities:

```
print (whole (i, -4))
```

which might print "`0`", "`99`", "`-99`", "`9999`" or, if  $i$  were greater than `9999`, "`****`", where "`*`" is the yield of `errorchar`;

```
print (whole (i, 4))
```

which would print "`+99`" rather than "`99`";

```
print (whole (i, 0))
```

which might print "`0`", "`99`", "`-99`", "`9999`" or "`99999`";

```
print (fixed (x, -6, 3))
```

which might print "`2.718`", "`27.183`" or "`271.83`" (in which one place after the decimal point has been sacrificed in order to fit the number in);

```
print (fixed (x, 0, 3))
```

which might print "`2.718`", "`27.183`" or "`271.823`";

```
print (float (x, 9, 3, 2))
```

which might print "`-2.71810+0`", "`+2.71810-1`", or "`+2.7210+11`" (in which one place after the decimal point has been sacrificed in order to make room for the unexpectedly large exponent) .}

a) `MODE  $\aleph_0$  NUMBER= UNION (<<  $L$  REAL >>, <<  $L$  INT >> )`;

b) `PROC whole = (NUMBER v, INT width) STRING:`

```
  CASE v IN
```

```
    << ( $L$  INT x):
```

```
      (INT length := ABS width - (x <  $L$  0  $\vee$  width > 0 | 1 | 0),
```

```
       $L$  INT n := ABS x;
```

```
      IF width = 0 THEN
```

```
         $L$  INT m := n; length := 0;
```

```
        WHILE m  $\div$ := 10; length += 1; m  $\neq$   $L$  0
```

```
        DO SKIP OD
```

```
      FI;
```

```
      STRING s := subwhole (n, length);
```

```
      IF length = 0  $\vee$  char in string (errorchar, LOC INT, s)
```

```
      THEN ABS width*errorchar5
```

```
      ELSE (x <  $L$  0 | "-" |: width > 0 | "+" | "") PLUSTO s;
```

```
        (width  $\neq$  0 | (ABS width - UPB s)  $\times$  " " PLUSTO s);
```

---

<sup>5</sup>This line should read `ELSE undefined`;

## ALGOL 68 REVISED REPORT

```

        s
        FI) >>,
        << (L REAL x): fixed (x, width, 0) >>
    ESAC;

c) PROC fixed (NUMBER v, INT width, after) STRING:
    CASE v IN
        << (L REAL x):
            IF INT length := ABS width - (x < L 0 V width > 0 | 1 | 0);
               after ≥ 0 ∧ (length > after V width = 0)
            THEN L REAL y = ABS x;
                IF width = 0
                THEN length := (after = 0 | 1 | 0);
                    WHILE y + L .5 × L .1 ↑ after ≥ L10.0 ↑ length
                    DO length += 1 OD;
                    length += (after = 0 | 0 | after + 1)
                FI;
                STRING s := sub fixed (y, length, after);
                IF ¬ char in string (errorchar, LOC INT, s)
                THEN (length > UPB s ∧ y < L 1.0 | "0" PLUSTO s);6
                    (x < L 0 | "-" |: width > 0 | "+" | "") PLUSTO s;
                    (width ≠ 0 | (ABS width - UPB s) × " " PLUSTO s);
                    s
                ELIF after > 0
                THEN fixed (v, width, after - 1)
                ELSE ABS width × errorchar
                FI
                ELSE undefined; ABS width × errorchar7
            FI >>,
        << (L INT x): fixed (L REAL (x), width, after) >>
    ESAC;

d) PROC float = (NUMBER v, INT width, after, exp) STRING:
    CASE v IN
        << (L REAL x):
            IF INT before = ABS width - ABS exp -
               (after ≠ 0 | after + 1 | 0) -2;
               SIGN before + SIGN after > 0
            THEN STRING s, L REAL y := ABS x, INT p := 0;
                L standardize (y, before, after, p);
                s := fixed (SIGN x × y, SIGN width × (ABS width - ABS exp - 1),
                    after)
                    + "10" + whole (p, exp);8
                IF exp = 0 V char in string (errorchar, LOC INT, s)
                THEN float (x, width, (after ≠ 0 | after - 1 | 0),

```

---

<sup>6</sup>This line should read THEN (length > UPB s ∧ y + L 0.5 + L .1 ↑ after < L 1.0 | "0" PLUSTO s);

<sup>7</sup>This line should read ELIF undefined; width = 0 THEN errorchar ELSE ABS width × errorchar

<sup>8</sup>This line is debated



- ```

        (exp > 0 | exp + 1 | exp - 1))
    ELSE s
    FI
    ELSE undefined; ABS width × errorchar9
    FI >>,
    << (L INT x): float (L REAL (x), width, after, exp) >>
ESAC;

```
- e) PROC  $\aleph_0$  subwhole = (NUMBER v, INT width) STRING:  
 $\zeta$  returns a string of maximum length 'width' containing a decimal representation of the positive integer 'v'  $\zeta$
- ```

CASE v IN
    << (L INT x):
        BEGIN STRING s, L INT n := x;
            WHILE dig char (S (n ÷× L 10)) PLUSTO s;
                n ÷:= L 10; n ≠ L 0
            DO SKIP OD;
            (UPB s > width | width × errorchar | s)
        END >>
ESAC;

```
- f) PROC  $\aleph_0$  subfixed = (NUMBER v, INT width, after) STRING:  
 $\zeta$  returns a string of maximum length 'width' containing a rounded decimal representation of the positive real number 'v'; if 'after' is greater than zero, this string contains a decimal point followed by 'after' digits  $\zeta$
- ```

CASE v IN
    << (L REAL x):
        BEGIN STRING s, INT before := 0;
            L REAL y := x + L .5 × L .1 ↑ after;
            PROC choosedig = (REF L REAL y) CHAR:
                dig char ((INT c := S ENTIER (y *:= L 10.0);
                    (c > 9 | c := 9);
                    y -:= K c; c));
            WHILE y > L 10.0 ↑ before DO before += 1 OD;
            y /:= L 10.0 ↑ before;
            TO before DO s PLUSAB choosedig (y) OD;
            (after > 0 | s PLUSAB ".");
            TO after DO s PLUSAB choosedig (y) OD;
            (UPB s > width | width × errorchar | s)
        END >>
ESAC;

```
- g) PROC  $\aleph_0$  L standardize = (REF L REAL y,  
 INT before, after, REF INT p) VOID:  
 $\zeta$  adjusts the value of 'y' so that it may be transput according to the format \$n (before) d.n (after) d\$ ; 'p' is set so that  $y \times 10 \uparrow p$  is equal to the original value of 'y'  $\zeta$

---

<sup>9</sup>This line should read ELIF undefined; width = 0 THEN errorchar ELSE ABS width × errorchar

## ALGOL 68 REVISED REPORT

```

BEGIN
  L REAL g = L 10.0 ↑ before; L REAL h = g × L .1;
  WHILE y > g DO y *:= L .1; p +:= 1 OD;
  (y ≠ L 0.0 | WHILE y < h DO y *:= L 10.0; p -:= 1 OD);
  (y + L .5 × L .1 ↑ after ≥ g | y := h; p +:= 1)
END;

h) PROC N0 dig char = (INT x) CHAR: "0123456789abcdef"[x + 1];

i) PROC N0 string to L int = (STRING s, INT radix, REF L INT i) BOOL:
  ⚡ returns true if the absolute value of the result is ≤ L max int ⚡
BEGIN
  L INT lr = K radix; BOOL safe := TRUE;
  L INT n := L 0, L INT m = L max int div lr;
  L INT ml = L max int - m × lr;
  FOR i FROM 2 TO UPB s
  WHILE L INT dig = K char dig (s[i]);
    safe := n < m ∨ n = m ∧ dig ≤ ml
  DO n := n × lr + dig OD;
  IF safe THEN i := (s[1] = "+" | n | -n); TRUE ELSE FALSE FI
END;

j) PROC N0 string to L real = (STRING s, REF L REAL r) BOOL:
  ⚡ returns true if the absolute value of the result is ≤ max real ⚡
BEGIN
  INT e := UPB s + 1;
  char in string ("10", e, s);
  INT p := e; char in string (".", p, s);
  INT j := 1, length = 0, L REAL x := L 0.0;
  ⚡ skip leading zeroes: ⚡
  FOR i FROM 2 TO e - 1
  WHILE s[i] = "0" ∨ s[i] = "." ∨ s[i] = " "
  DO j := i OD;
  FOR i FROM j + 1 TO e - 1 WHILE length < L real width
  DO IF s [i] ≠ "."
    THEN x := x × L 10.0 + K char dig (s[j := i]); length += 1
    FI ⚡ all significant digits converted ⚡
  OD;
  ⚡ set preliminary exponent: ⚡
  INT exp := (p > j | p - j - 1 | p - j), expart := 0;
  ⚡ convert exponent part: ⚡
  BOOL safe := IF e < UPB s
    THEN string to int (s[e + 1: ], 10, expart)
    ELSE TRUE
  FI;
  ⚡ prepare a representation of L max real to compare with the
  L REAL value to be delivered: ⚡
  L REAL max stag := L max real, INT max exp := 0;
  L standardize (max stag, length, 0, max exp); exp += expart;
  IF ¬ safe ∨ (exp > max exp ∨ exp = max exp ∧ x > max stag)
  THEN FALSE

```

- ```

    ELSE r := (s[1] = "+" | x | -x) × L 10.0 ↑ exp; TRUE
  FI
END;

```
- k) PROC  $\aleph_0$  char dig = (CHAR x) INT:  
 (x = " " | 0 | INT i;  
 char in string (x, i, "0123456789abcdef"); i - 1);
- l) PROC char in string = (CHAR c, REF INT i, STRING s) BOOL:  
 (BOOL found := FALSE;  
 FOR k FROM LWB s TO UPB s WHILE  $\neg$  found  
 DO (c = s[k] | i := k; found := TRUE) OD;  
 found);
- m) INT  $L$  int width =  
 $\zeta$  the smallest integral value such that ' $L$  max int' may be  
 converted without error using the pattern n ( $L$  int width) d  $\zeta$   
 (INT c := 1;  
 WHILE  $L 10 \uparrow (c-1) < L .1 \times L$  maxint DO c += 1 OD;  
 c);
- n) INT  $L$  real width =  
 $\zeta$  the smallest integral value such that different strings are  
 produced by conversion of '1.0' and of '1.0 +  $L$  small real'  
 using the pattern d.n ( $L$  real width - 1) d  $\zeta$   
 $1 - S$  ENTIER ( $L \ln (L$  small real) /  $L \ln (L 10)$ );
- o) INT  $L$  exp width =  
 $\zeta$  the smallest integral value such that ' $L$  max real' may be  
 converted without error using the pattern  
 d.n ( $L$  real width - 1) d e n ( $L$  exp width) d  $\zeta$   
 $1 + S$  ENTIER ( $L \ln (L \ln (L$  max real) /  $L \ln (L 10)$ ) /  $L \ln (L 10)$ );

### 25.3.2.2 Transput modes

- a) MODE  $\aleph_0$  SIMPLOUT = UNION ( $\ll L$  INT  $\gg$ ,  $\ll L$  REAL  $\gg$ ,  $\ll L$  COMPL  $\gg$ ,  
 BOOL,  $\ll L$  BITS  $\gg$ , CHAR, [ ] CHAR);
- b) MODE  $\aleph_0$  OUTTYPE =  
**C** an actual-declarer specifying a mode united from {16<sub>2</sub>.1.3.6.a} a sufficient set of modes  
 none of which is 'void' or contains 'flexible', 'reference to', 'procedure' or 'union of' C ;
- c) MODE  $\aleph_0$  SIMPLIN = UNION ( $\ll$  REF  $L$  INT  $\gg$ ,  $\ll$  REF  $L$  REAL  $\gg$ ,  
 $\ll$  REF  $L$  COMPL  $\gg$ , REF BOOL,  $\ll$  REF  $L$  BITS  $\gg$ ,  
 REF CHAR, REF [ ] CHAR, REF STRING);
- d) MODE  $\aleph_0$  INTYPE =  
**C** an actual-declarer specifying a mode united from {16<sub>2</sub>.1.3.6.a} 'reference to flexible row  
 of character' together with a sufficient set of modes each of which is 'reference to' followed  
 by a mode which does not contain 'flexible', 'reference to', 'procedure' or 'union of' C ;

{See the remarks after 25<sub>10</sub>.2.3.1 concerning the term "sufficient set".}

### 25.3.2.3 Straightening

a) OP  $\aleph_0$  STRAIGHTOUT = (OUTTYPE  $\times$ ) [] SIMPLOUT:  
**C** the result of "straightening" 'x' **C** ;

b) OP  $\aleph_0$  STRAIGHTIN = (INTYPE  $\times$ ) [] SIMPLIN:  
**C** the result of straightening 'x' **C** ;

c) The result of "straightening" a given value  $V$  is a multiple value  $W$  {of one dimension} obtained as follows:

- it is required that  $V$  (if it is a name) be not nil;
- a counter  $i$  is set to 0:
- $V$  is "traversed" {d} using  $i$
- $W$  is composed of, a descriptor  $((1, i))$  and the elements obtained by traversing  $V$ :
- if  $V$  is not (is) a name, then the mode of the result is the mode specified by [] SIMPLOUT ( [] SIMPLIN ).

d) A value  $V$  is "traversed", using a counter  $i$ , as follows:

If  $V$  is (refers to) a value from whose mode that specified by SIMPLOUT is united, then

- $i$  is increased by one:
- the element of  $W$  selected by  $(i)$  is  $V$ :

otherwise,

Case A:  $V$  is (refers to) a multiple value (of one dimension) having a descriptor  $((l, u))$ :

- for  $j = l, \dots, u$ , the element (the subname) of  $V$  selected by  $(j)$  is traversed using  $i$ ;

Case B:  $V$  is (refers to) a multiple value {of  $n$  dimensions,  $n \geq 2$ } whose descriptor is  $((l_1, u_1), \dots, (l_n, u_n))$  where  $n \geq 2$ :

- for  $j = l_1, \dots, u_1$ , the multiple value selected {16<sub>2</sub>.1.3.4.i} by (the name generated {16<sub>2</sub>.1.3.4.j} by) the trim  $(j, (l_2, u_2, 0), \dots, (l_n, u_n, 0))$  is traversed using  $i$ ;

Case C:  $V$  is (refers to) a structured value  $V1$ :

- the fields (the subnames of  $V$  referring to the fields) of  $V1$ , taken in order, are traversed using  $i$ .

### 25.3.3 Formatless transput

{In formatless transput, the elements of a "data list" are transput, one after the other, via a specified file. Each element of the data list is either a layout routine of the mode specified by PROC (REF FILE) VOID (25<sub>10</sub>.3.1.6) or a value of the mode specified by OUTTYPE (on output) or INTYPE (on input) . On encountering a layout routine in the data list, that routine is called with the specified file as parameter. Other values in the data list are first straightened (25<sub>10</sub>.3.2.3) and the resulting values are then transput via the given file one after the other.

Transput normally takes place at the current position but, if there is no room on the current line (on output) or if a readable value is not present there (on input), then first, the event routine corresponding to on line end (or, where appropriate, to on page end, on physical file end or on logical file end) is called, and next, if this returns false, the next "good" character position of the book is found, viz., the first character position of the next nonempty line.}

#### 25.3.3.1 Formatless output

{For formatless output, put (a) and print (or write) (25<sub>10</sub>.5.1.d) may be used. Each straightened value  $V$  from the data list is output as follows:

aa) If the mode of  $V$  is specified by  $L$  INT, then first, if there is not enough room for  $L$  int width + 2 characters on the remainder of the current line, a good position is found on a subsequent line (see 25<sub>10</sub>.3.3): next, when not at the beginning of a line, a space is given and then  $V$  is output as if under the control of the **picture** `n (L int width - 1) z + d`.

bb) If the mode of  $V$  is specified by  $L$  REAL, then first, if there is not enough room for  $L$  real width +  $L$  exp width + 5 characters on the current line, then a good position is found on a subsequent line: next, when not at the beginning of a line, a space is given and then  $V$  is output as if under control of the **picture** `+d.n (L real width - 1) den (L exp width - 1) z+d`.

cc) If the mode of  $V$  is specified by  $L$  COMPL, then first, if there is not enough room for  $2 \times (L$  real width +  $L$  exp width) + 11 characters on the current line, then a good position is found on a subsequent line: next, when not at the beginning of a line, a space is given and then  $V$  is output as if under control of the **picture** `+d.n (L real width - 1) den (L exp width - 1) z+d" "i`  
`+d.n (L real width - 1) den (L exp width - 1) z+d .`

dd) If the mode of  $V$  is specified by BOOL, then first, if the current line is full, a good position is found on a subsequent line: next, if  $V$  is true (false), the character yielded by flip (flop) is output (with no intervening space) .

## ALGOL 68 REVISED REPORT

ee) If the mode of  $V$  is specified by  $L$  BITS, then the elements of the only field of  $V$  are output (as in dd) one after the other (with no intervening spaces, and with new lines being taken as required),

ff) If the mode of  $V$  is specified by CHAR, then first, if the current line is full, a good position is found on a subsequent line: next  $V$  is output (with no intervening space) .

gg) If the mode of  $V$  is specified by  $[]$  CHAR, then the elements of  $V$  are output (as in ff) one after the other (with no intervening spaces, and with new lines being taken as required) .}

```

a) PROC put = (REF FILE f, [] UNION (OUTTYPE,
  PROC (REF FILE) VOID) x) VOID:
  IF opened OF f THEN
    FOR i TO UPB x
      DO CASE set write mood (f); set char mood (f); x[i] IN
        (PROC (REF FILE) VOID pf): pf (f),
        (OUTTYPE ot):
          BEGIN [] SIMPLOUT y = STRAIGHTOUT ot;
            << PROC L real conv = (L REAL r) STRING:
              float (r, L real width + L exp width + 4,
                L real width - 1, L exp width + 1) >> ;
            FOR j TO UPB y
              DO CASE y[j] IN
                (UNION (NUMBER, << L COMPL >> ) nc):
                  BEGIN STRING s :=
                    CASE nc IN
                      << (L INT k): whole (k, L int width + 1) >>,
                      << (L REAL r): L real conv (r) >>,
                      << (L COMPL z): L real conv (RE z) + " I"
                        + L real conv (IM z) >>
                    ESAC;
                  REF REF POS cpos = cpos OF f, INT n = UPB s;
                  WHILE nextpos (f);
                    (n > c OF book bounds (f) | undefined);
                    c OF cpos + (c OF cpos = 1 | n | n + 1) >
                      c OF book bounds (f) + 1
                  DO (¬ (line mended OF f) (f) | put (f, newline));
                    set write mood (f)
                  OD;
                    (c OF cpos ≠ 1 | " " PLUSTO s);
                    FOR k TO UPB s DO put char (f, s[k]) OD
                  END ⚡ numeric ⚡,
                (BOOL b): (next pos (f); put char (f, (b | flip | flop))),
                << (L BITS lb):
                  FOR k TO L bits width
                    DO put (f, (L F OF lb) [k]) OD >>,
                (CHAR k): (nextpos (f); put char (f, k)),
                ([] CHAR ss):
                  FOR k FROM LWB ss TO UPB ss

```

```

        DO next pos (f); put char (f, ss[k]) OD
      ESAC
    OD
  END
  ESAC
OD
ELSE undefined
FI;

b) PROC N0 put char = (REF FILE f, CHAR char) VOID:
  IF opened OF f ∧ ¬ line ended (f)
  THEN REF POS cpos = cpos OF f, lpos = lpos OF book OF f;
    set char mood (f); set write mood (f);
    REF INT p = p OF cpos, l = l OF cpos, c = c OF cpos;
    CHAR k; BOOL found := FALSE;
    CASE text OF f IN
      (TEXT): (k := char; found := TRUE),
      (FLEXTEXT):
        FOR i TO UPB F OF conv OF f WHILE ¬ found
          DO STRUCT (CHAR internal, external) key = (F OF conv OF f) [i];
            (internal OF key = char | k := external OF key; found := TRUE)
          OD
        ESAC;
      IF found THEN
        CASE text OF f IN
          (TEXT t1): t1 [p][l][c] := k,
          (FLEXTEXT t2): t2[p][l][c] := k
        ESAC;
        c += 1;
        IF cpos BEYOND lpos THEN lpos := cpos
        ELIF ¬ set possible (f) ∧
          POS (p OF lpos, l OF lpos, 1) BEYOND cpos
        THEN lpos := cpos;
          (compressible (f) |
            C the size of the line and page containing the logical
            end of the book and of all subsequent lines and
            pages may be increased {e.g., to the sizes with
            which the book was originally established
            {2510.3.1.4.cc} or to the sizes implied by maxpos OF
            chan OF f} C)
        FI
      ELSE k := " ";
        IF ¬ (char error mended OF f) (f, k)
        THEN undefined; k := " "
        FI;
        check pos (f); put char (f, k)
      FI
    ELSE undefined
  FI
  ⚡ write mood is still set ⚡ ;

c) PROC N0 next pos = (REF FILE f) VOID:

```

## ALGOL 68 REVISED REPORT

```
(¬ get good line (f, read mood OF f) | undefined)
ç the line is now good {2510.3.1.6.dd} and the read/write mood is
  as on entry ç ;
```

### 25.3.3.2 Formatless input

{For formatless input, `get` (a) and `read` (25<sub>10</sub>.5.1.e) may be used. Values from the book are assigned to each straightened name  $N$  from the data list as follows:

aa) If the mode of  $N$  is specified by `REF L INT`, then first, the book is searched for the first character that is not a space (finding good positions on subsequent lines as necessary); next, the largest string is read from the book that could be "indited" (25<sub>10</sub>.3.4.1.1.kk) under the control of some **picture** of the form `+n(k1) "␣" n(k2)d d` or `n(k2) d d` (where  $k_1$  and  $k_2$  yield arbitrary nonnegative integers): this string is converted to an integer and assigned to  $N$ ; if the conversion is unsuccessful, the event routine corresponding to `on value error` is called.

bb) If the mode of  $N$  is specified by `REF L REAL`, then first, the book is searched for the first character that is not a space (finding good positions on subsequent lines as necessary); next, the largest string is read from the book that could be indited under the control of some **picture** of the form

```
+n(k1) "␣" n(k2)d or n(k2)d followed by .n(k3)d d or by ds., possibly followed
again by
e n(k4) "␣"+n(k5) "␣" n(k6)d d or by
e n(k5) "␣" n(k6)d d;
```

this string is converted to a real number and assigned to  $N$ ; if the conversion is unsuccessful, the event routine corresponding to `on value error` is called.

cc) If the mode of  $N$  is specified by `REF L COMPL`, then first, a real number is input (as in bb) and assigned to the first subname of  $N$ ; next, the book is searched for the first character that is not a space; next, a character is input and, if it is not "I" or "i", then the event routine corresponding to `on char error` (25<sub>10</sub>.3.1.3.cc) is called, the suggestion being "I"; finally, a real number is input and assigned to the second subname of  $N$ .

dd) If the mode of  $N$  is specified by `REF BOOL`, then first, the book is searched for the first character that is not a space (finding good positions on subsequent lines as necessary); next, a character is read: if this character is the same as that yielded by `flip(flop)`, then `true` (`false`) is assigned to  $N$ ; otherwise, the event routine corresponding to `on char error` is called, the suggestion being `flop`.

ee) If the mode of  $N$  is specified by `REF L BITS`, then input takes place (as in dd) to the subnames of  $N$  one after the other (with new lines being taken as required).



ff) If the mode of  $N$  is specified by REF CHAR, then first, if the current line is exhausted, a good position is found on a subsequent line; next, a character is read and assigned to  $N$ .

gg) If the mode of  $N$  is specified by REF [] CHAR, then input takes place (as in ff) to the subnames of  $N$  one after the other (with new lines being taken as required) .

hh) If the mode of  $N$  is specified by REF STRING, then characters are read until either

- (i) a character is encountered which is contained in the string associated with the file by a call of the routine make term, or
- (ii) the current line is exhausted, whereupon the event routine corresponding to on line end (or, where appropriate, to on page end, on physical file end or on logical file end) is called; if the event routine moves the current position to a good position (see 25<sub>10</sub>.3.3), then input of characters is resumed.

The string consisting of the characters read is assigned to  $N$  (note that, if the current line has already been exhausted, or if the current position is at the start of an empty line or outside the logical file, then an empty string is assigned to  $N$ ) .}

```
a) PROC get = (REF FILE f, [] UNION (INTYPE,
  PROC (REF FILE) VOID) x) VOID:
  IF opened OF f THEN
  FOR i TO UPB x
  DO CASE set read mood (f); set char mood (f); x[i] IN
  (PROC (REF FILE) VOID pf): pf (f),
  (INTYPE it) :
  BEGIN
  [] SIMPLIN y = STRAIGHTIN it; CHAR k; BOOL k empty;
  OP ? = (STRING s) BOOL: C true if the next character, if any,
  in the current line is contained in 's'
  (the character is assigned to 'k')
  and false otherwise C
  IF k empty ^ (line ended (f) V logical file ended (f))
  THEN FALSE
  ELSE (k empty | get char (f, k));
  k empty := char in string (k, LOC INT, s)
  FI;
  OP ? = (CHAR c) BOOL: ? STRING (c);
  PRIO ! = 8;
  OP ! = (STRING s, CHAR c) CHAR:
  C expects a character contained in 's'; if the character
  read is not in 's', the event routine corresponding to
  'on char error' is called with the suggestion 'c' C
  IF (k empty | check pos (f); get char (f, k));
  k empty := TRUE;
  char in string (k, LOC INT, s)
  THEN k
```

## ALGOL 68 REVISED REPORT

```

ELSE CHAR sugg := c;
  IF (char error mended OF f) (f, sugg) THEN
    (char in string (sugg, LOC INT, s) | sugg | undefined; c)
  ELSE undefined; c
  FI;
  set read mood (f)10
FI;
OP ! = (CHAR s, c) CHAR: STRING (s) ! c;
PROC skip initial spaces = VOID:
  WHILE (k empty | next pos (f)); ? " " DO SKIP OD;
PROC skip spaces = VOID:
  WHILE ? " " DO SKIP OD;
PROC read dig = STRING:
  (STRING t := "0123456789" ! "0";
  WHILE ? "0123456789" DO t PLUSAB k OD; t);
PROC read sign = CHAR:
  (CHAR t = (skipspace; ? "+-" | k | "+");
  skip spaces; t);
PROC read num = STRING:
  (CHAR t = read sign; t + read dig);
PROC read real = STRING:
  (STRING t := read sign;
  (¬ ? "." | t PLUSAB read dig | k empty := FALSE);
  (N0 "." | t PLUSAB "." + read dig);
  (N0 "10\e" | t PLUSAB "10" + read num); t);
FOR j TO UPB y
DO BOOL incomp := FALSE; k empty := TRUE;
CASE y[j] IN
  << (REF L INT ii) :
    (skip initial spaces;
    incomp := ¬ string to L int (read num, 10, ii)) >>,
  << (REF L REAL rr) :
    (skip initial spaces;
    incomp := ¬ string to L real (read real, rr)) >>,
  << (REF L COMPL zz) :
    (skip initial spaces;
    incomp := ¬ string to L real (read real, re OF zz);
    skip spaces; "iI" ! "I;";
    incomp := incomp V
    string to L real (read real, im OF zz)) >>
  (REF BOOL bb) :
    (skip initial spaces;

```

---

<sup>10</sup>This line and the four preceding ones should read

```

CHAR cc = IF (char error mended OF f) (f, sugg) THEN
(char in string (sugg, LOC INT, s) | sugg | undefined; c)
ELSE undefined; c
FI;
set read mood (f); cc

```

```

        bb := (flip + flop) ! flop = flip),
    << (REF L BITS lb) :
        FOR i TO L bits width
            DO get (f, (L F OF lb) [i]) OD >>,
    (REF CHAR cc): (next pos (f); get char (f, cc)),
    (REF []CHAR ss) :
        FOR i FROM LWB ss TO UPB ss
            DO next pos (f); get char (f, ss [i]) OD,
    (REF STRING ss):
        BEGIN STRING t;
            WHILE check pos (f);
                IF line ended (f) ∨ logical file ended (f)
                    THEN FALSE
                ELSE get char (f, k);
                    k empty := ¬ char in string (k, LOC INT, term OF f)
                FI
            DO t PLUSAB k OD;
            ss := t
        END
    ESAC;
    (¬ k empty | backspace (f));
    IF incomp
    THEN (¬ (value error mended OF f) (f) | undefined);
        set read mood (f)
    FI
    OD
    END
    ESAC
    OD
    ELSE undefined
    FI;

```

- b) PROC  $\aleph_0$  get char = (REF FILE f, REF CHAR char) VOID:
- ```

    IF opened OF f ∧ ¬ line ended (f) ∧ ¬ logical file ended (f)
    THEN REF POS cpos = cpos OF f;
        set char mood (f); set read mood (f);
        INT p = p OF cpos, l = l OF cpos, c = c OF cpos;
        c OF cpos += 1;
        char := CASE text OF f IN
            (TEXT t1): t1[p][l][c],
            (FLEXTEXT t2) :
                (CHAR k := t2[p][l][c];
                BOOL found := FALSE;
                FOR i TO UPB F OF conv OF f WHILE ¬ found
                DO STRUCT (CHAR internal, external) key =
                    (F OF conv OF f) [i];
                    (external OF key = k | k := internal OF key; found := TRUE)
                OD;
                IF found THEN k
                ELSE k := " ";

```

## ALGOL 68 REVISED REPORT

```
        IF (char error mended OF f) (f, k)
        THEN k
        ELSE undefined; " "
        FI; set read mood (f);
    FI)
    ESAC
ELSE undefined
FI ç read mood is still set ç ;
```

```
c) PROC N0 check pos = (REF FILE f) VOID:
    BEGIN BOOL reading = read mood OF f;
        BOOL not ended := TRUE;
        WHILE not ended := not ended ^ get good page (f, reading);
            line ended (f) ^ not ended
        DO not ended := (line mended OF f) (f) OD
    END;
```

{The routine `check pos` is used in formatted transput before each call of `put char` or `get char`. If the position is not good (25<sub>10</sub>.3.1.6.dd), it calls the appropriate event routine, and may call further event routines if `true` is returned. If the event routine corresponding to `on page end` returns `false`, `newpage` is called but, if any other event routine returns `false`, no default action is taken and no more event routines are called. On exit, the read/write mood is as on entry, but the current position may not be good, in which case `undefined` will be called in the following `put char` or `get char`. However, `check pos` is also called when getting strings (hh), in which case the string is then terminated if the current position is not good.}

### 25.3.4 Format texts

{In formatted transput, each straightened value from a data list (cf. 25<sub>10</sub>.3.3) is matched against a constituent **picture** of a **format-text** provided by the user. A **picture** specifies how a value is to be converted to or from a sequence of characters and prescribes the layout of those characters in the book. Features which may be specified include the number of digits, the position of the decimal point and of the sign, if any, suppression of zeroes and the insertion of arbitrary strings. For example, using the **picture** `-d.3d " " 3d " " e z+d`, the value 1234.567 would be transput as the string `"_1.234 56710+3"`.

A "format" is a structured value (i.e., an internal object) of mode **'FORMAT'**, which mirrors the hierarchical structure of a **format-text** (which is an external object). In this section are given the syntax of **format-texts** and the semantics for obtaining their corresponding formats. The actual formatted transput is performed by the routines given in section 25<sub>10</sub>.3.5 but, for convenience, a description of their operation is given here, in association with the corresponding syntax.}

### 25.3.4.1 Collections and pictures

#### 25.3.4.1.1. Syntax

{The following **mode-declarations** (taken from rr10.3.5.a) are reflected in the metaproduction rules A to K below.

- A) `MODE FORMAT = STRUCT (FLEX [1 : 0] PIECE F);`
- B) `MODE PIECE = STRUCT (INT cp, count, bp, FLEX [1: 0] COLLECTION c);`
- C) `MODE COLLECTION = UNION (PICTURE, COLLITEM);`
- D) `MODE COLLITEM = STRUCT (INSERTION i1, PROC INT rep, INT p,  
INSERTION i2);`
- E) `MODE INSERTION = FLEX [1 : 0] STRUCT (PROC INT rep,  
UNION (STRING, CHAR) sa);`
- F) `MODE PICTURE = STRUCT (UNION (PATTERN, CPATTERN, FPATTERN, GPATTERN,  
VOID) p, INSERTION i);`
- G) `MODE PATTERN = STRUCT (INT type, FLEX [1 : 0] FRAME frames);`
- H) `MODE FRAME = STRUCT (INSERTION i, PROC INT rep, BOOL supp, CHAR marker);`
- I) `MODE CPATTERN = STRUCT (INSERTION i, INT type,  
FLEX [1 : 0] INSERTION c);`
- J) `MODE FPATTERN = STRUCT (INSERTION i, PROC FORMAT pf);`
- K) `MODE GPATTERN = STRUCT (INSERTION i, FLEX [1 : 0] PROC INT spec); }`

A) **FORMAT :: structured with row of PIECE field letter aleph mode.**

B) **PIECE ::  
structured with integral field letter c letter p  
integral field letter c letter o letter u letter n letter t  
integral field letter b letter p  
row of COLLECTION field letter c mode.**

C) **COLLECTION :: union of PICTURE COLLITEM mode.**

D) **COLLITEM ::  
structured with  
INSERTION field letter i digit one  
procedure yielding integral field letter r letter e letter p  
integral field letter p  
INSERTION field letter i digit two mode.**

E) **INSERTION ::  
row of structured with  
procedure yielding integral field letter r letter e letter p  
union of row of character character mode field letter s letter a mode.**

- F) **PICTURE** ::  
structured with  
union of **PATTERN** **CPATTERN** **FPATTERN** **GPATTERN** void mode field  
letter p  
**INSERTION** field letter i mode.
- G) **PATTERN** ::  
structured with  
integral field letter t letter y letter p letter e  
row of **FRAME** field letter f letter r letter a letter m letter e letter s mode.
- H) **FRAME** ::  
structured with  
**INSERTION** field letter i  
procedure yielding integral field letter r letter e letter p  
boolean field letter s letter u letter p letter p  
character field letter m letter a letter r letter k letter e letter r mode.
- I) **CPATTERN** ::  
structured with  
**INSERTION** field letter i  
integral field letter t letter y letter p letter e  
row of **INSERTION** field letter c mode.
- J) **FPATTERN** ::  
structured with  
**INSERTION** field letter i  
procedure yielding **FIVMAT** field letter p letter f mode.
- K) **GPATTERN** ::  
structured with  
**INSERTION** field letter i  
row of procedure yielding integral field letter s letter p letter e letter c  
mode.
- L) **FIVMAT** ::  
mui definition of structured with  
row of structured with integral field letter c letter p  
integral field letter c letter o letter u letter n letter t  
integral field letter b letter p  
row of union of  
structured with  
union of **PATTERN** **CPATTERN**  
structured with **INSERTION** field letter i  
procedure yielding mui application field  
letter p letter f

**mode**  
**GPATTERN void**  
**mode field letter p**  
**INSERTION field letter i**  
**mode**  
**COLLITEM**  
**mode field letter c**  
**mode field letter aleph**  
**mode.**  
 {'FIVMAT' is equivalent (16<sub>2</sub>.1.1.2.a) to 'FORMAT'.}

- M) **MARK :: sign ; point ; exponent ; complex ; boolean.**
- N) **COMARK :: zero ; digit ; character.**
- O) **UNSUPPRESSETY :: unsuppressible ; EMPTY.**
- P) **TYPE :: integral ; real ; boolean ; complex ; string ; bits ; integral choice ; boolean choice ; format ; general.**
- a) **FORMAT NEST format text {19<sub>5</sub>.D} :**  
**formatter {24<sub>9</sub>.4.f} token, NEST collection {b} list, formatter {24<sub>9</sub>.4.f} token.**
- b) **NEST collection {a,b} :**  
**pragment {24<sub>9</sub>.2.a} sequence option, NEST picture {c};**  
**pragment {24<sub>9</sub>.2.a} sequence option, NEST insertion {d},**  
**NEST replicator {g}, NEST collection {b} list brief pack,**  
**pragment {24<sub>9</sub>.2.a} sequence option, NEST insertion {d}.**
- c) **NEST picture {b} :**  
**NEST TYPE pattern {25<sub>10</sub>.3.4.2.a, 25<sub>10</sub>.3.4.3.a, 25<sub>10</sub>.3.4.4.a, 25<sub>10</sub>.3.4.5.a,**  
**25<sub>10</sub>.3.4.6.a, 25<sub>10</sub>.3.4.7.a, 25<sub>10</sub>.3.4.8.a,b, 25<sub>10</sub>.3.4.9.a, 25<sub>10</sub>.3.4.10.a}**  
**option, NEST insertion {d}.**
- d) **NEST insertion {b,c,j,k, 25<sub>10</sub>.3.4.7.b, 25<sub>10</sub>.3.4.3.a,b, 25<sub>10</sub>.3.4.9.a, 25<sub>10</sub>.3.4.10.a} :**  
**NEST literal {i} option, NEST alignment {e} sequence option.**
- e) **NEST alignment {d} :**  
**NEST replicator {g}, alignment code {f}, NEST literal {i} option.**
- f) **alignment code {e} :**  
**letter k {24<sub>9</sub>.4.a} symbol ; letter x {24<sub>9</sub>.4.a} symbol ; letter y {24<sub>9</sub>.4.a} symbol ;**  
**letter l {24<sub>9</sub>.4.a} symbol ; letter p {24<sub>9</sub>.4.a} symbol ; letter q {24<sub>9</sub>.4.a} symbol.**
- g) **NEST replicator {b,e,i,k} : NEST unsuppressible replicator {h} option.**
- h) **NEST unsuppressible replicator {g,i} :**  
**fixed point numeral {23<sub>8</sub>.1.1.b} ; letter n {24<sub>9</sub>.4.a} symbol,**

**meek integral NEST ENCLOSED clause** {17<sub>3</sub>.1.a, 17<sub>3</sub>.4.a, -},  
**pragma** {24<sub>9</sub>.2.a} **sequence option.**

- i) **NEST UNSUPPRESSETY literal** {d,e,i, 25<sub>10</sub>.3.4.8.c} :  
**NEST UNSUPPRESSETY replicator** {g,h},  
**strong row of character NEST denoter** {23<sub>8</sub>.0.a} **coercee** {21<sub>6</sub>.1.a} ,  
**NEST unsuppressible literal** {i} **option.**
- j) **NEST UNSUPPRESSETY MARK frame**  
{25<sub>10</sub>.3.4.2.c, 25<sub>10</sub>.3.4.3.b,c, 25<sub>10</sub>.3.4.4.a, 25<sub>10</sub>.3.4.5.a} :  
**NEST insertion** {d}, **UNSUPPRESSETY suppression** {l} ,  
**MARK marker** {25<sub>10</sub>.3.4.2.e, 25<sub>10</sub>.3.4.3.d,e, 25<sub>10</sub>.3.4.4.b, 25<sub>10</sub>.3.4.5.b} .
- k) **NEST UNSUPPRESSETY COMARK frame** {25<sub>10</sub>.3.4.2.b,c, 25<sub>10</sub>.3.4.6.a} :  
**NEST insertion** {d} , **NEST replicator** {g},  
**UNSUPPRESSETY suppression** {l} ,  
**COMARK marker** {25<sub>10</sub>.3.4.2.d,f, 25<sub>10</sub>.3.4.6.b} .
- l) **UNSUPPRESSETY suppression** {j,k, 25<sub>10</sub>.3.4.7.b} :  
**where (UNSUPPRESSETY) is (unsuppressible), EMPTY ;**  
**where (UNSUPPRESSETY) is (EMPTY), letter s** {24<sub>9</sub>.4.a} **symbol option.**
- m) **\*frame :**  
**NEST UNSUPPRESSETY MARK frame** {j} ;  
**NEST UNSUPPRESSETY COMARK frame** {k};  
**NEST RADIX frame** {25<sub>10</sub>.3.4.7.b} .
- n) **\*marker :**  
**MARK marker** {25<sub>10</sub>.3.4.2.e, 25<sub>10</sub>.3.4.3.d,e, 25<sub>10</sub>.3.4.4.b, 25<sub>10</sub>.3.4.5.b} ;  
**COMARK marker** {25<sub>10</sub>.3.4.2.d,f, 25<sub>10</sub>.3.4.6.b} ; **radix marker** {25<sub>10</sub>.3.4.7.c} .
- o) **\*pattern : NEST TYPE pattern**  
{25<sub>10</sub>.3.4.2.a, 25<sub>10</sub>.3.4.3.a, 25<sub>10</sub>.3.4.4.a, 25<sub>10</sub>.3.4.5.a, 25<sub>10</sub>.3.4.6.a,  
25<sub>10</sub>.3.4.7.a, 25<sub>10</sub>.3.4.8.a,h, 25<sub>10</sub>.3.4.9.a, 25<sub>10</sub>.3.4.10.a} .

{Examples:

- a) \$p"table of"x 10a, l n(lim-1) ("x="12z+d 2x,  
+.12de+2d3q"+j\*"3" "si+.10de+2d l)p\$
- b) p"table of"x10a • l n(lim-1) ("x="12z+d 2x,  
+.12de+2d3q"+j\*"3" "si+.10de+2d l)p
- c) 120k c("mon", "tues", "wednes", "thurs",  
"fri", "satur", "sun") "day"
- d) p"table of"x



- e) p"table of"
- h) 10 • n(lim-1)
- i) "+j\*"3" "
- j) si
- k) "x="12z
- l) s }

{The positions where pragments {24<sub>9</sub>.2.1.a} may occur in **format-texts** are restricted. In general (as elsewhere in the language), a pragment may not occur between two **DIGIT-** or **LETTER-symbols.**}

{aa) For formatted output, `putf` {25<sub>10</sub>.3.5.1.a} and `printf` (or `writeln`) {25<sub>10</sub>.5.1.f} may be used and, for formatted input, `getf` {25<sub>10</sub>.3.5.2.a} and `readf` {25<sub>10</sub>.5.1.g}. Each element in the data list (cf. 25<sub>10</sub>.3.3) is either a format to be associated with the file or a value to be transput (thus a format may be included in the data list immediately before the values to be transput using that format).

bb) During a call of `putf` or `getf` transput proceeds as follows:  
For each element of the data list, considered in turn,

If it is a format,

then it is made to be the current format of the file by `associate format` (25<sub>10</sub>.3.5.k):

otherwise, the element is straightened (25<sub>10</sub>.3.2.3.c) and each element of the resulting multiple value is output {hh} or input {ii} using the next "picture" {cc, gg} from the current format.

cc) A "picture" is the yield of a **picture**. It is composed of a "pattern" of some specific **'TYPE'** (according to the syntax of the **TYPE-pattern** of that **picture**), followed by an "insertion" {ee}. Patterns, apart from **'choice'**, **'format'** and **'general'** patterns, are composed of "frames", possibly "suppressed", each of which has an insertion, a "replicator" {dd}, and a "marker" to indicate whether it is a "d", "z", "i" etc. frame. The frames of each pattern may be grouped into "sign moulds", "integral moulds", etc., according to the syntax of the corresponding pattern.

dd) A "replicator" is a routine, returning an integer, constructed from a **replicator** {25<sub>10</sub>.3.4.1.2.c}, For example, the **replicator** 10 gives rise to a routine composed from `INT: 10;` moreover, `n (lim - 1)` is a "dynamic" **replicator** and gives rise to `INT: (lim - 1)`. Note that the scope of a replicator restricts the scope of any format containing it, and thus it may be necessary to take a local copy of a file before associating a format with it (see, e.g.. 11.D)

. A replicator which returns a negative value is treated as if it had returned zero ("k" alignments apart) .

When a **picture** is "staticized", all of its replicators and other routines (including those contained in its insertions) are called collaterally. A staticized pattern may be said to "control" a string, and there is then a correspondence between the frames of that pattern, taken in order, and the characters of the string. Each frame controls  $n$  consecutive characters of the string, where  $n$  is 0 for an "r" frame and, otherwise, is the integer returned by the replicator of the frame (which is always 1 for a "+", "-", ".", "e", "i" or "b" frame) . Each controlled character must be one of a limited set appropriate to that frame.

ee) An "insertion", which is the yield of an **insertion** {25<sub>10</sub>.3.4.1.2.d}, is a sequence of replicated "alignments" and strings: an insertion containing no alignments is termed a "literal". An insertion is "performed" by performing its alignments {ff} and on output (input) writing ("expecting" {ll}) each character of its replicated strings (a string is replicated by repeating it the number of times returned by its replicator) .

ff) An "alignment" is the character yielded by an **alignment-code** {25<sub>10</sub>.3.4.1.2.d}. An alignment which has been replicated  $n$  times is performed as follows:

- "k" causes `set char number` to be called, with  $n$  as its second parameter:
- "x" causes `space` to be called  $n$  times:
- "y" causes `backspace` to be called  $n$  times:
- "l" causes `newline` to be called  $n$  times:
- "p" causes `newpage` to be called  $n$  times;
- "q" on output (input) causes the character `blank` to be written (expected)  $n$  times.

gg) A format may consist of a sequence of **pictures**, each of which is selected in turn by `get next picture` {25<sub>10</sub>.3.5.b}. In addition, a set of pictures may be grouped together to form a replicated "collection" (which may contain further such collections) . When the last picture in a collection has been selected, its first picture is selected again, and so on until the whole collection has been repeated  $n$  times, where  $n$  is the integer returned by its replicator. A collection may be provided with two insertions, the first to be performed before the collection, the second afterwards.

A format may also invoke other formats by means of **'format'** patterns {25<sub>10</sub>.3.4.9.1}.

When a format has been exhausted, the event routine corresponding to `on format end` is called; if this returns `false`, the format is repeated; otherwise, if the event routine has failed to provide a new format, `undefined` is called.

hh) A value  $V$  is output, using a picture  $P$ , as follows:

If the pattern  $Q$  of  $P$  is a **'choice'** or **'general'** pattern,

then  $V$  is output using  $P$  (See 25<sub>10</sub>.3.4.3.1.aa,dd, 25<sub>10</sub>.3.4.10.1.aa);

otherwise,  $V$  is output as follows:

- $P$  is staticized;

If the mode of  $V$  is "output compatible" with  $Q$  (see the separate section dealing with each type of pattern),

then

- $V$  is converted into a string controlled {dd} by  $Q$  (See the appropriate section);

If the mode is not output compatible, or if the conversion is unsuccessful,

then

- the event routine corresponding to `on value error` is called;
- if this returns `false`,  $V$  is output using `put` and `undefined` is called;

otherwise, the string is "edited" (jj) using  $Q$ :

- the insertion of  $P$  is performed.

ii) A value is input to a name  $N$ , using a picture  $P$ , as follows:

If the pattern  $Q$  of  $P$  is a **'choice'** or **'general'** pattern,

then a value is input to  $N$  using  $P$  (see 25<sub>10</sub>.3.4.8.1.bb,ee, 25<sub>10</sub>.3.4.10.1.bb);

otherwise,

- $P$  is staticized;
- a string controlled by  $Q$  is "indited" {kk};

If the mode of  $N$  is "input compatible" with  $Q$  (see the appropriate section),

then

- the string is converted to an appropriate value suitable for  $N$  using  $Q$  (see the appropriate section);
- if the conversion is successful, the value is assigned to  $N$ ;

If the mode is not input-compatible, or if the conversion is unsuccessful,

then

- the event routine corresponding to `on value error` is called;
- if this returns `false`, `undefined` is called;

- the insertion of  $P$  is performed.

## ALGOL 68 REVISED REPORT

jj) A string is "edited", using a pattern  $P$ , as follows:

In each part of the string controlled by a sign mould,

- if the first character of the string (which indicates the sign) is "+" and the sign mould contains a "-" frame, then that character is replaced by "\_"
- the first character (i.e., the sign) is shifted to the right across all leading zeroes in this part of the string and these zeroes are replaced by spaces (for example, using the sign mould  $4z+$ , the string "+0003" becomes "+\_ \_ \_3")

In each part of the string controlled by an integral mould,

- zeroes controlled by "z" frames are replaced by spaces as follows:
  - between the start of the string and the first nonzero digit;
  - between each "d", "e" or "i" frame and the next nonzero digit; (for example, using the pattern  $zdzd2d$  the string "180168" becomes "18\_168");

For each frame  $F$  of  $P$ ,

- the insertion of  $F$  is performed.
- if  $F$  is not suppressed, the characters controlled by  $F$  are written; (for example, the string "+0003.5", when edited using the pattern  $4z+ s. \_ \_ \_ d$ , causes the string "+\_ \_ \_+3, 5" to be written and the string "180168", using the pattern  $zd"-z d"-19"2d$ , gives rise to "18-\_1-1968").

kk) A string is "indited", using a pattern  $P$ , as follows:

For each frame  $F$  of  $P$ ,

- the insertion of  $F$  is performed.

For each element of the string controlled by  $F$ , a character is obtained as follows:

If  $F$  is contained in a sign mould,

then

- if a sign has been found, a digit is expected, with "0" as suggestion;
- otherwise, either a "+" or a "-" is expected, with "0" as suggestion, and, in addition, if the sign mould contains a frame, then a space preceding the first digit will be accepted as the sign (and replaced by "+");

otherwise, if  $F$  is contained in an integral mould,

then

If  $F$  is suppressed,  
 then "0" is supplied,  
 otherwise:

Case A:  $F$  is a "d" frame:

- a digit is expected, with "0" as suggestion;

Case B: if  $F$  is "z" frame:

- a digit or space is expected, with "0" as suggestion, but a space is only acceptable as follows:
  - between the start of the string and the first nonzero digit;
  - between each "d", "e" or "i" frame and the next nonzero digit;
- such spaces are replaced by zeroes.

otherwise, if  $F$  is an "a" frame,

then

if  $F$  is not suppressed, a character is read and supplied;  
 otherwise " " is supplied;

otherwise, if  $F$  is not suppressed,

then

if  $F$  is a "." ("e", "i", "b") frame, a "." ("10" or "  
 " or "e", "I" or "i", flip or flop) is expected, with "." ("10", "I", flop) as  
 suggestion;

otherwise,

if  $F$  is a suppressed "." ("e", "i") frame, the character "." ("10", "I") is  
 supplied.

II) A member of a set of characters  $S$  is "expected", with the character  $C$  as suggestion, as follows:

- a character is read;

If that character is one of the expected characters (i.e., a member of  $S$ ),

then that character is supplied;

otherwise, the event routine corresponding to `on char error` is called, with  $C$  as suggestion; if this returns `true` and  $C$ , as possibly replaced, is one of the expected characters, then that character is supplied; otherwise, `undefined` is called. }

**25.3.4.1.2. Semantics**

{A format is brought into being by means of a **format-text**. A format is best regarded as a tree, with a collection at each node and a picture at each tip. In order to avoid violation of the scope restrictions, each node of this tree is, in this Report, packed into a value of mode **'PIECE'**. A format is composed of a row of such pieces and the pieces contain pointers to each other in the form of indices selecting from that row. An implementer will doubtless store the tree in a more efficient manner. This is possible because the **field-selector** of a format is hidden from the user in order that he may not break it open.

Although a **format-text** may contain **ENCLOSED-clauses** (in **replicators** and **format-patterns**) or **units** (in **general-patterns**,) these are not elaborated at this stage but are, rather, turned into routines for subsequent calling as and when they are encountered during formatted transput. Indeed, the elaboration of a **format-text** does not result in any actions of any significance to the user.}

a) The yield of a **format-text**  $F$ , in an environ  $E$ , is a structured value whose only field is a multiple value  $W$ , whose mode is **'row of PIECE'**, composed of a descriptor  $((1, n))$  and  $n$  elements determined as follows:

- a counter  $i$  is set to 1;
- $F$  is "transformed" {b} in  $E$  into  $W$ , using  $i$ .

b) A **format-text** or a **collection-list-pack**  $C$  is "transformed" in an environ  $E$  into a multiple value  $W$  whose mode is **'row of PIECE'**, using a counter  $i$ , as follows:

•the element of  $W$  selected by  $(i)$  is a structured value, whose mode is **'PIECE'** and whose fields, taken in order, are

- $\{cp\}$  undefined;
- $\{count\}$  undefined;
- $\{bp\}$  undefined;
- $\{c\}$  a multiple value  $V$  whose mode is **"row of COLLECTION"**, having a descriptor  $((1,m))$ , where  $m$  is the number of constituent collections of  $C$ , and elements determined as follows:  
For  $j = 1, \dots, m$ , letting  $C_j$  be the  $j$ -th constituent collection of  $C$ ,

Case A: The direct descendents of  $C_j$  include a **picture**  $P$ .

- the constituent **pattern**  $P$  if any and the **insertion**  $I$  of  $P$  are elaborated collaterally;
- the  $j$ -th element of  $V$  is a structured value, whose mode is **'PICTURE'** and whose fields, taken in order are

- $\{p\}$ ; the yield of  $T$ , if any, {e, 25<sub>10</sub>.3.4.8.2, 25<sub>10</sub>.3.4.9.2, 25<sub>10</sub>.3.4.10.2} and, otherwise, empty.
- $\{i\}$  the yield of  $I \{d\}$ ;

Case B: The direct descendents of  $C_j$ , include a first **insertion**  $I1$ , a **replicator**  $REP$ , a **collection-list-pack**  $P$  and a second **insertion**  $I2$ :

- $i$  is increased by 1;
- $I1$ ,  $REP$  and  $I2$  are elaborated collaterally;
- the  $j$ -th element of  $V$  is a structured value whose mode is '**COLLITEM**' and whose fields, taken in order, are
  - $\{i1\}$  the yield of  $I1 \{d\}$ ;
  - $\{rep\}$  the yield of  $REP \{c\}$ ;
  - $\{p\}$   $i$ ;
  - $\{i2\}$  the yield of  $I2$ ;
- $P$  is transformed in  $E$  into  $W$ , using  $i$ .

c) The yield, in an environ  $E$ , of a **NEST-UNSUPPRESSETY-replicator**  $R$  {25<sub>10</sub>.3.4.1.1.g, h} is a routine whose mode is '**procedure yielding integral**', composed of a **procedure-yielding-integral-NEST-routine-text** whose **unit** is  $U$ , together with the environ necessary {22<sub>7</sub>.2.2.c} for  $U$  in  $E$ , where  $U$  is determined as follows:

Case A:  $R$  contains a **meek-integral-ENCLOSED-clause**  $C$ :

- $U$  is a new **unit** akin {15<sub>1</sub>.1.3.2.k} to  $C$ ;

Case B:  $R$  contains a **fixed-point-numeral**  $D$ , but no **ENCLOSED-clause**:

- $U$  is a new **unit** akin to  $D$ ;

Case C:  $R$  is invisible:

- $U$  is a new **unit** akin to a **fixed-point-numeral** which has an intrinsic value {23<sub>8</sub>.1.1.2} of 1.

d) The yield of an **insertion**  $I$  {25<sub>10</sub>.3.4.1.1.d} is a multiple value  $W$  whose mode is '**INSERTION**', determined as follows:

- let  $u_1, \dots, U_n$  be the constituent **UNSUPPRESSETY-replicators** of  $I$ , and let  $A_i, i = 1, \dots, n$ , be the **denoter-coercee** or **alignment-code** {immediately} following  $U_i$ ;
- let  $R_1, \dots, R_n$  and  $D_1, \dots, D_n$  be the {collateral} yields of  $u_1, \dots, U_n$  and  $A_1, \dots, A_n$ , where the yield of an **alignment-code** is the {character which is the} intrinsic value {23<sub>8</sub>.1.4.2.b} of its **LETTER-symbol**;

## ALGOL 68 REVISED REPORT

- the descriptor of  $W$  is  $((1, n))$ ;
- the element of  $W$  selected by  $(i)$ ,  $i = 1, \dots, n$ , is a structured value (of the mode specified by  
STRUCT (PROC INT rep, UNION (STRING, CHAR) sa)) whose fields, taken in order, are
  - $\{rep\} R_i$ ;
  - $\{sa\} D_i$ .

e) The yield of an **integral-, real-, boolean-, complex-, string- or bits-pattern** <sup>$P$</sup>  {25<sub>10</sub>.3.4.2.1.a, 25<sub>10</sub>.3.4.3.1.a, 25<sub>10</sub>.3.4.4.1.a, 25<sub>10</sub>.3.4.5.1.a, 25<sub>10</sub>.3.4.6.1.a, 25<sub>10</sub>.3.4.7.1.a} is a structured value  $W$  whose mode is **'PATTERN'**, determined as follows:

- let  $V_1, \dots, V_n$  be the {collateral} yields of the constituent frames of  $P$  {f};
- the fields of  $W$ , taken in order, are
  - $\{type\}$  1(2, 3, 4, 5) if  $P$  is an **integral- (real-, boolean-, complex-, string-) -pattern** and 6(8, 12, 20) if  $P$  is a **bits-pattern** whose constituent **RADIX** is a **radix-two (-four, -eight, -sixteen)**;
  - $\{frames\}$  a multiple value, whose mode is **'row of FRAME'**, having a descriptor  $((1, n))$  and  $n$  elements, that selected by (i) being  $V_i$ .

f) The yield of a **frame**  $F$  {25<sub>10</sub>.3.4.1.1.m} is a structured value  $W$  whose mode is **'FRAME'**, determined as follows:

- the **insertion** and the **replicator**, if any, of  $F$  are elaborated collaterally;
- the fields of  $W$ , taken in order, are
  - $\{i\}$  the yield of its **insertion**;
  - $\{rep\}$  the yield of its **replicator** {c}, if any, and, otherwise, the yield of an invisible **replicator**;
  - $\{supp\}$  true if its first **UNSUPPRESSED-suppression** contains a **letter-s-symbol** and, otherwise, false;
  - $\{marker\}$  (the character which is the intrinsic value {23<sub>8</sub>.1.4.2.b} of a **symbol**  $S$  determined as follows:

Case A:  $F$  is a constituent **unsuppressible-zero-frame** of a **sign-mould** {such as 3z+} whose constituent **sign-marker** contains a **plus-symbol**:

- $S$  is a **letter-u-symbol**.



Case B:  $F$  is a constituent **unsuppressible-zero-frame** of a **sign-mould** {such as  $3z-$ } whose constituent **sign-marker** contains a **minus-symbol**:

- $S$  is a **letter-v-symbol**.

Other cases: • $S$  is the constituent **symbol** of the **marker** of  $F$ .

{Thus the **zero-marker**  $z$  may be passed on as the character "u", "v" or "z" according to whether it forms part of a **sign-mould** (with descendent **plus-symbol** or **minus-symbol**) or of an **integral-mould**.}

## 25.3.4.2 Integral patterns

### 25.3.4.2.1. Syntax

- a) **NEST integral pattern** { $25_{10}.3.4.1.c, 25_{10}.3.4.3.c$ } :  
**NEST sign mould** { $c$ } **option**, **NEST integral mould** { $b$ }.
- b) **NEST integral mould** { $a, 25_{10}.3.4.3.b, c, 25_{10}.3.4.7.a$ } :  
**NEST digit frame** { $25_{10}.3.4.1.k$ } **sequence**.
- c) **NEST sign mould** { $a, 25_{10}.3.4.3.a$ } :  
**NEST unsuppressible zero frame** { $25_{10}.3.4.1.k$ } **sequence option**,  
**NEST unsuppressible sign frame** { $25_{10}.3.4.1.j$ } .
- d) **zero marker** { $f, 25_{10}.3.4.1.k$ } : **letter z** { $24_9.4.a$ } **symbol**.
- e) **sign marker** { $25_{10}.3.4.1.j$ } : **plus** { $24_9.4.c$ } **symbol** ; **minus** { $24_9.4.c$ } **symbol**.
- f) **digit marker** { $25_{10}.3.4.1.k$ } : **letter d** { $24_9.4.a$ } **symbol** ; **zero marker** { $d$ }.

{Examples:

a) "x=" 12z+d

b) d

c) "x=" 12z+ }

{For the semantics of **integral-patterns** see  $25_{10}.3.4.1.2.e$  .}

{aa) The modes which are output (input) compatible with an **'integral'** pattern are those specified by  $L$  INT (by REF  $L$  INT) .

bb) A value  $V$  is converted to a string  $S$  using an **'integral'** pattern  $P$  as follows:

- if  $P$  contains a sign mould, the first character of  $S$  is the sign of  $V$ ; otherwise, if  $V < 0$ , the conversion is unsuccessful;
- the remainder of  $S$  contains a decimal representation of  $V$  determined as follows:
  - the elements of  $S$  controlled by "d" and "z" frames are the appropriate digits (thus the pattern specifies the number of digits to be used);
  - if  $V$  cannot be represented by such a string, the conversion is unsuccessful;

(For example, the value 99 could be converted to a string using the pattern `zzd`, but `9999` and `-99` could not.)

cc) A string  $S$  is converted to an integer suitable for a name  $N$ , using an **'integral'** pattern, as follows:

- the integer  $I$  for which  $S$  contains a decimal representation {23<sub>8</sub>.1.1.2} is considered;
- if  $I$  is greater than the largest value to which  $N$  can refer, the conversion is unsuccessful; otherwise,  $I$  is the required integer (e.g., if the mode of  $N$  is specified by `REF SHORT INT`, and the value of `short max int` is 65535, then no string containing a decimal representation of a value greater than 65535 may be converted) .)

### 25.3.4.3 Real patterns

#### 25.3.4.3.1. Syntax

- a) **NEST real pattern** {25<sub>10</sub>.3.4.1.c, 25<sub>10</sub>.3.4.5.a} :  
**NEST sign mould** {25<sub>10</sub>.3.4.2.c} **option**,  
**NEST variable point mould** {b}  
**or alternatively NEST floating point mould** {c}.
- b) **NEST variable point mould** {a,c} :  
**NEST integral mould** {25<sub>10</sub>.3.4.2.b} ;  
**NEST point frame** {25<sub>10</sub>.3.4.1.j} , **NEST integral mould** {25<sub>10</sub>.3.4.2.b} **option**;  
**NEST point frame** {25<sub>10</sub>.3.4.1.j} , **NEST integral mould** {25<sub>10</sub>.3.4.2.b} .
- c) **NEST floating point mould** {a} :  
**NEST variable point mould** {b}  
**or alternatively NEST integral mould** {25<sub>10</sub>.3.4.2.b} ,  
**NEST exponent frame** {25<sub>10</sub>.3.4.1.j} , **NEST integral pattern** {25<sub>10</sub>.3.4.2.a} .
- d) **point marker** {25<sub>10</sub>.3.4.1.j} : **point** {24<sub>9</sub>.4.b} **symbol**.
- e) **exponent marker** {25<sub>10</sub>.3.4.1.j} : **letter e** {24<sub>9</sub>.4.a} **symbol**.

{Examples:

- a) `+zd.11d • +.12de+2d`
- b) `zd.11d • +.12d`
- c) `.12de+2d }`

{For the semantics of **real-patterns** see 25<sub>10</sub>.3.4.1.2.e .}

{aa) The modes which are output (input) compatible with a **'real'** pattern are those specified by  $L$  REAL and  $L$  INT (by REF  $L$  REAL) .

bb) A value  $V$  is converted to a string  $S$  using a **'real'** pattern  $P$  as follows:

- if  $P$  contains a sign mould, the first character of  $S$  is the sign of  $V$ ; otherwise, if  $V < 0$ , the conversion is unsuccessful;
- the remainder of  $S$  contains a decimal representation of  $V$  determined as follows:
  - if necessary,  $V$  is widened to a real number;
  - the element of  $S$  controlled by the ". " ("e") frame, if any, of  $P$  is ". " ("10");  
If  $P$  contains an "e" frame,  
then
    - let  $W$  be the sequence of frames preceding, and  $IP$  be the **'integral'** pattern following, that "e" frame;
    - an exponent  $E$  is calculated by standardizing  $V$  to the largest value convertible using  $W$  (see below);
    - the part of  $S$  controlled by  $IP$  is obtained by converting  $E$  using  $IP$  (see 25<sub>10</sub>.3.4.2.bb)

otherwise,

- let  $W$  be the whole of  $P$ ;
- the elements of  $S$  controlled by the "d" and "z" frames of  $W$  are the appropriate digits to be used, and the number of digits to be placed after the decimal point, if any);
- if  $V$  cannot be represented by such a string, the conversion is unsuccessful.

cc) A string  $S$  is converted to a real number suitable for a name  $N$ , using a **'real'** pattern  $P$ , as follows:

- the real number  $R$  for which  $S$  contains a decimal representation is considered;
- if  $R$  is greater than the largest value to which  $N$  can refer, the conversion is unsuccessful; otherwise,  $R$  is the required real number.}

### 25.3.4.4 Boolean patterns

#### 25.3.4.4.1. Syntax

- a) **NEST boolean pattern** {25<sub>10</sub>.3.4.1.c} :  
    **NEST unsuppressible boolean frame** {25<sub>10</sub>.3.4.1.j} .
- b) **boolean marker** {25<sub>10</sub>.3.4.1.j, 25<sub>10</sub>.3.4.8.b} : **letter b** {24<sub>9</sub>.4.a} **symbol**.

{Example:

- a) 14x b }

{For the semantics of **boolean-patterns** see 25<sub>10</sub>.3.4.1.2.e .}

{aa) The mode which is output (input) compatible with a '**boolean**' pattern is that specified by **BOOL** (**REF BOOL**). bb) A value *V* is converted to a string using a '**boolean**' pattern as follows:

- if *V* is **true** (**false**), then the string is that yielded by **flip** (**flop**) .

cc) A string *S* is converted to a boolean value, using a '**boolean**' pattern, as follows:

- if *S* is the same as the string yielded by **flip** (**flop**), then the required value is **true** (**false**) .}

### 25.3.4.5 Complex patterns

#### 25.3.4.5.1. Syntax

- a) **NEST complex pattern** {25<sub>10</sub>.3.4.1.c} :  
    **NEST real pattern** {25<sub>10</sub>.3.4.3.a} ,  
    **NEST complex frame** {25<sub>10</sub>.3.4.1.j} ,  
    **NEST real pattern** {25<sub>10</sub>.3.4.3.a} .
- b) **complex marker** {25<sub>10</sub>.3.4.1.j} : **letter i** {24<sub>9</sub>.4.a} **symbol**.

{Example:

- a) +.12de+2d 3q"+j\*"3" "si+.10de+2d }

{For the Semantics of **complex-patterns** see 25<sub>10</sub>.3.4.1.2.e .}

{aa) The modes which are output (input) compatible with a '**complex**' pattern are those specified by  $L_{\text{COMPL}}$ ,  $L_{\text{REAL}}$  and  $L_{\text{INT}}$  (by REF  $L_{\text{COMPL}}$ ).

bb) A value  $V$  is converted to a string  $S$  using a '**complex**' pattern  $P$  as follows:

- if necessary,  $V$  is widened to a complex number;
- the element of  $S$  controlled by the "i" frame of  $P$  is "I";
- the part of  $S$  controlled by the first (second) '**real**' pattern of  $P$  is that obtained by converting the first (second) field of  $V$  to a string using the first (second) '**real**' pattern of  $P$  {25<sub>10</sub>.3.4.3.1.bb}
- if either conversion is unsuccessful, the conversion of  $V$  is unsuccessful.

cc) A string is converted to a complex value  $C$  suitable for a name  $N$ , using a '**complex**' pattern  $P$ , as follows:

- the part of the string controlled by the first (second) '**real**' pattern of  $P$  is converted to a suitable real number {25<sub>10</sub>.3.4.3.1.cc}, which then forms the first (second) field of  $C$ .
- if either conversion is unsuccessful, the conversion to  $C$  is unsuccessful.}

## 25.3.4.6 String patterns

### 25.3.4.6.1. Syntax

- a) **NEST string pattern** {25<sub>10</sub>.3.4.1.c} :  
**NEST character frame** {25<sub>10</sub>.3.4.1.k} **sequence.**
- b) **character marker** {25<sub>10</sub>.3.4.1.k} :  
**letter a** {24<sub>9</sub>.4.a} **symbol.**

{Example:

- a) p "table of"x 10a }

{For the semantics of **string-patterns** see 25<sub>10</sub>.3.4.1.2.e .} {

aa) The modes which are output (input) compatible with a '**string**' pattern are those specified by CHAR and [ ] CHAR (by REF CHAR, REF [ ] CHAR and REF STRING).

bb) A value  $V$  is converted to a string using a '**string**' pattern  $P$  as follows:

## ALGOL 68 REVISED REPORT

- if necessary,  $V$  is rowed to a string:
- if the length of the string  $V$  is equal to the length of the string controlled by  $P$ , then  $V$  is supplied: otherwise, the conversion is unsuccessful,

cc) A string  $S$  is converted to a character or a string suitable for a name  $N$ , using a '**string**' pattern, as follows:

Case A: The mode of  $N$  is specified by REF CHAR:

- if  $S$  does not consist of one character, the conversion is unsuccessful: otherwise, that character is supplied:

Case B: The mode of  $N$  is specified by REF [ ] CHAR:

- if the length of  $S$  is not equal to the number of characters referred to by  $N$ , the conversion is unsuccessful: otherwise,  $S$  is supplied,

Case C: The mode of  $N$  is specified by REF STRING:

- $S$  is supplied.}

### 25.3.4.7 Bits patterns

#### 25.3.4.7.1. Syntax

- a) **NEST bits pattern** {25<sub>10</sub>.3.4.1.c} :  
**NEST RADIX frame** {b}, **NEST integral mould** {25<sub>10</sub>.3.4.2.b} .
- b) **NEST RADIX frame** {a} :  
**NEST insertion** {A341d} , **RADIX** {23<sub>8</sub>.2.d,e,f,g} ,  
**unsuppressible suppression** {25<sub>10</sub>.3.4.1.1} , **radix marker** {c}.
- c) **radix marker** {b} : **letter r** {24<sub>9</sub>.4.a} **symbol**.

{Examples:

- a) 2r6d26sd
- b) 2r }

{For the semantics of **bits-patterns**, see 25<sub>10</sub>.3.4.1.2.e .}

{aa) The modes which are output (input) compatible with a '**bits**'-pattern are those specified by  $L$  BITS (REF  $L$  BITS) .

bb) A value  $V$  is converted to a string using a '**bits**' pattern  $P$  as follows:

- the integer  $I$  corresponding to  $V$  is determined, using the **operator** ABS {25<sub>10</sub>.2.3.8.i};

If the "r" frame of  $P$  was yielded by a **radix-two- (-four-, -eight-, -sixteen-) -frame**,

then

$I$  is converted to a string, controlled by the integral mould of  $P$ , containing a binary (quaternary, octal, hexadecimal) representation of  $I$  (cf. 25<sub>10</sub>.3.4.2.1.bb);

- if  $I$  cannot be represented by such a string, the conversion is unsuccessful.

cc) A string  $S$  is converted to a bits value suitable for a name  $N$ , using a '**bits**' pattern  $P$ , as follows:

- if the "r" frame of  $P$  was yielded by a **radix-two- (-four-, -eight-, -sixteen-) -frame**, then the integer  $I$  for which  $S$  contains a binary (quaternary, octal, hexadecimal) representation is determined;
- the bits value  $B$  corresponding to  $I$  is determined, using the operator BIN (25<sub>10</sub>.2.3.8.j) :
- if the width of  $B$  is greater than that of the value to which  $N$  refers, the conversion is unsuccessful.}

## 25.3.4.8 Choice patterns

### 25.3.4.8.1. Syntax

- NEST integral choice pattern** {25<sub>10</sub>.3.4.1.c} :  
**NEST insertion** {25<sub>10</sub>.3.4.1.d} ,  
**letter c** {24<sub>9</sub>.4.a} **symbol**, **NEST pragli** {c} **list brief pack**,  
**pragment** {24<sub>9</sub>.2.a} **sequence option**.
- NEST boolean choice pattern** {25<sub>10</sub>.3.4.1.c} :  
**NEST insertion** {25<sub>10</sub>.3.4.1.d} , **boolean marker** {25<sub>10</sub>.3.4.4.b} ,  
**brief begin** {24<sub>9</sub>.4.f} **token**, **NEST pragli** {c}, **and also** {24<sub>9</sub>.4.f} **token**,  
**NEST pragli** {c}, **brief end** {24<sub>9</sub>.4.f} **token**,  
**pragment** {24<sub>9</sub>.2.a} **sequence option**.
- NEST pragli** {a,b} : **pragment** {24<sub>9</sub>.2.a} **sequence option**, **NEST literal** {25<sub>10</sub>.3.4.1.i}

{Examples:

## ALGOL 68 REVISED REPORT

- a) 120kc ("mon", "tues", "wednes", "thurs", "fri", "satur", "sun")
- b) b ("", "error")
- c) "mon" }

{aa) A value  $V$  is output using a **picture**  $P$  whose pattern  $Q$  was yielded by an **integral-choice-pattern**  $C$ , as follows:

- the **insertion** of  $Q$  is staticized {25<sub>10</sub>.3.4.1.1.dd} and performed {25<sub>10</sub>.3.4.1.1.ee};

If the mode of  $V$  is specified by INT, if  $V > 0$  and if the number of constituent **literals** in the **praglit-list-pack** of  $C$  is at least  $V$ ,

then

- the literal yielded by the  $V$ -th **literal** is staticized and performed;
- otherwise,
- the event routine corresponding to on value error is called;
  - if this returns false,  $V$  is output using put and undefined is called;
- the **insertion** of  $P$  is staticized and performed.

bb) A value is input to a name  $N$  using a picture  $P$  whose pattern  $Q$  was yielded by an **integral-choice-pattern**  $C$  as follows:

- the **insertion** of  $Q$  is staticized and performed,
- each of the literals yielded by the constituent **literals** of the **praglit-list-pack** of  $C$  is staticized and "searched for" {cc} in turn:

If the mode of  $N$  is specified by REF INT and the  $i$ -th literal is the first one present.

then  $i$  is assigned to  $N$ :

otherwise,

- the event routine corresponding to on value error is called;
  - if this returns false, undefined is called;
- the **insertion** of  $P$  is staticized and performed.

cc) A literal is "searched for" by reading characters and matching them against successive characters of the literal. If the end of the current line or the logical end of the file is reached, or if a character fails to match, the search is unsuccessful and the current position is returned to where it started from.

dd) A value  $V$  is output using a picture  $P$  whose pattern  $Q$  was yielded by a **boolean-choice-pattern**  $C$  as follows:



- the insertion of  $Q$  is staticized and performed:

If the mode of  $V$  is specified by `BOOL`,

then

- if  $V$  is `true` (`false`), the literal yielded by the first (second) constituent **literal** of  $C$  is staticized and performed:

otherwise,

- the event routine corresponding to `on value error` is called;
- if this returns `false`,  $V$  is output using `put` and `undefined` is called;

- the insertion of  $P$  is staticized and performed.

ee) A value is input to a name  $N$  using a picture  $P$  whose pattern  $Q$  was yielded by a **boolean-choice-pattern**  $C$  as follows:

- the insertion of  $Q$  is staticized and performed;
- each of the literals yielded by the constituent **literals** of  $C$  is staticized and searched for in turn:

If the mode of  $N$  is specified by `REF BOOL`, and the first (second) insertion is present,

then `true` (`false`) is assigned to  $N$ :

otherwise,

- the event routine corresponding to `on value error` is called;
- if this returns `false`, `undefined` is called;

- the insertion of  $P$  is staticized and performed.}

#### 25.3.4.8.2. Semantics

The yield of a **choice-pattern**  $P$  is a structured value  $W$  whose mode is **'CPATTERN'**, determined as follows:

- let  $n$  be the number of constituent **NEST-literals** of the **praglit-list-pack** of  $P$ ;
- let  $S_i, i = 1, \dots, n$ , be a **NEST-insertion** akin {15<sub>1</sub>.1.3.2.k} to the  $i$ -th of those constituent **NEST-literals**:
- the **insertion**  $I$  of  $P$  and all of  $S_1, S_2, \dots, S_n$  are elaborated collaterally:
- the fields of  $W$ , taken in order, are

- $\{i\}$  the yield of  $I$ ;
- $\{type\}$  1(2) if  $P$  is a **boolean- (integral-) -choice-pattern**;
- $\{c\}$  a multiple value whose mode is **'row of INSERTION'**, having a descriptor  $((1, n))$  and  $n$  elements, that selected by  $(i), i = 1, \dots, n$ , being the yield of  $S_i$ .

### 25.3.4.9 Format patterns

#### 25.3.4.9.1. Syntax

- a) **NEST format pattern** {25<sub>10</sub>.3.4.1.c} :
- NEST insertion** {25<sub>10</sub>.3.4.1.d} , **letter f** {24<sub>9</sub>.4.a} **symbol**,  
**meek FORMAT NEST ENCLOSED clause** {17<sub>3</sub>.1.a, 17<sub>3</sub>.4.a} ,  
**pragment** {24<sub>9</sub>.2.a} **sequence option**.

{Example:

- a) `f (uir | (INT): $5d$, (REAL): $d.3d$) }`

{A **format-pattern** may be used to provide formats dynamically for use in transput. When a **'format'** pattern is encountered during a call of `get next picture`, it is staticized and its insertion is performed. The first picture of the format returned by the routine of the pattern is supplied as the next picture, and subsequent pictures are taken from that format until it has been exhausted.}

#### 25.3.4.9.2. Semantics

The yield, in an environ  $E$ , of a **NEST-format-pattern**  $P$  is a structured value whose mode is **'FPATTERN'** and whose fields, taken in order, are

- $\{i\}$  the yield of its **insertion**;
- $\{pf\}$  a routine whose mode is **'procedure yielding FORMAT'**, composed of a **procedure-yielding-FORMAT-NEST-routine-text** whose **unit**  $U$  is a new **unit** akin {15<sub>1</sub>.1.3.2.k} to the **meek-FORMAT-ENCLOSED-clause** of  $P$ , together with the environ necessary for  $U$  in  $E$ .

### 25.3.4.10 General patterns

#### 25.3.4.10.1. Syntax

- a) **NEST general pattern** {25<sub>10</sub>.3.4.1.c} :  
**NEST insertion** {25<sub>10</sub>.3.4.1.d} ,  
**letter g** {24<sub>9</sub>.4.a} **symbol**, **NEST width specification** {b} **option**.
- b) **NEST width specification** {a} :  
**brief begin** {24<sub>9</sub>.4.f} **token** ;  
**meek integral NEST unit** {17<sub>3</sub>.2.d} ,  
**NEST after specification** {c} **option**, **brief end** {24<sub>9</sub>.4.f} **token**,  
**pragment** {24<sub>9</sub>.2.a} **sequence option**.
- c) **NEST after specification** {b} :  
**and also** {24<sub>9</sub>.4.f} **token**, **meek integral NEST unit** {17<sub>3</sub>.2.d} ,  
**NEST exponent specification** {d} **option**.
- d) **NEST exponent specification** {c} :  
**and also** {24<sub>9</sub>.4.f} **token**, **meek integral NEST unit** {17<sub>3</sub>.2.d} .

{Examples:

- a)  $g \bullet g(-18, 12, -3)$   
b)  $-18, 12, -3$   
c)  $, 12, -3$   
d)  $, -3 \}$

{aa) A value  $V$  is output using a picture  $P$  whose pattern  $Q$  was yielded by a **general-pattern**  $G$  as follows:

- $P$  is staticized;
  - the insertion of  $Q$  is performed;
- If  $Q$  is not parametrized (i.e.,  $Q$  contains no **width-specification**) .

then  $V$  is output using `put`;

otherwise, if the mode of  $V$  is specified by  $L$  INT or  $L$  REAL,

then

- if  $Q$  contains one (two, three) parameter (s),  $V$  is converted to a string using `whole(fixed, float)`;
- the string is written using `put`;

otherwise,

## ALGOL 68 REVISED REPORT

- the event routine corresponding to `on value error` is called;
- if this returns `false`, `V` is output using `put`, and `undefined` is called;
- the insertion of `P` is performed.

bb) A value is input to a name `N` using a **picture** `P` whose pattern is a **'general'** pattern as follows:

- `P` is staticized;
- (any parameters are ignored and) the value is input to `N` using `get`.)

### 25.3.4.10.2. Semantics

The yield, in an environ `E`, of a **NEST-general-pattern** `P` is a structured value whose mode is **'GPATTERN'** and whose fields, taken in order, are

- `{i}` the yield of the **insertion** of `P`;
- `{spec}` a multiple value `W` whose mode is **'row of procedure yielding integral'**, having a descriptor  $((1, n))$ , where `n` is the number of constituent **meek-integral-units** of the **width-specification-option** of `P`, and `n` elements determined as follows: For  $i = 1, \dots, n$ ,
  - the  $i$ -th element of `W` is a routine, whose mode is **'procedure yielding integral'**, composed of a **procedure-yielding-integral-NEST-routine-text** whose **unit** `U` is a new **unit** akin {15<sub>1</sub>.1.3.2.k} to the  $i$ -th of those **meek-integral-units**, together with the environ necessary for `U` in `E`.

## 25.3.5 Formatted transput

```
a) MODE FORMAT = STRUCT (FLEX [1: 0] PIECE F);
MODE N0 PIECE = STRUCT (INT cp ç pointer to current collection ç,
    count ç number of times piece is to be repeatedç,
    bp ç back pointer ç,
    FLEX [1 : 0] COLLECTION c);
MODE N0 COLLECTION = UNION (PICTURE, COLLITEM) :
MODE N0 COLLITEM = STRUCT (INSERTION i1,
    PROC INT rep; ç replicator ç,
    INT p ç pointer to another piece ç, INSERTION i2);
MODE N0 INSERTION = FLEX [1: 0] STRUCT (PROC INT rep ç replicator ç,
    UNION (STRING, CHAR) sa);
MODE N0 PICTURE = STRUCT (
    UNION (PATTERN, CPATTERN, FPATTERN, GPATTERN, VOID) p,
```

```

INSERTION i);
MODE N0 PATTERN = STRUCT (INT type c of pattern c,
  FLEX [1: 0] FRAME frames),
MODE N0 FRAME = STRUCT (INSERTION i,
  PROC INT rep c replicator c,
  BOOL supp c true if suppressed c,
  CHAR marker);
MODE N0 CPATTERN = STRUCT (INSERTION i,
  INT type c boolean or integral c,
  FLEX [1 : 0] INSERTION c);
MODE N0 FPATTERN = STRUCT (INSERTION i, PROC FORMAT pf);
MODE N0 GPATTERN = STRUCT (INSERTION i, FLEX [1 : 0] PROC INT spec);

b) PROC N0 get next picture = (REF FILE f, BOOL read,
  REF PICTURE picture) VOID:
BEGIN
  BOOL picture found := FALSE, format ended := FALSE;
  WHILE ¬ picture found
  DO IF forp OF f = 0 THEN
    IF format ended
    THEN undefined
    ELIF ¬ (format mended OF f) (f)
    THEN REF INT (forp OF f) := 1;
      cp OF (F OF format OF f) [1] := 1;
      count OF (F OF format OF f) [1] := 1
    ELSE format ended := TRUE
    FI
  ELSE REF INT forp = forp OF f;
    REF FLEX [] PIECE aleph = F OF format OF f;
    CASE (c OF aleph [forp]) [cp OF aleph [forp]] IN
      (COLLITEM cl) :
        ([1 : UPB (il OF cl)] SINSERT si;
          bp OF alph[p OF cl] := forp; forp := SKIP;
          (staticize insertion (il OF cl, si),
            count OF aleph [p OF cl] := rep OF cl);
          (aleph ≠: F OF format OF f | undefined);
          (read | get insertion (f, si) | put insertion (f, si));
          cp OF aleph[p OF cl] := 0;
          forp := p OF cl),
      (PICTURE pict): (picture found := TRUE; picture := pict)
    ESAC;
  WHILE
    (forp ≠ 0 | cp OF aleph[forp] = UPB c OF aleph[forp]) | FALSE)
  DO IF (count OF aleph[forp] -= 1) ≤ 0
  THEN
    IF (forp := bp OF aleph [forp]) ≠ 0
    THEN
      INSERTION extra =
        CASE (c OF aleph [forp]) [cp OF aleph [forp]] IN
          (COLLITEM cl) :

```

## ALGOL 68 REVISED REPORT

```

        (bp OF aleph [p OF c1] := 0; i2 OF c1),
    (PICTURE pict) :
        CASE p OF pict IN
            (FPATTERN fpatt):
                (INT k := forp;
                 WHILE bp OF aleph [k] ≠ forp DO k += 1 OD;
                 aleph := aleph [ : k - 1];
                 i OF pict)
            ESAC
        ESAC;
    INT m = UPB i OF picture, n = UPB extra;
    [1 : m + n] STRUCT (PROC INT rep, UNION (STRING, CHAR) sa) c;
    c[1 : m] := i OF picture; c[m + 1 : m + n] := extra;
    i OF picture := c
    FI
    ELSE cp OF aleph [forp] := 0
    FI OD;
    (forp ≠ 0 | cp OF aleph [forp] += 1)
    FI OD
END;

```

- c) MODE  $\aleph_0$  SINSERT = STRUCT (INT rep, UNION (STRING, CHAR) sa);
- d) PROC  $\aleph_0$  staticize insertion = (INSERTION ins, REF [] SINSERT sins) VOID:  
 ¢ calls collaterally all the replicators in 'frames' ¢  
 IF UPB ins = 1  
 THEN  
 rep OF sins[1] := rep OF ins[1];  
 sa OF sins[1] := sa OF ins[1]  
 ELIF UPB ins > 1  
 THEN (staticize insertion (ins[1], sins[1]),  
 staticize insertion (ins[2 : ], sins[2 : ]))  
 FI;
- e) MODE  $\aleph_0$  SFRAME = STRUCT (FLEX [1 : 0] SINSERT si,  
 INT rep, BOOL supp, CHAR marker);
- f) PROC  $\aleph_0$  staticize frames =  
 ([] FRAME frames, REF [] SFRAME sframes) VOID:  
 ¢ calls collaterally all the replicators in 'frames' ¢  
 IF UPB frames = 1  
 THEN  
 [1 : UPB (i OF frames [1])] SINSERT si;  
 (staticize insertion (i OF frames[1], si),  
 rep OF sframes [1] := rep OF frames [1]);  
 si OF sframes [1] := si;  
 supp OF sframes [1] := supp OF frames [1];  
 marker OF sframes [1] := marker OF frames[1]  
 ELIF UPB frames > 1  
 THEN (staticize frames (frames[1], sframes[1]) .

```

        staticize frames (frames[2 : ], sframes[2 : ]))
    FI;
g) PROC N0 put insertion = (REF FILE f, [] SINSERT si) VOID:
    BEGIN set write mood (f);
        FOR k TO UPB si
        DO
            CASE sa OF si[k] IN
                (CHAR a): alignment (f, rep OF si[k], a, FALSE),
                (STRING s) :
                    TO rep OF si[k]
                    DO
                        FOR i TO UPB s
                        DO checkpos (f); putchar (f, s[i]) OD
                    OD
            ESAC
        OD
    END;
h) PROC N0 get insertion (REF FILE f, [] SINSERT si) VOID:
    BEGIN set read mood (f);
        FOR k TO UPB si
        DO
            CASE sa OF si[k] IN
                (CHAR a): alignment (f, rep OF si [k], a, TRUE),
                (STRING s) :
                    (CHAR c;
                    TO rep OF si[k]
                    DO
                        FOR i TO UPB s
                        DO checkpos (f); get char (f, c);
                            (c ≠ s[i]
                            | (¬ (char error mended OF f) (f, c := s[i])
                            | undefined);
                            set read mood (f))
                        OD
                    OD)
            ESAC
        OD
    END;
i) PROC N0 alignment = (REF FILE f, INT r, CHAR a, BOOL read) VOID:
    IF a = "x" THEN TO r DO space (f) OD
    ELIF a = "y" THEN TO r DO backspace (f) OD
    ELIF a = "l" THEN TO r DO newline (f) OD
    ELIF a = "p" THEN TO r DO newpage (f) OD
    ELIF a = "k" THEN set char number (f, r)
    ELIF a = "q"
    THEN TO r
        DO

```

## ALGOL 68 REVISED REPORT

```

    IF read
    THEN CHAR c; check pos (f); get char (f, c);
        (c ≠ blank
        | (¬ (char error mended OF f) (f, c := blank)
        | undefined); set read mood (f))
    ELSE check pos (f); put char (f, blank)
    FI
  OD
FI;

```

j) PROC  $\aleph_0$  do fpattern = (REF FILE f, FPATTERN fpattern, BOOL read) VOID:  
 BEGIN FORMAT pf;  
 [1 : UPB (i OF fpattern)] SINSERT si;  
 (staticize insertion (i OF fpattern, si),  
 pf := pf OF fpattern);  
 (read | get insertion (f, si) | put insertion (f, si));  
 REF INT forp = forp OF f;  
 REF FLEX [] PIECE aleph =  $F$  OF format OF f;  
 INT m = UPB aleph, n = UPB ( $F$  OF pf);  
 [1 : m + n] PIECE c; c[1 : m] := aleph;  
 c[m + 1 : m + n] :=  $F$  OF pf;  
 aleph := c; bp OF aleph[m + 1] := forp;  
 forp := m + 1; cp OF aleph[forp] := 0;  
 count OF aleph[forp] := 1;  
 FOR i FROM m + 1 TO m + n  
 DO  
 FOR j TO UPB c OF aleph[i]  
 DO  
 CASE (c OF aleph[i]) [j] IN  
 (c OF aleph[i]) [j] :=  
 COLLITEM (i1 OF c1, rep OF c1, p OF c1 + m, i2 OF c1)  
 ESAC  
 OD  
 OD  
 END;

k) PROC  $\aleph_0$  associate format = (REF FILE f, FORMAT format) VOID:  
 BEGIN  
 format OF f :=  
 C a newly created name which is made to refer to the yield  
 of an actual-format-declarer and whose scope is equal to  
 the scope of the value yielded by 'format' C  
 := format;  
 forp OF f := HEAP INT := 1;  
 cp OF ( $F$  OF format OF f) [1] := 1;  
 count OF ( $F$  OF format OF f) [1] := 1;  
 bp OF OF ( $F$  OF format OF f) [1] := 1  
 END;



## 25.3.5.1 Formatted output

```

a) PROC putf = (REF FILE f, [] UNION (OUTTYPE, FORMAT) x) VOID:
  IF opened OF f THEN
  FOR k TO UPB x
  DO CASE set write mood (f); set char mood (f); x[k] IN
    (FORMAT format): associate format (f, format),
    (OUTTYPE ot) :
  BEGIN INT j := 0;
    PICTURE picture, [] SIMPLOUT y = STRAIGHTOUT ot;
    WHILE (j += 1) ≤ UPB y
    DO BOOL incomp := FALSE;
      get next picture (f, FALSE, picture);
      set write mood (f);
      [1 : UPB (i OF picture)] SINSERT sinsert;
      CASE p OF picture IN
        (PATTERN pattern) :
      BEGIN INT rep, sfp := 1;
        [1 : UPB (frames OF pattern)] SFRAME sframes;
        (staticize frames (frames OF pattern, sframes),
         staticize insertion (i OF picture, sinsert));
        STRING s;

        OP  $\aleph_0$  = (STRING s) BOOL:
           $\zeta$  true if the next marker is one of the elements of
            's' and false otherwise  $\zeta$ 
          IF sfp > UPB sframes
          THEN FALSE
          ELSE SFRAME sf = sframes[sfp];
            rep := rep OF sf:
            IF char in string (marker OF sf, LOC INT, s)
            THEN sfp += 1; TRUE
            ELSE FALSE
          FI
        FI;

      OP  $\aleph_0$  = (CHAR c) BOOL:  $\aleph_0$  STRING (c);
      PROC int pattern (REF BOOL sign mould) INT:
        (INT l := 0;
         WHILE  $\aleph_0$  "zuv" DO (rep ≥ 0 | l += rep) OD;
         sign mould :=  $\aleph_0$  "+-";
         WHILE  $\aleph_0$  "zd" DO (rep ≥ 0 | l += rep) OD; l);

      << PROC edit L int (L INT i) VOID:
        (BOOL sign mould; INT l := int pattern (sign mould);
         STRING t = subwhole (ABS i, l);
         IF char in string (errorchar, LOC INT, t) ∨ l = 0
           ∨ ¬ sign mould ∧ i < L 0
         THEN incomp := TRUE
         ELSE t PLUSTO s;

```

## ALGOL 68 REVISED REPORT

```

        (1 - UPB t) × "0" PLUSTO s;
        (sign mould | (i < L 0 | "-" | "+") PLUSTO s)
    FI) >> ;

<< PROC edit L real (L REAL r) VOID:
    (INT b := 0, a := 0, e := 0, exp := 0,
     L REAL y := ABS r,
     BOOL sign1, STRING point := "";
     b := int pattern (sign1);
     (N₀ "." | a := int pattern (LOC BOOL); point := ".");
     IF N₀ "e"
     THEN L standardize (y, b, a, exp);
          edit int (exp);
          "10" PLUSTO s
     FI;
     STRING t = subfixed (y, b + a + (a ≠ 0 | 1 | 0), a);
     IF char in string (errorchar, LOC INT, t) ∨ a + b = 0
        ∨ ¬ sign1 ∧ r < L 0 ∧ exp < 0
     THEN incomp := TRUE
     ELSE t[ : b ] + point + t[b + 2 : ] PLUSTO s;
          (b + a + (a ≠ 0 | 1 | 0) - UPB t) × "0" PLUSTO s;
          (sign1 | (r < L 0 | "-" | "+") PLUSTO s)
     FI) >> ;

<< PROC edit L compl = (L COMPL z) VOID:
    (WHILE ¬ N₀ "i" DO sfp += 1 OD; edit L real (IM z);
     "I" PLUSTO s; sfp := 1; edit L real (RE z)) >> ;

<< PROC edit L bits (L BITS lb, INT radix) VOID:
    (L INT n := ABS lb; N₀ "r";
     INT l := intpattern (LOC BOOL);
     WHILE dig char (S (n ÷× K radix)) PLUSTO s;
          n %:= K radix; n ≠ L 0
     DO SKIP OD;
     IF UPB s ≤ 1
     THEN (1 - UPB s) × "0" PLUSTO s
     ELSE incomp := TRUE
     FI) >> ;

PROC charcount = INT: (INT l := 0;
    WHILE N₀ "a" DO (rep ≥ 0 | l += rep) OD; l);

CASE type OF pattern IN
  † integral †
    (y[j] |
     << (L INT i): edit L int (i) >>
     | incomp := TRUE),
  † real †
    (y[j] |
     << (L REAL r): edit L real (r) >>

```

```

    << (L INT i): edit L real (i) >>
    | incomp := TRUE),
ç boolean ç
    (y[j] |
    (BOOL b): s := (b | flip | flop)
    | incomp := TRUE),
ç complex ç
    (y[j] |
    << (L COMPLEX z): edit L complex (z) >>
    << (L REAL r): edit L complex (r) >>
    << (L INT i): edit L complex (i) >>
    | incomp := TRUE),
ç string ç
    (y[j] |
    (CHAR c): (charcount = 1 | s := c | incomp := TRUE),
    ([] CHAR t) :
        (charcount = UPB t - LWB t + 1
        | s := t[@1]
        | incomp := TRUE)
        | incomp := TRUE)
    OUT
ç bits ç
    (y[j] |
    << (L BITS lb): edit L bits (lb, type OF pattern - 4) >>
    | incomp := TRUE)
ESAC;
IF ¬ incomp
THEN edit string (f, s, sframes)
FI
END,
(CPATTERN choice) :
BEGIN
    [1 : UPB (i OF choice)] SINSERT si;
    staticize insertion (i OF choice, si) :
    put insertion (f, si);
    INT l =
        CASE type OF choice IN
ç boolean ç
        (y [j] |
        (BOOL b): (b | 1 | 2)
        | incomp := TRUE; SKIP),
ç integral ç
        (y [j] |
        (INT i): i
        | incomp := TRUE; SKIP)
    ESAC;
    IF ¬ incomp
    THEN
        IF l > UPB (c OF choice) ∨ l ≤ 0

```

## ALGOL 68 REVISED REPORT

```

        THEN incomp := TRUE
        ELSE
            [1 : UPB ((c OF choice) [1])] SINSERT ci:
                staticize insertion ((c OF choice) [1], ci);
                put insertion (f, ci)
            FI
        FI;
        staticize insertion (i OF picture, sinsert)
    END,
    (FPATTERN fpattern:
    BEGIN
        do fpattern (f, fpattern, FALSE);
        FOR TO UPB sinsert DO sinsert[i] := (0, "") OD;
    END,
    (GPATTERN gpattern) :
    BEGIN
        [1 : UPB (i OF gpattern)] SINSERT si;
        [] PROC INT spec = spec OF gpattern; INT n = UPB spec;
        [1 : n] INT s;
        (staticize insertion (i OF gpattern, si),
        staticize insertion (i OF picture, sinsert),
        s := (n | spec[1], (spec[1], spec[2]),
        (spec[1], spec[2], spec[3]) | ()));
        put insertion (f, si);
        IF n = 0 THEN put (f, y[j])
        ELSE
            NUMBER yj =
                (y[j] | << (L INT i): i >>, << (L REAL r): r >>
                | incomp := TRUE; SKIP);
            IF ¬ incomp
            THEN CASE n IN
                put (f, whole (yj, s[1])),
                put (f, fixed (yj, s[1], s[2])),
                put (f, float (yj, s[1], s[2], s[3]))
            ESAC
            FI
        FI
    END,
    (VOID) :
        (j -= 1; staticize insertion (i OF picture, sinsert))
    ESAC;
    IF incomp
    THEN set write mood (f);
        (¬ (value error mended OF f) (f) | put (f, y [j]) :
        undefined)
    FI;
    put insertion (f, sinsert)
    OD
END

```

```

ESAC OD
ELSE undefined
FI;

b) PROC  $\aleph_0$  edit string (REF FILE f, STRING s, [] SFRAME sf) VOID:
BEGIN BOOL supp, zs := TRUE, signput := FALSE, again,
  INT j := 0, sign;
  PROC copy = (CHAR c) VOID:
    ( $\neg$  supp | check pos (f); put char (f, c));
  FOR k TO UPB sf
  DO SFRAME sfk = sf[k]; supp := supp OF sfk:
    put insertion (f, si OF sfk);
    TO rep OF sfk
    DO again := TRUE;
      WHILE again
      DO j += 1; again := FALSE;
        CHAR sj = s[j], marker = marker OF sfk;
        IF marker = "d"
        THEN copy (sj); zs := TRUE
        ELIF marker = "z" THEN
          (sj = "0" | copy ((zs | " " | sj))
           |: sj = "+" | again := TRUE
           | zs := FALSE; copy (sj))
        ELIF marker = "u"  $\vee$  marker = "v" THEN
          (sj = "+" | sign := 1; again := TRUE
           |: sj = "-" | sign := 2; again := TRUE
           |: sj = "0" | copy ((zs | " " | sj))
           | ( $\neg$  signput |
            copy ((sign | (marker = "u" | "+" | " "), "-"));
            signput := TRUE);
            copy (sj); zs := FALSE)
        ELIF marker = "+" then
          (sj = "+"  $\vee$  sj = "-" | copy (sj)
           | ( $\neg$  signput | copy ((sign | "+", "-")));
           j -= 1)
        ELIF marker = "-" THEN
          (sj = "+" | copy (" ")
           |: sj = "-" | copy (sj)
           | ( $\neg$  signput | copy ((sign | " ", "-")));
           j -= 1)
        ELIF marker = "." THEN
          copy (".")
        ELIF marker = "e"  $\vee$  marker = "i"
           $\vee$  marker = "a"  $\vee$  marker = "b"
        THEN copy (sj); zs := TRUE; signput := FALSE
        ELIF marker = "r"
        THEN j -= 1
      FI
    OD
  OD
OD

```

## ALGOL 68 REVISED REPORT

```
OD
END;
```

### 25.3.5.2 Formatted input

```
a) PROC getf = (REF FILE f, [] UNION (INTYPE, FORMAT) x) VOID:
  IF opened OF f THEN
    FOR k TO UPB x
      DO CASE set read mood (f); set char mood (f); x[k] IN
        (FORMAT format): associate format (f, format),
        (INTYPE it) :
      BEGIN INT j := 0;
        PICTURE picture, [] SIMPLIN y = STRAIGHTIN it;
        WHILE (j +:=- 1) ≤ UPB y
          DO BOOL incomp := FALSE;
            get next picture (f, TRUE, picture); set read mood (f);
            [1 : UPB (i OF picture)] SINSERT sinsert;
            CASE p OF picture IN
              (PATTERN pattern) :
                BEGIN
                  [1 : UPB (frames OF pattern)] SFRAME sframes;
                  (staticize frames (frames OF pattern, sframes),
                   staticize insertion (i OF picture, sinsert));
                  STRING s;
                  INT radix =
                    (type OF pattern ≥ 6 | type OF pattern - 4 | 10);
                  indit string (f, s, sframes, radix);
                  CASE type OF pattern IN
                    † integral †
                      (y[j] |
                       « (REF L INT ii) :
                         incomp := ¬ string to L int (s, 10, ii) »
                       | incomp := TRUE),
                    † real †
                      (y[j] |
                       « (REF L REAL rr) :
                         incomp := ¬ string to L real (s, rr) »
                       | incomp := TRUE),
                    † boolean †
                      (y[j] |
                       (REF BOOL bb): bb := s = flip
                       | incomp := TRUE),
                    † complex †
                      (y[j] |
                       « (REF L COMPL zz) :
                         (INT i, BOOL bi, b2; char in string ("I", i, s);
                          b1 := string to L real (s [ : i - 1], re OF zz);
                          b2 := string to L real (s [i + 1 : ], im OF zz);
```

```

        incomp :=  $\neg$  (b1  $\wedge$  b2))  $\gg$ 
    | incomp := TRUE),
 $\zeta$  string  $\zeta$ 
    (y[j] |
    (REF CHAR cc) :
        (UPB s = 1 | cc := s[1] | incomp := TRUE),
    (REF [] CHAR ss) :
        (UPB ss - LWB ss + 1 = UPB s | ss[@1] := s
        | incomp := TRUE),
    (REF STRING ss): ss := s
    | incomp := TRUE)
    OUT
 $\zeta$  bits  $\zeta$ 
    (y[j] |
     $\ll$  (REF L BITS lb) :
        IF L INT i; string to L int (s, radix, i)
        THEN lb := BIN i
        ELSE incomp := TRUE
        FI  $\gg$ 
    | incomp := TRUE)
    ESAC
END,
(CPATTERN choice) :
BEGIN
    [1 : UPB (i OF choice) SINSERT si;
    staticize insertion (i OF choice, si);
    get insertion (f, si);
    INT c = c OF cpos OF f, CHAR kk;
    INT k := 0, BOOL found := FALSE;
    WHILE k < UPB (c OF choice)  $\wedge$   $\neg$  found
    DO k += 1;
        [1 : UPB ((c OF choice) [k])] SINSERT si;
        BOOL bool := TRUE;
        staticize insertion ((c OF choice, [k], si);
        STRING s;
        FOR i TO UPB si
        DO s PLUSAB
            (sa OF si[i] | (STRING ss): ss)  $\times$  rep OF si[i]
        OD;
        FOR jj TO UPB s
        WHILE bool := bool  $\wedge$   $\neg$  line ended (f)
             $\wedge$   $\neg$  logical file ended (f)
        DO get char (f, kk); bool := kk = s[jj] OD;
        ( $\neg$  (found := bool) | set char number (f, c))
        OD;
    IF  $\neg$  found THEN incomp := TRUE
    ELSE
        CASE type OF choice IN
     $\zeta$  boolean  $\zeta$ 

```

## ALGOL 68 REVISED REPORT

```

        (y [j] |
        (REF BOOL b): b := k = 1
        | incomp := TRUE),
    ç integral ç
        (y [j] |
        (REF INT i): i := k
        | incomp := TRUE)
    ESAC
    FI;
    staticize insertion (i OF picture, sinsert)
END,
(FPATTERN fpattern):
BEGIN do fpattern (f, fpattern, TRUE);
    FOR i TO UPB sinsert DO sinsert [i] := (0, "") OD;
    j -= 1
END,
(GPATTERN gpattern) :
    ([1 : UPB (i OF gpattern)] SINSERT si;
    (staticize insertion (i OF gpattern, si),
    staticize insertion (i OF picture, sinsert));
    get insertion (f, si);
    get (f, y[j])),
(VOID) :
    (j -= 1; staticize insertion (i OF picture, sinsert))
ESAC;
IF incomp
THEN set read mood (f);
    (¬ (value error mended OF f) (f) | undefined)
FI;
get insertion (f, sinsert)
OD
END
ESAC OD
ELSE undefined
FI;

```

- b) PROC  $\aleph_0$  indit string = (REF FILE f, REF STRING s, [] SFRAME sf, INT radix) VOID:
- ```

BEGIN
    BOOL supp, zs := TRUE, sign found := FALSE, space found := FALSE,
    nosign := FALSE, INT sp := 1, rep;
    PRIO ! = 8;

    OP ! = (STRING s, CHAR c) CHAR :
        ç expects a character contained in 's'; if the character
        read is not in 's', the event routine corresponding to 'on
        char error' is called with the suggestion 'c' ç
    IF CHAR k; checkpos (f); get char (f, k);
        char in string (k, LOC INT, s)
    THEN k

```



```

ELSE CHAR sugg := c;
  IF (char error mended OF f) (f, sugg) THEN
    (char in string (sugg, LOC INT, s) | sugg | undefined; c)
  ELSE undefined; c
  FI;
  set read mood (f);11
FI;
OP != (CHAR s, c) CHAR: STRING (s) ! c;
[] CHAR good digits = "0123456789abcdef"[ : radix];
s := "+";
FOR k TO UPB sf
DO SFRAME sfk = sf[k]; supp := supp OF fsk;
  get insertion (f, si OF sfk);
  TO rep OF sfk
DO CHAR marker = marker OF sfk;
  IF marker = "d" THEN
    s PLUSAB (supp | "0" | good digits ! "0"); zs := TRUE
  ELIF marker = "z" THEN
    s PLUSAB (supp | "0"
      | CHAR c = ((zs | " " | "") + gooddigits) ! "0";
      (c ≠ " " | zs := FALSE); c)
  ELIF marker = "u" ∨ marker = "+" THEN
    IF sign found
    THEN zs := FALSE; a PLUSAB ("0123456789" ! "0")
    ELSE CHAR c = ("+-" + (marker = "u" | " " | "")) ! "+";
      (c = "+" ∨ c = "-" | sign found := TRUE; s[sp] := c)
    FI
  ELIF marker = "v" ∨ marker = "-" THEN
    IF sign found
    THEN zs := FALSE; z PLUSAB ("0123456789" ! "0")
    ELIF CHAR c; space found
    THEN c := "+- 0123456789" ! "+";
      (c = "+" ∨ c = "-" | sign found := TRUE; s[sp] := c
      |: c ≠ " " | zs := FALSE; sign found := TRUE; s PLUSAB c)
    ELSE c := "+- " ! "+";
      (c = "+" ∨ c = "-" | sign found := TRUE; s[sp] := c
      | space found := TRUE)
    FI
  ELIF marker = "." THEN
    s PLUSAB (supp | "." | ".! ".)
  ELIF marker = "e" THEN
    s PLUSAB (supp | "10" | "10\e" ! "10"; "10");

```

---

<sup>11</sup>This line and the four preceding ones should read

```

CHAR cc = IF (char error mended OF f) (f, sugg) THEN
  (char in string (sugg, LOC INT, s) | sugg | undefined; c)
ELSE undefined; c
FI;
set read mood (f); cc

```

## ALGOL 68 REVISED REPORT

```
        sign found := FALSE;
        zs := TRUE; s PLUSAB "+"; sp := UPB s
    ELIF marker = "i" THEN
        s PLUSAB (supp | "I" | "iI"! "I"; "I");
        sign found := FALSE; zs := TRUE; s PLUSAB "+"; sp := UPB s
    ELIF marker = "b" THEN
        s PLUSAB (flip + flop) ! flop; no sign := TRUE
    ELIF marker = "a" THEN
        s PLUSAB (supp | " "
            | CHAR c; check pos (f); get char (f, c);
            c);
        no sign := TRUE
    ELIF marker = "r"
    THEN SKIP
    FI
    OD
OD;
IF no sign THEN s := s[2 : ] FI
END;
```

### 25.3.6 Binary transput

{In binary transput, the values obtained by straightening the elements of a data list (cf. 25<sub>10</sub>.3.3) are transput, via the specified file, one after the other. The manner in which such a value is stored in the book is defined only to the extent that a value of mode  $M$  (being some mode from which that specified by SIMPLOUT is united) output at a given position may subsequently be re-input from that same position to a name of mode 'reference to  $M$ '. Note that, during input to the name referring to a multiple value, the number of elements read will be the existing number of elements referred to by that name.

The current position is advanced after each value by a suitable amount and, at the end of each line or page, the appropriate event routine is called, and next, if this returns false, the next good character position of the book is found (cf. 25<sub>10</sub>.3.3).

For binary output, put bin (25<sub>10</sub>.3.6.1.a) and write bin (25<sub>10</sub>.5.1.h) may be used and, for binary input, get bin (25<sub>10</sub>.3.6.2.a) and read bin (25<sub>10</sub>.5.1.i).}

- a) PROC  $\aleph_0$  to bin = (REF FILE f, SIMPLOUT x) [] CHAR:  
**C** a value of mode 'row of character' whose lower bound is one and whose upper bound depends on the value of 'book OF f' and on the mode and the value of 'x'; furthermore, x = from bin (f, x, to bin (f, x)) **C** ;
- b) PROC  $\aleph_0$  from bin = (REF FILE f, SIMPLOUT y. [] CHAR c) SIMPLOUT:  
**C** a value, if one exists, of the mode of the value yielded by 'y', such that c = to bin (f, from bin (f, y, c)) **C** ;

**25.3.6.1 Binary output**

```

a) PROC put bin = (REF FILE f, [] OUTTYPE ot) VOID:
  IF opened OF f THEN
    set bin mood (f);
    set write mood (f);
    FOR k TO UPB ot DO
      [] SIMPLOUT y = STRAIGHTOUT ot [k];
      FOR j TO UPB y DO
        [] CHAR bin = to bin (f, y[j]);
        FOR i TO UPB bin DO
          next pos (f);
          set bin mood (f);
          REF POS cpos = cpos OF f, lpos = lpos OF book OF f;
          CASE text OF f IN
            (FLEXTEXT t2):
              t2 [p OF cpos][l OF cpos][c OF cpos] := bin [i]
          ESAC;
          c OF cpos += 1;
          IF cpos BEYOND lpos THEN
            lpos := cpos
          ELIF ¬ set possible (f) ^
            POS (p OF lpos, l OF lpos, 1) BEYOND cpos THEN
            lpos := cpos; (compressible (f) |

```

**C** the size of the line and page containing the logical end of the book and of all subsequent lines and pages may be increased **C** )

```

          FI
        OD
      OD
    OD
  ELSE undefined
  FI;

```

**25.3.6.2 Binary input**

```

a) PROC get bin = (REF FILE f, [] INTYPE it) VOID:
  IF opened OF f THEN
    set bin mood (f); set read mood (f);
    FOR k TO UPB it
      DO [] SIMPLIN y = STRAIGHTIN it [k];
        FOR j TO UPB y
          DO
            SIMPLOUT yj = CASE y[j] IN
              << (REF L INT i): i >>,
              << (REF L REAL r): r >>,
              << (REF L COMPL z): z >>,

```

# ALGOL 68 REVISED REPORT

```

        (REF BOOL b): b,
        << (REF L BITS lb): lb >>,
        (REF CHAR c): c,
        (REF [] CHAR s): s,
        (REF STRING ss): ss ESAC;
    [1 : UPB (to bin (f, yj))] CHAR bin;
    FOR i TO UPB bin
    DO next pos (f); set bin mood (f);
        REF POS cpos = cpos OF f;
        bin[i] :=
            CASE text OF f IN
                (FLEXTEXT t2) :
                    t2[p OF cpos][l OF cpos][c OF cpos]
            ESAC;
        c OF cpos += 1
    OD;
    CASE y[j] IN
        << (REF L INT ii):
            (from bin (f, ii, bin) | (L INT i): ii := i) >>,
        << (REF L REAL rr) :
            (from bin (f, rr, bin) | (L REAL r): rr := r) >>,
        << (REF L COMPL zz) :
            (from bin (f, zz, bin) | (L COMPL z): zz := z) >>,
        (REF BOOL bb): (from bin (f, bb, bin) | (BOOL b): bb := b),
        << (REF L BITS lb) :
            (from bin (f, lb, bin) | (L BITS b): lb := b) >>,
        (REF CHAR cc): (from bin (f, cc, bin) | (CHAR c): cc := c),
        (REF []CHAR ss) :
            (from bin (f, ss, bin) | ([] CHAR s): ss := s),
        (REF STRING ssss) :
            (from bin (f, ssss, bin) | ([] CHAR s): ssss := s)
    ESAC
    OD
    OD
    ELSE undefined
    FI;

```

{



{But Eeyore wasn't listening. He was taking the balloon out, and putting it back again, as happy as could be. ...

Winnie-the-Pooh,

A.A. Milne. }

}

## 25.4 The system prelude and task list

### 25.4.0.3 The system prelude

The representation of the **system-prelude** is obtained from the following form, to which may be added further forms not defined in this Report. {The syntax of **program-texts** ensures that a **declaration** contained in the **system-prelude** may not contradict any **declaration** contained in the **standard-prelude**. It is intended that the further forms should contain declarations that are needed for the correct operation of any **system-tasks** that may be added (by the implementer, as provided in 25<sub>10</sub>.1.2.d) .}

```
a) SEMA N0 gremlins = (SEMA s; F OF s := PRIM INT := 0; s);
```

### 25.4.0.4 The system task list

The representation of the {first} constituent **system-task** or the **system-task-list** is obtained from the following form. The other **system-tasks**, if any, are not defined by this Report {but may be defined by the implementer in order to account for the particular features of his operating environment, especially in so far as they interact with the running of **particular-programs** (see, e.g., 25<sub>10</sub>.3.1.1.dd) } .

```
a) DO DOWN gremlins; undefined; UP bfileprotect OD
```

{The intention is that this call of `undefined`, which is released by an `UP gremlins` whenever a book is closed, may reorganize the chain of backfiles and the chain of locked backfiles, such as by removing the book if it is not to be available for further opening, or by inserting it into the chain of backfiles several times over if it is to be permitted for several **particular-programs** to read it simultaneously. Note that, when an `UP gremlins` is given, `bfileprotect` is always down and remains so until such reorganization has taken place.}

```
{From ghoulies and ghosties and long-
leggety beasties and things that go bump
in the night,
Good Lord, deliver us!
Ancient Cornish litany }
```

## 25.5 The particular preludes and postludes

### 25.5.1 The particular preludes

The representation of the **particular-prelude** of each **user-task** is obtained from the following forms, to which may be added such other forms as may be needed for the proper functioning of the facilities defined in the constituent **library-prelude** of the **program-text** {, e.g., **declarations** and calls of `open` for additional standard files}. However, for each **QUALITY-new-new-PROPS1-LAYER2-defining-indicator-with-TAX** contained in such an additional form, the predicate '**where QUALITY TAX independent PROPS1**' (7.1.1.a, c) must hold [i.e., no **declaration** contained in the **standard-prelude** may be contradicted] .

- a) `L INT L last random := ROUND (L maxint / L 2);`
- b) `PROC L random = L REAL: L next random (L last random);`
- c) `FILE stand in, stand out, stand back;`  
`open (stand in, "", stand in channel);`  
`open (stand out, "", stand out channel);`  
`open (stand back, "", stand back channel);`
- d) `PROC print = ([ UNION (OUTTYPE, PROC (REF FILE) VOID) x) VOID:`  
`put (stand out, x),`  
`PROC write = ([ UNION (OUTTYPE, PROC (REF FILE) VOID) x) VOID:`  
`put (stand out, x);`
- e) `PROC read = ([ UNION (INTYPE, PROC (REF FILE) VOID) x) VOID:`  
`get (stand in, x);`
- f) `PROC printf = ([ UNION (OUTTYPE, FORMAT) x) VOID:`  
`putf (stand out, x),`  
`PROC writef = ([ UNION (OUTTYPE, FORMAT) x) VOID:`  
`putf (stand out, x);`
- g) `PROC readf = ([ UNION (INTYPE, FORMAT) x) VOID:`  
`getf (stand in, x);`
- h) `PROC write bin = ([ OUTTYPE x) VOID:`  
`put bin (stand back, x);`
- i) `PROC read bin = ([ INTYPE x) VOID:`  
`get bin (stand back, x);`

### 25.5.2 The particular postludes

The representation of the **particular-postlude** of each **user-task** is obtained from the following form, to which may be added such other forms as may be needed for the proper functioning of the facilities defined in the constituent **library-prelude** of the **program-text** {, e.g., **calls** of `lock` for additional standard files}.

- a) `stop: lock (stand in); lock (stand out); lock (stand back)`

# Chapter 26

## Examples

### 26.1 Complex square root

```
PROC compsqrt = (COMPL z) COMPL :  
  ⚭ the square root whose real part is nonnegative  
  of the complex number 'z' ⚭  
  BEGIN REAL x = RE z, y = IM z;  
    REAL rp = sqrt ((ABS x + sqrt (x ↑ 2 + y ↑ 2)) / 2);  
    REAL ip = (rp = 0 | 0 | y / (2 × rp));  
    IF x ≥ 0 THEN rp ⊥ ip ELSE ABS ip ⊥ (y ≥ 0 | rp | -rp) FI  
  END
```

Calls {19.4.3} using compsqrt:

```
compsqrt (w)  
compsqrt (-3.14)  
compsqrt (-1)
```

### 26.2 Innerproduct 1

```
PROC innerproduct1 = INT n, PROC INT) REAL x, y) REAL :  
  ⚭ the innerproduct of two vectors, each with 'n' components, x (i),  
  y (i), i = 1, ..., n, where 'x' and 'y' are arbitrary mappings  
  from integer to real numbers ⚭  
  BEGIN LONG REAL s := LONG 0;  
    FOR i TO n DO s += LENG × (i) × LENG y (i) OD;  
  SHORTEN s
```

## ALGOL 68 REVISED REPORT

END

Real-calls using innerproduct 1:

```
innerproduct1 (m, (INT j) REAL: x1[j], (INT j) REAL: y1[j])
innerproduct1 (n, nsin, ncos)
```

### 26.3 Innerproduct 2

```
PROC innerproduct 2 = (REF [ ] REAL a, b) REAL:
  IF UPB a - LWB a = UPB b - LWB b
  THEN  $\zeta$  the innerproduct of two vectors 'a' and 'b' with equal numbers
    of elements  $\zeta$ 
    LONG REAL s := LONG 0;
    REF [ ] REAL a1 = a[@1], b1 = b[@1];
     $\zeta$  note that the bounds of 'a[@1]' are [1 : UPB a - LWB a + 1]  $\zeta$ 
    FOR i TO UPB a1 DO s += LENG a1[i]  $\times$  LENG b1[i] OD;
    SHORTEN s
  FI
```

Real-calls using innerproduct 2:

```
innerproduct2 (x1, y1)
innerproduct2 (y2[2, ], y2[, 3])
```

### 26.4 Largest element

```
PROC absmax = (REF [, ] REAL a,  $\zeta$  result  $\zeta$  REF REAL y,
   $\zeta$  subscripts  $\zeta$  REF INT i, k) VOID:
   $\zeta$  the absolute value of the element of greatest absolute value of
  the matrix 'a' is assigned to 'y', and the subscripts of this
  element to 'i' and 'k' $\zeta$ 
  BEGIN y := -1;
  FOR p FROM 1 LWB a TO 1 UPB a
  DO
    FOR q FROM 2 LWB a TO 2 UPB a
    DO
      IF ABS a[p, q] > y THEN y := ABS a[i := p, k := q] FI
    OD
  OD
  END
```



Calls using `absmax`:

```
absmax (x2, x, i, j)
absmax (x2, x, LOC INT, LOC INT)
```

## 26.5 Euler summation

```
PROC euler = (PROC (INT) REAL f, REAL eps, INT tim) REAL:
  ⚭ the sum for 'i' from 1 to infinity of 'f (i) ', computed by means
  of a suitably refined Euler transformation. The summation is
  terminated when the absolute values of the terms of the
  transformed series are found to be less than 'eps' 'tim' times in
  succession. This transformation is particularly efficient in the
  case of a slowly convergent or divergent alternating series ⚭
BEGIN INT n := 1, t; REAL mn, ds := eps; [1 : 16] REAL m;
REAL sum := (m[1] := f (1)) / 2;
FOR i FROM 2 WHILE (t := ABS ds < eps | t + 1 | 1) ≤ tim
DO mn := f (i);
  FOR k TO n DO mn := ((ds := mn) + m[k]) / 2; m[k] := ds OD;
  sum += (ds := (ABS mn < ABS m[n] ∧ n < 16 | n += 1; m[n] := mn;
    mn / 2 | mn))
OD;
sum
END
```

A call using `euler`:

```
euler ((INT i) REAL: (ODD i | -1 / i | 1 / i), 1.e-5, 2)
```

## 26.6 The norm of a vector

```
PROC norm = (REF [ ] REAL a) REAL :
  ⚭ the euclidean norm of the vector 'a' ⚭
(LONG REAL s := LONG 0;
FOR k FROM LWB a TO UPB a DO s += LENG a[k] ↑ 2 OD;
SHORTEN long sqrt (s))
```

For a use of `norm` in a call, see 26<sub>11</sub>.7.

## 26.7 Determinant of a matrix

```

PROC det = (REF [, ] REAL x, REF [ ] INT p) REAL :
  IF REF [, ] REAL a = x[@1, @1];
    1 UPB a = 2 UPB a ^ 1 UPB a = UPB p - LWB p + 1
  THEN INT n = 1 UPB a;
    ⚭ the determinant of the square matrix 'a' of order 'n' by the
    method of Crout with row interchanges: 'a' is replaced by its
    triangular decomposition,  $l \times u$ , with all  $u[k, k] = 1$ .
    The vector 'p' gives as output the pivotal row indices; the k-th
    pivot is chosen in the k-th column of T such that
     $ABS\ l[i, k] / \text{row norm}$  is maximal ⚭
  [1 : n] REAL v; REAL d := 1, s, pivot;
  FOR i TO n DO v[i] := norm (a[i, ]) OD;
  FOR k TO n
  DO INT k1 = k - 1; REF INT pk = p[@1][k]; REAL r := -1;
    REF [, ] REAL al = a[, 1 : k1], au = a[1 : k1,];
    REF [ ] REAL ak = a[k, ], ka = a[, k],
      alk = al[k, ], kau = au[, k];
    FOR i FROM k TO n
    DO REF REAL aik = ka[i];
      IF (s := ABS (aik -:= innerproduct2 (al[i, ], kau)) / v[i]) > r
      THEN r := s; pk := i
      FI
    OD;
    v[pk] := v[k]; pivot := ka[pk]; REF [ ] REAL apk = a[pk, ];
    FOR j TO n
    DO REF REAL akj = ak[j], apkj = apk[j];
      r := akj;
      akj := IF j ≤ k THEN apkj
        ELSE (apkj - innerproduct2 (alk, au[, j])) / pivot FI;
      IF pk /= k THEN apkj := -r FI
    OD;
    d ×:= pivot
  OD;
  d
FI

```

A call using det:

```
det (y2, i1)
```

## 26.8 Greatest common divisor

```
PROC gcd = (INT a, b) INT:
  ⚭ the greatest common divisor of two integers ⚭
  (b = 0 | ABS a | gcd (b, a MOD b))
```

A call using gcd:

```
gcd (n, 124)
```

## 26.9 Continued fraction

```
OP / = ([ ] REAL a, [ ] REAL b) REAL:
  ⚭ the value of a / b is that of the continued fraction
  a1 / (b1 + a2 / (b2 + ... an / bn) ...) ⚭
  IF LWB a = 1 ^ LWB b = 1 ^ UPB a = UPB b
  THEN (UPB a = 0 | 0 | a[1] / (b[1] + a[2 : ] / b[2 : ]))
  FI
```

A formula using /:

```
x1 / y1
```

{The use of recursion may often be elegant rather than efficient as in the recursive procedure 26<sub>11</sub>.8 and the recursive operation 26<sub>11</sub>.9 . See, however, 26<sub>11</sub>.10 and 26<sub>11</sub>.13 for examples in which recursion is of the essence.}

## 26.10 Formula manipulation

```
BEGIN
  MODE FORM = UNION (REF CONST, REF VAR, REF TRIPLE, REF CALL);
  MODE CONST = STRUCT (REAL value);
  MODE VAR = STRUCT (STRING name, REAL value);
  MODE TRIPLE = STRUCT (FORM left operand, INT operator,
    FORM right operand);
  MODE FUNCTION = STRUCT (REF VAR bound var, FORM body);
  MODE CALL = STRUCT (REF FUNCTION function name, FORM parameter);
  INT plus = 1, minus = 2, times = 3, by = 4, to = 5;
  HEAP CONST zero, one; value OF zero := 0; value OF one := 1;
  OP = = (FORM a, REF CONST b) BOOL: (a | (REF CONST ec): ec :=: b | FALSE);
  OP + = (FORM a, b) FORM:
```

## ALGOL 68 REVISED REPORT

```
a = zero | b | : b = zero | a | HEAP TRIPLE := (a, plus, b));
OP - = (FORM a, b) FORM: (b = zero | a | HEAP TRIPLE := (a, minus, b));
OP × = (FORM a, b) FORM: (a = zero ∨ b = zero | zero | : a = one | b
| : b = one | a | HEAP TRIPLE := (a, times, b));
OP / = (FORM a, b) FORM: (a = zero ∧ ¬ (b = zero) | zero | :
b = one | a | HEAP TRIPLE := (a, by, b));
OP ↑ = (FORM a, REF CONST b) FORM:
(a = one ∨ (b :=: zero) | one | :
b :=: one | a | HEAP TRIPLE := (a, to, b));
PROC derivative of = (FORM e, ϕ with respect to ϕ REF VAR x) FORM:
CASE e IN
  (REF CONST): zero,
  (REF VAR ev): (ev :=: x | one | zero),
  (REF TRIPLE et) :
    CASE FORM u = left operand OF et, v = right operand OF et;
      FORM udash = derivative of (u, ϕ with respect to ϕ x),
      vdash = derivative of (v, ϕ with respect to ϕ x);
      operator OF et
    IN
      udash + vdash,
      udash - vdash,
      u × vdash + udash × v,
      (udash - et × vdash) / v,
      (v | (REF CONST ec): v × u ↑ (HEAP CONST c;
        value OF c := value OF ec - 1; c) × udash)
    ESAC,
  (REF CALL ef) :
    BEGIN REF FUNCTION f = function name OF ef;
      FORM g = parameter OF ef; REF VAR y = bound var OF f;
      HEAP FUNCTION fdash := (y, derivative of (body OF f, y));
      HEAP CALL := (fdash, g) × derivative of (g, x)
    END
ESAC;
PROC value of = (FORM e) REAL:
CASE e IN
  (REF CONST ec): value OF ec,
  (REF VAR ev): value OF ev,
  (REF TRIPLE et) :
    CASE REAL u = value of (left operand OF et),
      v = value of (right operand OF et);
      operator OF et
    IN u + v, u - v, u × v, u / v, exp (v × ln (u))
    ESAC,
  (REF CALL ef) :
```

```

BEGIN REF FUNCTION f = function name OF ef;
  value OF bound var OF f := value of (parameter OF ef);
  value of (body OF f)
END
ESAC;
HEAP FORM f, g;
HEAP VAR a := ("a", SKIP), b := ("b", SKIP), x := ("x", SKIP);
ç start here ç
read ((value OF a, value OF b, value OF x));
f := a + x / (b + x);
g := (f + one) / (f - one);
print ((value OF a, value OF b, value OF x,
  value of (derivative of (g, ç with respect to ç x))))
END ç example of formula manipulation ç

```

## 26.11 Information retrieval

```

BEGIN
MODE RA = REF AUTH, RB = REF BOOK;
MODE AUTH = STRUCT (STRING name, RA next, RB book),
BOOK = STRUCT (STRING title, RB next);
RA auth, first auth := NIL, last auth;
RB book; STRING name, title; INT i; FILE input, output;
open (input, "", remote in); open (output, "", remote out);
putf (output, ($p
  "to enter a new author, type "author", a space,"x
  "and his name."l
  "to enter a new book, type "book", a space,"x
  "the name of the author, a new line, and the title."l
  "for a listing of the books by an author, type "list", "x
  "a space, and his name."l
  "to find the author of a book, type "find", a new line,"x
  "and the title."l
  "to end, type "end""al$, ".));
PROC update = VOID:
  IF RA (first auth) :=: NIL
  THEN auth := first auth := last auth := HEAP AUTH := (name, NIL, NIL)
  ELSE auth := first auth;
  WHILE RA (auth) :≠: NIL
  DO
    (name = name OF auth | GO TO known | auth := next OF auth)
  OD;
  lastauth := next OF lastauth := auth :=
  HEAP AUTH := (name, NIL, NIL);

```

## ALGOL 68 REVISED REPORT

```
known:  SKIP
        FI;

DO
try again:
    getf (input, ($c ("author", "book", "list", "find", "end", ""),
        x30a1, 80a1$, i));

CASE i IN

    ç authorç
    getf (input, name); update),

    ç book ç
    BEGIN getf (input, (name, title)); update;
        IF RB (book OF auth) :=: NIL
        THEN book OF auth := HEAP BOOK := (title, NIL)
        ELSE book := book OF auth;
            WHILE RB (next OF book) :=: NIL
            DO
                (title = title OF book
                | GO TO try again | book := next OF book)
            OD;
            (title /= title OF book
            | next OF book := HEAP BOOK := (title, NIL))
        FI
    END,

    ç list ç
    BEGIN getf (input, name); update;
        putf (output, ($p"author: "30a11$, name));
        IF RB (book := book OF auth) :=: NIL
        THEN put (output, ("no publications", newline))
        ELSE on page end (output,
            (REF FILE f) BOOL:
                (putf (f, ($p"author: "30a41k"continued"11$, name));
                TRUE));
            WHILE RB (book) :=: NIL
            DO putf (output, ($180a$, title OF book)); book := next OF book
            OD;
            on page end (output, (REF FILE f) BOOL: FALSE)
        FI
    END,

    ç find ç
    BEGIN getf (input, (LOC STRING, title)); auth := first auth;
        WHILE RA (auth) :=: NIL
        DO book := book OF auth;
            WHILE RB (book) :=: NIL
            DO
                IF title = title OF book
```

```

        THEN putf (output, ($1"author:  "30a$, name OF auth));
        GO TO try again
    ELSE book := next OF book
    FI
    OD;
    auth := next OF auth
    OD;
    put (output, (newline, "unknown", newline))
END,

ç end ç
    (put (output, (new page, "signed off", close)); close (input);
    GOTO stop),

ç error ç
    (put (output, (newline, "mistake, try again")); newline (input))
ESAC
OD
END

```

## 26.12 Cooperating sequential processes

```

BEGIN INT nmb magazine slots, nmb producers, nmb consumers;
    read ((nmb magazine slots, nmb producers, nmb consumers));
    [1 : nmb producers] FILE in file; [1 : nmb consumers] FILE out file;
    FOR i TO nmb producers DO open (in file[i], "", inchannel[i]) OD;
    ç 'inchannel' and 'outchannel' are defined in a surrounding range ç
    FOR i TO nmb consumers
    DO open (out file[i], "", outchannel[i]) OD;
    MODE PAGE = [1 : 60, 1 : 132] CHAR ;
    [1 : nmb magazine slots] REF PAGE magazine;
    INT ç pointers of a cyclic magazine ç index:= 1, exdex := 1;
    SEMA full slots = LEVEL 0, free slots = LEVEL nmb magazine slots,
    in buffer busy = LEVEL 1, out buffer busy = LEVEL 1;
    PROC par call = (PROC (INT) VOID p, INT n) VOID:
        ç call 'n' incarnations of 'p' in parallel ç
        (n > 0 | PAR (p (n), par call (p, n - 1)));
    PROC producer = (INT i) VOID:
        DO HEAP PAGE page;
            get (in file[i], page);
            DOWN free slots; DOWN in buffer busy;
            magazine[index] := page;
            index MODAB nmb magazine slots PLUSAB 1;
            UP full slots; UP in buffer busy

```

## ALGOL 68 REVISED REPORT

```
OD;
PROC consumer = (INT i) VOID:
  DO PAGE page;
    DOWN full slots; DOWN out buffer busy;
    page := magazine[exdex];
    exdex MODAB nmb magazine slots PLUSAB 1;
    UP free slots; UP out buffer busy;
    put (out file[i], page)
  OD;
PAR (par call (producer, nmb producers),
    par call (consumer, nmb consumers))
END
```

### 26.13 Towers of Hanoi

```
FOR k TO 8
DO FILE f := stand out;
  PROC p = (INT me, de, ma) VOID:
    IF ma > 0 THEN
      p (me, 6 - me - de, ma - 1);
      putf (f, (me, de, ma));
      ¢ move from peg 'me' to peg 'de' piece 'ma' ¢
      p (6-me-de, de, ma -1)
    FI ;
  putf (f, ($1"k = "dl,n ((2 ↑ k + 15) ÷ 16) (2(2(4(3(d)x)x)x)l)$, k));
  p (1, 2, k)
OD
```



# Chapter 27

## Glossaries

### 27.1 Technical terms

Given below are the locations of the defining occurrences of a number of words which, in this Report, have a specific technical meaning. A word appearing in different grammatical forms is given once, usually as the infinitive. Terms which are used only within pragmatic remarks are enclosed within braces.

|                                                    |                              |
|----------------------------------------------------|------------------------------|
| abstraction (a protonotion of a protonotion)       | 15 <sub>1</sub> .1.4.2.b     |
| acceptable to (a value acceptable to a mode)       | 16 <sub>2</sub> .1.3.6.d     |
| access (inside a locale)                           | 16 <sub>2</sub> .1.2.c       |
| action                                             | 16 <sub>2</sub> .1.4.1.a     |
| active (action)                                    | 16 <sub>2</sub> .1.4.3.a     |
| after (in the textual order)                       | 15 <sub>1</sub> .1.3.2.i     |
| akin (a production tree to a production tree)      | 15 <sub>1</sub> .1.3.2.k     |
| {alignment}                                        | 25 <sub>10</sub> .3.4.1.1.ff |
| alternative                                        | 15 <sub>1</sub> .1.3.2.c     |
| apostrophe                                         | 15 <sub>1</sub> .1.3.1.a     |
| arithmetic value                                   | 16 <sub>2</sub> .1.3.1.a     |
| ascribe (a value or scene to an <b>indicator</b> ) | 18 <sub>4</sub> .8.2.a       |
| assign (a value to a name)                         | 19 <sub>5</sub> .2.1.2.b     |
| asterisk                                           | 15 <sub>1</sub> .1.3.1.a     |
| {balancing}                                        | 17 <sub>3</sub> .4.1.b       |

## ALGOL 68 REVISED REPORT

|                                                         |                              |
|---------------------------------------------------------|------------------------------|
| before (in the textual order)                           | 15 <sub>1</sub> .1.3.2.i     |
| blind alley                                             | 15 <sub>1</sub> .1.3.2.d     |
| {book}                                                  | 25 <sub>10</sub> .3.1.1.b    |
| bound                                                   | 16 <sub>2</sub> .1.3.4.b     |
| bound pair                                              | 16 <sub>2</sub> .1.3.4.b     |
| built (the name built from a name)                      | 21 <sub>6</sub> .6.2.c       |
| built (the multiple value built from a value)           | 21 <sub>6</sub> .6.2.b       |
| calling (of a routine)                                  | 19 <sub>5</sub> .4.3.2.b     |
| {channel}                                               | 25 <sub>10</sub> .3.1.2.b    |
| character                                               | 16 <sub>2</sub> .1.3.1.g     |
| chosen (scene of a <b>chooser-clause</b> )              | 17 <sub>3</sub> .4.2.b       |
| {close (a file)}                                        | 25 <sub>10</sub> .3.1.4.ff   |
| collateral action                                       | 16 <sub>2</sub> .1.4.2.a     |
| collateral elaboration                                  | 16 <sub>2</sub> .1.4.2.f     |
| {collection}                                            | 25 <sub>10</sub> .3.4.1.1.gg |
| colon                                                   | 15 <sub>1</sub> .1.3.1.a     |
| comma                                                   | 15 <sub>1</sub> .1.3.1.a     |
| complete (an action)                                    | 16 <sub>2</sub> .1.4.3.c, d  |
| {compressible}                                          | 25 <sub>10</sub> .3.1.3.ff   |
| consistent substitute                                   | 15 <sub>1</sub> .1.3.4.e     |
| constituent                                             | 15 <sub>1</sub> .1.4.2.d     |
| construct                                               | 15 <sub>1</sub> .1.3.2.e     |
| construct in a representation language                  | 24 <sub>9</sub> .3.b         |
| contain (by a hypernotation)                            | 15 <sub>1</sub> .1.4.1.b     |
| contain (by a production tree)                          | 15 <sub>1</sub> .1.3.2.g     |
| contain (by a protonotation)                            | 15 <sub>1</sub> .1.4.1.b     |
| {control (a string by a pattern)}                       | 25 <sub>10</sub> .3.4.1.1.dd |
| {conversion key}                                        | 25 <sub>10</sub> .3.1.2.b    |
| {create (a file on a channel)}                          | 25 <sub>10</sub> .3.1.4.cc   |
| {cross-reference (in the syntax)}                       | 15 <sub>1</sub> .1.3.4.f     |
| {data list}                                             | 25 <sub>10</sub> .3.3.b      |
| defining <b>range (of an indicator)</b>                 | 22 <sub>7</sub> .2.2.a       |
| deflex (a mode to a mode)                               | 16 <sub>2</sub> .1.3.6.b     |
| {deproceduring}                                         | 21 <sub>6</sub>              |
| {dereferencing}                                         | 21 <sub>6</sub>              |
| descendent                                              | 15 <sub>1</sub> .1.3.2.g     |
| descendent action                                       | 16 <sub>2</sub> .1.4.2.b     |
| descriptor                                              | 16 <sub>2</sub> .1.3.4.b     |
| designate (a hypernotation designating a protonotation) | 15 <sub>1</sub> .1.4.1.a     |
| designate (a paranotation designating a construct)      | 15 <sub>1</sub> .1.4.2.a     |

|                                                    |                              |
|----------------------------------------------------|------------------------------|
| develop (a scene from a <b>declarer</b> )          | 18 <sub>4</sub> .6.2.c       |
| direct action                                      | 16 <sub>2</sub> .1.4.2.a     |
| direct descendent                                  | 15 <sub>1</sub> .1.3.2.f     |
| direct parent                                      | 16 <sub>2</sub> .1.4.2.c     |
| divided by (of arithmetic values)                  | 16 <sub>2</sub> .1.3.1.e     |
| {dynamic ( <b>replicator</b> )}                    | 25 <sub>10</sub> .3.4.1.1.dd |
| {edit (a string)}                                  | 25 <sub>10</sub> .3.4.1.1.jj |
| elaborate collaterally                             | 16 <sub>2</sub> .1.4.2.f     |
| elaboration                                        | 16 <sub>2</sub> .1.4.1.a     |
| element (of a multiple value)                      | 16 <sub>2</sub> .1.3.4.a     |
| elidable hypernotation                             | 15 <sub>1</sub> .1.4.2.c     |
| endow with subnames                                | 16 <sub>2</sub> .1.3.4.g     |
| envelop (a protonotion enveloping a hypernotation) | 15 <sub>1</sub> .1.4.1.c     |
| environ                                            | 16 <sub>2</sub> .1.1.1.c     |
| {environment enquiry}                              | 25 <sub>10</sub> .2          |
| equivalence (of a character and an integer)        | 16 <sub>2</sub> .1.3.1.g     |
| equivalence (of modes)                             | 16 <sub>2</sub> .1.1.2.a     |
| equivalence (of protonotions)                      | 16 <sub>2</sub> .1.1.2.a     |
| establish (an environ around an environ)           | 17 <sub>3</sub> .2.2.b       |
| {establish (a file on a channel)}                  | 25 <sub>10</sub> .3.1.4.cc   |
| {event routine}                                    | 25 <sub>10</sub> .3.1.3.b    |
| {expect}                                           | 25 <sub>10</sub> .3.4.1.1.ll |
| {external object}                                  | 16 <sub>2</sub> .1.1.b       |
| field                                              | 16 <sub>2</sub> .1.3.3.a     |
| {file}                                             | 25 <sub>10</sub> .3.1.3.b    |
| {firm (position)}                                  | 21 <sub>6</sub> .1.1.b       |
| {firmly related}                                   | 22 <sub>7</sub> .1.1.b       |
| fixed name (referring to a multiple value)         | 16 <sub>2</sub> .1.3.4.f     |
| flat descriptor                                    | 16 <sub>2</sub> .1.3.4.c     |
| flexible name (referring to a multiple value)      | 16 <sub>2</sub> .1.3.4.f     |
| follow (in the textual order)                      | 15 <sub>1</sub> .1.3.2.j     |
| {format}                                           | 25 <sub>10</sub> .3.4.b      |
| {frame}                                            | 25 <sub>10</sub> .3.5.1.bb   |
| generate (a <b>'TAG'</b> generating a name)        | 16 <sub>2</sub> .1.3.4.l     |
| generate (a trim generating a name)                | 16 <sub>2</sub> .1.3.4.j     |
| ghost element                                      | 16 <sub>2</sub> .1.3.4.c     |
| halt (an action)                                   | 16 <sub>2</sub> .1.4.3.f     |
| hardware language                                  | 24 <sub>9</sub> .3.a         |
| {heap}                                             | 19 <sub>5</sub> .2.3.b       |
| hold (of a predicate)                              | 15 <sub>1</sub> .3.2.b       |

## ALGOL 68 REVISED REPORT

|                                                          |                                                  |
|----------------------------------------------------------|--------------------------------------------------|
| hold (of a relationship)                                 | 16 <sub>2</sub> .1.2.a                           |
| hyper-rule                                               | 15 <sub>1</sub> .1.3.4.b                         |
| hyperalternative                                         | 15 <sub>1</sub> .1.3.4.c                         |
| hypernotation                                            | 15 <sub>1</sub> .1.3.1.e                         |
| hyphen                                                   | 15 <sub>1</sub> .1.3.1.a                         |
| identify (an <b>indicator identifying an indicator</b> ) | 22 <sub>7</sub> .2.2.b                           |
| implementation (of Algol 68)                             | 16 <sub>2</sub> .2.2.c                           |
| implementation of the reference language                 | 24 <sub>9</sub> .3.c                             |
| in (a construct in an environ)                           | 16 <sub>2</sub> .1.5.b                           |
| in place of                                              | 19 <sub>5</sub> .4.4.2.b                         |
| inactive (action)                                        | 16 <sub>2</sub> .1.4.3.a                         |
| incompatible actions                                     | 16 <sub>2</sub> .1.4.2.e                         |
| {independence (of properties)}                           | 22 <sub>7</sub> .1.1.b                           |
| index (to select an element)                             | 16 <sub>2</sub> .1.3.4.a                         |
| {indit (a string)}                                       | 25 <sub>10</sub> .3.4.1.1.kk                     |
| initiate (an action)                                     | 16 <sub>2</sub> .1.4.3.b, c                      |
| {input compatible}                                       | 25 <sub>10</sub> .3.4.1.1.ii                     |
| inseparable action                                       | 16 <sub>2</sub> .1.4.2.a                         |
| {insertion}                                              | 25 <sub>10</sub> .3.4.1.1.ee                     |
| integer                                                  | 16 <sub>2</sub> .1.3.1.a                         |
| integral equivalent (of a character)                     | 16 <sub>2</sub> .1.3.1.g                         |
| {internal object}                                        | 16 <sub>2</sub> .1.1.b                           |
| interrupt (an action)                                    | 16 <sub>2</sub> .1.4.3.h                         |
| intrinsic value                                          | 23 <sub>8</sub> .1.1.2, 23 <sub>8</sub> .2.2.b,c |
| invisible                                                | 15 <sub>1</sub> .1.3.2.h                         |
| is (of hypernotations)                                   | 16 <sub>2</sub> .1.5.e                           |
| large syntactic mark                                     | 15 <sub>1</sub> .1.3.1.a                         |
| largest integral equivalent (of a character)             | 16 <sub>2</sub> .1.3.1.g                         |
| lengthening (of arithmetic values)                       | 16 <sub>2</sub> .1.3.1.e                         |
| {link (a book with a file)}                              | 25 <sub>10</sub> .3.1.4.bb                       |
| {literal}                                                | 25 <sub>10</sub> .3.4.1.1.ee                     |
| local environ                                            | 19 <sub>5</sub> .2.3.2.b                         |
| locale                                                   | 16 <sub>2</sub> .1.1.1.b                         |
| {lock (a file)}                                          | 25 <sub>10</sub> .3.1.4.gg                       |
| {logical end}                                            | 25 <sub>10</sub> .3.1.1.aa                       |
| {logical file}                                           | 25 <sub>10</sub> .3.1.5.dd                       |
| lower bound                                              | 16 <sub>2</sub> .1.3.4.b                         |
| make to access (a value inside a locale)                 | 16 <sub>2</sub> .1.2.c                           |
| make to refer to (of a name)                             | 16 <sub>2</sub> .1.3.2.a                         |
| {marker}                                                 | 25 <sub>10</sub> .3.4.1.1.cc                     |

|                                        |                                                |
|----------------------------------------|------------------------------------------------|
| meaning                                | 15 <sub>1</sub> .1.4, 16 <sub>2</sub> .1.4.1.a |
| meaningful program                     | 15 <sub>1</sub> .1.4.3.c                       |
| {meek (position)}                      | 21 <sub>6</sub> .1.1.b                         |
| member                                 | 15 <sub>1</sub> .1.3.2.d                       |
| metanotion                             | 15 <sub>1</sub> .1.3.1.d                       |
| metaproduction rule                    | 15 <sub>1</sub> .1.3.3.b                       |
| minus (of arithmetic values)           | 16 <sub>2</sub> .1.3.1.e                       |
| mode                                   | 16 <sub>2</sub> .1.5.f                         |
| {multiple selection}                   | 19 <sub>5</sub> .3.1.b                         |
| multiple value                         | 16 <sub>2</sub> .1.3.4.a                       |
| name                                   | 16 <sub>2</sub> .1.3.2.a                       |
| necessary for (an environ for a scene) | 22 <sub>7</sub> .2.2.c                         |
| nest                                   | 17 <sub>3</sub> .0.2.b                         |
| newer (of scopes)                      | 16 <sub>2</sub> .1.2.f                         |
| newly created (name)                   | 16 <sub>2</sub> .1.3.2.a                       |
| nil                                    | 16 <sub>2</sub> .1.3.2.a                       |
| nonlocal                               | 17 <sub>3</sub> .2.2.b                         |
| notion                                 | 15 <sub>1</sub> .1.3.1.c                       |
| number of extra lengths                | 16 <sub>2</sub> .1.3.1.d                       |
| number of extra shorths                | 16 <sub>2</sub> .1.3.1.d                       |
| number of extra widths                 | 25 <sub>10</sub> .2.1.j, l                     |
| numerical analysis, in the sense of    | 16 <sub>2</sub> .1.3.1.e                       |
| object                                 | 16 <sub>2</sub> .1.1.b                         |
| of (construct of a construct)          | 16 <sub>2</sub> .1.5.a                         |
| of (construct of a scene)              | 16 <sub>2</sub> .1.1.1.d                       |
| of (environ of a scene)                | 16 <sub>2</sub> .1.1.1.d                       |
| of (nest of a construct)               | 17 <sub>3</sub> .0.2.b                         |
| older (of scopes)                      | 16 <sub>2</sub> .1.2.f                         |
| {on routine}                           | 25 <sub>10</sub> .3.1.3.b                      |
| {open (a file)}                        | 25 <sub>10</sub> .3.1.4.dd                     |
| original                               | 15 <sub>1</sub> .1.3.2.f                       |
| other syntactic mark                   | 15 <sub>1</sub> .1.3.1.a                       |
| {output compatible}                    | 25 <sub>10</sub> .3.4.1.1.hh                   |
| {overflow}                             | 16 <sub>2</sub> .1.4.3.h                       |
| {overload}                             | 18 <sub>4</sub> .5                             |
| parallel action                        | 25 <sub>10</sub> .2.4.b                        |
| paranotion                             | 15 <sub>1</sub> .1.4.2.a                       |
| {perform (an alignment)}               | 25 <sub>10</sub> .3.4.1.1.ff                   |
| {perform (an insertion)}               | 25 <sub>10</sub> .3.4.1.1.ee                   |
| {pattern}                              | 25 <sub>10</sub> .3.4.1.1.cc                   |

## ALGOL 68 REVISED REPORT

|                                                    |                              |
|----------------------------------------------------|------------------------------|
| permanent relationship                             | 16 <sub>2</sub> .1.2.a       |
| {physical file}                                    | 25 <sub>10</sub> .3.1.5.cc   |
| {picture}                                          | 25 <sub>10</sub> .3.4.1.1.cc |
| plain value                                        | 16 <sub>2</sub> .1.3.1.a     |
| point                                              | 15 <sub>1</sub> .1.3.1.a     |
| pragmatic remark                                   | 15 <sub>1</sub> .1.2.b       |
| {pre-elaboration}                                  | 16 <sub>2</sub> .1.4.1.c     |
| precede (in the textual order)                     | 15 <sub>1</sub> .1.3.2.j     |
| predicate                                          | 15 <sub>1</sub> .3.2.b       |
| primal environ                                     | 16 <sub>2</sub> .2.2.a       |
| process                                            | 25 <sub>10</sub> .2.4.b      |
| produce                                            | 15 <sub>1</sub> .1.3.2.f     |
| production rule                                    | 15 <sub>1</sub> .1.3.2.b     |
| production tree                                    | 15 <sub>1</sub> .1.3.2.f     |
| program in the strict language                     | 25 <sub>10</sub> .1.2.b      |
| {property}                                         | 17 <sub>3</sub> .0.2.b       |
| protonotion                                        | 15 <sub>1</sub> .1.3.1.b     |
| pseudo-comment                                     | 25 <sub>10</sub> .1.3.7      |
| publication language                               | 24 <sub>9</sub> .3.a         |
| {random access}                                    | 25 <sub>10</sub> .3.1.3.ff   |
| {reach}                                            | 17 <sub>3</sub> .0.2.b       |
| real number                                        | 16 <sub>2</sub> .1.3.1.a     |
| refer to                                           | 16 <sub>2</sub> .1.3.2.a     |
| reference language                                 | 24 <sub>9</sub> .3.a         |
| relationship                                       | 16 <sub>2</sub> .1.2.a       |
| {replicator}                                       | 25 <sub>10</sub> .3.4.1.1.dd |
| representation language                            | 24 <sub>9</sub> .3.a         |
| required                                           | 15 <sub>1</sub> .1.4.3.b     |
| resume (an action)                                 | 16 <sub>2</sub> .1.4.3.g     |
| routine                                            | 16 <sub>2</sub> .1.3.5.a     |
| {rowing}                                           | 21 <sub>6</sub>              |
| same as (of scopes)                                | 16 <sub>2</sub> .1.2.f       |
| scene                                              | 16 <sub>2</sub> .1.1.1.d     |
| scope (of a value)                                 | 16 <sub>2</sub> .1.1.3.a     |
| scope (of an environ)                              | 16 <sub>2</sub> .1.1.3.b     |
| {scratch (a file)}                                 | 25 <sub>10</sub> .3.1.4.hh   |
| select (a <b>"TAG"</b> selecting a field)          | 16 <sub>2</sub> .1.3.3.a     |
| select (a <b>"TAG"</b> selecting a multiple value) | 16 <sub>2</sub> .1.3.4.k     |
| select (a <b>"TAG"</b> selecting a surname)        | 16 <sub>2</sub> .1.3.3.e     |
| select (a <b>field-selector</b> selecting a field) | 16 <sub>2</sub> .1.5.g       |

|                                                     |                              |
|-----------------------------------------------------|------------------------------|
| select (an index selecting a subname)               | 16 <sub>2</sub> .1.3.4.g     |
| select (an index selecting an element)              | 16 <sub>2</sub> .1.3.4.a     |
| select (a trim selecting a multiple value)          | 16 <sub>2</sub> .1.3.4.i     |
| semantics                                           | 15 <sub>1</sub> .1.1.b       |
| semicolon                                           | 15 <sub>1</sub> .1.3.1.a     |
| sense of numerical analysis                         | 16 <sub>2</sub> .1.3.1.e     |
| {sequential access}                                 | 25 <sub>10</sub> .3.1.3.ff   |
| serial action                                       | 16 <sub>2</sub> .1.4.2.a     |
| simple substitute                                   | 15 <sub>1</sub> .1.3.3.d     |
| size (of an arithmetic value)                       | 16 <sub>2</sub> .1.3.1.b     |
| small syntactic mark                                | 15 <sub>1</sub> .1.3.1.a     |
| smaller (descendent smaller than a production tree) | 15 <sub>1</sub> .1.3.2.g     |
| smaller than (of arithmetic values)                 | 16 <sub>2</sub> .1.3.1.e     |
| {soft (position)}                                   | 21 <sub>6</sub> .1.1.b       |
| {sort}                                              | 21 <sub>6</sub>              |
| specify (a <b>declarer</b> specifying a mode)       | 18 <sub>4</sub> .6.2.d       |
| {spelling (of a mode)}                              | 16 <sub>2</sub> .1.1.2.b     |
| standard environment 10                             |                              |
| {standard function}                                 | 25 <sub>10</sub> .2          |
| {standard mode}                                     | 25 <sub>10</sub> .2          |
| {standard operator}                                 | 25 <sub>10</sub> .2          |
| {state}                                             | 25 <sub>10</sub> .3.1.3.b    |
| {staticize (a picture)}                             | 25 <sub>10</sub> .3.4.1.1.dd |
| stowed name                                         | 16 <sub>2</sub> .1.3.2.b     |
| stowed value                                        | 16 <sub>2</sub> .1.1.1.a     |
| straightening                                       | 25 <sub>10</sub> .3.2.3.c    |
| strict language                                     | 25 <sub>10</sub> .1.2.b      |
| {string}                                            | 23 <sub>8</sub> .3           |
| {strong (position)}                                 | 21 <sub>6</sub> .1.1.b       |
| structured value                                    | 16 <sub>2</sub> .1.3.3.a     |
| sublanguage                                         | 16 <sub>2</sub> .2.2.c       |
| subname                                             | 16 <sub>2</sub> .1.2.g       |
| substitute consistently                             | 15 <sub>1</sub> .1.3.4.e     |
| substitute simply                                   | 15 <sub>1</sub> .1.3.3.d     |
| superlanguage                                       | 16 <sub>2</sub> .2.2.c       |
| {suppressed frame}                                  | 25 <sub>10</sub> .3.4.1.1.cc |
| symbol                                              | 15 <sub>1</sub> .1.3.1.f     |
| {synchronization operation}                         | 25 <sub>10</sub> .2          |
| syntax                                              | 15 <sub>1</sub> .1.1         |
| terminal metaproduction (of a metanotion)           | 15 <sub>1</sub> .1.3.3.c     |

## ALGOL 68 REVISED REPORT

|                                            |                             |
|--------------------------------------------|-----------------------------|
| terminal production (of a notion)          | 15 <sub>1</sub> .1.3.2.f    |
| terminal production (of a production tree) | 15 <sub>1</sub> .1.3.2.f    |
| terminate (an action)                      | 16 <sub>2</sub> .1.4.3.e    |
| textual order                              | 15 <sub>1</sub> .1.3.2.i    |
| times (of arithmetic values)               | 16 <sub>2</sub> .1.3.1.e    |
| transform                                  | 25 <sub>10</sub> .3.4.1.2.b |
| {transient name}                           | 16 <sub>2</sub> .1.3.6.c    |
| transitive relationship                    | 16 <sub>2</sub> .1.2.a      |
| {transput declaration}                     | 25 <sub>10</sub> .2         |
| {transput}                                 | 25 <sub>10</sub> .3         |
| traverse                                   | 25 <sub>10</sub> .3.2.3.d   |
| trim                                       | 16 <sub>2</sub> .1.3.4.h    |
| truth value                                | 16 <sub>2</sub> .1.3.1.f    |
| typographical display feature              | 24 <sub>9</sub> .4.d        |
| undefined                                  | 15 <sub>1</sub> .1.4.3.a    |
| united from (of modes)                     | 16 <sub>2</sub> .1.3.6.a    |
| {uniting}                                  | 21 <sub>6</sub>             |
| upper bound                                | 16 <sub>2</sub> .1.3.4.b    |
| vacant locale                              | 16 <sub>2</sub> .1.1.1.b    |
| value                                      | 16 <sub>2</sub> .1.1.1.a    |
| variant (of a value)                       | 18 <sub>4</sub> .4.2.c      |
| variant of Algol 68                        | 15 <sub>1</sub> .1.5.b      |
| version (of an operator)                   | 25 <sub>10</sub> .1.3.3     |
| visible                                    | 15 <sub>1</sub> .1.3.2.h    |
| void value                                 | 16 <sub>2</sub> .1.3.1.h    |
| {voiding}                                  | 21 <sub>6</sub>             |
| {weak (position)}                          | 21 <sub>6</sub> .1.1.b      |
| {well formed}                              | 22 <sub>7</sub> .4          |
| widenable to (an integer to a real number) | 16 <sub>2</sub> .1.3.1.e    |
| {widening}                                 | 21 <sub>6</sub>             |
| yield (of a scene)                         | 16 <sub>2</sub> .1.2.b      |

{Denn eben, wo Begriffe fehlen,  
Da stellt ein Wort zur rechten Zeit sich ein.  
Faust, J.W. von Goethe. }

## 27.2 Paranotions

Given below are short paranotions representing the notions defined in this Report, with references to their hyper-rules.



|                                |                              |
|--------------------------------|------------------------------|
| <b>after-specification</b>     | 25 <sub>10</sub> .3.4.10.1.c |
| <b>alignment</b>               | 25 <sub>10</sub> .3.4.1.1.e  |
| <b>alignment-code</b>          | 25 <sub>10</sub> .3.4.1.1.f  |
| <b>alternate-CHOICE-clause</b> | 17 <sub>3</sub> .4.1.d       |
| <b>assignation</b>             | 19 <sub>5</sub> .2.1.1.a     |
| <b>bits-denotation</b>         | 23 <sub>8</sub> .2.1.l       |
| <b>bits-pattern</b>            | 25 <sub>10</sub> .3.4.7.1.a  |
| <b>boolean-choice-pattern</b>  | 25 <sub>10</sub> .3.4.8.1.b  |
| <b>boolean-marker</b>          | 25 <sub>10</sub> .3.4.4.1.b  |
| <b>boolean-pattern</b>         | 25 <sub>10</sub> .3.4.4.1.a  |
| <b>boundscript</b>             | 19 <sub>5</sub> .3.2.1.j     |
| <b>call</b>                    | 19 <sub>5</sub> .4.3.1.a     |
| <b>case-clause</b>             | 17 <sub>3</sub> .4.1.p       |
| <b>case-part-of-CHOICE</b>     | 17 <sub>3</sub> .4.1.i       |
| <b>cast</b>                    | 19 <sub>5</sub> .5.1.1.a     |
| <b>character-glyph</b>         | 23 <sub>8</sub> .1.4.1.c     |
| <b>character-marker</b>        | 25 <sub>10</sub> .3.4.6.1.b  |
| <b>choice-clause</b>           | 17 <sub>3</sub> .4.1.n       |
| <b>chooser-CHOICE-clause</b>   | 17 <sub>3</sub> .4.1.b       |
| <b>closed-clause</b>           | 17 <sub>3</sub> .1.1.a       |
| <b>coercee</b>                 | 21 <sub>6</sub> .1.1.g       |
| <b>coercend</b>                | 21 <sub>6</sub> .1.1.h       |
| <b>collateral-clause</b>       | 17 <sub>3</sub> .3.1.a, d, e |
| <b>collection</b>              | 25 <sub>10</sub> .3.4.1.1.b  |
| <b>complex-marker</b>          | 25 <sub>10</sub> .3.4.5.1.b  |
| <b>complex-pattern</b>         | 25 <sub>10</sub> .3.4.5.1.a  |
| <b>conditional-clause</b>      | 17 <sub>3</sub> .4.1.o       |
| <b>conformity-clause</b>       | 17 <sub>3</sub> .4.1.q       |
| <b>constant</b>                | 17 <sub>3</sub> .0.1.d       |

|                               |                                       |
|-------------------------------|---------------------------------------|
| <b>declaration</b>            | 18 <sub>4</sub> .1.1.a                |
| <b>declarative</b>            | 19 <sub>5</sub> .4.1.1.e              |
| <b>declarator</b>             | 18 <sub>4</sub> .6.1.c, d, g, h, o, s |
| <b>declarer</b>               | 18 <sub>4</sub> .6.1.a, b             |
| <b>definition</b>             | 18 <sub>4</sub> .1.1.d                |
| <b>denotation</b>             | 23 <sub>8</sub> .3.1.a                |
| <b>denoter</b>                | 23 <sub>8</sub> .0.1.a                |
| <b>deprocedured-to-FORM</b>   | 21 <sub>6</sub> .3.1.a                |
| <b>dereferenced-to-FORM</b>   | 21 <sub>6</sub> .2.1.a                |
| <b>destination</b>            | 19 <sub>5</sub> .2.1.1.b              |
| <b>digit-cypher</b>           | 23 <sub>8</sub> .1.1.1.c              |
| <b>digit-marker</b>           | 25 <sub>10</sub> .3.4.2.1.f           |
| <b>display</b>                | 17 <sub>3</sub> .3.1.j                |
| <b>do-part</b>                | 17 <sub>3</sub> .5.1.h                |
| <b>dyadic-operator</b>        | 19 <sub>5</sub> .4.2.1.e              |
| <b>enquiry-clause</b>         | 17 <sub>3</sub> .4.1.c                |
| <b>establishing-clause</b>    | 17 <sub>3</sub> .2.1.i                |
| <b>exponent-marker</b>        | 25 <sub>10</sub> .3.4.3.1.e           |
| <b>exponent-part</b>          | 23 <sub>8</sub> .1.2.1.g              |
| <b>exponent-specification</b> | 25 <sub>10</sub> .3.4.10.1.d          |
| <b>expression</b>             | 17 <sub>3</sub> .0.1.b                |
| <b>field-selector</b>         | 18 <sub>4</sub> .8.1.f                |
| <b>fixed-point-numeral</b>    | 23 <sub>8</sub> .1.1.1.b              |
| <b>floating-point-mould</b>   | 25 <sub>10</sub> .3.4.3.1.c           |
| <b>floating-point-numeral</b> | 23 <sub>8</sub> .1.2.1.e              |
| <b>for-part</b>               | 17 <sub>3</sub> .5.1.b                |
| <b>format-pattern</b>         | 25 <sub>10</sub> .3.4.9.1.a           |
| <b>format-text</b>            | 25 <sub>10</sub> .3.4.1.1.a           |
| <b>formula</b>                | 19 <sub>5</sub> .4.2.1.d              |
| <b>fractional-part</b>        | 23 <sub>8</sub> .1.2.1.d              |
| <b>frame</b>                  | 25 <sub>10</sub> .3.4.1.1.m           |
| <b>general-pattern</b>        | 25 <sub>10</sub> .3.4.10.1.a          |
| <b>generator</b>              | 19 <sub>5</sub> .2.3.1.a              |
| <b>go-to</b>                  | 19 <sub>5</sub> .4.4.1.b              |
| <b>hip</b>                    | 19 <sub>5</sub> .1.a                  |
| <b>identifier-declaration</b> | 18 <sub>4</sub> .4.1.g                |
| <b>identity-declaration</b>   | 18 <sub>4</sub> .4.1.a                |
| <b>identity-definition</b>    | 18 <sub>4</sub> .4.1.c                |
| <b>identity-relation</b>      | 19 <sub>5</sub> .2.2.1.a              |
| <b>identity-relator</b>       | 19 <sub>5</sub> .2.2.1.b              |
| <b>in-part-of-CHOICE</b>      | 17 <sub>3</sub> .4.1.f, g, h          |

|                                |                             |
|--------------------------------|-----------------------------|
| <b>in-CHOICE-clause</b>        | 17 <sub>3</sub> .4.1.e      |
| <b>indexer</b>                 | 19 <sub>5</sub> .3.2.1.i    |
| <b>indicator</b>               | 18 <sub>4</sub> .8.1.e      |
| <b>insertion</b>               | 25 <sub>10</sub> .3.4.1.1.d |
| <b>integral-choice-pattern</b> | 25 <sub>10</sub> .3.4.8.1.a |
| <b>integral-mould</b>          | 25 <sub>10</sub> .3.4.2.1.b |
| <b>integral-part</b>           | 23 <sub>8</sub> .1.2.1.c    |
| <b>integral-pattern</b>        | 25 <sub>10</sub> .3.4.2.1.a |
| <b>intervals</b>               | 17 <sub>3</sub> .5.1.c      |
| <b>joined-label-definition</b> | 25 <sub>10</sub> .1.1.h     |
| <b>joined-portrait</b>         | 17 <sub>3</sub> .3.1.b      |
| <b>jump</b>                    | 19 <sub>5</sub> .4.4.1.a    |
| <b>label-definition</b>        | 17 <sub>3</sub> .2.1.c      |
| <b>literal</b>                 | 25 <sub>10</sub> .3.4.1.1.i |
| <b>loop-clause</b>             | 17 <sub>3</sub> .5.1.a      |
| <b>lower-bound</b>             | 18 <sub>4</sub> .6.1.m      |
| <b>marker</b>                  | 25 <sub>10</sub> .3.4.1.1.n |
| <b>mode-declaration</b>        | 18 <sub>4</sub> .2.1.a      |
| <b>mode-definition</b>         | 18 <sub>4</sub> .2.1.b      |
| <b>monadic-operator</b>        | 19 <sub>5</sub> .4.2.1.f    |
| <b>nihil</b>                   | 19 <sub>5</sub> .2.4.1.a    |
| <b>operand</b>                 | 19 <sub>5</sub> .4.2.1.g    |
| <b>operation-declaration</b>   | 18 <sub>4</sub> .5.1.a      |
| <b>operation-definition</b>    | 18 <sub>4</sub> .5.1.c      |
| <b>other-string-item</b>       | 23 <sub>8</sub> .1.4.1.d    |
| <b>other-PRAGMENT-item</b>     | 24 <sub>9</sub> .2.1.d      |
| <b>parallel-clause</b>         | 17 <sub>3</sub> .3.1.c      |
| <b>parameter</b>               | 19 <sub>5</sub> .4.3.1.c    |
| <b>parameter-definition</b>    | 19 <sub>5</sub> .4.1.1.f    |
| <b>particular-postlude</b>     | 25 <sub>10</sub> .1.1.i     |
| <b>particular-program</b>      | 25 <sub>10</sub> .1.1.g     |
| <b>pattern</b>                 | 25 <sub>10</sub> .3.4.1.1.o |
| <b>phrase</b>                  | 17 <sub>3</sub> .0.1.a      |
| <b>picture</b>                 | 25 <sub>10</sub> .3.4.1.1.c |
| <b>plain-denotation</b>        | 23 <sub>8</sub> .1.0.1.b    |
| <b>plan</b>                    | 18 <sub>4</sub> .6.1.p      |
| <b>plusminus</b>               | 23 <sub>8</sub> .1.2.1.j    |
| <b>point-marker</b>            | 25 <sub>10</sub> .3.4.3.1.d |
| <b>power-of-ten</b>            | 23 <sub>8</sub> .1.2.1.i    |
| <b>praglit</b>                 | 25 <sub>10</sub> .3.4.8.1.c |

|                                    |                              |
|------------------------------------|------------------------------|
| <b>pragma</b>                      | 24 <sub>9</sub> .2.1.a       |
| <b>preludes</b>                    | 25 <sub>10</sub> .1.1.b      |
| <b>priority-declaration</b>        | 18 <sub>4</sub> .3.1.a       |
| <b>priority-definition</b>         | 18 <sub>4</sub> .3.1.b       |
| <b>program</b>                     | 16 <sub>2</sub> .2.1.a       |
| <b>program-text</b>                | 25 <sub>10</sub> .1.1.a      |
| <b>radix-digit</b>                 | 23 <sub>8</sub> .2.1.m       |
| <b>radix-marker</b>                | 25 <sub>10</sub> .3.4.7.1.c  |
| <b>range</b>                       | 17 <sub>3</sub> .0.1.f       |
| <b>real-pattern</b>                | 25 <sub>10</sub> .3.4.3.1.a  |
| <b>repeating-part</b>              | 17 <sub>3</sub> .5.1.e       |
| <b>replicator</b>                  | 25 <sub>10</sub> .3.4.1.1.g  |
| <b>revised-lower-bound</b>         | 19 <sub>5</sub> .3.2.1.g     |
| <b>routine-declarer</b>            | 18 <sub>4</sub> .4.1.b       |
| <b>routine-plan</b>                | 18 <sub>4</sub> .5.1.b       |
| <b>routine-text</b>                | 19 <sub>5</sub> .4.1.1.a, b  |
| <b>row-display</b>                 | 17 <sub>3</sub> .3.1.i       |
| <b>row-rouer</b>                   | 18 <sub>4</sub> .6.1.j, k, l |
| <b>row-ROWS-rouer</b>              | 18 <sub>4</sub> .6.1.i       |
| <b>rowed-to-FORM</b>               | 21 <sub>6</sub> .6.1.a       |
| <b>sample-generator</b>            | 19 <sub>5</sub> .2.3.1.b     |
| <b>selection</b>                   | 19 <sub>5</sub> .3.1.1.a     |
| <b>serial-clause</b>               | 17 <sub>3</sub> .2.1.a       |
| <b>series</b>                      | 17 <sub>3</sub> .2.1.b       |
| <b>sign-marker</b>                 | 25 <sub>10</sub> .3.4.2.1.e  |
| <b>sign-mould</b>                  | 25 <sub>10</sub> .3.4.2.1.c  |
| <b>skip</b>                        | 19 <sub>5</sub> .5.2.1.a     |
| <b>slice</b>                       | 19 <sub>5</sub> .3.2.1.a     |
| <b>softly-deprocedured-to-FORM</b> | 21 <sub>6</sub> .3.1.b       |
| <b>source</b>                      | 19 <sub>5</sub> .2.1.1.c     |
| <b>source-for-MODINE</b>           | 18 <sub>4</sub> .4.1.d       |
| <b>specification</b>               | 17 <sub>3</sub> .4.1.j, k    |
| <b>stagnant-part</b>               | 23 <sub>8</sub> .1.2.1.f     |
| <b>statement</b>                   | 17 <sub>3</sub> .0.1.c       |
| <b>string</b>                      | 23 <sub>8</sub> .3.1.b       |
| <b>string-denotation</b>           | 23 <sub>8</sub> .3.1.c       |
| <b>string-item</b>                 | 23 <sub>8</sub> .1.4.1.b     |
| <b>string-pattern</b>              | 25 <sub>10</sub> .3.4.6.1.a  |
| <b>structure-display</b>           | 17 <sub>3</sub> .3.1.h       |
| <b>subscript</b>                   | 19 <sub>5</sub> .3.2.1.e     |

|                                      |                                 |
|--------------------------------------|---------------------------------|
| <b>suppression</b>                   | 25 <sub>10</sub> .3.4.1.1.l     |
| <b>symbol</b>                        | 24 <sub>9</sub> .1.1.h          |
| <b>system-task</b>                   | 25 <sub>10</sub> .1.1.e         |
| <b>tasks</b>                         | 25 <sub>10</sub> .1.1.d         |
| <b>times-ten-to-the-power-choice</b> | 23 <sub>8</sub> .1.2.1.h        |
| <b>token</b>                         | 24 <sub>9</sub> .1.1.g          |
| <b>trimmer</b>                       | 19 <sub>5</sub> .3.2.1.f        |
| <b>trimscript</b>                    | 19 <sub>5</sub> .3.2.1.h        |
| <b>unchanged-from-FORM</b>           | 21 <sub>6</sub> .1.1.f          |
| <b>unit</b>                          | 17 <sub>3</sub> .2.1.d          |
| <b>unitary-clause</b>                | 17 <sub>3</sub> .2.1.h          |
| <b>united-to-FORM</b>                | 21 <sub>6</sub> .4.1.a          |
| <b>unsuppressible-literal</b>        | 25 <sub>10</sub> .3.4.1.1.i     |
| <b>unsuppressible-replicator</b>     | 25 <sub>10</sub> .3.4.1.1.h     |
| <b>unsuppressible-suppression</b>    | 25 <sub>10</sub> .3.4.1.1.l     |
| <b>upper-bound</b>                   | 18 <sub>4</sub> .6.1.n          |
| <b>user-task</b>                     | 25 <sub>10</sub> .1.1.f         |
| <b>vacuum</b>                        | 17 <sub>3</sub> .3.1.k          |
| <b>variable</b>                      | 17 <sub>3</sub> .0.1.e          |
| <b>variable-declaration</b>          | 18 <sub>4</sub> .4.1.e          |
| <b>variable-definition</b>           | 18 <sub>4</sub> .4.1.f          |
| <b>variable-point-mould</b>          | 25 <sub>10</sub> .3.4.3.1.b     |
| <b>variable-point-numeral</b>        | 23 <sub>8</sub> .1.2.1.b        |
| <b>voided-to-FORM</b>                | 21 <sub>6</sub> .7.1.a, b       |
| <b>while-do-part</b>                 | 17 <sub>3</sub> .5.1.f          |
| <b>while-part</b>                    | 17 <sub>3</sub> .5.1.g          |
| <b>widened-to-FORM</b>               | 21 <sub>6</sub> .5.1.a, b, c, d |
| <b>width-specification</b>           | 25 <sub>10</sub> .3.4.10.1.b    |
| <b>zero-marker</b>                   | 25 <sub>10</sub> .3.4.2.1.d     |
| <b>ADIC-operand</b>                  | 19 <sub>5</sub> .4.2.1.c        |
| <b>CHOICE-again</b>                  | 24 <sub>9</sub> .1.1.c          |
| <b>CHOICE-finish</b>                 | 24 <sub>9</sub> .1.1.e          |
| <b>CHOICE-in</b>                     | 24 <sub>9</sub> .1.1.b          |
| <b>CHOICE-out</b>                    | 24 <sub>9</sub> .1.1.d          |
| <b>CHOICE-start</b>                  | 24 <sub>9</sub> .1.1.a          |
| <b>CHOICE-clause</b>                 | 17 <sub>3</sub> .4.1.a          |
| <b>COMMON-joined-definition</b>      | 18 <sub>4</sub> .1.1.b, c       |
| <b>DYADIC-formula</b>                | 19 <sub>5</sub> .4.2.1.a        |
| <b>EXTERNAL-prelude</b>              | 25 <sub>10</sub> .1.1.c         |
| <b>FIELDS-definition-of-FIELD</b>    | 18 <sub>4</sub> .6.1.f          |

|                                       |                                    |
|---------------------------------------|------------------------------------|
| <b>FIELDS-portrait</b>                | 17 <sub>3</sub> .3.1.f, g          |
| <b>FIELDS-portrayer-of-FIELDS1</b>    | 18 <sub>4</sub> .6.1.e             |
| <b>FORM-coercee</b>                   | 21 <sub>6</sub> .1.1.a, b, c, d, e |
| <b>FROBYT-part</b>                    | 17 <sub>3</sub> .5.1.d             |
| <b>INDICATOR</b>                      | 18 <sub>4</sub> .8.1.a, b          |
| <b>MOIDS-joined-declarer</b>          | 18 <sub>4</sub> .6.1.t, u          |
| <b>MONADIC-formula</b>                | 19 <sub>5</sub> .4.2.1.b           |
| <b>NOTETY-pack</b>                    | 15 <sub>1</sub> .3.3.d             |
| <b>NOTION-bracket</b>                 | 15 <sub>1</sub> .3.3.e             |
| <b>NOTION-list</b>                    | 15 <sub>1</sub> .3.3.c             |
| <b>NOTION-option</b>                  | 15 <sub>1</sub> .3.3.a             |
| <b>NOTION-sequence</b>                | 15 <sub>1</sub> .3.3.b             |
| <b>NOTION-token</b>                   | 24 <sub>9</sub> .1.1.f             |
| <b>PARAMETERS</b>                     | 19 <sub>5</sub> .4.3.1.b           |
| <b>PARAMETERS-joined-declarer</b>     | 18 <sub>4</sub> .6.1.q, r          |
| <b>PRAGMENT</b>                       | 24 <sub>9</sub> .2.1.b             |
| <b>PRAGMENT-item</b>                  | 24 <sub>9</sub> .2.1.c             |
| <b>QUALITY-FIELDS-field-selector</b>  | 18 <sub>4</sub> .8.1.c, d          |
| <b>RADIX</b>                          | 23 <sub>8</sub> .2.1.d, e, f, g    |
| <b>RADIX-digit</b>                    | 23 <sub>8</sub> .2.1.h, i, j, k    |
| <b>RADIX-frame</b>                    | 25 <sub>10</sub> .3.4.7.1.b        |
| <b>ROWS-leaving-ROWSETY-indexer</b>   | 19 <sub>5</sub> .3.2.1.b, c, d     |
| <b>TALLY-declarer</b>                 | 18 <sub>4</sub> .2.1.c             |
| <b>THING1-or-alternatively-THING2</b> | 15 <sub>1</sub> .3.3.f             |
| <b>UNSUPPRESSETY-literal</b>          | 25 <sub>10</sub> .3.4.1.1.i        |
| <b>UNSUPPRESSETY-suppression</b>      | 25 <sub>10</sub> .3.4.1.1.l        |
| <b>UNSUPPRESSETY-COMARK-frame</b>     | 25 <sub>10</sub> .3.4.1.1.k        |
| <b>UNSUPPRESSETY-MARK-frame</b>       | 25 <sub>10</sub> .3.4.1.1.j        |

## 27.3 Predicates

Given below are abbreviated forms of the predicates defined in this Report.

|                    |                                                      |
|--------------------|------------------------------------------------------|
| 'and'              | 15 <sub>1</sub> .3.1.c, e                            |
| 'balances'         | 17 <sub>3</sub> .2.1.f, g                            |
| 'begins with'      | 15 <sub>1</sub> .3.1.h, i, j                         |
| 'coincides with'   | 15 <sub>1</sub> .3.1.k, l                            |
| 'contains'         | 15 <sub>1</sub> .3.1.m, n                            |
| 'counts'           | 18 <sub>4</sub> .3.1.c, d                            |
| 'deflexes to'      | 18 <sub>4</sub> .7.1.a, b, c, d, e                   |
| 'deprefts to firm' | 22 <sub>7</sub> .1.1.n                               |
| 'develops from'    | 22 <sub>7</sub> .3.1.c                               |
| 'equivalent'       | 22 <sub>7</sub> .3.1.a, b, d, e, f, g, h, i, j, k, q |
| 'false'            | 15 <sub>1</sub> .3.1.b                               |
| 'firmly related'   | 22 <sub>7</sub> .1.1.k                               |
| 'identified in'    | 22 <sub>7</sub> .2.1.a                               |
| 'incestuous'       | 18 <sub>4</sub> .7.1.f                               |
| 'independent'      | 22 <sub>7</sub> .1.1.a, b, c, d                      |
| 'is'               | 15 <sub>1</sub> .3.1.g                               |
| 'is derived from'  | 19 <sub>5</sub> .3.1.1.b, c                          |
| 'is firm'          | 22 <sub>7</sub> .1.1.l, m                            |
| 'like'             | 19 <sub>5</sub> .4.1.1.c, d                          |
| 'may follow'       | 17 <sub>3</sub> .4.1.m                               |
| 'number equals'    | 22 <sub>7</sub> .3.1.o, p                            |
| 'or'               | 15 <sub>1</sub> .3.1.d, f                            |
| 'ravel to'         | 18 <sub>4</sub> .7.1.g                               |
| 'related'          | 22 <sub>7</sub> .1.1.e, f, g, h, i, j                |
| 'resides in'       | 22 <sub>7</sub> .2.1.b, c                            |
| 'shields'          | 22 <sub>7</sub> .4.1.a, b, c, d                      |
| 'subset of'        | 22 <sub>7</sub> .3.1.l, m, n                         |
| 'true'             | 15 <sub>1</sub> .3.1.a                               |
| 'unites to'        | 21 <sub>6</sub> .4.1.b                               |

## 27.4 Index to the standard prelude

|       |                                                                                                                                                                                                                                                                                                                                            |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.a, 25 <sub>10</sub> .2.3.5.c, 25 <sub>10</sub> .2.3.5.c, d,<br>25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.a, g, h                                                                                                                                   |
| <=    | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.b, 25 <sub>10</sub> .2.3.4.b, 25 <sub>10</sub> .2.3.5.c, d,<br>25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.8.e, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.b, g, h                                                                                                        |
| +     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.i, 25 <sub>10</sub> .2.3.4.i, j, 25 <sub>10</sub> .2.3.5.a, b,<br>25 <sub>10</sub> .2.3.6.b, 25 <sub>10</sub> .2.3.7.j, k, p, q, r, s, 25 <sub>10</sub> .2.3.10.i, j, k                                                                                                                 |
| + :=  | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.d, e, f, o, p, q, s                                                                                                                                                                                                                                                                    |
| +=:   | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.r, t                                                                                                                                                                                                                                                                                   |
| +*    | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.u, 25 <sub>10</sub> .2.3.4.s, 25 <sub>10</sub> .2.3.5.e, f                                                                                                                                                                                                                              |
| &     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.2.b, 25 <sub>10</sub> .2.3.8.d                                                                                                                                                                                                                                                            |
| ^     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.2.b, 25 <sub>10</sub> .2.3.8.d                                                                                                                                                                                                                                                            |
| □     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.9.b                                                                                                                                                                                                                                                                                       |
| ┌     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.1.c, e                                                                                                                                                                                                                                                                                    |
| ↓     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.4.d                                                                                                                                                                                                                                                                                         |
| └     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.1.b, d                                                                                                                                                                                                                                                                                    |
| ≥     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.e, 25 <sub>10</sub> .2.3.4.e, 25 <sub>10</sub> .2.3.5.c, d,<br>25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.8.f, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.e, g, h                                                                                                        |
| ≤     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.b, 25 <sub>10</sub> .2.3.4.b, 25 <sub>10</sub> .2.3.5.c, d,<br>25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.8.e, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.b, g, h                                                                                                        |
| ≠     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.2.e, 25 <sub>10</sub> .2.3.3.d, 25 <sub>10</sub> .2.3.4.d,<br>25 <sub>10</sub> .2.3.5.c, d, 25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.7.g, u, v, w, x,<br>25 <sub>10</sub> .2.3.8.b, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.d, g, h, 25 <sub>10</sub> .2.3.10.d, g, h |
| ∨     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.2.a, 25 <sub>10</sub> .2.3.8.c                                                                                                                                                                                                                                                            |
| ⊥     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.u, 25 <sub>10</sub> .2.3.4.s, 25 <sub>10</sub> .2.3.5.e, f                                                                                                                                                                                                                              |
| ÷     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.m                                                                                                                                                                                                                                                                                       |
| ÷×    | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.n                                                                                                                                                                                                                                                                                       |
| ÷× := | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.k                                                                                                                                                                                                                                                                                      |
| ÷*    | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.n                                                                                                                                                                                                                                                                                       |
| ÷* := | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.k                                                                                                                                                                                                                                                                                      |
| ÷ :=  | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.j                                                                                                                                                                                                                                                                                      |
| ×     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.l, 25 <sub>10</sub> .2.3.4.l, 25 <sub>10</sub> .2.3.5.a, b,<br>25 <sub>10</sub> .2.3.7.l, p, q, r, s, 25 <sub>10</sub> .2.3.10.l, m, n, o                                                                                                                                               |
| × :=  | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.g, h, i, n, o, p, u                                                                                                                                                                                                                                                                    |
| ~     | 25 <sub>10</sub> .2.3.2.c, 25 <sub>10</sub> .2.3.8.m                                                                                                                                                                                                                                                                                       |
| ↑     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.p, 25 <sub>10</sub> .2.3.5.g, 25 <sub>10</sub> .2.3.7.t, 25 <sub>10</sub> .2.3.7.t                                                                                                                                                                                                      |
| *     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.l, 25 <sub>10</sub> .2.3.4.l, 25 <sub>10</sub> .2.3.5.a, b,<br>25 <sub>10</sub> .2.3.7.l, p, q, r, s, 25 <sub>10</sub> .2.3.10.l, m, n, o                                                                                                                                               |
| **    | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.p, 25 <sub>10</sub> .2.3.5.g, 25 <sub>10</sub> .2.3.7.t, 25 <sub>10</sub> .2.3.7.t                                                                                                                                                                                                      |
| * :=  | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.g, h, i, n, o, p, u                                                                                                                                                                                                                                                                    |



|         |                                                                                                                                                                                                                                                                                                                                            |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ¬       | 25 <sub>10</sub> .2.3.2.c, 25 <sub>10</sub> .2.3.8.m                                                                                                                                                                                                                                                                                       |
| -       | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.g, h, 25 <sub>10</sub> .2.3.4.g, h, 25 <sub>10</sub> .2.3.5.a, b,<br>25 <sub>10</sub> .2.3.7.h, i, p, q, r, s                                                                                                                                                                           |
| - :=    | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.a, b, c, n, o, p                                                                                                                                                                                                                                                                       |
| /       | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.o, 25 <sub>10</sub> .2.3.4.m, 25 <sub>10</sub> .2.3.5.a, b,<br>25 <sub>10</sub> .2.3.7.m, p, q, r, s                                                                                                                                                                                    |
| / :=    | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.l, m, n, o, p                                                                                                                                                                                                                                                                          |
| / =     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.2.e, 25 <sub>10</sub> .2.3.3.d, 25 <sub>10</sub> .2.3.4.d,<br>25 <sub>10</sub> .2.3.5.c, d, 25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.7.g, u, v, w, x,<br>25 <sub>10</sub> .2.3.8.b, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.d, g, h, 25 <sub>10</sub> .2.3.10.d, g, h |
| %       | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.m                                                                                                                                                                                                                                                                                       |
| % ×     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.n                                                                                                                                                                                                                                                                                       |
| % × :=  | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.k                                                                                                                                                                                                                                                                                      |
| % *     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.n                                                                                                                                                                                                                                                                                       |
| % * :=  | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.k                                                                                                                                                                                                                                                                                      |
| % :=    | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.j                                                                                                                                                                                                                                                                                      |
| >       | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.f, 25 <sub>10</sub> .2.3.4.f, 25 <sub>10</sub> .2.3.5.c, d,<br>25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.f, g, h                                                                                                                                   |
| > =     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.e, 25 <sub>10</sub> .2.3.4.e, 25 <sub>10</sub> .2.3.5.c, d,<br>25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.8.f, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.e, g, h                                                                                                        |
| =       | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.2.d, 25 <sub>10</sub> .2.3.3.c, 25 <sub>10</sub> .2.3.4.c,<br>25 <sub>10</sub> .2.3.5.c, d, 25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.7.f, u, v, w, x,<br>25 <sub>10</sub> .2.3.8.a, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.c, g, h                                   |
| ABS     | 25 <sub>10</sub> .2.1.n, 25 <sub>10</sub> .2.3.2.f, 25 <sub>10</sub> .2.3.3.k, 25 <sub>10</sub> .2.3.4.k,<br>25 <sub>10</sub> .2.3.7.c, 25 <sub>10</sub> .2.3.8.i                                                                                                                                                                          |
| AND     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.2.b, 25 <sub>10</sub> .2.3.8.d                                                                                                                                                                                                                                                            |
| ARG     | 25 <sub>10</sub> .2.3.7.d                                                                                                                                                                                                                                                                                                                  |
| BIN     | 25 <sub>10</sub> .2.3.8.j                                                                                                                                                                                                                                                                                                                  |
| BITS    | 25 <sub>10</sub> .2.2.g                                                                                                                                                                                                                                                                                                                    |
| BOOL    | 25 <sub>10</sub> .2.2.b                                                                                                                                                                                                                                                                                                                    |
| BYTES   | 25 <sub>10</sub> .2.2.h                                                                                                                                                                                                                                                                                                                    |
| CHANNEL | 25 <sub>10</sub> .3.1.2.a                                                                                                                                                                                                                                                                                                                  |
| CHAR    | 25 <sub>10</sub> .2.2.e                                                                                                                                                                                                                                                                                                                    |
| COMPL   | 25 <sub>10</sub> .2.2.f                                                                                                                                                                                                                                                                                                                    |
| CONJ    | 25 <sub>10</sub> .2.3.7.e                                                                                                                                                                                                                                                                                                                  |
| DIVAB   | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.l, m, n, o, p                                                                                                                                                                                                                                                                          |
| DOWN    | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.4.d                                                                                                                                                                                                                                                                                         |
| ELEM    | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.9.b                                                                                                                                                                                                                                                                                       |
| ENTIER  | 25 <sub>10</sub> .2.3.4.r                                                                                                                                                                                                                                                                                                                  |

## ALGOL 68 REVISED REPORT

|         |                                                                                                                                                                                                                                                                                                                                            |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| EQ      | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.2.d, 25 <sub>10</sub> .2.3.3.c, 25 <sub>10</sub> .2.3.4.c,<br>25 <sub>10</sub> .2.3.5.c, d, 25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.7.f, u, v, w, x,<br>25 <sub>10</sub> .2.3.8.a, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.c, g, h, 25 <sub>10</sub> .2.3.10.c, g, h |
| FILE    | 25 <sub>10</sub> .3.1.3.a                                                                                                                                                                                                                                                                                                                  |
| FORMAT  | 25 <sub>10</sub> .3.5.a                                                                                                                                                                                                                                                                                                                    |
| GE      | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.e, 25 <sub>10</sub> .2.3.4.e, d, 25 <sub>10</sub> .2.3.6.a,<br>25 <sub>10</sub> .2.3.8.f,<br>25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.e, g, h                                                                                                                                |
| GT      | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.f, 25 <sub>10</sub> .2.3.4.f, 25 <sub>10</sub> .2.3.5.c, d,<br>25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.f, g, h                                                                                                                                   |
| I       | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.u, 25 <sub>10</sub> .2.3.4.s, 25 <sub>10</sub> .2.3.5.e, f                                                                                                                                                                                                                              |
| IM      | 25 <sub>10</sub> .2.3.7.b                                                                                                                                                                                                                                                                                                                  |
| INT     | 25 <sub>10</sub> .2.2.c                                                                                                                                                                                                                                                                                                                    |
| LE      | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.b, 25 <sub>10</sub> .2.3.4.b, 25 <sub>10</sub> .2.3.5.c, d,<br>25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.8.e, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.b, g, h                                                                                                        |
| LENG    | 25 <sub>10</sub> .2.3.3.q, 25 <sub>10</sub> .2.3.4.n, 25 <sub>10</sub> .2.3.7.n, 25 <sub>10</sub> .2.3.8.n, 25 <sub>10</sub> .2.3.9.d                                                                                                                                                                                                      |
| LEVEL   | 25 <sub>10</sub> .2.4.b, c                                                                                                                                                                                                                                                                                                                 |
| LT      | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.a, 25 <sub>10</sub> .2.3.5.c, 25 <sub>10</sub> .2.3.5.c, d,<br>25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.a, g, h                                                                                                                                   |
| LWB     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.1.b, d                                                                                                                                                                                                                                                                                    |
| MINUSAB | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.a, b, c, n, o, p                                                                                                                                                                                                                                                                       |
| MOD     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.n                                                                                                                                                                                                                                                                                       |
| MODAB   | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.k                                                                                                                                                                                                                                                                                      |
| NE      | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.2.e, 25 <sub>10</sub> .2.3.3.d, 25 <sub>10</sub> .2.3.4.d,<br>25 <sub>10</sub> .2.3.5.c, d, 25 <sub>10</sub> .2.3.6.a, 25 <sub>10</sub> .2.3.7.g, u, v, w, x,<br>25 <sub>10</sub> .2.3.8.b, 25 <sub>10</sub> .2.3.9.a, 25 <sub>10</sub> .2.3.10.d, g, h, 25 <sub>10</sub> .2.3.10.d, g, h |
| NOT     | 25 <sub>10</sub> .2.3.2.c, 25 <sub>10</sub> .2.3.8.m                                                                                                                                                                                                                                                                                       |
| ODD     | 25 <sub>10</sub> .2.3.3.s                                                                                                                                                                                                                                                                                                                  |
| OR      | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.2.a, 25 <sub>10</sub> .2.3.8.c                                                                                                                                                                                                                                                            |
| OVER    | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.m                                                                                                                                                                                                                                                                                       |
| OVERAB  | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.j                                                                                                                                                                                                                                                                                      |
| PLUSAB  | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.d, e, f, n, o, p, q, s                                                                                                                                                                                                                                                                 |
| PLUSTO  | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.r, t                                                                                                                                                                                                                                                                                   |
| RE      | 25 <sub>10</sub> .2.3.7.a                                                                                                                                                                                                                                                                                                                  |
| REAL    | 25 <sub>10</sub> .2.2.d                                                                                                                                                                                                                                                                                                                    |
| REPR    | 25 <sub>10</sub> .2.1.o                                                                                                                                                                                                                                                                                                                    |
| ROUND   | 25 <sub>10</sub> .2.3.4.p                                                                                                                                                                                                                                                                                                                  |
| SEMA    | 25 <sub>10</sub> .2.4.a                                                                                                                                                                                                                                                                                                                    |
| SHL     | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.8.g                                                                                                                                                                                                                                                                                       |

|                |                                                                                                                                                                   |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SHORTEN        | 25 <sub>10</sub> .2.3.3.r, 25 <sub>10</sub> .2.3.4.o, 25 <sub>10</sub> .2.3.7.o, 25 <sub>10</sub> .2.3.8.o,<br>25 <sub>10</sub> .2.3.9.e                          |
| SHR            | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.8.h                                                                                                              |
| SIGN           | 25 <sub>10</sub> .2.3.4.q                                                                                                                                         |
| STRING         | 25 <sub>10</sub> .2.2.i                                                                                                                                           |
| TIMESAB        | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.11.g, h, i, n, o, p, u                                                                                           |
| UP             | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.3.p, 25 <sub>10</sub> .2.3.5.g, 25 <sub>10</sub> .2.3.7.t,<br>25 <sub>10</sub> .2.3.8.g, 25 <sub>10</sub> .2.4.e |
| UPB            | 25 <sub>10</sub> .2.3.0.a, 25 <sub>10</sub> .2.3.1.c, e                                                                                                           |
| VOID           | 25 <sub>10</sub> .2.2.a                                                                                                                                           |
| arccos         | 25 <sub>10</sub> .2.3.12.f                                                                                                                                        |
| arcsin         | 25 <sub>10</sub> .2.3.12.h                                                                                                                                        |
| arctan         | 25 <sub>10</sub> .2.3.12.j                                                                                                                                        |
| associate      | 25 <sub>10</sub> .3.1.4.e                                                                                                                                         |
| backspace      | 25 <sub>10</sub> .3.1.6.b                                                                                                                                         |
| bin possible   | 25 <sub>10</sub> .3.1.3.d                                                                                                                                         |
| bits lengths   | 25 <sub>10</sub> .2.1.h                                                                                                                                           |
| bits pack      | 25 <sub>10</sub> .2.3.8.l                                                                                                                                         |
| bits shorths   | 25 <sub>10</sub> .2.1.i                                                                                                                                           |
| bits width     | 25 <sub>10</sub> .2.1.j                                                                                                                                           |
| blank          | 25 <sub>10</sub> .2.1.u                                                                                                                                           |
| bytes lengths  | 25 <sub>10</sub> .2.1.k                                                                                                                                           |
| bytes pack     | 25 <sub>10</sub> .2.3.9.c                                                                                                                                         |
| bytes shorths  | 25 <sub>10</sub> .2.1.l                                                                                                                                           |
| bytes width    | 25 <sub>10</sub> .2.1.m                                                                                                                                           |
| chan           | 25 <sub>10</sub> .3.1.3.i                                                                                                                                         |
| char in string | 25 <sub>10</sub> .3.2.1.l                                                                                                                                         |
| char number    | 25 <sub>10</sub> .3.1.5.a                                                                                                                                         |
| close          | 25 <sub>10</sub> .3.1.4.n                                                                                                                                         |
| compressible   | 25 <sub>10</sub> .3.1.3.e                                                                                                                                         |
| cos            | 25 <sub>10</sub> .2.3.12.e                                                                                                                                        |
| create         | 25 <sub>10</sub> .3.1.4.c                                                                                                                                         |
| errorchar      | 25 <sub>10</sub> .2.1.t                                                                                                                                           |
| estab possible | 25 <sub>10</sub> .3.1.2.c                                                                                                                                         |
| establish      | 25 <sub>10</sub> .3.1.4.b                                                                                                                                         |
| exp            | 25 <sub>10</sub> .2.3.12.c                                                                                                                                        |
| exp width      | 25 <sub>10</sub> .3.2.1.o                                                                                                                                         |
| fixed          | 25 <sub>10</sub> .3.2.1.c                                                                                                                                         |
| flip           | 25 <sub>10</sub> .2.1.r                                                                                                                                           |
| float          | 25 <sub>10</sub> .3.2.1.d                                                                                                                                         |
| flop           | 25 <sub>10</sub> .2.1.s                                                                                                                                           |
| get            | 25 <sub>10</sub> .3.3.2.a                                                                                                                                         |

## ALGOL 68 REVISED REPORT

|                      |                            |
|----------------------|----------------------------|
| get bin              | 25 <sub>10</sub> .3.6.2.a  |
| get possible         | 25 <sub>10</sub> .3.1.3.b  |
| getf                 | 25 <sub>10</sub> .3.5.2.a  |
| int shorths          | 25 <sub>10</sub> .2.1.b    |
| int width            | 25 <sub>10</sub> .3.2.1.m  |
| last random          | 25 <sub>10</sub> .5.1.a    |
| line number          | 25 <sub>10</sub> .3.1.5.b  |
| ln                   | 25 <sub>10</sub> .2.3.12.d |
| lock                 | 25 <sub>10</sub> .3.1.4.o  |
| make conv            | 25 <sub>10</sub> .3.1.3.j  |
| make term            | 25 <sub>10</sub> .3.1.3.k  |
| max abs char         | 25 <sub>10</sub> .2.1.p    |
| max int              | 25 <sub>10</sub> .2.1.c    |
| max real             | 25 <sub>10</sub> .2.1.f    |
| newline              | 25 <sub>10</sub> .3.1.6.c  |
| newpage              | 25 <sub>10</sub> .3.1.6.d  |
| next random          | 25 <sub>10</sub> .2.3.12.k |
| null character       | 25 <sub>10</sub> .2.1.q    |
| on char error        | 25 <sub>10</sub> .3.1.3.r  |
| on format end        | 25 <sub>10</sub> .3.1.3.p  |
| on line end          | 25 <sub>10</sub> .3.1.3.o  |
| on logical file end  | 25 <sub>10</sub> .3.1.3.l  |
| on page end          | 25 <sub>10</sub> .3.1.3.n  |
| on physical file end | 25 <sub>10</sub> .3.1.3.m  |
| on value error       | 25 <sub>10</sub> .3.1.3.q  |
| open                 | 25 <sub>10</sub> .3.1.4.d  |
| page number          | 25 <sub>10</sub> .3.1.5.c  |
| pi                   | 25 <sub>10</sub> .2.3.12.a |
| print                | 25 <sub>10</sub> .5.1.d    |
| printf               | 25 <sub>10</sub> .5.1.f    |
| put                  | 25 <sub>10</sub> .3.3.1.a  |
| put bin              | 25 <sub>10</sub> .3.6.1.a  |
| put possible         | 25 <sub>10</sub> .3.1.3.c  |
| putf                 | 25 <sub>10</sub> .3.5.1.a  |
| random               | 25 <sub>10</sub> .5.1.b    |
| read                 | 25 <sub>10</sub> .5.1.e    |
| read bin             | 25 <sub>10</sub> .5.1.i    |
| readf                | 25 <sub>10</sub> .5.1.g    |
| real lengths         | 25 <sub>10</sub> .2.1.d    |
| real shorths         | 25 <sub>10</sub> .2.1.e    |

|                      |                            |
|----------------------|----------------------------|
| real width           | 25 <sub>10</sub> .3.2.1.n  |
| reidf                | 25 <sub>10</sub> .3.1.3.s  |
| reidf possible       | 25 <sub>10</sub> .3.1.3.h  |
| reset                | 25 <sub>10</sub> .3.1.6.j  |
| reset possible       | 25 <sub>10</sub> .3.1.3.f  |
| scratch              | 25 <sub>10</sub> .3.1.4.p  |
| set                  | 25 <sub>10</sub> .3.1.6.i  |
| set char number      | 25 <sub>10</sub> .3.1.6.k  |
| set possible         | 25 <sub>10</sub> .3.1.3.g  |
| sin                  | 25 <sub>10</sub> .2.3.12.g |
| small real           | 25 <sub>10</sub> .2.1.g    |
| space                | 25 <sub>10</sub> .3.1.6.a  |
| sqrt                 | 25 <sub>10</sub> .2.3.12.b |
| stand back           | 25 <sub>10</sub> .5.1.c    |
| stand back channel   | 25 <sub>10</sub> .3.1.2.g  |
| stand in             | 25 <sub>10</sub> .5.1.c    |
| stand in channel     | 25 <sub>10</sub> .3.1.2.e  |
| stand out            | 25 <sub>10</sub> .5.1.c    |
| stand out channel    | 25 <sub>10</sub> .3.1.2.f  |
| standconv            | 25 <sub>10</sub> .3.1.2.d  |
| stop                 | 25 <sub>10</sub> .5.2.a    |
| tan                  | 25 <sub>10</sub> .2.3.12.i |
| whole                | 25 <sub>10</sub> .3.2.1.b  |
| write                | 25 <sub>10</sub> .5.1.d    |
| write bin            | 25 <sub>10</sub> .5.1.h    |
| writef               | 25 <sub>10</sub> .5.1.f    |
| <i>L</i> BITS        | 25 <sub>10</sub> .2.2.g    |
| <i>L</i> BYTES       | 25 <sub>10</sub> .2.2.h    |
| <i>L</i> COMPL       | 25 <sub>10</sub> .2.2.f    |
| <i>L</i> INT         | 25 <sub>10</sub> .2.2.c    |
| <i>L</i> REAL        | 25 <sub>10</sub> .2.2.d    |
| <i>L</i> arccos      | 25 <sub>10</sub> .2.3.12.f |
| <i>L</i> arcsin      | 25 <sub>10</sub> .2.3.12.h |
| <i>L</i> arctan      | 25 <sub>10</sub> .2.3.12.j |
| <i>L</i> bits pack   | 25 <sub>10</sub> .2.3.8.l  |
| <i>L</i> bits width  | 25 <sub>10</sub> .2.1.j    |
| <i>L</i> bytes pack  | 25 <sub>10</sub> .2.3.9.c  |
| <i>L</i> bytes width | 25 <sub>10</sub> .2.1.m    |
| <i>L</i> cos         | 25 <sub>10</sub> .2.3.12.e |
| <i>L</i> exp         | 25 <sub>10</sub> .2.3.12.c |

## ALGOL 68 REVISED REPORT

|                            |                            |
|----------------------------|----------------------------|
| <i>L</i> exp width         | 25 <sub>10</sub> .3.2.1.o  |
| <i>L</i> int width         | 25 <sub>10</sub> .3.2.1.m  |
| <i>L</i> last random       | 25 <sub>10</sub> .5.1.a    |
| <i>L</i> ln                | 25 <sub>10</sub> .2.3.12.d |
| <i>L</i> max int           | 25 <sub>10</sub> .2.1.c    |
| <i>L</i> max real          | 25 <sub>10</sub> .2.1.f    |
| <i>L</i> next random       | 25 <sub>10</sub> .2.3.12.k |
| <i>L</i> pi                | 25 <sub>10</sub> .2.3.12.a |
| <i>L</i> random            | 25 <sub>10</sub> .5.1.b    |
| <i>L</i> real width        | 25 <sub>10</sub> .3.2.1.n  |
| <i>L</i> sin               | 25 <sub>10</sub> .2.3.12.g |
| <i>L</i> small real        | 25 <sub>10</sub> .2.1.g    |
| <i>L</i> sqrt              | 25 <sub>10</sub> .2.3.12.b |
| <i>L</i> tan               | 25 <sub>10</sub> .2.3.12.i |
| N <sub>0</sub> BEYOND      | 25 <sub>10</sub> .3.1.1.d  |
| N <sub>0</sub> BFILE       | 25 <sub>10</sub> .3.1.1.e  |
| N <sub>0</sub> BOOK        | 25 <sub>10</sub> .3.1.1.a  |
| N <sub>0</sub> COLLECTION  | 25 <sub>10</sub> .3.5.a    |
| N <sub>0</sub> COLLITEM    | 25 <sub>10</sub> .3.5.a    |
| N <sub>0</sub> CONV        | 25 <sub>10</sub> .3.1.2.b  |
| N <sub>0</sub> CPATTERN    | 25 <sub>10</sub> .3.5.a    |
| N <sub>0</sub> FLEXTEXT    | 25 <sub>10</sub> .3.1.1.b  |
| N <sub>0</sub> FPATTERN    | 25 <sub>10</sub> .3.5.a    |
| N <sub>0</sub> FRAME       | 25 <sub>10</sub> .3.5.a    |
| N <sub>0</sub> GPATTERN    | 25 <sub>10</sub> .3.5.a    |
| N <sub>0</sub> INSERTION   | 25 <sub>10</sub> .3.5.a    |
| N <sub>0</sub> INTYPE      | 25 <sub>10</sub> .3.2.2.d  |
| N <sub>0</sub> NUMBER      | 25 <sub>10</sub> .3.2.1.a  |
| N <sub>0</sub> OUTTYPE     | 25 <sub>10</sub> .3.2.2.b  |
| N <sub>0</sub> PATTERN     | 25 <sub>10</sub> .3.5.a    |
| N <sub>0</sub> PICTURE     | 25 <sub>10</sub> .3.5.a    |
| N <sub>0</sub> PIECE       | 25 <sub>10</sub> .3.5.a    |
| N <sub>0</sub> POS         | 25 <sub>10</sub> .3.1.1.c  |
| N <sub>0</sub> ROWS        | 25 <sub>10</sub> .2.3.1.a  |
| N <sub>0</sub> SFRAME      | 25 <sub>10</sub> .3.5.e    |
| N <sub>0</sub> SIMPLIN     | 25 <sub>10</sub> .3.2.2.c  |
| N <sub>0</sub> SIMPLOUT    | 25 <sub>10</sub> .3.2.2.a  |
| N <sub>0</sub> SINSERT     | 25 <sub>10</sub> .3.5.c    |
| N <sub>0</sub> STRAIGHTIN  | 25 <sub>10</sub> .3.2.3.b  |
| N <sub>0</sub> STRAIGHTOUT | 25 <sub>10</sub> .3.2.3.a  |

|                                    |                           |
|------------------------------------|---------------------------|
| N <sub>0</sub> TEXT                | 25 <sub>10</sub> .3.1.1.b |
| N <sub>0</sub> alignment           | 25 <sub>10</sub> .3.5.i   |
| N <sub>0</sub> associate format    | 25 <sub>10</sub> .3.5.k   |
| N <sub>0</sub> bfileprotect        | 25 <sub>10</sub> .3.1.1.h |
| N <sub>0</sub> book bounds         | 25 <sub>10</sub> .3.1.5.e |
| N <sub>0</sub> chainbfile          | 25 <sub>10</sub> .3.1.1.f |
| N <sub>0</sub> char dig            | 25 <sub>10</sub> .3.2.1.k |
| N <sub>0</sub> check pos           | 25 <sub>10</sub> .3.3.2.c |
| N <sub>0</sub> current pos         | 25 <sub>10</sub> .3.1.5.d |
| N <sub>0</sub> dig char            | 25 <sub>10</sub> .3.2.1.h |
| N <sub>0</sub> do fpattern         | 25 <sub>10</sub> .3.5.j   |
| N <sub>0</sub> edit string         | 25 <sub>10</sub> .3.5.1.b |
| N <sub>0</sub> false               | 25 <sub>10</sub> .3.1.4.i |
| N <sub>0</sub> file available      | 25 <sub>10</sub> .3.1.4.f |
| N <sub>0</sub> from bin            | 25 <sub>10</sub> .3.6.b   |
| N <sub>0</sub> get char            | 25 <sub>10</sub> .3.3.2.b |
| N <sub>0</sub> get good file       | 25 <sub>10</sub> .3.1.6.g |
| N <sub>0</sub> get good line       | 25 <sub>10</sub> .3.1.6.e |
| N <sub>0</sub> get good page       | 25 <sub>10</sub> .3.1.6.f |
| N <sub>0</sub> get insertion       | 25 <sub>10</sub> .3.5.h   |
| N <sub>0</sub> get next picture    | 25 <sub>10</sub> .3.5.b   |
| N <sub>0</sub> gremlins            | 25 <sub>10</sub> .4.1.a   |
| N <sub>0</sub> idf ok              | 25 <sub>10</sub> .3.1.4.g |
| N <sub>0</sub> indit string        | 25 <sub>10</sub> .3.5.2.b |
| N <sub>0</sub> line ended          | 25 <sub>10</sub> .3.1.5.f |
| N <sub>0</sub> lockedbfile         | 25 <sub>10</sub> .3.1.1.g |
| N <sub>0</sub> logical file ended  | 25 <sub>10</sub> .3.1.5.i |
| N <sub>0</sub> match               | 25 <sub>10</sub> .3.1.4.h |
| N <sub>0</sub> next pos            | 25 <sub>10</sub> .3.3.1.c |
| N <sub>0</sub> page ended          | 25 <sub>10</sub> .3.1.5.g |
| N <sub>0</sub> physical file ended | 25 <sub>10</sub> .3.1.5.h |
| N <sub>0</sub> put char            | 25 <sub>10</sub> .3.3.1.b |
| N <sub>0</sub> put insertion       | 25 <sub>10</sub> .3.5.g   |
| N <sub>0</sub> set bin mood        | 25 <sub>10</sub> .3.1.4.m |
| N <sub>0</sub> set char mood       | 25 <sub>10</sub> .3.1.4.l |
| N <sub>0</sub> set mood            | 25 <sub>10</sub> .3.1.6.h |
| N <sub>0</sub> set read mood       | 25 <sub>10</sub> .3.1.4.k |
| N <sub>0</sub> set write mood      | 25 <sub>10</sub> .3.1.4.j |
| N <sub>0</sub> standardize         | 25 <sub>10</sub> .3.2.1.g |
| N <sub>0</sub> staticize frames    | 25 <sub>10</sub> .3.5.f   |

|                                |                           |
|--------------------------------|---------------------------|
| $\aleph_0$ staticize insertion | 25 <sub>10</sub> .3.5.d   |
| $\aleph_0$ string to $L$ int   | 25 <sub>10</sub> .3.2.1.i |
| $\aleph_0$ string to $L$ real  | 25 <sub>10</sub> .3.2.1.j |
| $\aleph_0$ subfixed            | 25 <sub>10</sub> .3.2.1.f |
| $\aleph_0$ subwhole            | 25 <sub>10</sub> .3.2.1.e |
| $\aleph_0$ to bin              | 25 <sub>10</sub> .3.6.a   |
| $\aleph_0$ undefined           | 25 <sub>10</sub> .3.1.4.a |
| $\aleph_0$ $L$ standardize     | 25 <sub>10</sub> .3.2.1.g |

## 27.5 Alphabetic listing of metaproduction rules

**ABC** {24<sub>9</sub>.4.2.L} :: a ; b ; c ; d ; e ; f ; g ; h ; i ; j ; k ; l ; m ; n ; o ; p ; q ; r ; s ; t ; u ; v ; w ; x ; y ; z.

**ADIC** {19<sub>5</sub>.4.2.C} :: **DYADIC** ; **MONADIC**.

**ALPHA** {15<sub>1</sub>.3.B} :: a ; b ; c ; d ; e ; f ; g ; h ; i ; j ; k ; l ; m ; n ; o ; p ; q ; r ; s ; t ; u ; v ; w ; x ; y ; z.

**BECOMESETY** {24<sub>9</sub>.4.2.J} :: cum becomes ; cum assigns to ; **EMPTY**.

**BITS** {21<sub>6</sub>.5.A} :: structured with row of boolean field **SITHEITY** letter aleph mode.

**BYTES** {21<sub>6</sub>.5.B} :: structured with row of character field **SITHEITY** letter aleph mode.

**CASE** {17<sub>3</sub>.4.B} :: choice using integral ; choice using **UNITED**.

**CHOICE** {17<sub>3</sub>.4.A} :: choice using boolean ; **CASE**.

**COLLECTION** {25<sub>10</sub>.3.4.1.C} :: union of **PICTURE COLLITEM** mode.

**COLLITEM** {25<sub>10</sub>.3.4.1.D} :: structured with **INSERTION** field letter i digit one procedure yielding integral field letter r letter e letter p integral field letter p

**INSERTION** field letter i digit two mode.

**COMARK** {25<sub>10</sub>.3.4.1.N} :: zero ; digit ; character.

**COMMON** {18<sub>4</sub>.1.A} :: mode ; priority ; **MODINE** identity ; reference to **MODINE** variable ;

**MODINE** operation ; **PARAMETER** ; **MODE FIELDS**.

**COMORF** {21<sub>6</sub>.1.G} :: **NEST** assignation ; **NEST** identity relation ;

**NEST LEAP** generator ; **NEST** cast ; **NEST** denoter ; **NEST** format text.

**CPATTERN** {25<sub>10</sub>.3.4.1.I} :: structured with **INSERTION** field letter i integral field letter t letter y letter p letter e

row of **INSERTION** field letter c mode.

**DEC** {15<sub>1</sub>.2.3.E} :: **MODE TAG** ; priority **PRIO TAD** ; **MOID TALLY TAB** ; **DUO TAD** ; **MONO TAM**.

**DECS** {15<sub>1</sub>.2.3.D} :: **DEC** ; **DECS DEC**.

**DECSETY** {15<sub>1</sub>.2.3.C} :: **DECS** ; **EMPTY**.

**DEFIED** {18<sub>4</sub>.8.B} :: defining ; applied.



**DIGIT** {24<sub>9</sub>.4.2.C} :: **digit zero ; digit one ; digit two ; digit three ; digit four ; digit five ; digit six ; digit seven ; digit eight ; digit nine.**

**DOP** {24<sub>9</sub>.4.2.M\*} :: **DYAD ; DYAD cum NOMAD.**

**DUO** {15<sub>1</sub>.2.3.H} :: **procedure with PARAMETER1 PARAMETER2 yielding MOID.**

**DYAD** {24<sub>9</sub>.4.2.G} :: **MONAD ; NOMAD.**

**DYADIC** {19<sub>5</sub>.4.2.A} :: **priority PRIO.**

**EMPTY** {15<sub>1</sub>.2.G} :: **.**

**ENCLOSED** {15<sub>1</sub>.2.2.A} :: **closed ; collateral ; parallel ; CHOICE ; loop.**

**EXTERNAL** {25<sub>10</sub>.1.A} :: **standard ; library ; system ; particular.**

**FIELD** {15<sub>1</sub>.2.J} :: **MODE field TAG.**

**FIELDS** {15<sub>1</sub>.2.I} :: **FIELD ; FIELDS FIELD.**

**FIRM** {21<sub>6</sub>.1.B} :: **MEEK ; united to.**

**FIVMAT** {25<sub>10</sub>.3.4.1.L} :: **mui definition of structured with row of structured with integral field letter c letter p integral field letter c letter o letter u letter n letter t integral field letter b letter p row of union of structured with union of PATTERN CPATTERN structured with INSERTION field letter i procedure yielding mui application field letter p letter f mode GPATTERN void mode field letter p INSERTION field letter i mode COLLITEM mode field letter c mode field letter aleph mode. FLEXETY** {15<sub>1</sub>.2.K} :: **flexible ; EMPTY.**

**FORM** {21<sub>6</sub>.1.E} :: **MORF ; COMORF.**

**FORMAT** {25<sub>10</sub>.3.4.1.A} :: **structured with row of PIECE field letter aleph mode.**

**FPATTERN** {25<sub>10</sub>.3.4.1.J} :: **structured with INSERTION field letter i procedure yielding FIVMAT field letter p letter f mode.**

**FRAME** {25<sub>10</sub>.3.4.1.H} :: **structured with INSERTION field letter i procedure yielding integral field letter r letter e letter p boolean field letter s letter u letter p letter p character field letter m letter a letter r letter k letter e letter r mode.**

**FROBYT** {17<sub>3</sub>.5.A} :: **from ; by ; to.**

**GPATTERN** {25<sub>10</sub>.3.4.1.K} :: **structured with INSERTION field letter i row of procedure yielding integral field letter s letter p letter e letter c mode. HEAD** {22<sub>7</sub>.3.B} :: **PLAIN ; PREF ; structured with ; FLEXETY ROWS of ; procedure with ; union of ; void.**

**INDICATOR** {18<sub>4</sub>.8.A} :: **identifier ; mode indication ; operator.**

**INSERTION** {25<sub>10</sub>.3.4.1.E} :: **row of structured with procedure yielding integral field letter r letter e letter p union of row of character character mode field letter s letter a mode.**

**INTREAL** {15<sub>1</sub>.2.C} :: **SIZETY integral ; SIZETY real.**

**LAB** {15<sub>1</sub>.2.3.K} :: label TAG.  
**LABS** {15<sub>1</sub>.2.3.J} :: LAB ; LABS LAB.  
**LABSETY** {15<sub>1</sub>.2.3.I} :: LABS ; EMPTY.  
**LAYER** {15<sub>1</sub>.2.3.B} :: new DECSETY LABSETY.  
**LEAP** {18<sub>4</sub>.4.B} :: local ; heap ; primal.  
**LENGTH** {21<sub>6</sub>.5.D} :: letter l letter o letter n letter g.  
**LENGTHETY** {21<sub>6</sub>.5.F} :: LENGTH LENGTHETY ; EMPTY.  
**LETTER** {24<sub>9</sub>.4.2.B} :: letter ABC ; letter aleph ; style TALLY letter ABC.  
**LONGSETY** {15<sub>1</sub>.2.E} :: long LONGSETY ; EMPTY.  
**MARK** {25<sub>10</sub>.3.4.1.M} :: sign ; point ; exponent ; complex ; boolean.  
**MEEK** {21<sub>6</sub>.1.C} :: unchanged from ; dereferenced to ; deprocedured to.  
**MODE** {15<sub>1</sub>.2.A} :: PLAIN ; STOWED ; REF to MODE ; PROCEDURE ; UNITED ;  
 MU definition of MODE ; MU application. **MODINE** {18<sub>4</sub>.4.A} :: MODE ; routine.  
**MOID** {15<sub>1</sub>.2.R} :: MODE ; void.  
**MOIDS** {18<sub>4</sub>.6.C} :: MOID ; MOIDS MOID.  
**MOIDSETY** {18<sub>4</sub>.7.C} :: MOIDS ; EMPTY.  
**MONAD** {24<sub>9</sub>.4.2.H} :: or ; and ; ampersand ; differs from ; is at most ; is at least ;  
 over ; percent ; window ; floor ; ceiling ; plus i times ; not ; tilde ;  
 down ; up ; plus ; minus ; style TALLY monad. **MONADIC** {19<sub>5</sub>.4.2.B} :: priority  
 iii iii iii i.  
**MONO** {15<sub>1</sub>.2.3.G} :: procedure with PARAMETER yielding MOID.  
**MOOD** {15<sub>1</sub>.2.U} :: PLAIN ; STOWED ; reference to MODE ; PROCEDURE ; void.  
**MOODS** {15<sub>1</sub>.2.T} :: MOOD ; MOODS MOOD.  
**MOODSETY** {18<sub>4</sub>.7.B} :: MOODS ; EMPTY.  
**MORF** {21<sub>6</sub>.1.F} :: NEST selection ; NEST slice ; NEST routine text ; NEST ADIC  
 formula ; NEST call ; NEST applied identifier with TAG.  
**MU** {15<sub>1</sub>.2.V} :: muTALLY.  
**NEST** {15<sub>1</sub>.2.3.A} :: LAYER ; NEST LAYER.  
**NOMAD** {24<sub>9</sub>.4.2.I} :: is less than ; is greater than ; divided by ; equals ; times ;  
 asterisk.  
**NONPREF** {22<sub>7</sub>.1.B} :: PLAIN ; STOWED ;  
 procedure with PARAMETERS yielding MOID ; UNITED ; void.  
**NONPROC** {21<sub>6</sub>.7.A} :: PLAIN ; STOWED ;  
 REF to NONPROC ; procedure with PARAMETERS yielding MOID ; UNITED.  
**NONSTOWED** {18<sub>4</sub>.7.A} :: PLAIN ; REF to MODE ; PROCEDURE ; UNITED ; void.  
**NOTETY** {15<sub>1</sub>.3.C} :: NOTION ; EMPTY.  
**NOTION** {15<sub>1</sub>.3.A} :: ALPHA ; NOTION ALPHA.  
**NUMERAL** {23<sub>8</sub>.1.0.B\*} :: fixed point numeral ; variable point numeral ; floating  
 point numeral.  
**PACK** {17<sub>3</sub>.1.B} :: STYLE pack.  
**PARAMETER** {15<sub>1</sub>.2.Q} :: MODE parameter.  
**PARAMETERS** {15<sub>1</sub>.2.P} :: PARAMETER ; PARAMETERS PARAMETER.  
**PARAMETY** {15<sub>1</sub>.2.O} :: with PARAMETERS ; EMPTY.  
**PART** {22<sub>7</sub>.3.E} :: FIELD ; PARAMETER.

**PARTS** {22<sub>7</sub>.3.D} :: **PART** ; **PARTS PART**.  
**PATTERN** {25<sub>10</sub>.3.4.1.G} :: structured with  
integral field letter t letter y letter p letter e  
row of **FRAME** field  
letter f letter r letter a letter m letter e letter s mode. **PICTURE** {25<sub>10</sub>.3.4.1.F} ::  
structured with  
union of **PATTERN CPATTERN FPATTERN GPATTERN** void mode field letter  
**p**  
**INSERTION** field letter i mode. **PIECE** {25<sub>10</sub>.3.4.1.B} :: structured with  
integral field letter c letter p  
integral field letter c letter o letter u letter n letter t  
integral field letter b letter p  
row of **COLLECTION** field letter c mode.  
**PLAIN** {15<sub>1</sub>.2.B} :: **INTREAL** ; boolean ; character.  
**PRAGMENT** {24<sub>9</sub>.2.A} :: **pragmat** ; comment.  
**PRAM** {18<sub>4</sub>.5.A} :: **DUO** ; **MONO**.  
**PREF** {22<sub>7</sub>.1.A} :: **procedure yielding** ; **REF to**.  
**PREFSETY** {22<sub>7</sub>.1.C\*} :: **PREF PREFSETY** ; **EMPTY**.  
**PRIMARY** {19<sub>5</sub>.D} :: slice coercee ; call coercee ; cast coercee ; denoter coercee ;  
format text coercee ; applied identifier with **TAG** coercee ; **ENCLOSED** clause.  
**PRIO** {15<sub>1</sub>.2.3.F} :: i ; ii ; iii ; iii i ; iii ii ; iii iii ; iii iii i ; iii iii ii ; iii iii iii.  
**PROCEDURE** {15<sub>1</sub>.2.N} :: procedure **PARAMETY** yielding **MOID**.  
**PROP** {18<sub>4</sub>.8.E} :: **DEC** ; **LAB** ; **FIELD**.  
**PROPS** {18<sub>4</sub>.8.D} :: **PROP** ; **PROPS PROP**.  
**PROPSETY** {18<sub>4</sub>.8.C} :: **PROPS** ; **EMPTY**.  
**QUALITY** {18<sub>4</sub>.8.F} :: **MODE** ; **MOID TALLY** ; **DYADIC** ; label ; **MODE** field.  
**RADIX** {23<sub>8</sub>.2.A} :: radix two ; radix four ; radix eight ; radix sixteen.  
**REF** {15<sub>1</sub>.2.M} :: reference ; transient reference.  
**REFETY** {19<sub>5</sub>.3.1.A} :: **REF to** ; **EMPTY**.  
**REFLEXETY** {19<sub>5</sub>.3.1.B} :: **REF to** ; **REF to flexible** ; **EMPTY**.  
**ROWS** {15<sub>1</sub>.2.L} :: row ; **ROWS row**.  
**ROWSETY** {19<sub>5</sub>.3.2.A} :: **ROWS** ; **EMPTY**.  
**SAFE** {22<sub>7</sub>.3.A} :: safe ; **MU** has **MODE SAFE** ; yin **SAFE** ; yang **SAFE** ; remember  
**MOID1 MOID2 SAFE**.  
**SECONDARY** {19<sub>5</sub>.C} :: **LEAP** generator coercee ; selection coercee ; **PRIMARY**.  
**SHORTH** {21<sub>6</sub>.5.E} :: letter s letter h letter o letter r letter t.  
**SHORTHETY** {21<sub>6</sub>.5.G} :: **SHORTH SHORTHETY** ; **EMPTY**.  
**SHORTSETY** {15<sub>1</sub>.2.F} :: short **SHORTSETY** ; **EMPTY**.  
**SITHETY** {21<sub>6</sub>.5.C} :: **LENGTH LENGTHETY** ; **SHORTH SHORTHETY** ; **EMPTY**.  
**SIZE** {23<sub>8</sub>.1.0.A} :: long ; short.  
**SIZETY** {15<sub>1</sub>.2.D} :: long **LONGSETY** ; short **SHORTSETY** ; **EMPTY**.  
**SOFT** {21<sub>6</sub>.1.D} :: unchanged from ; softly deprocedured to.  
**SOID** {17<sub>3</sub>.1.A} :: **SORT MOID**.  
**SOME** {15<sub>1</sub>.2.2.B} :: **SORT MOID NEST**.

**SORT** {15<sub>1</sub>.2.2.C} :: **strong ; firm ; meek ; weak ; soft.**  
**STANDARD** {24<sub>9</sub>.4.2.E} :: **integral ; real ; boolean ; character ; format ; void ; complex ;**  
    **bits ; bytes ; string ; sema ; file ; channel.**  
**STOP** {25<sub>10</sub>.1.B} :: **label letter s letter t letter o letter p.**  
**STOWED** {15<sub>1</sub>.2.H} :: **structured with FIELDS mode ; FLEXETY ROWS of MODE.**  
**STRONG** {21<sub>6</sub>.1.A} :: **FIRM ; widened to ; rowed to ; voided to.**  
**STYLE** {15<sub>1</sub>.3.3.A} :: **brief ; bold ; style TALLY.**  
**TAB** {24<sub>9</sub>.4.2.D} :: **bold TAG ; SIZETY STANDARD.**  
**TAD** {24<sub>9</sub>.4.2.F} :: **bold TAG ; DYAD BECOMESETY ; DYAD cum NOMAD BECOMESETY.**  
**TAG** {24<sub>9</sub>.4.2.A} :: **LETTER ; TAG LETTER ; TAG DIGIT.**  
**TAILETY** {22<sub>7</sub>.3.C} :: **MOID ; FIELDS mode ; PARAMETERS yielding MOID ; MOODS mode ; EMPTY.**  
**TALLETY** {19<sub>5</sub>.4.2.D} :: **TALLY ; EMPTY.**  
**TALLY** {15<sub>1</sub>.2.W} :: **i ; TALLY i.**  
**TAM** {24<sub>9</sub>.4.2.K} :: **bold TAG ; MONAD BECOMESETY ; MONAD cum NOMAD BECOMESETY.**  
**TAO** {18<sub>4</sub>.5.B} :: **TAD ; TAM.**  
**TAX** {18<sub>4</sub>.8.G} :: **TAG ; TAB ; TAD ; TAM.**  
**TERTIARY** {19<sub>5</sub>.B} :: **ADIC formula coercee ; nihil ; SECONDARY.**  
**THING** {15<sub>1</sub>.3.D} :: **NOTION ; (NOTETY1) NOTETY2 ; THING (NOTETY1) NOTETY2.**  
**TYPE** {25<sub>10</sub>.3.4.1.P} :: **integral ; real ; boolean ; complex ; string ; bits ;**  
    **integral choice ; boolean choice ; format ; general.**  
**UNIT** {19<sub>5</sub>.A} :: **assignation coercee ; identity relation coercee ; routine text coercee ;**  
    **jump ; skip ; TERTIARY.**  
**UNITED** {15<sub>1</sub>.2.S} :: **union of MOODS mode.**  
**UNSUPPRESSETY** {25<sub>10</sub>.3.4.1.O} :: **unsuppressible ; EMPTY.**  
**VICTAL** {18<sub>4</sub>.6.A} :: **VIRACT ; formal.**  
**VIRACT** {18<sub>4</sub>.6.B} :: **virtual ; actual.**  
**WHETHER** {15<sub>1</sub>.3.E} :: **where ; unless.**

**Part V**

**Appendices**



# Appendix A

## Bibliography

For other informal texts on the language, see for instance:

1. D. F. Brailsford, A. N. Walker. Introductory Algol 68 programming. Ellis Horwood Ltd, Chichester [1979].
2. C. H. Lindsey and S. G. van der Meulen, Informal Introduction to Algol 68, North-Holland [1977].
3. A. D. McGettrick. Algol 68: a first and second course. Cambridge University Press [1978].
4. C. H. A. Koster, H. Meijer. Systematisch programmeren in Algol 68. Deel 1, 2. Kluwer, Deventer [1978][1981].

*a68g* implements extensions that are documented in following material:

1. C. H. Lindsey. Specification of partial parametrization proposal. This specification is not a part of the Algol 68 Revised Report, and is reproduced in this book as section 20 with kind permission of C.H. Lindsey. The original publication is ALGOL BULLETIN 39.3.1 pages 6-9.
2. S. G. van der Meulen, M. Veldhorst. TORRIX - A programming language for operations on vectors and matrices over arbitrary fields and of variable size. Rijksuniversiteit Utrecht [1977].

Some interesting material on the history of Algol 68:

1. C. H. A. Koster, The Making of Algol 68 [1996].  
<http://www.cs.ru.nl/kees/home/biblio.html>

## APPENDICES

2. C. H. Lindsey, A History of Algol 68. ACM SIGPLAN Notices **28**(3) 97 [1993].

Online information on Algol 68:

1. The Algol 68 Genie web pages.  
<http://www.xs4all.nl/~jmvdveer/algol.html>



2. An overview of the development of Algol, and implementations, can be found at Paul McJones's page.  
<http://www.softwarepreservation.org/projects/ALGOL/>
3. Ample Algol 68 code is available from the Rosetta Code project.  
<http://www.softwarepreservation.org/projects/ALGOL/>
4. Paul McJones's page on the development of Algol, and an overview of implementations. <http://rosettacode.org>
5. Greg Nunan maintains a page on Algol 68 web resources.  
<http://www.algol68.org/>
6. HOPL - History of programming languages. Online encyclopaedia of programming languages has an entry for Algol 68.  
<http://hop1.murdoch.edu.au/>
7. Karl Kleine collects and posts Algol 68 historic documents.  
<http://www.fh-jena.de/~kleine/history/history.html>
8. Brian Wichmann maintains the ALGOL BULLETIN online archive.  
[http://archive.computerhistory.org/resources/text/algol/algol\\_bulletin/](http://archive.computerhistory.org/resources/text/algol/algol_bulletin/)
9. The Algol 68 directory from the Open Directory Project.  
[http://dmoz.org/Computers/Programming/Languages/Algol\\_68/](http://dmoz.org/Computers/Programming/Languages/Algol_68/)
10. The Wikipedia Project has entries for Algol 68, Algol 68 Genie, Adriaan van Wijngaarden, and two level grammars.  
<http://en.wikipedia.org/>
11. Dick Grune has posted an Algol 68 program collection, including the MC Test Set.  
<http://www.cs.vu.nl/pub/dick/Algol68/>
12. The Algol 68 project on SourceForge. <http://sourceforge.net/projects/algol68>

Next to Algol 68 Genie, other implementations are still available:

1. *ALGOL68S* compiler by Charles Lindsey.  
<http://www.cs.man.ac.uk/~chl/>
2. Algol 68 to C translator *a68toc*, a ported compiler by Sian Leitch. The front end essentially is the *ALGOL68RS* portable compiler originally written by the Defence Research Agency when it was known as the RSRE (Royal Signals and Radar Establishment) but formats are unavailable. *ALGOL68RS* did not implement the parallel clause. The *a68toc* distribution includes the  $\text{\TeX}$  source of *Programming Algol 68 Made Easy* which is available under the GNU General Public License. The *a68toc* distribution is available from Greg Nunan's site  
<http://www.algol68.org/>

## APPENDICES

This book refers to legacy implementations of Algol 68:

1. L. Ammeraal, An interpreter for simple Algol 68 programs. Stichting Mathematisch Centrum Amsterdam [1973].
2. *CONTROL DATA ALGOL 68*. Zoethout et al., CDC Netherlands 1974. Full implementation (apparently it only lacked the format pattern), used mainly in teaching in Germany and the Netherlands on CDC mainframes.
3. *ALGOL68C*. Bourne et al., Cambridge 1980. Subset implementation for IBM mainframes, DECsystems 10 and 20, VAX/UNIX, CYBER 205 and other systems. Compiler front end that generated code for a ZCODE back end.
4. *ALGOL68R*. Currie et al., RSRE (Royal Signals and Radar Establishment, Malvern) 1970. Single pass compiler, used extensively in military and scientific circles in the UK on ICL machines.
5. *ALGOL68S*. Hibbard et al., UK and Carnegie Mellon 1976. Subset, ported to small machines as the PDP-11, (apparently) mostly used in teaching.
6. *ALGOL68RS*. For ICL 2900, VAX/VMS and other systems.
7. *FLACC*. Full Language Algol 68 Checkout Compiler. Mailloux et al., Edmonton, Canada 1978. An interpreter for the full language, available on IBM mainframes and used in teaching, both in North America and in Europe.

This book contains smaller or larger parts from different open sources:

1. *Programming Algol 68 Made Easy* by Sian Leitch (GNU GPL).
2. GNU Compiler Collection (GNU GPL).
3. GNU Debugger (GNU GPL).
4. GNU C Library (GNU GPL).
5. GNU Scientific Library documentation (GNU GPL).
6. GNU Plotting Utilities documentation (GNU GPL).
7. PostgreSQL documentation (BSD license).
8. Wikipedia (GNU FDL).
9. Biografisch Woordenboek van Nederlandse Wiskundigen (Public domain).

# Appendix B

## Reporting bugs

{Les erreurs passent, il n’y a que le vrai qui reste.  
Denis Diderot. }

Algol 68 Genie is regression tested using a test suite containing over 1,500 programs comprising circa 190,000 lines of Algol 68 source code. This suite is composed of programs from various sources such as Dick Grune’s *The Mathematisch Centrum Algol 68 Test Set*, test sets from Bernard Houssais and Paul Branquart, the personal applications and test programs of a68g’s author, contributions from a68g users and code from <http://rosettacode.org> and <http://www.codecodex.com>. As you will be aware, no matter how large a test suite, you may uncover a bug that the current suite did not detect. Therefore your bug reports play an essential role in making Algol 68 Genie reliable and keeping this book up-to-date. When you encounter a problem, then please report it as outlined in below section B.2.

### B.1 Have you found a bug?

If you are not sure whether you have found a bug, here are some guidelines:

- Check using section 9.11 that your issue is not a known one.
- If you specified option `--optimise`, use the default `--no-optimise` to check whether you get a runtime error from the interpreter proper. The unit compiler omits many runtime checks, making a faulty Algol 68 program behave erratically.
- If `a68g` gets a fatal signal, and for instance produces a core dump, for any input, that is most likely a bug. For example:

```
$ a68g hello.a68
```

### **Segmentation fault**

But please check section 9.11 for some known conditions that can crash *a68g*.

- If *a68g* produces an error message for valid input, that is a bug. An example would be a syntax error issued for valid Algol 68 source code.
- If *a68g* does not produce an error message for invalid input, that is a bug. However, you should note that your idea of invalid input might be someone else's idea of an extension or support for traditional practice.
- If *a68g* just terminates without producing expected output, this may indicate a bug. But please check that the bug is not an error in the Algol 68 source program.
- If you are an experienced user of Algol 68, your suggestions for improvement of Algol 68 Genie are welcome in any case.

## **B.2 How and where to report bugs**

If you want to report a bug please send an e-mail to [algol68g@xs4all.nl](mailto:algol68g@xs4all.nl). Please include in your bug report:

- At least the version number of *a68g* that produced the bug, but preferably the output of the command:  

```
$ a68g {-}{-}version
```
- A description of the bug.
- If possible, submit the Algol 68 source code that produced the bug.
- The hardware and operating system.

Your feedback is much appreciated.

# Appendix C

## Release history

{Were I to await perfection, my book would never be finished.  
Tai T'ung, 13<sup>th</sup> century. }

Development of Algol 68 Genie started in 1992. Algol 68 Genie was released under GNU GPL in 2001.

Version 2.0.0, September 2010

1. Adds compiler for units to speed execution
2. Adds options `optimise` and `no-optimise`
3. Adds options `compile` and `no-compile`
4. Adds options `object` and `no-object`
5. Adds option `strict`
6. Improves parser
7. Improves diagnostics
8. Updates documentation
9. Regression fixes:
  1. Zero-suppression sometimes failed in formatted transput
  2. The default draw fill style was 1, not 0 as stated in the manual

Version 1.18.0, August 2009

1. Regression fixes:
  1. Rowing failed for some flexible rows

## APPENDICES

2. Generator failed for some structures
  3. Garbage collector could crash under rare circumstances
  4. Some flexible mode declarations were not accepted
  5. Some modes were incorrectly marked as equivalent
  6. Unnecessary size limitations in some matrix operations
  7. Build error on Mac OS X
  8. Alignment error on AMD64
2. Adds **field-selections** as an alternative to **selections**
  3. Adds `NEW` as alternative to `HEAP`
  4. More efficient use of memory during compilation
  5. Documentation updates
  6. Upon request, re-post the HTML translation of the Revised Report
  7. Change to a common Linux-style version numbering  
(former *Mark j* will now read *Version 1.j.0*, and former *Mark j.k* will now read *Version 1.j.k*)

### Version 1.17.0, May 2009

1. Adds C-style placeholders in formats {see 7.11.2}
2. Adds warnings when tags hide declarations with larger reach
3. Regression fixes:
  1. The **while-part** range was not visible from deeper ranges in case a **loop-clause** was the sole **unit** of a **routine-text**
  2. `SKIP` did not generate a value for some modes
4. Lint-free source code {see 9.10}
5. Documentation updates

### Version 1.16.0, April 2009

1. Regression fixes
2. Documentation updates,  
adds Revised Report to the  $\text{\LaTeX}$  documentation,  
discontinues HTML translation of the Revised Report.

### Version 1.15.1, January 2009

1. Adds Laplace transform
2. Regression fixes and documentation updates

Version 1.15.0, December 2008

1. Regression fixes and documentation updates
2. Improves diagnostics

Version 1.14.1, November 2008

1. Implements zero replicators as required by the Revised Report
2. Decommissions (undocumented) VMS option syntax

Version 1.14.0, October 2008

1. Adds first edition of the *Algol 68 Genie User Manual*
2. Adds Revised Report (HTML translation 1.2) to the documentation
3. Adds option `link` to the monitor

Version 1.13.0, June 2008

1. Adds procedures to interrogate files and directories
2. Adds various operators and procedures
3. Adds `apropos` to command line options and monitor options

Version 1.12.0, May 2008

1. Improves code for the parallel clause
2. Improves interpreter efficiency
3. Improves breakpoint commands in the monitor
4. Adds options `UNTIL`, `FINISH` and `OUT` to the monitor
5. Adds Revised Report (HTML translation 1.1) to the documentation

Version 1.11.1, March 2008

## APPENDICES

1. Adds options `RERUN`, `RESET` and `RESTART` to monitor
2. Adds Revised Report (HTML translation 1.0) to the documentation

### Version 1.11.0, November 2007

1. Adds support for sound
2. Adds Fourier transforms
3. Adds pseudo operator `DIAG`
4. Adds pseudo operator `TRNSP`
5. Adds pseudo operator `COL`
6. Adds pseudo operator `ROW`
7. Improves diagnostics in formatted transput
8. Improves monitor

### Version 1.10.2, April 2007

1. Adds procedure `real`
2. Adds format pattern `h`
3. Adds operator `OP SET = (INT, L BITS) L BITS`
4. Adds operator `OP CLEAR = (INT, L BITS) L BITS`

### Version 1.10.1, December 2006

1. Improves interpreter efficiency

### Version 1.10.0, August 2006

1. Adds basic linear algebra
2. Adds procedures `arctan2`, `long arctan2` and `long long arctan2`
3. Adds procedure `execve output`
4. Improves diagnostics
5. Adds option `no-backtrace`



Version 1.9.2, July 2006

1. Adds typographical display features

Version 1.9.1, May 2006

1. 64-bit safe interpreter
2. Improves monitor/debugger
3. Adds option `monitor` (or `debug`)
4. Adds procedure `monitor` (or `debug`)

Version 1.9.0, March 2006

1. Adds basic PostgreSQL support
2. Adds procedure `utc time` (UNIX)
3. Adds procedure `local time` (UNIX)
4. Improves diagnostic messages

Version 1.8.1, November 2005

1. Adds procedure `sub in string` for substituting regular expressions in a string (UNIX)
2. Adds option `portcheck cf. standard` hardware representation
3. Adds option `pedantic`

Version 1.8.0, July 2005

1. Adds procedure `http content` for fetching web page contents (UNIX)
2. Adds procedure `tcp request` for sending requests via TCP (UNIX)
3. Adds procedure `grep in string` for matching regular expressions in a string (UNIX)
4. Adds procedure `last char in string` to find last occurrence of a character in a string
5. Adds procedure `string in string` to find first occurrence of a string in a string

## APPENDICES

6. Adds keyword `UNTIL` to implement post checked loop
7. Adds keyword `DOWNTO` to complement `TO`
8. Adds keyword `ANDTH` as alternative for `ANDE` and keyword `OREL` as alternative for `ORF`
9. Makes `a SKIP` when used as a unit (cf. `RR`)

### Version 1.7.0, May 2005

1. Adds partial parametrisation following C. H. Lindsey's proposal
2. Adds elementary and trigonometric functions for complex numbers, independent of `GSL`

### Version 1.6.0, March 2005

1. Adds parallel clause on platforms that support POSIX threads

### Version 1.5.0, October 2004

1. Improves interpreter efficiency
2. Adds transput on `STRING` as well as `FILE` and `PIPE`
3. Adds dynamic scope checking
4. Adds basic preprocessor
5. Adds curses support

### Version 1.4.0, June 2004

1. Adds more `GSL` support
2. Adds modes `LONG BITS` and `LONG LONG BITS`
3. Makes parser accept `( .. )` as alternative for `[ .. ]`
4. Makes parser optionally treat `..`, `[ .. ]` and `( .. )` as equivalent
5. Makes parser accept loop clause as encloses clause
6. Makes mode checker accept `UNION` with components relates through deflexing
7. Changes to thread safe `plotutils` interface

Version 1.3.2, March 2004

1. Adds overflow checks for primitive modes

Version 1.3.1, January 2004

1. Adds optional system stack overflow handling to interpreter
2. Improves multiprecision library

Version 1.3.0, September 2003

1. Adds formatted transput
2. Improves speed of multiprecision library

Version 1.2.0, March 2003

1. Adds mode `BYTES` and `LONG BYTES`
2. Adds mode `FORMAT` and parsing of `FORMAT` texts
3. Adds straightening in unformatted transput
4. Adds mode `PIPE` and `UNIX` support
5. Changes to `UNIX` system level IO

Version 1.1.0, November 2002

Version 1.0.0, September 2002

Beta releases, from November 2001 onward

## APPENDICES

# Appendix D

## Algol 68 Genie manual page

The text below results from next command, if *a68g* is installed by the superuser on your system:

```
$ man a68g
```

```
A68G(1) BSD General Commands Manual A68G(1)

NAME
    a68g - Algol 68 Genie, an Algol 68 interpreter

SYNOPSIS
    a68g [--apropos | --help | --info -[string]]
    [--assertions | --no-assertions] [--backtrace | --no-backtrace]
    [--brackets] [--check | --no-run] [--compile | --no-compile]
    [--debug | --monitor] [--echo string] [--execute unit | e unit]
    [--extensive] [--frame number] [--handles number] [--heap number]
    [--listing] [--moids] [--object | --no-object] [--overhead number]
    [--optimise | --no-optimise] [--pragmats | --no-pragmats]
    [--precision number] [--prelude-listing] [--print unit | p unit]
    [--quote-stropping] [--reductions] [--rerun] [--run]
    [--source | --no-source] [--stack number] [--statistics] [--strict]
    [--terminal] [--time-limit number] [--trace | --no-trace]
    [--tree | --no-tree] [--unused] [--upper-stropping] [--verbose]
    [--version] [--warnings | --no-warnings] [--xref | --no-xref] filename

DESCRIPTION
    Algol 68 Genie (Algol68G) is an Algol 68 compiler-interpreter. It can be
    used for executing Algol 68 programs or scripts. Algol 68 is a rather
    lean orthogonal general-purpose language that is a beautiful means for
    denoting algorithms. Algol 68 was designed as a general-purpose program-
    ming language by IFIP Working Group 2.1 (Algorithmic Languages and Cal-
    culi) that has continuing responsibility for Algol 60 and Algol 68.

    Algol 68 Genie and its documentation can be obtained from

        www.xs4all.nl/~jmvdveer/algol.html

OPTIONS
```

## APPENDICES

Options are passed to a68g either from the file .a68g.rc in the working directory, the environment variable A68G\_OPTIONS, the command-line or from pragmat.

Option precedence is as follows: pragmat options supersede command-line options, command-line options supersede options in environment variable A68G\_OPTIONS, A68G\_OPTIONS supersedes options in .a68g.rc.

Listing options, tracing options and -pragmat, -nopragmat, take their effect when they are encountered in a left-to-right pass of the program text, and can thus be used, for example, to generate a cross reference for a particular part of the program.

Where numeric arguments are required, suffices k, M or G are allowed for multiplication with 2 \*\* 10, 2 \*\* 20 or 2 \*\* 30 respectively.

```
--apropos | --help | --info [string]
    Print info on options if string is omitted, or print info on
    string otherwise.

--assertions | --no-assertions
    Control elaboration of assertions.

--backtrace | --no-backtrace
    Control whether a stack backtrace is done in case a runtime-error
    occurs.

--brackets
    Consider [ .. ] and { .. } as being equivalent to ( .. ). Tradi-
    tional Algol 68 syntax allows ( .. ) to replace [ .. ] in bounds
    and slices.

--check | --no-run
    Check syntax only, the interpreter does not start.

--compile | --no-compile
    Switch compilation of units on or off. Compilation omits many of
    the runtime checks offered by the interpreter proper. The program
    is not executed and a shell script is generated combining source
    code and its shared library. This shell script can be used as a
    pseudo-executable.

--debug | --monitor
    Start in the monitor. Invoke the monitor in case a runtime-error
    occurs; the program will pause in the monitor on the line that
    contains the error.

--echo string
    Echo string to stdout.

--execute unit | -e unit
    Execute the Algol 68 unit. In this way one-liners can be executed
    from the command line.

--extensive
    Generate an extensive listing.

--frame number
    Set the frame stack size to number bytes.

--handles number
    Set the handle space size to number bytes.
```

```

--heap number
    Set the heap size to number bytes.

--listing
    Generate a concise listing.

--moids
    Generate an overview of modes in the listing file.

--object | --no-object
    Control the listing of C code in the listing file.

--optimise | --no-optimise
    Switch compilation of units on or off. Compilation omits many of
    the runtime checks offered by the interpreter proper.

--overhead number
    Set overhead for stack checking.

--portcheck | --no-portcheck
    Enable or disable portability warning messages.

--pragmats | --no-pragmats
    Control elaboration of pragmats.

--precision number
    Set the precision for LONG LONG modes to number significant dig-
    its.

--prelude-listing
    Generate a listing of preludes.

--print unit | -p unit
    Print the value yielded by the Algol 68 unit. In this way one-
    liners can be executed from the command line.

--quote-stropping
    Use quote stropping.

--reductions
    Print reductions made by the parser.

--rerun
    Use compiled code of a previous run.

--run
    Override the --no-run option.

--source | --no-source
    Control the listing of source lines in the listing file.

--stack number
    Set the stack size to number bytes.

--statistics
    Generate statistics in the listing file.

--strict
    Ignores extensions to Algol 68 syntax.

--time-limit number
    Interrupt the interpreter after number seconds, generating a time

```

## APPENDICES

```
    limit exceeded error.

--trace | --no-trace
    Control tracing of the running program.

--tree | --no-tree
    Control listing of the syntax tree in the listing file.

--unused
    Generate an overview of unused tags in the listing file.

--upper-stropping
    Use upper stropping, which is the default stropping regime.

--verbose
    Use verbose mode.

--version
    Print the version of the running image of a68g.

--warnings | --no-warnings
    Enable warning messages or suppress suppressible warning mes-
    sages.

--xref | -no-xref
    Control generation of a cross-reference in the listing file.
```

### AUTHOR

Author of Algol68G is Marcel van der Veer <algol68g@xs4all.nl>.

September 6, 2010



# Appendix E

## GNU General Public License

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

## APPENDICES

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

## TERMS AND CONDITIONS

### 0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

#### 1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require,

## APPENDICES

such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

### 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

### 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

### 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all

notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

#### 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”.
- (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
- (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

#### 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- (a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- (b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three

## APPENDICES

years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

- (c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- (d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- (e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning

of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

## 7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- (a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- (b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

## APPENDICES

- (c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- (d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- (e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- (f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

### 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.



Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

#### 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

#### 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

#### 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of

## APPENDICES

this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

## 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

## 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

## 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

## 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS

## APPENDICES

WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

### 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

### 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

## END OF TERMS AND CONDITIONS

### How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
```

```
Copyright (C) <textyear> <name of author>
```

```
This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
```

```
This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
```

## GNU GENERAL PUBLIC LICENSE

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see [<http://www.gnu.org/licenses/>](http://www.gnu.org/licenses/).

**Also add information on how to contact you by electronic and paper mail.**

**If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:**

```
<program> Copyright (C) <year> <name of author>
```

```
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.  
This is free software, and you are welcome to redistribute it  
under certain conditions; type 'show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

## APPENDICES

# Appendix F

## GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000, 2001, 2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “**Document**”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “**you**”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “**Modified Version**” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “**Secondary Section**” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “**Invariant Sections**” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD,



and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover

## APPENDICES

**Texts:** **Front-Cover Texts** on the front cover, and **Back-Cover Texts** on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
  - I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
  - J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
  - K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
  - L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
  - M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
  - N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to

## APPENDICES

the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## **7. AGGREGATION WITH INDEPENDENT WORKS**

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## **8. TRANSLATION**

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## **9. TERMINATION**

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

### **ADDENDUM: How to use this License for your documents**

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with ... Texts.” line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

# Index

- \***, 20, 25, 60, 218, 219, 221, 224, 256
- \*\***, 22, 78, 218
- +**, 25, 59, 64, 78, 224, 225
- +\***, 218, 219
- , 78
- /**, 22, 218, 219, 221, 256
- /=**, 26, 60, 216, 218, 220, 221, 223, 255
- :/=:**, 44
- :=:**, 44
- <**, 26, 60, 218, 220, 221
- <=**, 26, 60, 218, 220, 221, 223
- =**, 26, 216, 218, 219, 221, 223, 254
- >**, 26, 60, 218, 220, 221
- >=**, 26, 60, 218, 220, 221, 223
- ^**, 22, 24, 214
- ^=**, 60, 214
- ~**, 24, 214
- ~=**, 26, 60, 214
- declaration**, 15
- sign**, 35
- actual-declarer**, 34, 52, 64, 72
- applied-identifiers**, 27
- assertion**, 169
- assignments**, 165
- assignment**, 31, 35
- becomes-symbol**, 31
- bits-pattern**, 136
- bound**, 52
- call**, 88
- cast**, 44, 54, 148, 160
- choice-pattern**, 137
- closed-clause**, 28, 54, 85, 153
- collateral-clause**, 13
- comma-symbol**, 15
- comment**, 36
- complex-pattern**, 136
- conditional-clause**, 39, 153
- conformity-clause**, 109
- declarer**, 12
- denotation**, 11, 12, 27
- dyadic-operators**, 18
- enclosed-clause**, 32, 39, 46, 153
- enquiry-clause**, 41, 42, 46, 49
- field-selection**, 74
- formal-declarer**, 52, 72
- format-pattern**, 137
- general-pattern**, 133
- generator**, 29, 107
- go-on-symbol**, 15
- identifier**, 14, 110
- identity-declaration**, 13, 17, 19
- identity-relation**, 44, 156, 165
- indexer**, 56
- integer-case-clause**, 39, 45, 154
- integer-pattern**, 134
- jump**, 166, 167
- label**, 167
- loop-clause**, 39, 48, 153
- mode-declaration**, 74
- mode-indicant**, 14, 18
- monadic-operators**, 18, 21
- operand**, 18, 20, 41, 92
- parameter**, 34, 85
- pragmat**, 37
- primary**, 88
- priority-declaration**, 21
- real-pattern**, 135
- refinement**, 83, 172, 202
- replicator**, 130
- routine-texts**, 165
- routine-text**, 83
- row-display**, 13, 35, 53, 55, 61

## INDEX

**row-of-character-denotation**, 24  
**selection**, 69  
**serial-clause**, 27, 46, 168  
**sign-mould**, 134  
**skip**, 166  
**slice**, 56, 64  
**string-pattern**, 136  
**structure-display**, 68, 71  
**subscript**, 56  
**tertiary**, 65  
**trimmer**, 59  
**united-case-clause**, 108, 109, 154  
**unit**, 27, 31, 32, 48, 54, 84, 85, 90, 110  
**until-part**, 49

ABS, 19, 24, 25, 78, 80, 216, 219, 220, 222,  
223

Ackermann, 95

Ada, 6

airy ai, 251

airy ai derivative, 251

airy bi, 251

airy bi derivative, 251

Algol 68 Genie, xiv

alias, 30

alternative representation, 21

AND, 24, 43, 80, 214, 216, 223

ANDF, 43

ANDTH, 43

arccos, 23, 227

arccosh, 23, 228

arcsin, 23, 227

arcsinh, 23, 228

arctan, 23, 228

arctanh, 23, 228

ARG, 78, 220

argc, 271

argument list, 88

argv, 272

ASCII, 25

associate, 138, 180, 235

AT, 58

back-tracking, 95

backspace, 140, 242

balancing, 42, 44, 156

BEGIN, 54, 153

bessel exp il, 252

bessel exp in, 251

bessel exp inu, 252

bessel exp kl, 252

bessel exp kn, 252

bessel exp knu, 252

bessel in, 251

bessel inu, 252

bessel jl, 252

bessel jn, 251

bessel jnu, 252

bessel kn, 252

bessel knu, 252

bessel yl, 252

bessel yn, 251

bessel ynu, 252

beta, 251

BIN, 80, 222

bin possible, 237

binary file, 138

bit-wise operators, 80

BITS, 79, 209

bits lengths, 210

bits pack, 230

bits shorths, 210

bits width, 79, 211

blank, 24, 213

BOOL, 208

boolean-expression, 137

bound, 62

break, 231

bug reports, 649

BYTES, 79, 209

bytes lengths, 210

bytes pack, 230

bytes shorths, 210

bytes width, 211

cgs acre, 267

cgs angstrom, 265

cgs astronomical unit, 264



- cgs bar, 268
- cgs barn, 265
- cgs bohr magneton, 265
- cgs bohr radius, 265
- cgs boltzmann, 263
- cgs btu, 268
- cgs calorie, 268
- cgs canadian gallon, 267
- cgs carat, 267
- cgs curie, 269
- cgs day, 266
- cgs dyne, 270
- cgs electron charge, 264
- cgs electron magnetic moment, 265
- cgs electron volt, 265
- cgs erg, 270
- cgs faraday, 263
- cgs fathom, 266
- cgs foot, 266
- cgs footcandle, 269
- cgs footlambert, 269
- cgs gauss, 264
- cgs gram force, 268
- cgs grav accel, 264
- cgs gravitational constant, 264
- cgs hectare, 264
- cgs horsepower, 268
- cgs hour, 266
- cgs inch, 266
- cgs inch of mercury, 268
- cgs inch of water, 268
- cgs joule, 270
- cgs kilometres per hour, 264
- cgs kilopound force, 268
- cgs knot, 266
- cgs lambert, 269
- cgs light year, 264
- cgs liter, 267
- cgs lumen, 269
- cgs lux, 269
- cgs mass electron, 265
- cgs mass muon, 265
- cgs mass neutron, 265
- cgs mass proton, 265
- cgs meter of mercury, 268
- cgs metric ton, 267
- cgs micron, 264
- cgs mil, 266
- cgs mile, 266
- cgs miles per hour, 264
- cgs minute, 266
- cgs molar gas, 263
- cgs nautical mile, 266
- cgs newton, 270
- cgs nuclear magneton, 265
- cgs ounce mass, 267
- cgs parsec, 264
- cgs phot, 269
- cgs pint, 267
- cgs planck constant, 263
- cgs planck constant bar, 263
- cgs point, 271
- cgs poise, 269
- cgs pound force, 268
- cgs pound mass, 267
- cgs poundal, 268
- cgs proton magnetic moment, 265
- cgs psi, 269
- cgs quart, 267
- cgs rad, 269
- cgs roentgen, 269
- cgs rydberg, 265
- cgs solar mass, 264
- cgs speed of light, 263
- cgs standard gas volume, 263
- cgs std atmosphere, 268
- cgs stilb, 269
- cgs stokes, 269
- cgs texpoint, 271
- cgs therm, 268
- cgs ton, 267
- cgs torr, 268
- cgs troy ounce, 267
- cgs uk gallon, 267
- cgs uk ton, 267
- cgs unified atomic mass, 265
- cgs us gallon, 267
- cgs week, 266

## INDEX

cgs yard, 266  
CHANNEL, 116, 209, 233  
CHANNELS, 289  
CHAR, 24, 208  
char in string, 230  
**character denotation**, 24  
cholesky decomp, 260  
cholesky solve, 260  
CLEAR, 81  
clock, 232  
close, 119, 121, 236  
COL, 66  
**collateral elaboration**, 15  
**collaterally**, 32  
collect seconds, 232  
**collection**, 130  
collections, 232  
**column**, 55  
**comparison operator**, 26  
COMPL, 77  
**completer**, 168  
COMPLEX, 209  
complex arc cosh, 250  
complex arc sinh, 249  
complex arc tanh, 250  
complex arccos, 79, 229  
complex arccosh, 229  
complex arcsin, 79, 229  
complex arcsinh, 229  
complex arctan, 79, 229  
complex arctanh, 229  
complex cos, 79, 229  
complex cosh, 229, 250  
complex exp, 79, 228  
complex ln, 79, 228  
complex lu decomp, 257  
complex lu det, 258  
complex lu inv, 258  
complex lu solve, 258  
**complex number**, 78  
complex sin, 79, 228  
complex sinh, 229, 249  
complex sqrt, 79, 228  
complex tan, 79, 229  
complex tanh, 229, 250  
compressible, 238  
**concatenation**, 64  
CONJ, 78, 220  
**context**, 17  
**core dump**, 205, 649  
cos, 23, 227  
cosh, 23, 228  
cpu time, 232  
create, 235  
curses clear, 279  
curses columns, 280  
curses end, 279  
curses getchar, 279  
**curses library**, 279  
curses lines, 280  
curses move, 279  
curses putchar, 279  
curses refresh, 279  
curses start, 279  
curt, 23, 227  
  
d, 115  
debug, 232  
**deproceduring**, 18, 94, 147, 148  
**deproceduring coercion**, 87  
**dereferenced**, 34  
**dereferencing**, 31, 86, 94, 147, 148  
DET, 254  
DIAG, 66  
**diagnostic**, 188  
**dimension**, 55  
DIVAB, 32, 78, 215, 257  
**division**, 21  
DOWN, 157, 227  
draw aspect, 245  
draw background color, 247  
draw background color name, 247  
draw background colour, 247  
draw background colour name, 247  
draw circle, 248  
draw color, 247  
draw color name, 247  
draw colour, 247

- draw colour name, 247
- draw device, 244, 245
- draw erase, 245
- draw fill style, 245
- draw font name, 249
- draw font size, 249
- draw line, 248
- draw line style, 246
- draw linewidth, 246
- draw move, 245
- draw point, 247
- draw possible, 244
- draw rect, 248
- draw show, 245
- draw text, 248
- draw text angle, 249
- DYAD, 255
- dyadic, 21, 92
  
- elaborated collaterally, 33
- elaboration, 110
- ELEM, 58, 81, 223–225
- element, 51
- ELEMS, 57
- elliptic integral e, 253
- elliptic integral k, 253
- elliptic integral rc, 253
- elliptic integral rd, 253
- elliptic integral rf, 253
- elliptic integral rj, 253
- ELSE, 40
- ELSF, 43
- EMPTY, 87
- END, 54, 153
- end of file, 122
- end of file, 241
- end of format, 122
- end of line, 122
- end of line, 241
- end of page, 122
- end-of-line, 121
- engineers notation, 126
- ENTIER, 19, 219
- environment enquiry, 79
  
- EQ, 214
- erf, 250
- erfc, 250
- errno, 272
- error char, 125, 213
- establish, 120, 180, 235
- evaluate, 232
- event-driven, 116
- example, 31
- execve, 273
- execve child, 274
- execve child pipe, 274
- execve output, 274
- EXIT, 168
- exp, 23, 227
- exp width, 212
  
- factorial, 250
- FALSE, 23, 208
- Fast Fourier Transform, 96
- fft backward, 262
- fft complex backward, 262
- fft complex forward, 262
- fft complex inverse, 262
- fft forward, 262
- fft inverse, 262
- FI, 40
- field, 67
- field selector, 68, 69
- FILE, 209, 233
- file, 116
- file is block device, 272
- file is char device, 272
- file is directory, 272
- file is fifo, 272
- file is link, 272
- file is regular, 272
- file mode, 272
- file open error, 122
- firm context, 34, 94, 107
- firmly related, 94, 107
- fixed, 125, 240, 321
- flat row, 53, 54
- FLEX, 62, 90

## INDEX

- flexible, 61
- flip, 213
- float, 125, 179, 240, 321
- flop, 213
- fork, 273
- formal-declarer, 64
- FORMAT, 128, 181, 209
- format, 128
- format error, 122
- formatted transput, 128
- fractional part, 19
- frame, 134
- free format, 27
  
- gamma, 250
- garbage, 232
- garbage collector, 30, 44
- GE, 215
- get, 118, 138, 236
- get bin, 138, 237
- get directory, 273
- get env, 272
- get int, 179
- get possible, 238
- get pwd, 273
- get sound, 290
- getf, 128, 181, 237
- global, 29
- grep in string, 276
- GT, 215
  
- HEAP, 30
- http content, 275
  
- I, 78, 214
- IBM 360, 79
- identification, 94
- idf, 139, 179, 232, 238
- IEEE-754 compatibility, 205
- IFIP, 4
- IM, 77, 220
- incarnation, 97
- incomplete beta, 251
- incomplete gamma, 250
- INT, 20, 208
  
- int lengths, 210
- int shorths, 210
- int width, 211
- integer-mould, 134
- INTYPE, 118
- INV, 254
- inverse erf, 250
- inverse erfc, 250
- IS, 44
- is alnum, 230
- is alpha, 230
- is cntrl, 231
- is digit, 231
- is graph, 231
- is lower, 231
- is print, 231
- is punct, 231
- is space, 231
- is upper, 231
- is xdigit, 231
- ISNT, 44
  
- Java, 175
  
- laplace, 262
- last char in string, 230
- LE, 214
- legacy implementations, 648
- LENG, 217, 219, 220, 222, 225
- LEVEL, 157, 226, 227
- Linux, 271
- Lisp, 101, 103
- ln, 23, 227
- ln gamma, 250
- LOC, 29
- local, 29
- local time, 271
- lock, 139, 236
- log, 23, 227
- LONG BITS, 209
- long bits width, 211
- LONG BYTES, 209
- long bytes width, 212
- LONG COMPLEX, 209

- long exp width, 212
- LONG INT, 208
- long int width, 211
- LONG LONG BITS, 209
- long long bits width, 211
- LONG LONG COMPLEX, 209
- long long exp width, 212
- LONG LONG INT, 208
- long long int width, 211
- long long max int, 211
- long long max real, 211
- long long min real, 211
- long long pi, 212
- LONG LONG REAL, 208
- long long real width, 212
- long long small real, 211
- long max int, 210
- long max real, 211
- long min real, 211
- long pi, 212
- LONG REAL, 208
- long real width, 212
- long small real, 211
- LT, 215
- lu decomp, 257
- lu det, 258
- lu inv, 258
- lu solve, 258
- LWB, 57
  
- make device, 245
- make term, 121, 138, 237, 238
- MATLAB**, 175
- max abs char, 24, 25, 213
- max int, 12, 210
- max real, 15, 211
- meek context**, 154
- meek-integral-unit**, 59
- min real, 15, 211
- MINUSAB, 32, 78, 215, 255
- mksa acre, 267
- mksa angstrom, 265
- mksa astronomical unit, 264
- mksa bar, 268
- mksa barn, 265
- mksa bohr magneton, 265
- mksa bohr radius, 265
- mksa boltzmann, 263
- mksa btu, 268
- mksa calorie, 268
- mksa canadian gallon, 267
- mksa carat, 267
- mksa curie, 269
- mksa day, 266
- mksa dyne, 270
- mksa electron charge, 264
- mksa electron magnetic moment, 265
- mksa electron volt, 265
- mksa erg, 270
- mksa faraday, 263
- mksa fathom, 266
- mksa foot, 266
- mksa footcandle, 269
- mksa footlambert, 269
- mksa gauss, 264
- mksa gram force, 268
- mksa grav accel, 264
- mksa gravitational constant, 264
- mksa hectare, 264
- mksa horsepower, 268
- mksa hour, 266
- mksa inch, 266
- mksa inch of mercury, 268
- mksa inch of water, 268
- mksa joule, 270
- mksa kilometres per hour, 264
- mksa kilopound force, 268
- mksa knot, 266
- mksa lambert, 269
- mksa light year, 264
- mksa liter, 267
- mksa lumen, 269
- mksa lux, 269
- mksa mass electron, 265
- mksa mass muon, 265
- mksa mass neutron, 265
- mksa mass proton, 265
- mksa meter of mercury, 268

## INDEX

- mksa metric ton, 267
- mksa micron, 264
- mksa mil, 266
- mksa mile, 266
- mksa miles per hour, 264
- mksa minute, 266
- mksa molar gas, 263
- mksa nautical mile, 266
- mksa newton, 270
- mksa nuclear magneton, 265
- mksa ounce mass, 267
- mksa parsec, 264
- mksa phot, 269
- mksa pint, 267
- mksa planck constant, 263
- mksa planck constant bar, 263
- mksa point, 271
- mksa poise, 269
- mksa pound force, 268
- mksa pound mass, 267
- mksa poundal, 268
- mksa proton magnetic moment, 265
- mksa psi, 269
- mksa quart, 267
- mksa rad, 269
- mksa roentgen, 269
- mksa rydberg, 265
- mksa solar mass, 264
- mksa speed of light, 263
- mksa standard gas volume, 263
- mksa std atmosphere, 268
- mksa stilb, 269
- mksa stokes, 269
- mksa texpoint, 271
- mksa therm, 268
- mksa ton, 267
- mksa torr, 268
- mksa troy ounce, 267
- mksa uk gallon, 267
- mksa uk ton, 267
- mksa unified atomic mass, 265
- mksa us gallon, 267
- mksa vacuum permeability, 263
- mksa vacuum permittivity, 263
- mksa week, 266
- mksa yard, 266
- MOD, 21, 215
- MODAB, 32, 215
- mode, 51
- monad, 93
- monadic, 92
- monitor, 190, 194, 197, 231, 232
- monitor, 232
- multiplication, 20
- mutex, 157
- name, 28, 29, 35, 91, 151
- NE, 26, 60, 214
- necessary environment, 92
- nesting, 86
- NEW, 30
- new line, 13, 53, 118, 120, 130, 242
- new page, 53, 118, 120, 130, 242
- new sound, 289
- NIL, 28, 36, 44, 176
- nomad, 93
- NORM, 254
- NOT, 24, 80, 216, 222
- null character, 213
- num atto, 270
- num avogadro, 263
- num exa, 270
- num femto, 270
- num fine structure, 265
- num giga, 270
- num kilo, 270
- num mega, 270
- num micro, 270
- num milli, 270
- num nano, 270
- num peta, 270
- num pico, 270
- num tera, 270
- num yocto, 270
- num yotta, 270
- num zepto, 270
- num zetta, 270
- ODD, 24, 217

- OF, 69
- on char error mended, 181
- on file end, 122, 180, 238
- on format end, 239
- on format error, 240
- on line end, 123, 239
- on logical file end, 122, 238
- on open error, 239
- on page end, 123, 239
- on physical file end, 122, 238
- on transput error, 240
- on value error, 239
- open, 118, 235
- operator, 18
- operator overloading, 94
- operator symbol, 93
- OR, 24, 27, 43, 80, 216, 223
- order of elaboration, 20
- OREL, 43
- ORF, 43
- orthogonality, 21
- OUSE, 47
- OUT CASE, 47
- OUTTYPE, 120
- OVER, 21, 215
- OVERAB, 32, 215
- overflow, 20
  
- PAR, 157
- parallel, 15
- parallel-clause, 178
- parentheses, 54, 89
- Pascal, 175
- PCM/WAVE, 290
- PDP 10, 79
- Perl, 175
- pi, 17, 212
- picture, 129
- PIPE, 209, 271
- plain value, 24, 51
- PLUSAB, 64, 78, 215, 255
- PLUSTO, 64, 215
- pq backed pid, 288
- pq cmd status, 287
- pq cmd tuples, 287
- pq connect db, 281
- pq db, 287
- pq error message, 287
- pq exec, 285
- pq fformat, 286
- pq finish, 284
- pq fname, 286
- pq fnumber, 286
- pq get is null, 286
- pq get value, 286
- pq host, 287
- pq nfields, 286
- pq ntuples, 286
- pq options, 287
- pq parameter status, 285
- pq pass, 287
- pq port, 287
- pq protocol version, 288
- pq reset, 285
- pq result error message, 287
- pq server version, 288
- pq socket, 288
- pq user, 287
- pre emptive sweep, 232
- pre emptive sweep heap, 232
- preprocessor, 201
- prime factors, 261
- print, 13, 23, 34, 53, 107, 108, 121, 236
- print bits, 243
- print bool, 243
- print char, 243
- print complex, 243
- print int, 179, 243
- print long bits, 243
- print long complex, 243
- print long int, 243
- print long long bits, 243
- print long long complex, 243
- print long long int, 243
- print long long real, 243
- print long real, 243
- print real, 243
- print string , 243

## INDEX

printf, 128, 181, 237  
PRIO, 93  
priority, 21, 22, 26, 27, 32, 60, 92, 93  
proceduring, 179  
pseudo-operator, 43  
put, 120, 138, 236  
put bin, 138, 237  
put int, 179  
put possible, 238  
putf, 128, 181, 236  
Python, 175

qr decomp, 259  
qr ls solve, 260  
qr solve, 259

radix, 80  
random number  
    random number  
    next random, 229  
random numbers  
    random numbers  
    first random, 229  
range, 27, 28, 40, 41, 46, 48, 49  
RATE, 289  
RE, 77, 220  
reach, 27  
read, 53, 107, 108, 119, 236  
read bin, 138, 237  
read bits, 243  
read bool, 23, 243  
read char, 24, 243  
read complex, 243  
read int, 14, 179, 242  
read long bits, 243  
read long complex, 243  
read long int, 14, 242  
read long long bits, 243  
read long long complex, 243  
read long long int, 15, 242  
read long long real, 18, 243  
read long real, 18, 243  
read real, 18, 243  
read string, 243

read-only, 117  
readf, 128, 181, 237  
reading, 116  
REAL, 20, 208  
real, 125, 179, 241, 321  
real lengths, 210  
real shorths, 210  
real width, 212  
recursion, 95  
REF, 28  
reidf possible, 238  
remainder, 21  
REPR, 25, 223  
reset, 139  
reset, 139, 180, 241  
reset errno, 272  
reset possible, 238  
RESOLUTION, 289  
rewind, 139  
rewind, 241  
rewind possible, 238  
RIFF, 290  
ROUND, 19, 219  
routine, 83  
routine-text, 90  
ROW, 66  
row, 51, 55  
rowing, 147  
rowing coercion, 54, 60  
Ruby, 175

SAMPLES, 289  
scientific format, 125  
scope, 29, 91  
scope checking, 91  
scratch, 139, 236  
seconds, 232  
selector, 109  
SEMA, 157, 209  
semaphore, 157  
sentinel, 101  
SET, 81  
set, 139, 241  
set possible, 140



- set possible, 238
- set pwd, 273
- set sound, 290
- shielded**, 111
- SHL, 81, 223
- SHORTEN, 217, 219, 221, 222, 225
- SHR, 81, 223
- SIGN, 19, 217, 219
- SIMPLIN, 108, 118, 181
- SIMPLOUT, 108, 120, 181
- sin, 23, 227
- sinh, 23, 228
- SKIP, 166
- small real, 15, 211
- soft context**, 31, 44, 87
- SORT, 230
- SOUND, 209, 289
- space**, 181
- space, 130, 140, 242
- specifier**, 110
- sqrt, 23, 227
- stack pointer, 233
- stand back, 116, 235
- stand back channel, 116, 233
- stand draw channel, 117, 244
- stand error, 116, 235
- stand error channel, 116, 233
- stand in, 116, 235
- stand in channel, 116, 233
- stand out, 116, 121, 235
- stand out channel, 116, 233
- standard-prelude**, 12, 18, 24–26, 31, 60, 63, 64, 77
- stderr, 235
- stdin, 235
- stdout, 235
- straightening**, 72, 118, 120, 126
- strerror, 272
- STRING, 63, 72, 75, 208
- string in string, 230
- string terminator**, 121
- strong context**, 31, 44, 54, 68, 77, 86, 121, 151
- STRUCT, 67
- structural equivalence**, 113
- structure**, 67
- sub in string, 276
- sv decomp, 258
- svd solve, 259
- sweep heap, 232
- syntax error**, 189
- system, 231
- system stack pointer, 233
- system stack size, 233
  
- T, 254
- tan, 23, 228
- tanh, 23, 228
- tcp request, 276
- term, 179, 238
- test suite**, 649
- THEF, 43
- THEN, 40
- times ten to the power symbol**, 16
- TIMESAB, 32, 78, 215, 256
- TO, 48
- to lower, 231
- to upper, 231
- TRACE, 254
- transient name**, 62
- transput error**, 122
- TRNSP, 66
- TRUE, 23, 208
  
- UNESCO, 4
- UNION, 105, 111
- uniting**, 94, 106, 147
- unworthy character**, 202
- UP, 22, 157, 227
- UPB, 57
- utc time, 271
  
- value**, 14
- value error**, 122
- VOID, 14, 28, 87, 151, 208
- voiding**, 87, 147
  
- wait pid, 273
- weak context**, 149

## *INDEX*

weak dereferencing, 59, 67, 71  
well-formed, 110  
WHILE, 49  
whole, 125, 240, 321  
widened, 54  
widening, 18, 19, 21, 31, 147  
widening coercion, 26  
WIN32 binaries, xiv  
write, 121  
write bin, 138, 237  
write-only, 117  
writef, 181  
writing, 116  
  
XOR, 24, 80, 216  
  
yang, 112  
yield, 91  
yin, 112