UNISYS

5000 and 7000 Series Operating Systems

Programming Guide

Priced Item

September 1987
Printed in U S America
UP-11221 Rev. 1

Chapter 2

C Language

Chapter 2

C LANGUAGE

LEXICAL CONVENTIONS

There are six classes of tokens - identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

Comments

The characters /* introduce a comment which terminates with the characters */. Comments do not nest.

Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are different. Although there is no limit on the length of a name, only initial characters are significant: at least eight characters of a non-external name, and perhaps fewer for external names. Moreover, some implementations may collapse case distinctions for external names. The external name sizes include:

5000 Series	>100 characters, 2 cases
7000 Series	>100 characters, 2 cases

Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

auto	**do**	**for**	**return**	**typedef**
break	**double**	**goto**	**short**	**union**
case	**else**	**if**	**sizeof**	**unsigned**
char	**enum**	**int**	**static**	**void**
continue	**external**	**long**	**struct**	**while**
default	**float**	**register**	**switch**	

Some implementations also reserve the words **fortran** and **asm**.

Constants

There are several kinds of constants. Each has a type; an introduction to types is given in "NAMES." Hardware characteristics that affect sizes are summarized in "Hardware Characteristics" under "LEXICAL CONVENTIONS."

Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero). An octal constant consists of the digits **0** through **7** only. A sequence of digits preceded by **0x** or **0X** (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include **a** or **A** through **f** or **F** with values 10 through 15. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be **long**. Otherwise, integer constants are **int**.

Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by **l** (letter ell) or **L** is a long constant.

Character Constants

A character constant is a character enclosed in single quotes, as in '**x**'. The value of a character constant is the numerical value of the character in the machine's character set.

Certain nongraphic characters, the single quote (') and the backslash (\), may be represented according to the following table of escape sequences:

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	*ddd*	*ddd*

The escape *ddd* consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is **\0** (not followed by a digit), which indicates the character **NUL**. If the character following a backslash is not one of those specified, the behavior is undefined. A new-line character is illegal in a character constant. The type of a character constant is **int**.

Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant has type **double**.

Enumeration Constants

Names declared as enumerators (see "Structure, Union, and Enumeration Declarations" under "DECLARATIONS") have type **int**.

Strings

A string is a sequence of characters surrounded by double quotes, as in "...". A string has type "array of **char**" and storage class **static** (see "NAMES") and is initialized with the given characters. The compiler places a null byte (**0**) at the end of each string so that programs which scan the string can find its end. In a string, the double quote character (") must be preceded by a \; in addition, the same escapes as described for character constants may be used.

A \ and the immediately following new-line are ignored. All strings, even when written identically, are distinct.

Hardware Characteristics

The following figure summarizes certain hardware properties.

5000 and 7000 Series
(ASCII)

char	8 bits	
int	32	
short	16	
long	32	
float	32	
double	64	
float range	± 10	± 38
double range	± 10	± 308*

*On 7000 double range is $\pm 10^{\pm 38}$

Figure 2-1. HARDWARE CHARACTERISTICS

SYNTAX NOTATION

Syntactic categories are indicated by *italic* type and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript "opt," so that

$$\{\ \textit{expression}\ _{\textit{opt}}\ \}$$

indicates an optional expression enclosed in braces. The syntax is summarized in "SYNTAX SUMMARY".

NAMES

The C language bases the interpretation of an identifier upon two attributes of the identifier - its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

Storage Class

There are four declarable storage classes:

Automatic
Static
External
Register

Automatic variables are local to each invocation of a block (see "Compound Statement or Block" in "STATEMENTS") and are discarded upon exit from the block. Static variables are local to a block but retain their

values upon reentry to a block even after control has left the block. External variables exist and maintain their values throughout the execution of the entire program and may be used for communications between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on the exit from the block.

Type

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implemenation's character set. If a genuine character from that character set is stored in a **char** variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables. In particular, **char** may be signed or unsigned by default.

Up to three sizes of integer, declared **short int, int,** and **long int,** are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs.

The properties of **enum** types (see "Structure, Union, and Enumeration Declarations" under "DECLARATIONS") are identical to those of some integer types. The implementation may use the range of values to determine how to allot storage.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation.

Chapter 5

COMPILER AND C LANGUAGE

This chapter describes the UNIX System's C compiler, **cc**, and the C programming language that the compiler translates.

The C compiler converts C programs into assembly language programs that are ultimately translated into object files by the assembler, **as**. The link editor, **ld**, collects and merges object files into executable load modules. Each of these tools preserves all symbolic information necessary for meaningful symbolic testing at C-language source level. In addition, a utility package aids in testing and debugging.

The current manual page for the C compiler can be obtained with the SGS command:

```
man cc
```

USE OF THE COMPILER

To use the compiler, first create a file (typically by using the UNIX system text editor) containing C source code. The name of the file created must have a special format; the last two characters of the file name must be **.c** as in *file1.c*.

Next, enter the command

cc *options file.c*

to invoke the compiler on the C source file *file.c* with the appropriate

options selected. The compilation process creates an absolute binary file named **a.out** that reflects the contents of *file.c* and any referenced library routines. The resulting binary file, **a.out**, can then be executed on the target system.

Options can control the steps in the compilation process. When none of the controlling options are used, and only one file is named, **cc** automatically calls the assembler, **as**, and the link editor, **ld**, thus resulting in an executable file, named **a.out**. If more than one file is named in a command,

cc *file1.c file2.c file3.c*

then the output will be placed on files *file1.o*, *file2.o*, and *file3.o*. These files can then be linked and executed through the **ld** command.

The **cc** compiler also accepts input file names with the last two characters **.s**. The **.s** signifies a source file in assembly language. The **cc** compiler passes this type of file directly to **as**, which assembles the file and places the output on a file of the same name with **.o** substituted for **.s**.

Cc is based on a portable C compiler and translates C source files into assembly code. Whenever the command **cc** is used, the standard C preprocessor (which resides on the file **/lib/cpp**) is called. The preprocessor performs file inclusion and macro substitution. The preprocessor is always invoked by **cc** and need not be called directly by the programmer. Then, unless the appropriate flags are set, **cc** calls the assembler and the link editor to produce an executable file.

COMPILER OPTIONS

All options recognized by the **cc** command are listed below:

Option	*Argument*	*Description*
-c	none	Suppress the link-editing phase of compilation and force an object file to be produced even if only one file is compiled.
-g	none	Produce symbolic debugging information.
-p	none	Reserved for invoking a profiler.
-D	*identifier[=constant]*	Define the external symbol *identifier* to the preprocessor, and give it the value *constant* (if specified).
-E	none	Same as the **-P** option except output is directed to the standard output.
-I	*directory*	Change the algorithm that searches for **#include** files whose names do not begin with / to look in the named *directory* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" are searched for first in the directory of the file being compiled, then in directories named by the **-I** options, and last in directories on the standard list. For **#include** files whose names are enclosed in <>, the directory of the *file* argument is not searched.

-O	none	Invoke an object code optimizer.
-P	none	Suppress compilation and loading; i.e., invoke only the preprocessor and leave out the output on corresponding files suffixed **.i**.
-U	*identifier*	Undefine the named *identifier* to the preprocessor.
-V	none	Print the version of the assembler that is invoked.
-W	*c,arg1[,arg2...]*	Pass along the argument(s) *argi* to pass *c*, where *c* is one of [**p012al**], indicating preprocessor, compiler first pass, compiler second pass, optimizer, assembler, or link editor, respectively.

This part provides additional information for those options not completely described above.

By using appropriate options, compilation can be terminated early to produce one of several intermediate translations such as relocatable object files (**-c option**), assembly source expansions for C code (**-S** option), or the output of the preprocessor (**-P** option). In general, the intermediate files may be saved and later resubmitted to the **cc** command, with other files or libraries included as necessary.

When compiling C source files, the most common practice is to use the **-c** option to save relocatable files. Subsequent changes to one file do not then require that the others be recompiled. A separate call to **cc** without the **-c** option then creates the linked executable **a.out** file. A relocatable object file created under the **-c** option is named by adding a **.o** suffix to the source file name.

The **-W** option provides the mechanism to specify options for each step that is normally invoked from the **cc** command line. These steps are preprocessing, the first pass of the compiler, the second pass of the compiler, optimization, assembly, and link editing. At this time,

only assembler and link editor options can be used with the **-W** option. The most common example of use of the **-W** option is "**-W***a,-m*", which passes the **-m** option to the assembler. Specifying "**-wl,-m**" passes the **-m** option to the link editor.

When the **-P** option is used, the compilation process stops after only preprocessing, with output left on *file.i*. This file will be unsuitable for subsequent processing by **cc**.

The **-O** option decreases the size and increases the execution speed of programs by moving, merging, and deleting code. However, line numbers used for symbolic debugging may be transposed when the optimizer is used.

The **-g** option produces information for a symbolic debugger.

Chapter 6

A C Program Checker—"lint"

Chapter 6

A C PROGRAM CHECKER—"lint"

GENERAL

The **lint** program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful or error prone constructions which nevertheless are legal. The **lint** program accepts multiple input files and library specifications and checks them for consistency.

Usage

The **lint** command has the form:

lint [options] files ... library-descriptors ...

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c** or **.ln**; and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the **lint** command are:

-a Suppress messages about assignments of long values to variables that are not long.

-b Suppress messages about break statements that cannot be reached.

-c Only check for intra-file bugs; leave external information in files suffixed with **.ln**.

-h Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).

-n Do not check for compatibility with either the standard or the portable **lint** library.

-O *name* Create a lint library from input files named **llib-l***name***.ln**.

-p Attempt to check portability to other dialects of C language.

-u Suppress messages about function and external variables used and not defined or defined and not used.

-v Suppress messages about unused arguments in functions.

-x Do not report variables referred to by external declarations but never used.

When more than one option is used, they should be combined into a single argument, such as, **-ab** or **-xha**.

The names of files that contain C language programs should end with the suffix **.c** which is mandatory or **lint** and the C compiler.

The **lint** program accepts certain arguments, such as:

```
-ly
```

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function

return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED comments can be used to specify features of the library functions.

The **lint** library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file but are not used on a source file do not result in messages. The **lint** program does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file which contains descriptions of the programs which are normally loaded when a C language program is run. When the **-p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **-n** option can be used to suppress all library checking.

TYPES OF MESSAGES

The following paragraphs describe the major categories of messages printed by **lint**.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

The **lint** program prints messages about variables and functions which are defined but not otherwise mentioned. An exception is

variables which are declared through explicit **extern** statements but are never referenced; thus the statement

```
extern double sin();
```

will evoke no comment if **sin** is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest and can be discovered by using the **-x** option with the **lint** command.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **-v** option is available to suppress the printing of messages about unused arguments. When **-v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

Messages about unused arguments can be suppressed for one function by adding the comment:

```
/* ARGSUSED */
```

to the program before the function. This has the effect of the **-v** option for only one function. Also, the comment:

```
/* VARARGS */
```

can be used to suppress messages about variable number of arguments in calls to a function. The comment should be added before the function definition. In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit giving the number of arguments which should be checked. For example:

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

There is one case where information about unused or undefined variables is more distracting than helpful. This is when **lint** is applied to some but not all files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used. Conversely, many functions and variables defined elsewhere may be used. The **-u** option may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

The **lint** program attempts to detect cases where a variable is used before it is set. The **lint** program detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use", since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement since the true flow of control need not be discovered. It does mean that **lint** can print messages about some programs which are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables which are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

Flow of Control

The **lint** program attempts to detect unreachable portions of the programs which it processes. It will print messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom and to recognize the special cases **while(1)** and **for(;;)** as infinite loops. The **lint** program also prints messages about loops which cannot be entered at the top. Some valid

programs may have such loops which are considered to be bad style at best and bugs at worst.

The **lint** program has no way of detecting functions which are called and never returned. Thus, a call to **exit** may cause an unreachable code which **lint** does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program cannot be reached but it is not apparent to **lint**, the comment

```
/* NOTREACHED */
```

can be added at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached.

The **lint** program will not print a message about unreachable **break** statements. Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements. The **−O** option in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the **lint** output. If these messages are desired, **lint** can be invoked with the **−b** option.

Function Values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function "values" that have never been returned. The **lint** program addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; the **lint** program will give the message

function *name* contains return(e) and return

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
        if ( a ) return ( 3 );
        g ();
        }
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a message from **lint**. If *g*, like **exit**, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem.

Type Checking

The **lint** program enforces the type checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- Across certain binary operators and implied assignments
- At the structure selection operators

- Between the definition and uses of functions

- In the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (**?:**), and relational operators have this property. The argument of a **return** statement and expressions used in initialization suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the -> be a pointer to structure, the left operand of the . be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

If it is desired to turn off strict type checking for an expression, the comment

```
/* NOSTRICT */
```

should be added to the program immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. The **lint** program will print a message as a result of detecting this. Consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signaled his intentions. It seems harsh for **lint** to continue to print messages about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **−c** flag controls the printing of comments about casts. When **−c** is in effect, casts are treated as though they were assignments subject to messages; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable Character Use

On some systems, characters are signed quantities with a range from -128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
    ...
if( (c = getchar( )) < 0 ) ...
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* as an integer since **getchar** is actually returning integer values. In any case, **lint** will print the message "nonportable character comparison".

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**

Assignments of "longs" to "ints"

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, which are truncated. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **-a** option.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **-h** option is used to supress these checks. For example, in the statement

```
*p++ ;
```

the * does nothing. This provokes the message "null effect" from **lint**. The following program fragment:

```
unsigned x ;
if( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

s equivalent to

```
if( x != 0 )
```

which may not be the intended action. The **lint** program will print the message "degenerate unsigned comparison" in these cases. If a program contains something similar to

```
if( 1 != 0 ) ...
```

lint will print the message "constant in conditional context" since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statement

```
if( x&077 == 0 ) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and **lint** encourages this by an appropriate message.

Finally, when the **-h** option has not been used, **lint** prints messages about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal but is considered to be bad style, usually unnecessary, and frequently a bug.

Old Syntax

Several forms of older syntax are now illegal. These fall into two classes - assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, ...) could cause ambiguous expressions, such as:

```
a =-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., +=, -=, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize *x* to 1. This also caused syntactic difficulties. For example, the initialization

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x ( y ) { . .
```

and the compiler must read past *x* in order to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. The **lint** program tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

The **lint** program checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will cause **lint** to print the message

```
warning: i evaluation order undefined
```

in order to call attention to this condition.

Chapter 7

Symbolic Debugging Program—"sdb"

Chapter 7

SYMBOLIC DEBUGGING PROGRAM—"sdb"

GENERAL

This chapter describes the symbolic debugger **sdb**(1) as implemented for C language and Fortran 77 programs on the UNIX operating system. The **sdb** program is useful both for examining "core images" of aborted programs and for providing an environment in which execution of a program can be monitored and controlled.

The **sdb** program allows interaction with a debugged program at the source language level. When debugging a core image from an aborted program, **sdb** reports which line in the source program caused the error and allows all variables to be accessed symbolically and to be displayed in the correct format.

Breakpoints may be placed at selected statements or the program may be single stepped on a line-by-line basis. To facilitate specification of lines in the program without a source listing, **sdb** provides a mechanism for examining the source text. Procedures may be called directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines which provided formatted printout of structured data.

USAGE

In order to use **sdb** to its full capabilities, it is necessary to compile the source program with the **-g** option. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **-g** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.

A typical sequence of **shell** commands for debugging a core image is

```
$ cc -g prgm.c -o prgm
$ prgm
Bus error - core dumped
$ sdb prgm
main:25:      x[i] = 0;
*
```

The program **prgm** was compiled with the **-g** option and then executed. An error occurred which caused a core dump. The **sdb** program is then invoked to examine the core dump to determine the cause of the error. It reports that the bus error occurred in function *main* at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. The **sdb** program then prompts the user with an * indicating that it awaits a command.

It is useful to know that **sdb** has a notion of current function and current line. In this example, they are initially set to *main* and "25", respectively.

In the above example, **sdb** was called with one argument, *prgm*. In general, it takes three arguments on the command line. The first is the name of the executable file which is to be debugged; it defaults to *a.out* when not specified. The second is the name of the core file, defaulting to *core*; and the third is the name of the directory containing the source of the program being debugged. The **sdb** program currently requires all source to reside in a single directory. The default is the working directory. In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

It is possible that the error occurred in a function which was not compiled with the **-g** option. In this case, **sdb** prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in *main*. The **sdb** program will print an error message if *main* was not compiled with the **-g** option, but debugging can continue for those routines compiled with the **-g** option. Figure 7-1 shows a typical example of **sdb** usage (see page 7-16).

Printing a Stack Trace

It is often useful to obtain a listing of the function calls which led to the error. This is obtained with the **t** command. For example:

```
*t
sub(x=2,y=3)     [prgm.c:25]
inter(i=16012)     [prgm.c:96]
main(argc=1,argv=0x7fffff54,envp=0x7fffff5c)[prgm.c:15]
```

This indicates that the error occurred within the function *sub* at line 25 in file *prgm.c*. The *sub* function was called with the arguments x=2 and y=3 from *inter* at line 96. The *inter* function was called from *main* at line 15. The *main* function is always called by the **shell** with three arguments often referred to as *argc*, *argv*, and *envp*. Note that *argv* and *envp* are pointers, so their values are printed in hexadecimal.

Examining Variables

The **sdb** program can be used to display variables in the stopped program. Variables are displayed by typing their name followed by a slash, so

```
*errflag/
```

causes **sdb** to display the value of variable *errflag*. Unless otherwise specified, variables are assumed to be either local to or accessible from the current function. To specify a different function, use the form

```
*sub:i/
```

to display variable *i* in function *sub*. F77 users can specify a common block variable in the same manner.

The **sdb** program supports a limited form of pattern matching for variable and function names. The symbol * is used to match any sequence of characters of a variable name and ? to match any single character. Consider the following commands

```
*x*/
*sub:y?/
**/
```

The first prints the values of all variables beginning with *x*, the second prints the values of all two letter variables in function *sub* beginning with *y*, and the last prints all variables. In the first and last examples, only variables accessible from the current function are printed. The command

```
**.*/
```

displays the variables for each function on the call stack.

The **sdb** program normally displays the variable in a format determined by its type as declared in the source program. To request a different format, a specifier is placed after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are:

b	One byte
h	Two bytes (half word)
l	Four bytes (long word).

The lengths are effective only with the formats **d**, **o**, **x**, and **u**. If no length is specified, the word length of the host machine is used. A numeric length specifier may be used for the **s** or **a** commands. These commands normally print characters until either a null is reached or 128 characters are printed. The number specifies how many characters should be printed.

There are a number of format specifiers available:

c	Character.
d	Decimal.
u	Decimal unsigned.
o	Octal.
x	Hexadecimal.
f	32-bit single-precision floating point.
g	64-bit double-precision floating point.
s	Assume variable is a string pointer and print characters starting at the address pointed to by the variable until a null is reached.
a	Print characters starting at the variable's address until a null is reached.
p	Pointer to function.
i	Interpret as a machine-language instruction.

For example, the variable *i* can be displayed with

```
*i/x
```

which prints out the value of *i* in hexadecimal.

The **sdb** program also knows about structures, arrays, and pointers so that all of the following commands work.

```
*array[2][3]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

The only restriction is that array subscripts must be numbers. Depending on your machine, accessing arrays may be limited to 1-dimensional arrays. Note that as a special case:

```
*psym->/d
```

displays the location pointed to by *psym* in decimal.

Core locations can also be displayed by specifying their absolute addresses. The command

```
*1024/
```

displays location 1024 in decimal. As in C language, numbers may also be specified in octal or hexadecimal so the above command is equivalent to both

```
*02000/
```

and

```
*0x400/
```

It is possible to mix numbers and variables so that

```
*1000.x/
```

refers to an element of a structure starting at address 1000, and

```
*1000->x/
```

refers to an element of a structure whose address is at 1000. For commands of the type *1000.x/ and *1000->x/, the **sdb** program uses the structure template of the last structured referenced.

The address of a variable is printed with the =, so

```
*i=
```

displays the address of *i*. Another feature whose usefulness will become apparent later is the command

```
*./
```

which redisplays the last variable typed.

SOURCE FILE DISPLAY AND MANIPULATION

The **sdb** program has been designed to make it easy to debug a program without constant reference to a current source listing. Facilities are provided which perform context searches within the source files of the program being debugged and to display selected portions of the source files. The commands are similar to those of the UNIX system text editor **ed**(1). Like the editor, **sdb** has a notion of current file and line within the file. The **sdb** program also knows how the lines of a file are partitioned into functions, so it also has a notion of current function. As noted in other parts of this document, the current function is used by a number of **sdb** commands.

Displaying the Source File

Four commands exist for displaying lines in the source file. They are useful for perusing the source program and for determining the context of the current line. The commands are:

p	Prints the current line.
w	Window; prints a window of ten lines around the current line.
z	Prints ten lines starting at the current line. Advances the current line by ten.
control-d	Scrolls; prints the next ten lines and advances the current line by ten. This command is used to cleanly display long segments of the program.

When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file but is also used as input by some **sdb** commands.

Changing the Current Source File or Function

The **e** command is used to change the current source file. Either of the forms

```
*e function
*e file.c
```

may be used. The first causes the file containing the named function to become the current file, and the current line becomes the first line of the function. The other form causes the named file to become current. In this case, the current line is set to the first line of the named file. Finally, an **e** command with no argument causes the current function and file named to be printed.

Changing the Current Line in the Source File

The **z** and **control-d** commands have a side effect of changing the current line in the source file. The following paragraphs describe other commands that change the current line.

There are two commands for searching for instances of regular expressions in source files. They are

```
*/regular expression/
*?regular expression?
```

The first command searches forward through the file for a line containing a string that matches the regular expression and the second searches backwards. The trailing / and **?** may be omitted from these commands. Regular expression matching is identical to that of **ed**(1).

The + and - commands may be used to move the current line forwards or backwards by a specified number of lines. Typing a

new-line advances the current line by one, and typing a number causes that line to become the current line in the file. These commands may be combined with the display commands so that

```
*+15z
```

advances the current line by 15 and then prints ten lines.

A CONTROLLED ENVIRONMENT FOR PROGRAM TESTING

One very useful feature of **sdb** is breakpoint debugging. After entering **sdb**, certain lines in the source program may be specified to be *breakpoints*. The program is then started with a **sdb** command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and **sdb** reports the breakpoint where the program stopped. Now, **sdb** commands may be used to display the trace of function calls and the values of variables. If the user is satisfied the program is working correctly to this point, some breakpoints can be deleted and others set; then program execution may be continued from the point where it stopped.

A useful alternative to setting breakpoints is single stepping. The **sdb** program can be requested to execute the next line of the program and then stop. This feature is especially useful for testing new programs, so they can be verified on a statement-by-statement basis. If an attempt is made to single step through a function which has not been compiled with the **-g** option, execution proceeds until a statement in a function compiled with the **-g** option is reached. It is also possible to have the program execute one machine level instruction at a time. This is particularly useful when the program has not been compiled with the **-g** option.

Setting and Deleting Breakpoints

Breakpoints can be set at any line in a function which contains executable code. The command format is:

```
*12b
*proc:12b
*proc:b
*b
```

The first form sets a breakpoint at line 12 in the current file. The line numbers are relative to the beginning of the file as printed by the source file display commands. The second form sets a breakpoint at line 12 of function *proc*, and the third sets a breakpoint at the first line of *proc*. The last sets a breakpoint at the current line.

Breakpoints are deleted similarly with the commands

```
*12d
*proc:12d
*proc:d
```

In addition, if the command **d** is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed, and a line is read from the user. If the line begins with a **y** or **d**, the breakpoint is deleted.

A list of the current breakpoints is printed in response to a **B** command, and the **D** command deletes all breakpoints. It is sometimes desirable to have **sdb** automatically perform a sequence of commands at a breakpoint and then have execution continue. This is achieved with another form of the **b** command.

```
*12b t;x/
```

causes both a trace back and the value of *x* to be printed each time execution gets to line 12. The **a** command is a variation of the above command. There are two forms:

```
*proc:a
*proc:12a
```

The first prints the function name and its arguments each time it is called, and the second prints the source line each time it is about to be executed. For both forms of the **a** command, execution continues after the function name or source line is printed.

Running the Program

The **r** command is used to begin program execution. It restarts the program as if it were invoked from the **shell**. The command

```
*r args
```

runs the program with the given arguments as if they had been typed on the **shell** command line. If no arguments are specified, then the arguments from the last execution of the program are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as INTERRUPT or QUIT occurs, or the program terminates. In all cases after an appropriate message is printed, control returns to **sdb**.

The **c** command may be used to continue execution of a stopped program. A line number may be specified, as in:

```
*proc:12c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **c** command which continues but passes the signal which stopped the

program back to the program. This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the **g** command. For example:

```
*17 g
```

continues at line 17 of the current function. A use for this command is to avoid executing a section of code which is known to be bad. The user should not attempt to continue execution in a function different than that of the breakpoint.

The **s** command is used to run the program for a single line. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the **S** command. This command is like the **s** command but does not stop within called functions. It is often used when one is confident that the called function works correctly but is interested in testing the calling routine.

The **i** command is used to run the program one machine level instruction at a time while ignoring the signal which stopped the program. Its uses are similar to the **s** command. There is also an **I** command which causes the program to execute one machine level instruction at a time, but also passes the signal which stopped the program back to the program.

Calling Functions

It is possible to call any of the functions of the program from **sdb**. This feature is useful both for testing individual functions with different arguments and for calling a function which prints structured data in a nice way. There are two ways to call a function:

```
*proc(arg1, arg2, ...)
*proc(arg1, arg2, ...)/m
```

The first simply executes the function. The second is intended for calling functions (it executes the function and prints the value that it returns). The value is printed in decimal unless some other format is specified by *m*. Arguments to functions may be integer, character or string constants, or values of variables which are accessible from the current function.

An unfortunate bug in the current implementation is that if a function is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged) all variables are initialized before the function is started. This makes it impossible to use a function which formats data from a dump.

MACHINE LANGUAGE DEBUGGING

The **sdb** program has facilities for examining programs at the machine language level. It is possible to print the machine language statements associated with a line in the source and to place breakpoints at arbitrary addresses. The **sdb** program can also be used to display or modify the contents of the machine registers.

Displaying Machine Language Statements

To display the machine language statements associated with line 25 in function *main*, use the command

```
*main:25?
```

The **?** command is identical to the / command except that it displays from text space. The default format for printing text space is the **i** format which interprets the machine language instruction. The **control-d** command may be used to print the next ten instructions.

Absolute addresses may be specified instead of line numbers by appending a **:** to them so that

```
*0x1024:?
```

displays the contents of address *0x1024* in text space. Note that the command

```
*0x1024?
```

displays the instruction corresponding to line *0x1024* in the current function. It is also possible to set or delete a breakpoint by specifying its absolute address;

```
*0x1024:b
```

sets a breakpoint at address *0x1024*.

Manipulating Registers

Individual registers may also be displayed. The 5000 Series uses the register name prepended with a % so that

```
* %d3
```

displays the value of register d3. The 7000 Series uses the register appended with a % so that

```
* r3%
```

displays the value of register r3.

OTHER COMMANDS

To exit **sdb**, use the **q** command.

The ! command is identical to that in **ed**(1) and is used to have the **shell** execute a command.

It is possible to change the values of variables when the program is stopped at a breakpoint. This is done with the command

```
*variable!value
```

which sets the variable to the given value. The value may be a number, character constant, register, or the name of another variable. If the variable is of type float or double, the value can also be a floating-point constant.

```
$ cat testdiv2.c
main(argc, argv, envp)
char **argv, **envp; {
   int i;
   i = div2(-1);
   printf(" -1/2  = %d\n" , i);
}
div2(i) {
   int j;
   j = i>>1;
   return(j);
}
$ cc -g  testdiv2.c
$ a.out
-1/2  = -1
$ sdb
No core image      # Warning message from sdb
*/^div2            # Search for function " div2"
7: div2(i) {       # It starts on line 7
*z                 # Print the next few lines
7: div2(i) {
8:    int j;
9:    j = i>>1;
10:   return(j);
11: }
*div2:b      # Place breakpoint at beginning of " div2"
div2:9 b     # Sdb echoes proc name and line number
*r           # Run the function
a.out        # Sdb echoes command line executed
Breakpoint at # Executions stops just before line 9
div2:9:    j = i>>1;
*t         # Print trace of subroutine calls
div2(i=-1)    [testdiv2.c:9]
main(argc=1,argv=0x7fffff50,envp=0x7fffff58)[testdiv2.c:4]
*i/          # Print i
-1
*s           # Single step
div2:10: return(j);   # Execution stops before line 10
*j/          # Print j
-1
*9d          # Delete the breakpoint
*div2(1)/ # Try running " div2"  with different arguments
0
```

```
*div2(-2)/
-1
*div2(-3)/
-2
*q
$
```

Figure 7-1. EXAMPLE OF sdb USAGE

Chapter 8

FORTRAN UNIX SYSTEM COMMANDS

A UNIX system Fortran 77 user should be familiar with the following commands:

- **f77** [options] files - This command invokes the UNIX system Fortran 77 compiler

- **ratfor** [options] [files] - This command invokes the Ratfor preprocessor

- **efl** [options] [files] - This command compiles a program written in Extended Fortran Language (EFL) into clean Fortran

- **asa** [files] - This command interprets the output of Fortran programs that utilize ASA carriage control characters

- **fsplit** options files - This command splits the named file(s) into separate files, with one procedure per file.

For more information about the above commands, see the *User Reference Manual* book.

Chapter 9

FORTRAN 77

Chapter 9

FORTRAN 77

This chapter describes the compiler and run-time system for Fortran 77 as implemented on the UNIX system. This chapter also describes the interfaces between procedures and the file formats assumed by the I/O system.

USAGE

The command to run the compiler is

f77 options file

The **f77**(1) command is a general purpose command for compiling and loading Fortran and Fortran-related files into an executable module. EFL (compiler) and Ratfor (preprocessor) source files will be translated into Fortran before being presented to the Fortran compiler. The **f77** command invokes the C compiler to translate C source files and invokes the assembler to translate assembler source files. Object files will be link edited. [The **f77**(1) and **cc**(1) commands have slightly different link editing sequences. Fortran programs need two extra libraries (libI77.a, libF77.a) and an additional startup routine.] The following file name suffixes are understood:

.f	Fortran source file
.e	EFL source file
.r	Ratfor source file
.c	C language source file
.s	Assembler source file
.o	Object file.

LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the Fortran 77 American National Standard, this compiler implements a few extensions. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C language procedures or to permit compilation of old (1966 Standard Fortran) programs.

Double Complex Data Type

The data type double complex is added. Each datum is represented by a pair of double-precision real variables. A double complex version of every complex built-in function is provided. The specific function names begin with % rather than c.

Internal Files

The Fortran 77 American National Standard introduces *internal files* (memory arrays) but restricts their use to formatted sequential I/O statements. This I/O system also permits internal files to be used in direct and unformatted reads and writes.

Implicit Undefined Statement

Fortran has a rule that the type of a variable that does not appear in a type statement is integer if its first letter is *i*, *j*, *k*, *l*, *m* or *n*. Otherwise, it is real. Fortran 77 has an implicit statement for overriding this rule. An additional type statement, *undefined*, is permitted. The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **-u** compiler option is equivalent to beginning each procedure with this statement.

Recursion

Procedures may call themselves directly or through a chain of other procedures.

Automatic Storage

Two new keywords recognized are **static** and **automatic.** These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements.

Variable Length Input Lines

The Fortran 77 American National Standard expects input to the compiler to be in a 72-column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next 66 are the body of the line. (If there are fewer than 72 characters on a line, the compiler pads it with blanks; characters after the first 72 are ignored.) In order to make it easier to type Fortran programs, this compiler also accepts input in variable length lines. An ampersand (**&**) in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Fortran 77 Standard, there are only 26 letters

— Fortran is a one-case language. Consistent with ordinary system usage, the new compiler expects lowercase input. By default, the compiler converts all uppercase characters to lowercase except those inside character constants. However, if the **-U** compiler option is specified, uppercase letters are not transformed. In this mode, it is possible to specify external names with uppercase letters in them and to have distinct variables differing only in case. Regardless of the setting of the option, keywords will only be recognized in lowercase.

Include Statement

The statement

```
include " stuff"
```

is replaced by the contents of the file *stuff*. Includes may be nested to a reasonable depth, currently ten.

Binary Initialization Constants

A **logical, real,** or **integer** variable may be initialized in a data statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is *b*, the string is binary, and only zeroes and ones are permitted. If the letter is *o*, the string is octal with digits *zero* through *seven*. If the letter is z or x, the string is hexadecimal with digits *zero* through *nine*, *a* through *f*. Thus, the statements

```
integer a(3)
data a/b'1010',o'12',z'a'/
```

initialize all three elements of a to ten.

Character Strings

For compatibility with C language usage, the following backslash escapes are recognized:

\n	New-line
\t	Tab
\b	Backspace
\f	Form feed

\0	Null
\'	Apostrophe (does not terminate a string)
\"	Quotation mark (does not terminate a string)
\\	\
\x	Where **x** is any other character.

Fortran 77 only has one quoting character — the apostrophe ('). This compiler and I/O system recognize both the apostrophe and the double quote ("). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an integer word boundary. Each character string constant appearing outside a data statement is followed by a null character to ease communication with C language routines.

Hollerith

Fortran 77 does not have the old Hollerith **(nh)** notation though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In this compiler, Hollerith data may be used in place of character string constants and may also be used to initialize non character variables in data statements.

Equivalence Statements

This compiler permits single subscripts in **equivalence** statements under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

One-Trip DO Loops

The Fortran 77 American National Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs though they were in violation of the 1966 Standard, the **-onetrip** compiler option causes nonstandard loops to be generated.

Commas in Formatted Input

The I/O system attempts to be more lenient than the Fortran 77 American National Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
-345,.05e-3,12
```

correctly.

Short Integers

On machines that support half word integers, the compiler accepts declarations of type **integer*2.** (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C language type **long int;** half word integers are of C language type **short int.**) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **-I2** flag, all small

nteger constants will be of type **integer*2.** If the precision of an nteger-valued intrinsic function is not determined by the generic 'unction rules, one will be chosen that returns the prevailing length **integer*2** when the **-I2** command flag is in effect). When the **-I2**)ption is in effect, all quantities of type **logical** will be short. Note :hat these short integer and logical quantities do not obey the standard rules for storage association.

Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or, and, xor,** and **not**) and for accessing :he command arguments (**getarg** and **iargc**).

The following lists the Fortran intrinsic function library plus some additional functions. These functions are automatically available to the Fortran programmer and require no special invocation of the compiler. The asterisk (*) beside some of the commands indicate they are not part of standard F77. In parentheses beside each function description listed below is the location for the command in the *Programmer Reference Manual* book. These functions are as follows:

```
abort*  Terminate program (ABORT(3F))
abs     Absolute value (MAX(3F)
acos    Arccosine (ACOS(3F))
aimag   Imaginary part of complex argument (AIMAG(3F))
aint    Integer part (AINT(3F))
alog    Natural logarithm (LOG(3F))
alog10  Common logarithm (ALOG10(3F))
amax0   Maximum value (MAX(3F))
amax1   Maximum value (MAX(3F))
amin0   Minimum value (MIN(3F))
amin1   Minimum value (MIN(3F))
amod    Remaindering (MOD(3F))
and*    Bitwise boolean (BOOL(3F))
anint   Nearest integer (ROUND(3F))
asin    Arcsine (ASIN(3F))
atan    Arctangent (ATAN(3F))
atan2   Arctangent (ATAN2(3F))
```

```
cabs     Complex absolute value (ABS(3F))
ccos     Complex cosine (COS(3F))
cexp     Complex exponential (EXP(3F))
char     Explicit type conversion (FTYPE(3F))
clog     Complex natural logarithm (LOG(3F))
cmplx    Explicit type conversion (FTYPE(3F))
conjg    Complex conjugate (CONJG(3F))
cos      Cosine (COS(3F))
cosh     Hyperbolic cosine (COSH(3F))
csin     Complex sine (SIN(3F))
csqrt    Complex square root (SQRT(3F))
dabs     Absolute value (ABS(3F))
dacos    Arccosine (ACOS(3F))
dasin    Arcsine (ASIN(3F))
datan    Arctangent (ATAN(3F))
datan2   Double precision arctangent (ATAN2(3F))
dble     Explicit type conversion (FTYPE(3F))
dcmplx*  Explicit type conversion (FTYPE(3F))
dconjg*  Complex conjugate (CONJG(3F))
dcos     Cosine (DCOS(3F))
dcosh    Hyperbolic cosine (COSH(3F))
ddim     Positive difference (DIM(3F))
dexp     Exponential (EXP(3F))
dim      Positive difference (DIM(3F))
dimag*   Imaginary part of complex argument ((AIMAG(3F))
dint     Integer part (AINT(3F))
dlog     Natural logarithm (LOG(3F))
dlog10   Common logarithm (LOG10(3F))
dmax1    Maximum value (MAX(3F))
dmin1    Minimum value (MIN(3F))
dmod     Remaindering (DMOD(3F))
dnint    Nearest integer (ROUND(3F))
dprod    Double precision product (DPROD(3F))
dsign    Transfer of sign (SIGN(3F))
dsin     Sine (SIN(3F))
dsinh    Hyperbolic sine (SINH(3F))
dsqrt    Square root (SQRT(3F))
dtan     Tangent (TAN(3F))
dtanh    Hyperbolic tangent (TANH(3F))
exp      Exponential (EXP(3F))
float    Explicit type conversion (FTYPE(3F))
getarg*  Return command-line argument (GETARG(3F))
getenv*  Return environment variable (GETENV(3F))
```

```
iabs     Absolute value (ABS(3F))
iargc    Return number of arguments (IARGC(3F))
ichar    Explicit type conversion (FTYPE(3F))
idim     Positive difference (DIM(3F))
idint    Explicit type conversion (FTYPE(3F))
idnint   Nearest integer (ROUND(3F))
ifix     Explicit type conversion (FTYPE(3F))
index    Return location of substring (INDEX(3F))
int      Explicit type conversion (FTYPE(3F))
irand*   Random number generator
isign    Transfer of sign (SIGN(3F))
len      Return location of string (LEN(3F))
lge      String comparison (STRCMP(3F))
lgt      String comparison (STRCMP(3F))
lle      String comparison (STRCMP(3F))
llt      String comparison (STRCMP(3F))
log      Natural logarithm (LOG(3F))
log10    Common logarithm (LOG10(3F))
lshift*  Bitwise boolean (BOOL(3F))
max      Maximum value (MAX(3F))
max0     Maximum value (MAX(3F))
max1     Maximum value (MAX(3F))
mclock*  Return Fortran time accounting (MCLOCK(3F))
min      Minimum value (MIN(3F))
min0     Minimum value (MIN(3F))
min1     Minimum value (MIN(3F))
mod      Remaindering (MOD(3F))
nint     Nearest integer (BOOL(3F))
not*     Bitwise boolean (BOOL(3F))
or*      Bitwise boolean (BOOL(3F))
rand*    Random number generator (RAND(3F))
real     Explicit type conversion (FTYPE(3F))
rshift*  Bitwise boolean (BOOL(3F))
sign     Transfer of sign (SIGN(3F))
signal*  Specify action on receipt of system signal
         (SIGNAL(3F))
sin      Sine (SINE(3F))
sinh     Hyperbolic sine (SINH(3F))
sngl     Explicit type conversion (FTYPE(3F))
sqrt     Square root (SQRT(3F))
srand*   Random number generator (RAND(3F))
system*  Issue a shell command (SYSTEM(3F))
```

```
tan     Tangent (TAN(3F))
tanh    Hyperbolic tangent (TANH(3F))
xor*    Bitwise boolean (BOOL(3F))
zabs*   Complex absolute value (ABS(3F)).
```

For more information on the Fortran intrinsic function commands, see the *Programmer Reference Manual* book.

VIOLATIONS OF THE STANDARD

The following paragraphs describe only three known ways in which the UNIX system implementation of Fortran 77 violates the new American National Standard.

Double Precision Alignment

The Fortran 77 American National Standard permits **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c
equivalence (a(1),b), (a(4),c)
```

Some machines require that double precision quantities be on double word boundaries; other machines run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double-precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use two separate operations. The first operation would be to move the upper and lower halves into the halves of an aligned temporary. The second would be to load that double-precision temporary. The reverse would be needed to store a result. All double-precision real and complex quantities are required to fall on even word boundaries on machines with corresponding hardware requirements and to issue a diagnostic if the source code demands a violation of the rule.

Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments. A warning is printed if a dummy procedure is not declared **external.** Code is correct if there are no **character** arguments.

T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. The implementation uses "seeks"; so if the unit is not one which allows seeks (such as a terminal) the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

INTERPROCEDURE INTERFACE

To be able to write C language procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C language procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

Data Representations

The following is a table of corresponding Fortran and C language declarations:

Fortran	C Language
`integer*2 x`	`short int x;`
`integer x`	`long int x;`
`logical x`	`long int x;`
`real x`	`float x;`
`double precision x`	`double x;`
`complex x`	`struct { float r, i; } x;`
`double complex x`	`struct { double dr, di; } x;`
`character*6 x`	`char x[6];`

By the rules of Fortran, **integer, logical,** and **real** data occupy the same amount of memory.

Return Values

A function of type **integer, logical, real,** or **double precision** declared as a C language function returns the corresponding type. A **complex** or **double complex** function is equivalent to a C language routine with an additional initial argument that points to the place where the return value is to be stored. Thus, the following:

```
complex function f( . . . )
```

is equivalent to

```
f_(temp,...)
struct { float r, i; } *temp;
...
```

A character-valued function is equivalent to a C language routine with two extra initial arguments - a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
g_(result, length,...)
char result [ ];
long int length;
```

and could be invoked in C language by

```
char chars[15];
. . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3), nret( )
```

Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value.) The order of arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A **long int** for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
...
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
...
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C language array always has subscript 0, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order; C language arrays are stored in row-major order.

FILE FORMATS

Structure of Fortran Files

Fortran requires four kinds of external files: *sequential formatted* and *unformatted*, and *direct formatted* and *unformatted*. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on "records." When a direct file is opened in a Fortran program, the record length of the records must be given; and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records but are treated as byte-addressable byte strings; i.e., as ordinary files on the UNIX system. (A read or write request on such a file keeps consuming bytes until satisfied rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each new-line as a record separator. The result of reading off the end of a record is undefined according to the Fortran 77 American National Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a new-line at the end of each record. It is also possible for programs to write new-lines for themselves. This is an error, but the only effect will be that the single record the user thought was written will be treated as more than one record when being read or backspaced over.

Preconnected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist nor will they be created unless their units are used without first executing an **open.** The default connection is for sequential formatted I/O.

The Fortran 77 Standard does not specify where a file which has been explicitly **open**ed for sequential I/O is initially positioned. In fact, the I/O system attempts to position the file at the end. A **write** will append to the file and a **read** will result in an "end of file" indication. To position a file to its beginning, use a **rewind** statement. The preconnected units 0, 5, and 6 are positioned as they come from the parent process.

Chapter 10

Ratfor

Chapter 10

RATFOR

GENERAL

This chapter describes the Ratfor(1) preprocessor. It is assumed that the user is familiar with the current implementation of Fortran 77 on the UNIX system.

The Ratfor language allows users to write Fortran programs in a fashion similar to C language. The Ratfor program is implemented as a preprocessor that translates this "simplified" language into Fortran. The facilities provided by Ratfor are:

- Statement grouping
- if-else and switch for decision making
- while, for, do, and repeat-until for looping
- break and next for controlling loop exits
- Free form input such as multiple statements/lines and automatic continuation
- Simple comment convention
- Translation of >, >=, etc., into .gt., .ge., etc.
- return statement for functions
- define statement for symbolic parameters
- include statement for including source files.

USAGE

The Ratfor program takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **-6x**, which uses x as a continuation character in column 6 (the UNIX system uses *&* in column 1), **-h**, which causes quoted strings to be turned into *n*H constructs and **-C**, which causes Ratfor comments to be copied into the generated Fortran.

STATEMENT GROUPING

The Ratfor language provides a statement grouping facility. A group of statements can be treated as a unit by enclosing them in the braces { and }. For example, the Ratfor code

```
if (x > 100)
    { call error(" x>100" ); err = 1; return }
```

will be translated by the Ratfor preprocessor into **Fortran** equivalent to

```
      if (x .le. 100) goto 10
          call error(5hx>100)
          err = 1
          return
10    ...
```

which should simplify programming effort. By using { and }, a group of statements can be used instead of a single statement.

Also note in the previous Ratfor example that the character > was used instead of **.GT.** in the **if** statement. The Ratfor preprocessor translates this C language type operator to the appropriate Fortran operator. More on relationship operators later.

In addition, many Fortran compilers permit character strings in quotes (like " *x>100*"). But others, like ANSI Fortran 66, do not. Ratfor converts it into the right number of *Hs*.

The Ratfor language is free form. Statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The previous example could also be written as

```
if (x > 100) {
    call error(" x>100" )
    err = 1
    return
}
```

which shows grouped statements spread over several lines. In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement, no braces are needed.

THE "if-else" CONSTRUCTION

The Ratfor language provides an **else** statement. The syntax of the **if-else** construction is:

if (*legal Fortran condition*)
    ratfor statement
else
    ratfor statement

where the **else** part is optional. The legal Fortran condition is anything that can legally go into a Fortran Logical **IF** statement. The Ratfor preprocessor does not check this clause since it does not know enough Fortran to know what is permitted. The "ratfor" statement is any Ratfor or Fortran statement or any collection of them in braces. For example:

```
if (a <= b)
    { sw = 0; write(6, 1) a, b }
else
    { sw = 1; write(6, 1) b, a }
```

is a valid Ratfor **if-else** construction. This writes out the smaller of a and b, then the larger, and sets sw appropriately.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed.

Nested "if" Statements

The statement that follows an **if** or an **else** can be any Ratfor statement including another **if** or **else** statement. In general, the structure

```
if (condition) action
else if (condition) action
else action
```

provides a way to write a multibranch in Ratfor. (The Ratfor language also provides a **switch** statement which could be used instead, under certain conditions.) The last **else** handles the "default" condition. If there is no default action, this final **else** can be omitted. Thus, only the actions associated with the valid condition are performed. For example:

```
if (x < 0)
    x = 0
else if (x > 100)
    x = 100
```

will ensure that x is not less than 0 and not greater than 100.

Nested **if** and **else** statements could result in ambiguous code. In Ratfor when there are more **if** statements than **else** statements, **else** statements are associated with the closest previous **if** statement that currently does not have an associated **else** statement. For example:

```
if (x > 0)
if (y > 0)
write(6,1) x, y
else
write(6,2) y
```

is interpreted by the Ratfor preprocessor as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
    else
        write(6, 2) y
}
```

in which the braces are assumed. If the other association is desired it must be written as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
}
else
    write(6, 2) y
```

with the braces specified.

THE "switch" STATEMENT

The **switch** statement provides a way to express multiway branches which branch on the value of some *integer*-valued expression. The syntax is

```
switch (expression) {
    case expr1 :
    statements
    case expr2, expr3 :
    statements
    ...
    default:
    statements
}
```

where each **case** is followed by an integer expression (or several integer expressions separated by commas). The **switch** *expression* is compared to each **case** *expr* until a match is found. Then the *statements* following that **case** are executed. If no **cases** match

expression, then the *statements* following **default** are executed. The **default** section of a **switch** is optional.

When the *statements* associated with a **case** are executed, the entire **switch** is exited immediately. This is different from C language.

THE "do" STATEMENT

The **do** statement in Ratfor is quite similar to the **DO** statement in Fortran except that it uses no statement number (braces are used to mark the end of the **do** instead of a statement number). The syntax of the **ratfor do** statement is

```
do legal-Fortran-DO-text {
   ratfor statements
}
```

The *legal-Fortran-DO-text* must be something that can legally be used in a Fortran **DO** statement. Thus if a local version of Fortran allows **DO** limits to be expressions (which is not currently permitted in ANSI Fortran 66), they can be used in a **ratfor do** statement. The *ratfor statements* are enclosed in braces; but as with the **if**, a single statement need not have braces around it. For example, the following code sets an array to zero:

```
do i = 1, n
   x(i) = 0.0
```

and the code

```
do i = 1, n
   do j = 1, n
      m(i, j) = 0
```

sets the entire array *m* to zero.

THE "break" AND "next" STATEMENTS

The Ratfor **break** and **next** statements provide a means for leaving a loop early and one for beginning the next iteration. The **break** causes an immediate exit from the **do**; in effect, it is a branch to the statement *after* the **do**. The **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array

```
do i = 1, n {
  if (x(i) < 0.0)
    next
  process positive element
}
```

The **break** and **next** statements will also work in the other Ratfor looping constructions and will be discussed with each looping construction.

The **break** and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop. For example:

```
break 2
```

exits from two levels of enclosing loops, and

```
break 1
```

is equivalent to **break**. The

```
next 2
```

iterates the second enclosing loop.

THE "while" STATEMENT

The Ratfor language provides a **while** statement. The syntax of the **while** statement is

while (*legal-Fortran-condition*)
  ratfor statement

As with the **if**, legal-Fortran-condition is something that can go into a Fortran Logical **IF**, and ratfor statement is a single statement which may be multiple statements enclosed in braces.

For example, suppose nextch is a function which returns the next input character both as a function value and in its argument. Then a **while** loop to find the first nonblank character could be

```
while (nextch(ich) == iblank)
    ;
```

where a semicolon by itself is a null statement (which is necessary here to mark the end of the **while**). If the semicolon were not present, the **while** would control the next statement. When the loop is exited, ich contains the first nonblank.

THE "for" STATEMENT

The **for** statement is another Ratfor loop. A **for** statement allows explicit initialization and increment steps as part of the statement.

The syntax of the **for** statement is

for (*init* ; *condition* ; *increment*)
  ratfor statement

where *init* is any single Fortran statement which is executed once before the loop begins. The increment is any single Fortran statement that is executed at the end of each pass through the loop before the test. The *condition* is again anything that is legal in a

Fortran Logical **IF**. Any of init, condition, and increment may be omitted although the semicolons must always be present. A nonexistent condition is treated as always true, so

```
for (;;)
```

is an infinite loop.

For example, a Fortran **DO** loop could be written as

```
for (i = 1; i <= n; i = i + 1) ...
```

which is equivalent to

```
i = 1
while (i <= n) {
    ...
    i = i + 1
}
```

The initialization and increment of *i* have been moved into the **for** statement.

The **for**, **do**, and **while** versions have the advantage that they will be done zero times if *n* is less than 1. In addition, the **break** and **next** statements work in a **for** loop.

The *increment* in a **for** need not be an arithmetic progression. The program

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
    sum = sum + value(i)
```

steps through a list (stored in an integer array *ptr*) until a zero pointer is found while adding up elements from a parallel array of values. Notice that the code also works correctly if the list is empty.

THE "repeat-until" STATEMENT

There are times when a test needs to be performed at the bottom of a loop after one pass through. This facility is provided by the **repeat-until** statement. The syntax for the **repeat-until** statement is

repeat
  ratfor statement
until (*legal-Fortran-condition*)

where ratfor-statement is done once, then the *condition* is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a **repeat** by itself is an infinite loop. A **repeat-until** loop can be exited by the use of a **stop**, **return**, or **break** statement or an implicit stop such as running out of input with a **READ** statement.

As stated before, a **break** statement causes an immediate exit from the enclosing **repeat-until** loop. A **next** statement will cause a skip to the bottom of a **repeat-until** loop (i.e., to the **until** part).

THE "return" STATEMENT

The standard Fortran mechanism for returning a value from a routine uses the name of the routine as a variable. This variable can be assigned a value. The last value stored in it is the value returned by the function. For example, in a Fortran routine named *equal*, the statements

```
equal = 0
return
```

cause *equal* to return zero.

The Ratfor language provides a **return** statement similar to the C language **return** statement. In order to return a value from any routine, the **return** statement has the syntax

return (*expression*)

where *expression* is the value to be returned.

If there is no parenthesized expression after **return**, no value is returned.

THE "define" STATEMENT

The Ratfor language provides a **define** statement similar to the C language version. Any string of alphanumeric characters can be defined as a name. Whenever that name occurs in the input (delimited by nonalphanumerics), it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off.) A defined name can be arbitrarily long and must begin with a letter.

Usually the **define** statement is used for symbolic parameters. The syntax of the **define** statement is

define name value

where *name* is a symbolic name that represents the quantity of *value*. For example:

```
define ROWS 100
define CLOS 50
dimension a(ROWS), b(ROWS, COLS)
    if (i > ROWS | j > COLS) ...
```

causes the preprocessor to replace the name *ROWS* with the value *100* and the name *COLS* with the value *50*. Alternately, definitions may be written as

```
define(ROWS, 100)
```

in which case the defining text is everything after the comma up to the right parenthesis. This allows multiple-line definitions.

THE "include" STATEMENT

The Ratfor language provides an **include** statement similar to the **#include** <...> statement in C language. The syntax for this statement is

include *file*

which inserts the contents of the named file into the Ratfor input file in place of the **include** statement. The standard usage is to place **COMMON** blocks on a file and use the **include** statement to include the common code whenever needed.

FREE-FORM INPUT

In Ratfor, statements can be placed anywhere on a line. Long statements are continued automatically as are long conditions in **if**, **for**, and **until** statements. Blank lines are ignored. Multiple statements may appear on one line if they are separated by semicolons. No semicolon is needed at the end of a line if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
= + - * , | & ( _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur. All other characters remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label and placed in columns 1 through 5 upon output. Thus:

```
write(6, 100); 100 format(" hello" )
```

is converted into

```
      write(6, 100)
100   format(5hhello)
```

TRANSLATIONS

When the **-h** option is chosen, text enclosed in matching single or double quotes is converted to *nH...* but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash (\) serves as an escape character; i.e., the next character is taken literally. This provides a way to get quotes and the backslash itself into quoted strings. For example:

```
"\'"
```

is a string containing a backslash and an apostrophe. (This is not the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character % is left absolutely unaltered except for stripping off the % and moving the line one position to the left. This is useful for inserting control cards and other things that should not be preprocessed (like an existing Fortran program). Use % only for ordinary statements not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made (except within single or double quotes or on a line beginning with a %):

```
==   .eq.
!=   .ne.
>    .gt.
>=   .ge.
<    .lt.
<=   .le.
&    .and.
|    .or.
!    .not.
```

In addition, the following translations are provided for input devices with restricted character sets:

```
[    {
]    }
$(   {
$)   }
```

WARNINGS

The Ratfor preprocessor catches certain syntax errors (such as missing braces), **else** statements without **if** statements, and most errors involving missing parentheses in statements.

All other errors are reported by the Fortran compiler. Unfortunately, the Fortran compiler prints messages in terms of generated Fortran code and not in terms of the Ratfor code. This makes it difficult to locate Ratfor statements that contain errors.

The keywords are deserved. Using **if**, **else**, **while**, etc., as variable names will cause considerable problems. Likewise, spaces within keywords and use of the Arithmetic **IF** will cause problems.

The Fortran *n*H convention is not recognized by Ratfor. Use quotes instead.

EXAMPLE OF RATFOR CONVERSION

As an example of how to use the Ratfor program, the following program **prog.r** (where the **.r** indicates a Ratfor source program), is written in the Ratfor language:

```
    ICNT=0
 10 WRITE(6,31)
 31 FORMAT(" INPUT FIRST NUMBER" )
    READ(5,32) A
 32 FORMAT(F10.2)
    WRITE(6,33)
 33 FORMAT(" INPUT SECOND NUMBER" )
    READ(5,34) B
 34 FORMAT(F10.2)
    IF(A<B)
      WRITE(6,36) A,B
    ELSE WRITE(6,37)A,B
 36 FORMAT(F10.2," < ",F10.2)
 37 FORMAT(F10.2," >= ",F10.2)
    ICNT=ICNT+1
    IF(ICNT.EQ.5)
      GOTO 100
    GOTO 10
100 END
```

The command

```
ratfor prog.r > prog.f
```

causes the Fortran translation program **prog.f** to be produced. (The Ratfor program **prog.r** remains intact.) The Fortran program **prog.f** follows:

```
      icnt=0
10    write(6,31)
31    format(" INPUT FIRST NUMBER" )
      read(5,32) a
32    format(f10.2)
      write(6,33)
33    format(" INPUT SECOND NUMBER" )
      read(5,34) b
34    format(f10.2)
      if(.not.(a.lt.b))goto 23000
      write(6,36) a,b
      goto 23001
23000 continue
      write(6,37)a,b
23001 continue
36    format(f10.2," < ",f10.2)
37    format(f10.2," >= ",f10.2)
      icnt=icnt+1
      if(.not.(icnt.eq.5))goto 23002
      goto 100
23002 continue
      goto 10
100   end
```

The Fortran program **prog.f** is compiled using the command

```
f77 prog.f
```

An object program file **prog.o** and a final output file **a.out** are produced. Since the output file **a.out** is an executable file, the command

```
a.out
```

causes the program to run.

The Ratfor program **prog.r** can also be translated and compiled with the single command

```
f77 prog.r
```

where the **.r** indicates a Ratfor source program. An object file **prog.o** and a final output file **a.out** are produced.

Chapter 11

The Programming Language EFL

Chapter 11

THE PROGRAMMING LANGUAGE EFL

INTRODUCTION

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. Thus, the EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment.

The name EFL originally stood for "Extended Fortran Language." The current compiler is much more than a simple preprocessor: it attempts to diagnose all syntax errors, to provide readable Fortran output, and to avoid a number of niggling restrictions.

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as **while**. Words in *italic* type indicate an item in a category, such as an *expression.* A construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation

[[*item*]]

could refer to any of the following:

item
item, *item*
item, *item*, *item*

The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to Fortran which may be ignored if the reader is unfamiliar with those languages.

LEXICAL FORM

Character Set

The following characters are legal in an EFL program:

letters	**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
digits	**0 1 2 3 4 5 6 7 8 9**
white space	*blank tab*
quotes	' "
sharp	#
continuation	_
braces	{ }
parentheses	( )
other	, ; : . + − * /
	= < > & ~ ¦ $

Letter case (upper or lower) is ignored except within strings, so "a" and "A" are treated as the same character. All of the examples below are printed in lower case. An exclamation mark ("!") may be used in place of a tilde ("~"). Square brackets ("[" and "]") may be used in place of braces ("{" and "}").

Lines

EFL is a line-oriented language. Except in special cases (discussed below), the end of a line marks the end of a token and the end of a statement. The trailing portion of a line may be used for a comment. There is a mechanism for diverting input from one source file to another, so a single line in the program may be replaced by a number of lines from the other file. Diagnostic messages are labeled with the line number of the file on which they are detected.

White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space. Such a space terminates a token.

Comments

A comment may appear at the end of any line. It is introduced by a sharp (#) character, and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

Include Files

It is possible to insert the contents of a file at a point in the source text, by referencing it in a line like

include joe

No statement or comment may follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. **Includes** may be nested at least ten deep.

Continuation

Lines may be continued explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space have been stripped) is an underscore, the end of a line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts (except inside of quoted strings). Thus

```
1_000_000_
000
```

equals 10^9.

There are also rules for continuing lines automatically: the end of line is ignored whenever it is obvious that the statement is not complete. To be specific, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. (A statement is not continued just because of unbalanced braces or parentheses.) Some compound statements are also continued automatically; these points are noted in the sections on executable statements.

Multiple Statements on a Line

A semicolon terminates the current statement. Thus, it is possible to write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless explicit continuation (see above) is signaled by an underscore.

Identifiers

An identifier is a letter or a letter followed by letters or digits. The following is a list of the reserved words that have special meaning in EFL. They will be discussed later.

array	**exit**	**precision**
automatic	**external**	**procedure**
break	**false**	**read**
call	**field**	**readbin**
case	**for**	**real**
character	**function**	**repeat**
common	**go**	**return**
complex	**goto**	**select**
continue	**if**	**short**
debug	**implicit**	**sizeof**
default	**include**	**static**
define	**initial**	**struct**
dimension	**integer**	**subroutine**
do	**internal**	**true**
double	**lengthof**	**until**
doubleprecision	**logical**	**value**
else	**long**	**while**
end	**next**	**write**
equivalence	**option**	**writebin**

The use of these words is discussed below. These words may not be used for any other purpose.

Strings

A character string is a sequence ot characters surrounded by quotation marks. If the string is bounded by single-quote marks ('), it may contain double quote marks ("), and vice versa. A quoted string may not be broken across a line boundary.

'hello there'
" ain't misbehavin'"

Integer Constants

An integer constant is a sequence of one or more digits.

0
57
123456

Floating Point Constants

A floating point constant contains a dot and/or an exponent field. An *exponent field* is a letter **d** or **e** followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

.I
I.
I.J
IE
I.E
.IE
I.JE

Punctuation

Certain characters are used to group or separate objects in the language. These are

parentheses	( )
braces	{ }
comma	,
semicolon	;
colon	:
end-of-line	

The end-of-line is a token (statement separator) when the line is neither blank nor continued.

Operators

The EFL operators are written as sequences of one or more non-alphanumeric characters.

```
+  -  *  /  **
<  <=  >  >=  ==  ~=
&&  ||  &  |
+=  -=  /=  **=
&&=  ||=  &=  |=
->  .  $
```

A dot (".") is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode (see " ATAVISMS") in which some of the operators may be represented by a string consisting of a dot, an identifier, and a dot (*e.g.,* **.lt.**).

Macros

EFL has a simple macro substitution facility. An identifier may be defined to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement like

```
define count    n += 1
```

Any time the name **count** appears in the program, it is replaced by the statement

```
n += 1
```

A **define** statement must appear alone on a line; the form is

define *name rest-of-line*

Trailing comments are part of the string.

PROGRAM FORM

Files

A *file* is a sequence of lines. A file is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed in " PROCEDURES."

Blocks

Statements may be formed into groups inside of a procedure. To describe the scope of names, it is convenient to introduce the ideas of *block* and of *nesting level.* The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations are also at level zero. The text immediately following a **procedure** statement is at level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. (Braces inside declarations do not mark blocks.) (See " Blocks" under " EXECUTABLE STATEMENTS.") An **end** statement marks the end of the procedure, level 1, and the return to level 0. A name (variable or macro) that is defined at level *K* is defined throughout that block

and in all deeper nested levels in which that name is not redefined or redeclared. Thus, a procedure might look like the following:

```
#  block 0
procedure george
real x
x = 2
...
if(x > 2)
        {       # new block
        integer x # a different variable
        do x = 1,7
                write(,x)
        ...
        }       # end of block
end     # end of procedure, return to block 0
```

Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define

Procedure
End

Declarative
Executable

The **option** statement is described in " COMPILER OPTIONS". The **include, define,** and **end** statements have been described above; they may not be followed by another statement on a line. Each procedure begins with a **procedure** statement and finishes with an **end** statement; these are discussed in " PROCEDURES". Declarations describe types and values of variables and procedures. Executable statements cause specific actions to be taken. A block is an example of an executable statement; it is made up of declarative and executable statements.

Labels

An executable statement may have a *label* which may be used in a branch statement. A label is an identifier followed by a colon, as in

```
        read(, x)
        if(x < 3) goto error
        ...
error:  fatal(" bad input" )
```

DATA TYPES AND VARIABLES

EFL supports a small number of basic (scalar) types. The programmer may define objects made up of variables of basic type; other aggregates may then be defined in terms of previously defined aggregates.

Basic Types

The basic types are

logical
integer
field(*m*:*n*)
real
complex
long real
long complex
character(*n*)

A logical quantity may take on the two values *true* and *false*. An integer may take on any whole number value in some machine-dependent range. A field quantity is an integer restricted to a particular closed interval ([*m*:*n*]). A "real" quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. (Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers.) A complex quantity is an approximation to a complex number, and is represented as a pair of reals. A character quantity is a fixed-length string of *n* characters.

Constants

There is a notation for a constant of each basic type.

A logical may take on the two values

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign, as in

17
−94
+6
0

A long real ("double precision") constant is a floating point constant containing an exponent field that begins with the letter **d.** A real ("single precision") constant is any other floating point constant. A real or long real constant may be preceded by a plus or minus sign. The following are valid **real** constants:

17.3
−.4
7.9e−6 (= 7.9×10^{-6})
14e9 (= 1.4×10^{10})

The following are valid **long real** constants

7.9d−6 (= 7.9×10^{-6})
5d3

A character constant is a quoted string.

Variables

A variable is a quantity with a name and a location. At any particular time the variable may also have a value. (A variable is said to be *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has certain attributes:

Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent (static). (A future extension of EFL may include dynamically allocated variables.)

Scope of Names

The names of common areas are global, as are procedure names: these names may be used anywhere in the program. All other names are local to the block in which they are declared.

Precision

Floating point variables are either of normal or **long** precision. This attribute may be stated independently of the basic type.

Arrays

It is possible to declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common.** A formal argument array may have intervals that are of length equal to one of the other formal arguments. An element of an array is denoted by the array name followed by a parenthesized comma-separated list of integer values, each of which must lie within the corresponding interval. (The intervals may include negative numbers.) Entire arrays may be passed as procedure arguments or in input/output lists, or they may be initialized; all other array references must be to individual elements.

Structures

It is possible to define new types which are made up of elements of other types. The compound object is known as a *structure;* its constituents are called *members* of the structure. The structure may be given a name, which acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they may be arrays of such objects. Entire structures may be passed to procedures or be used in input/output lists; individual elements of structures may be referenced. The uses of structures will be detailed below. The following structure might represent a symbol table:

```
struct tableentry
        {
        character(8) name
        integer hashvalue
        integer numberofelements
        field(0:1) initialized, used, set
        field(0:10) type
        }
```

EXPRESSIONS

Expressions are syntactic forms that yield a value. An expression may have any of the following forms, recursively applied:

primary
(*expression* **)**
unary-operator expression
expression binary-operator expression

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. The meanings of these operators are described in " Unary Operators" and " Binary Operators" under " EXPRESSIONS"

```
-> .
**
* /    unary + - ++ --
+ -
< <= > >= == ~=
& &&
| ||
$
= += -= *= /= **=   &= |= &&= ||=
```

Examples of expressions are

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

Primaries

Primaries are the basic elements of expressions. They include constants, variables, array elements, structure members, procedure invocations, input/output expressions, coercions, and sizes.

Constants

Constants are described in " Constants" under " DATA TYPES AND VARIABLES" .

Variables

Scalar variable names are primaries. They may appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) may appear only as procedure arguments and in input/output lists.

Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, one integer value for each declared dimension:

```
a(5)
b(6,-3,4)
```

Structure Members

A structure name followed by a dot followed by the name of a member of that structure constitutes a reference to that element. If that element is itself a structure, the reference may be further qualified.

```
a.b
x(3).y(4).z(5)
```

Procedure Invocations

A procedure is invoked by an expression of one of the forms

procedurename ( )
procedurename (*expression*)
procedurename (*expression-1*, ..., *expression-n*)

The *procedurename* is either the name of a variable declared **external** or it is the name of a function known to the EFL compiler (see " Known Functions" under " PROCEDURES"), or it is the actual name of a procedure, as it appears in a **procedure** statement. If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure. Each *expression* in the above is called an *actual argument*. Examples of procedure invocations are

```
f(x)
work()
g(x, y+3, 'xx')
```

When one of these procedure invocations is to be performed, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, the called procedure is permitted to use the corresponding

formal argument as the left side of an assignment or in an input list; otherwise it may only use the value. After the formal and actual arguments are associated, control is passed to the first executable statement of the procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure, which must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument. See " PROCEDURES" .

Input/Output Expressions

The EFL input/output syntactic forms may be used as integer primaries that have a non-zero value if an error occurs during the input or output. See " Input/Output Statements" under " EXECUTABLE STATEMENTS" .

Coercions

An expression of one precision or type may be converted to another by an expression of the form

attributes (*expression*)

At present, the only *attributes* permitted are precision and basic types. Attributes are separated by white space. An arithmetic value of one type may be coerced to any other arithmetic type; a character expression of one length may be coerced to a character expression of another length; logical expressions may not be coerced to a nonlogical type. As a special case, a quantity of **complex** or **long complex** type may be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

integer(5.3) = 5
long real(5) = 5.0d0
complex(5,3) = 5+3i

Most conversions are done implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are of most use when it is necessary to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

Sizes

There is a notation which yields the amount of memory required to store a datum or an item of specified type:

sizeof (*leftside*)
sizeof (*attributes*)

In the first case, *leftside* can denote a variable, array, array element, or structure member. The value of **sizeof** is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

sizeof(x) / sizeof(integer)

yields the size of the variable **x** in integer words.

The distance between consecutive elements of an array may not equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives this larger value, again in arbitrary units. The syntax is

lengthof (*leftside*)
lengthof (*attributes*)

Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression must be evaluated before an expression of which it is a part is evaluated.

Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

Arithmetic

Unary + has no effect. A unary − yields the negative of its operand.

The prefix operator ++ adds one to its operand. The prefix operator −− subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

Logical

The only logical unary operator is complement (~). This operator is defined by the equations

~ true = false
~ false = true

Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

Arithmetic

The binary arithmetic operators are

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative: a**b**c = a**(b**c) = $a^{(b^c)}$ The operations have the conventional meanings: 8+2 = 10, 8-2 = 6, 8*2 = 16, 8/2 = 4, 8**2 = 8^2 = 64.

The type of the result of a binary operation *A op B* is determined by the types of its operands:

	Type of B				
Type of A	i	r	l r	c	l c
i	i	r	l r	c	l c
r	r	r	l r	c	l c
l r	l r	l r	l r	l c	l c
c	c	c	l c	c	l c
l c	l c	l c	l c	l c	l c

i = integer
r = real
l r = long real
c = complex
l c = long complex

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer, and is computed exactly. (Quotients are truncated toward zero, so 8/3=2.)

Logical

The two binary logical operations in EFL, **and** and **or,** are defined by the truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Each of these operators comes in two forms. In one form, the order of evaluation is specified. The expression

a && b

is evaluated by first evaluating **a**; if it is false then the expression is false and **b** is not evaluated; otherwise, the expression has the value of **b.** The expression

a || b

is evaluated by first evaluating **a**; if it is true then the expression is true and **b** is not evaluated; otherwise, the expression has the value of **b.** The other forms of the operators (**&** for **and** and **|** for **or**) do not imply an order of evaluation. With the latter operators, the compiler may speed up the code by evaluating the operands in any order.

Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator		Meaning
<	<	less than
<=	≤	less than or equal to
==	=	equal to
~=	≠	not equal to
>	>	greater than
>=	≥	greater than or equal

Since the complex numbers are not ordered, the only relational operators that may take complex operands are == and ~= . The character collating sequence is not defined.

Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = *expression*

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side, and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. In each case, *a op= b* is equivalent to *a = a op b*. (The operator and equal sign must not be separated by blanks.) Thus, **n+=2** adds 2 to n. The location of the left side is evaluated only once.

Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures,

leftside –> *structurename*

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside.* In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way using the dot operator. Thus,

place(i) –> st.elt

refers to the **elt** member of the **st** structure starting at the i^{th} element of the array **place.**

Repetition Operator

Inside of a list, an element of the form

integer-constant-expression $ *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

(3, 3$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

DECLARATIONS

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two forms:

attributes variable-list
attributes { *declarations* }

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the declarations also has the specified attributes. A variable name may appear in more than one variable list, so long as the attributes are not contradictory. Each

name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Examples of declarations are

```
integer k=2

long real b(7,3)

common(cname)
    {
    integer i
    long real array(5,0:3) x, y
    character(7) ch
    }
```

Attributes

Basic Types

The following are basic types in declarations

```
logical
integer
field(m:n)
character(k)
real
complex
```

In the above, the quantities k, m, and n denote integer constant expressions with the properties $k>0$ and $n>m$.

Arrays

The dimensionality may be declared by an **array** attribute

array(b_1,...,b_n)

Each of the b_i may either be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to n, the number of bounds. All of the integer expressions

must be constants. An exception is permitted only if all of the variables associated with an array declarator are formal arguments of the procedure; in this case, each bound must have the property that *upper-lower* +1 is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it will recognize important cases such as **(0:n−1)**. The upper bound for the last dimension (b_n) may be marked by an asterisk (*) if the size of the array is not known. The following are legal **array** attributes:

```
array(5)
array(5, 1:5, −3:0)
array(5, *)
array(0:m−1, m)
```

Structures

A structure declaration is of the form

struct *structname* { *declaration statements* }

The *structname* is optional; if it is present, it acts as if it were the name of a type in the rest of its scope. Each name that appears inside the *declarations* is a *member* of the structure, and has a special meaning when used to qualify any variable declared with the structure type. A name may appear as a member of any number of structures, and may also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known. The following are valid structure attributes

```
struct xx
    {
    integer a, b
    real x(5)
    }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three **xx**'*s* and a character string.

Precision

Variables of floating point (**real** or **complex**) type may be declared to be **long** to ensure they have higher precision than ordinary floating point variables. The default precision is **short**.

Common

Certain objects called *common areas* have external scope, and may be referenced by any procedure that has a declaration for the name using a

common (*commonareaname*)

attribute. All of the variables declared with a particular **common** attribute are in the same block; the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it is necessary to declare it in a statement of the form

external [[*name*]]

If a name has the external attribute and it is a formal argument of the procedure, then it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, then that name is the actual name of a procedure, as it appears in the corresponding **procedure** statement.

Variable List

The elements of a variable list in a declaration consist of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification is the same form and meaning as the parenthesized list in an **array** attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign may be a parenthesized list of constant expressions, or repeated elements or lists; the total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

The Initial Statement

An initial value may also be specified for a simple variable, array, array element, or member of a structure using a statement of the form

initial [[*var* = *val*]]

The *var* may be a variable name, array element specification, or member of structure. The right side follows the same rules as for an initial value specification in other declaration statements.

EXECUTABLE STATEMENTS

Every useful EFL program contains executable statements, otherwise it would not do anything and would not need to be run. Statements are frequently made up of other statements. Blocks are the most obvious case, but many other forms contain statements as constituents.

To increase the legibility of EFL programs, some of the statement forms can be broken without an explicit continuation. A square (□) in the syntax represents a point where the end of a line will be ignored.

Expression Statements

Subroutine Call

A procedure invocation that returns no value is known as a subroutine call. Such an invocation is a statement. Examples are

```
work(in, out)
run( )
```

Input/output statements (see "Input/Output Statements" under "EXECUTABLE STATEMENTS") resemble procedure invocations but do not yield a value. If an error occurs the program stops.

Assignment Statements

An expression that is a simple assignment (=) or a compound assignment (+= etc.) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, optionally followed by executable statements, followed by a right brace. A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is

```
{
integer i  # this variable is unknown
        # outside the braces
big = 0
do i = 1,n
   if(big < a(i))
        big = a(i)
}
```

Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

If Statement

The simplest of the test statements is the **if** statement, of form

if (*logical-expression*) □ *statement*

The logical expression is evaluated; if it is true, then the *statement* is executed.

If-Else

A more general statement is of the form

if (*logical-expression*) □ *statement-1* □
else □ *statement-2*

If the expression is **true** then *statement-1* is executed, otherwise, *statement-2* is executed. Either of the consequent statements may itself be an **if-else** so a completely nested test sequence is possible:

```
if(x<y)
   if(a<b)
      k = 1
   else
      k = 2
else
   if(a<b)
      m = 1
   else
      m = 2
```

An **else** applies to the nearest preceding un-**elsed if**. A more common use is as a sequential test:

```
if(x==1)
   k = 1
else if(x==3 | x==5)
   k = 2
else
   k = 3
```

Select Statement

A multiway test on the value of a quantity is succinctly stated as a **select** statement, which has the general form

select(*expression* **)** □ *block*

Inside the block two special types of labels are recognized. A prefix of the form

case [*constant*] **:**

marks the statement to which control is passed if the expression in the select has a value equal to one of the case constants. If the expression equals none of these constants, but there is a label **default** inside the select, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until the next **case** or **default** is encountered. The **else-if** example above is better written as

```
select(x)
   {
   case 1:
      k = 1
   case 3,5:
      k = 2
   default:
      k = 3
   }
```

Note that control does not "fall through" to the next case.

Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest (**while**) form is theoretically sufficient, but it is very convenient to have the more general loops available, since each expresses a mode of control that arises frequently in practice.

While Statement

This construct has the form

while (*logical-expression*) ☐ *statement*

The expression is evaluated; if it is true, the statement is executed, and then the test is performed again. If the expression is false, execution proceeds to the next statement.

For Statement

The **for** statement is a more elaborate looping construct. It has the form

for (*initial-statement* , ☐ *logical-expression* ,
☐ *iteration-statement*) ☐ *body-statement*

Except for the behavior of the **next** statement (see "Branch Statement" under "EXECUTABLE STATEMENTS"), this construct is equivalent to

initial-statement
while (*logical-expression*)
{
body-statement
iteration-statement
}

This form is useful for general arithmetic iterations, and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
    n += i
```

Alternatively, the computation could be done by the single statement

```
for( { n = 0 ; i = 1 } , i<=100 , { n += i ; ++i } )
    ;
```

Note that the body of the **for** loop is a null statement in this case. An example of following a linked list will be given later.

Repeat Statement

The statement

repeat ☐ *statement*

executes the statement, then does it again, without any termination test. Obviously, a test inside the *statement* is needed to stop the loop.

Repeat ... Until Statement

The **while** loop performs a test before each iteration. The statement

repeat ☐ *statement* ☐ **until** (*logical-expression*)

executes the *statement,* then evaluates the logical; if the logical is true the loop is complete; otherwise, control returns to the *statement.*

Thus, the body is always executed at least once. The **until** refers t the nearest preceding **repeat** that has not been paired with an **until** In practice, this appears to be the least frequently used loopin construct.

Do Loop

The simple arithmetic progression is a very common one in numerica applications. EFL has a special loop form for ranging over ar ascending arithmetic sequence

do *variable* = *expression-1, expression-2, expression-3*
statement

The variable is first given the value *expression-1.* The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2.* If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

```
t2 = expression-2
t3 = expression-3
for(variable=expression-1, variable<=t2, variable+=t3)
    statement
```

(The compiler translates EFL **do** statements into Fortran DO statements, which are in turn usually compiled into excellent code.) The **do** *variable* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2.* The sum of the first hundred positive integers could be computed by

```
n = 0
do i = 1, 100
        n += i
```

Branch Statements

Most of the need for branch statements in programs can be averted by using the loop and test constructs, but there are programs where they are very useful.

Goto Statement

The most general, and most dangerous, branching statement is the simple unconditional

goto *label*

After executing this statement, the next statement performed is the one following the given label. Inside of a **select** the case labels of that block may be used as labels, as in the following example:

```
select(k)
        {
        case 1:
                error(7)

        case 2:
                k = 2
                goto case 4

        case 3:
                k = 5
                goto case 4

        case 4:
                fixup(k)
                goto default

        default:
                prmsg(" ouch" )
        }
```

(If two **select** statements are nested, the case labels of the outer **select** are not accessible from the inner one.)

Break Statement

A safer statement is one which transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```
repeat
    {
    do a computation
    if ( finished )
        break
    }
```

More general forms permit controlling a branch out of more than one construct.

```
break 3
```

transfers control to the statement following the third loop and/or **select** surrounding the statement. It is possible to specify which type of construct (**for**, **while**, **repeat**, **do**, or **select**) is to be counted. The statement

```
break while
```

breaks out of the first surrounding **while** statement. Either of the statements

```
break 3 for
break for 3
```

will transfer to the statement after the third enclosing **for** loop.

Next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration: the next operation performed is the test of a **while,** the *iteration-statement* of a **for,** the body of a **repeat,** the test of a **repeat...until,** or the increment of a **do.** Elaborations similar to those for **break** are available:

next
next 3
next 3 for
next for 3

A **next** statement ignores **select** statements.

Return

The last statement of a procedure is followed by a return of control to the caller. If it is desired to effect such a return from any other point in the procedure, a

return

statement may be executed. Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression*)

Input/Output Statements

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile**, **rewind**, and **backspace**). These forms may be used either as a primary with a **integer** value or as a statement. If an exception occurs when one of these forms is used as a statement, the result is undefined but will probably be treated as a fatal error. If they are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value an error. The input/output part of EFL very strongly reflects the facilities of Fortran.

Input/Output Units

Each I/O statement refers to a "unit," identified by a small positive integer. Two special units are defined by EFL, the *standard input unit* and the *standard output unit.* These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into *records.* These records may be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file.*

Binary Input/Output

The **readbin** and **writebin** statements transmit data in a machine-dependent but swift manner. The statements are of the form

writebin(*unit* **,** *binary-output-list* **)**
readbin(*unit* **,** *binary-input-list* **)**

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist* (see below) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers in which each of the expressions is a variable name, array element, or structure member.

Formatted Input/Output

The **read** and **write** statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

write(*unit* **,** *formatted-output-list* **)**
read(*unit* **,** *formatted-input-list* **)**

The lists are of the same form as for binary I/O, except that the lists may include format specifications. If the *unit* is omitted, the standard input or output unit is used.

Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form

expression
{ *iolist* }
do-specification { *iolist* }

For formatted I/O, an *ioexpression* may also have the forms

ioexpression : *format-specifier*
: *format-specifier*

A *do-specification* looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

Formats

The following are permissible *format-specifiers*. The quantities w, d, and k must be integer constant expressions.

i(*w*)	integer with *w* digits
f(*w*,*d*)	floating point number of *w* characters, *d* of them to the right of the decimal point.
e(*w*,*d*)	floating point number of *w* characters, *d* of them to the right of the decimal point, with the exponent field marked with the letter **e**
l(*w*)	logical field of width *w* characters, the first of which is **t** or **f** (the rest are blank on output, ignored on input) standing for **true** and **false** respectively

c	character string of width equal to the length of the datum
c(*w*)	character string of width *w*
s(*k*)	skip *k* lines
x(*k*)	skip *k* spaces
" ... "	use the characters inside the string as a Fortran format

If no format is specified for an item in a formatted input/output statement, a default form is chosen.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

Manipulation Statements

The three input/output statements

backspace(*unit*)
rewind(*unit*)
endfile(*unit*)

look like ordinary procedure calls, but may be used either as statements or as integer expressions which yield non-zero if an error is detected. **backspace** causes the specified unit to back up, so that the next read will re-read the previous record, and the next write will over-write it. **rewind** moves the device to its beginning, so that the next input statement will read the first record. **endfile** causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past is an error.

PROCEDURES

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

Procedures Statement

Each procedure begins with a statement of one of the forms

procedure
attributes **procedure** *procedurename*
attributes **procedure** *procedurename* ( )
attributes **procedure** *procedurename* ([*name*])

The first case specifies the main procedure, where execution begins. In the two other cases, the *attributes* may specify precision and type, or they may be omitted entirely. The precision and type of the procedure may be declared in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and no value may be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call. Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

End Statement

Each procedure terminates with a statement

end

Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument, and the procedure may reference the values in the object, and assign to it. Otherwise, the value of the actual is associated with the formal argument, but the procedure may not attempt to change the value of that formal argument.

If the value of one of the arguments is changed in the procedure, it is not permitted that the corresponding actual argument be associated with another formal argument or with a **common** element that is referenced in the procedure.

Execution and Return Values

After actual and formal arguments have been associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed. If the procedure is a function (has a declared type), and a **return**(*value*) is executed, the value is coerced to the correct type and precision and returned.

Known Functions

A number of functions are known to EFL, and need not be declared. The compiler knows the types of these functions. Some of them are *generic;* i.e., they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

Minimum and Maximum Functions

The generic functions are **min** and **max.** The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that may take different numbers of arguments in different calls. If any of the arguments are **long real** then the result is **long real**. Otherwise, if any of the arguments are **real** then the result is **real;** otherwise all the arguments and the result must be **integer.** Examples are

```
min(5, x, -3.20)
max(i, z)
```

Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments the type of the result is identical to the type of the argument; for complex arguments the type of the result is the real of the same precision.

Elementary Functions

The following generic functions take arguments of **real**, **long real**, or **complex** type and return a result of the same type:

sin	**sine function**
cos	**cosine function**
exp	**exponential function (e^x).**
log	**natural (base *e*) logarithm**
log10	**common (base 10) logarithm**
sqrt	**square root function ($\sqrt{x}$).**

In addition, the following functions accept only **real** or **long real** arguments:

atan	$atan(x)=\tan^{-1}x$
atan2	$atan2(x,y)=\tan^{-1}\frac{x}{y}$

Other Generic Functions

The **sign** functions takes two arguments of identical type; **sign**(x,y) = $sgn(y)|x|$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

ATAVISMS

Certain facilities are included in the EFL language to ease the conversion of old Fortran or Ratfor programs to EFL.

Escape Lines

In order to make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. A line that begins with a percent sign ("%") is copied through to the output, with the percent sign removed but no other change. Inside of a procedure, each escape line is treated as an executable statement. If a sequence of lines constitute a continued Fortran statement, they should be enclosed in braces.

Call Statement

A subroutine call may be preceded by the keyword **call.**

```
call joe
call work(17)
```

Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Fortran	EFL
double precision	**long real**
function	**procedure**
subroutine	**procedure** *(untyped)*

Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types may be given in an **implicit** statement, with syntax

implicit (*letter-list*) *type*

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement appears, the following rules are assumed:

```
implicit (a–h, o–z) real
implicit (i–n) integer
```

Computed Goto

Fortran contains an indexed multi-way branch; this facility may be used in EFL by the computed GOTO:

goto ([[*label*]]), *expression*

The expression must be of type integer and be positive but be no larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

Goto Statement

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in

go to xyz

Dot Names

Fortran uses a restricted character set, and represents certain operators by multi-character sequences. There is an option (**dots=on**; see " COMPILER OPTIONS") which forces the compiler to recognize the forms in the second column below:

<	**.lt.**
<=	**.le.**
>	**.gt.**
>=	**.ge.**
==	**.eq.**
~=	**.ne.**
&	**.and.**
¦	**.or.**
&&	**.andand.**
¦¦	**.oror.**
~	**.not.**
true	**.true.**
false	**.false.**

In this mode, no structure element may be named **lt, le,** etc. The readable forms in the left column are always recognized.

Complex Constants

A complex constant may be written as a parenthesized list of real quantities, such as

(1.5, 3.0)

The preferred notation is by a type coercion,

complex(1.5, 3.0)

Function Values

The preferred way to return a value from a function in EFL is the **return**(*value*) construct. However, the name of the function acts as a variable to which values may be assigned; an ordinary **return** statement returns the last value assigned to that name as the function value.

Equivalence

A statement of the form

equivalence $v_1, v_2, ..., v_n$

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

Minimum and Maximum Functions

There are a number of non-generic functions in this category, which differ in the required types of the arguments and the type of the return value. They may also have variable numbers of arguments, but all the arguments must have the same type.

Function	*Argument Type*	*Result Type*
amin0	**integer**	**real**
amin1	**real**	**real**
min0	**integer**	**integer**
min1	**real**	**integer**
dmin1	**long real**	**long real**
amax0	**integer**	**real**
amax1	**real**	**real**
max0	**integer**	**integer**
max1	**real**	**integer**
dmax1	**long real**	**long real**

COMPILER OPTIONS

A number of options can be used to control the output and to tailor it for various compilers and systems. The defaults chosen are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Options are set with statements of the form

option [*opt*]

where each *opt* is of one of the forms

optionname
optionname = *optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names **yes** and **no** apply to a number of options.

Default Options

Each option has a default setting. It is possible to change the whole set of defaults to those appropriate for a particular environment by using the **system** option. At present, the only valid values are **system=unix** and **system=gcos.**

Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no.**

Input/Output Error Handling

The **ioerror** option can be given three values: **none** means that none of the I/O statements may be used in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses ERR= and END= clauses. The implementation of the **fortran77** form uses IOSTAT= clauses.

Continuation Conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option **continue=column1** puts an ampersand (**&**) in the first column of the continued lines instead.

Default Formats

If no format is specified for a datum in an iolist for a **read** or **write** statement, a default is provided. The default formats can be changed by setting certain options

Option	*Type*
iformat	integer
rformat	real
dformat	long real
zformat	complex
zdformat	long complex
lformat	logical

The associated value must be a Fortran format, such as

option rformat=f22.6

Alignments and Sizes

In order to implement **character** variables, structures, and the **sizeof** and **lengthof** operators, it is necessary to know how much space various Fortran data types require, and what boundary alignment properties they demand. The relevant options are

Fortran Type	*Size Option*	*Alignment Option*
integer	**isize**	**ialign**
real	**rsize**	**ralign**
long real	**dsize**	**dalign**
complex	**zsize**	**zalign**
logical	**lsize**	**lalign**

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option **charperint** gives the number of characters per **integer** variable.

Default Input/Output Units

The options **ftnin** and **ftnout** are the numbers of the standard input and output units. The default values are **ftnin=5** and **ftnout=6.**

Miscellaneous Output Control Options

Each Fortran procedure generated by the compiler will be preceded by the value of the **procheader** option.

No Hollerith strings will be passed as subroutine arguments if **hollincall=no** is specified.

The Fortran statement numbers normally start at 1 and increase by 1. It is possible to change the increment value by using the **deltastno** option.

EXAMPLES

In order to show the flavor or programming in EFL, we present a few examples. They are short, but show some of the convenience of the language.

File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure  # main program
character(100) line

while( read( , line) == 0 )
                    write( , line)
end
```

Since **read** returns zero until the end of file (or a read error), this program keeps reading and writing until the input is exhausted.

Matrix Multiplication

The following procedure multiplies the $m \times n$ matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula $c_{ij} = \sum a_{ik} b_{kj}$.

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)
do i = 1,m
do j = 1,p
        {
        c(i,j) = 0
        do k = 1,n
                c(i,j) += a(i,k) * b(k,j)
        }
end
```

Searching a Linked List

Assume we have a list of pairs of numbers (x,y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value.

```
define LAST       0
define NOTFOUND          -1

integer procedure val(list, first, x)

#  list is an array of structures.
#  Each structure contains a thread index value,
#  an x, and a y value.
struct
          {
          integer nextindex
          integer x, y
          } list(*)
integer first, p, arg

for(p = first , p~=LAST && list(p).x<=x ,
   p = list(p).nextindex)
          if(list(p).x == x)
                    return( list(p).y )

return(NOTFOUND)
end
```

The search is a single **for** loop that begins with the head of the list and examines items until either the list is exhausted (p==LAST) or until it is known that the specified value is not on the list (list(p).x > x). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the **list(p)** reference. Therefore, the **&&** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex**.

Walking a Tree

As an example of a more complicated problem, let us imagine we have an expression tree stored in a common area, and that we want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a

left and a right descendant. In a recursive language, such a tree walk would be implement by the following simple pseudocode:

```
if this node is a leaf
        print its value
otherwise
        print a left parenthesis
        print the left node
        print the operator
        print the right node
        print a right parenthesis
```

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value.

```
procedure walk(first)       # print an expression tree

integer first           # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
        {
        character(1) op
        integer leftp, rightp
        real val
        } tree(100)     # array of structures

struct
        {
        integer nextstate
        integer nodep
        } stackframe(100)

define NODE tree(currentnode)
define STACK        stackframe(stackdepth)

#  nextstate values
define DOWN         1
define LEFT 2
define RIGHT        3
```

```
#  initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first

while( stackdepth > 0 )
        {
        currentnode = STACK.nodep
        select(STACK.nextstate)
                {
                case DOWN:
                        if(NODE.op == " " )  # a leaf
                                {
                                outval( NODE.val )
                                stackdepth -= 1
                                }
                        else    { # a binary operator node
                                outch( " (" )
                                STACK.nextstate = LEFT
                                stackdepth += 1
                                STACK.nextstate = DOWN
                                STACK.nodep = NODE.leftp
                                }
                case LEFT:
                        outch( NODE.op )
                        STACK.nextstate = RIGHT
                        stackdepth += 1
                        STACK.nextstate = DOWN
                        STACK.nodep = NODE.rightp
                case RIGHT:
                        outch( ")" )
                        stackdepth -= 1
                }
        }
end
```

PORTABILITY

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the **fortran77** option is specified).

Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

Character String Copying

The subroutine **eflasc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```
subroutine eflasc(a, la, b, lb)
integer a(*), la, b(*), lb
```

and it must copy the first **lb** characters from **b** to the first **la** characters of **a.**

Character String Comparisons

The function **eflcmc** is invoked to determine the order of two character strings. The declaration is

```
integer function eflcmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string **a** of length **la** precedes the string **b** of length **lb.** It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

DIFFERENCES BETWEEN RATFOR AND EFL

There are a number of differences between Ratfor and EFL, since EFL is a defined language while Ratfor is the union of the special control structures and the language accepted by the underlying Fortran compiler. Ratfor running over Standard Fortran is almost a subset of EFL. Most of the features described in the "ATAVISMS" are present to ease the conversion of Ratfor programs to EFL.

There are a few incompatibilities: The syntax of the **for** statement is slightly different in the two languages: the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements may be compound statements in EFL because of this change). The input/output syntax is quite different in the two languages, and there is no FORMAT statement in EFL. There are no ASSIGN or assigned GOTO statements in EFL.

The major linguistic additions are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions, and provides a more uniform syntax. (One need not worry about the Fortran/Ratfor restrictions on subscript or DO expression forms, for example.)

COMPILER

Current Version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described above except for **long complex** numbers.

Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) on which the error was detected. Warnings are given for variables that are used but not explicitly declared.

Quality of Fortran Produced

The Fortran produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the Fortran code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded GOTO and CONTINUE statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (See " EXAMPLES" .)

```
      subroutine matmul(a, b, c, m, n, p)
      integer m, n, p
      double precision a(m, n), b(n, p), c(m, p)
      integer i, j, k
      do  3 i = 1, m
          do  2 j = 1, p
              c(i, j) = 0
              do  1 k = 1, n
                  c(i, j) = c(i, j)+a(i, k)*b(k, j)
1                 continue
2             continue
3         continue
      end
```

The following is the procedure for the tree walk:

```
      subroutine walk(first)
      integer first
      common /nodes/ tree
      integer tree(4, 100)
      real tree1(4, 100)
      integer staame(2, 100), stapth, curode
      integer const1(1)
      equivalence (tree(1,1), tree1(1,1))
      data const1(1)/4h     /
c print out an expression tree
c index of root node
c array of structures
c   nextstate values
c   initialize stack with root node
      stapth = 1
      staame(1, stapth) = 1
      staame(2, stapth) = first
   1  if (stapth .le. 0) goto  9
         curode = staame(2, stapth)
         goto  7
   2        if (tree(1, curode) .ne. const1(1)) goto 3
               call outval(tree1(4, curode))
c a leaf
               stapth = stapth-1
               goto  4
   3           call outch(1h()
c a binary operator node
               staame(1, stapth) = 2
               stapth = stapth+1
               staame(1, stapth) = 1
               staame(2, stapth) = tree(2, curode)
   4        goto  8
   5        call outch(tree(1, curode))
            staame(1, stapth) = 3
            stapth = stapth+1
            staame(1, stapth) = 1
            staame(2, stapth) = tree(3, curode)
            goto  8
   6        call outch(1h))
            stapth = stapth-1
            goto  8
```

```
7           if (staame(1, stapth) .eq. 3) goto  6
            if (staame(1, stapth) .eq. 2) goto  5
            if (staame(1, stapth) .eq. 1) goto  2
8       continue
        goto  1
9   continue
    end
```

CONSTRAINTS ON EFL

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character external names), but others are sweeping (lack of pointer variables). The following paragraphs describe the major limitations imposed by Fortran.

External Names

External names (procedure and COMMON block names) must be no longer than six characters in Fortran. Further, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

Procedure Interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran: a procedure name may only be passed as an argument or be invoked; it cannot be stored. Fortran (and EFL) would be noticeably simpler if a procedure variable mechanism were available.

Pointers

The most grievous problem with Fortran is its lack of a pointer-like data type. The implementation of the compiler would have been far easier if certain hard cases could have been handled by pointers. Further, the language could have been simplified considerably if pointers were accessible in Fortran. (There are several ways of simulating pointers by using subscripts, but they founder on the problems of external variables and initialization.)

Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. (Recursive procedures with arguments can be simulated only with great pain.)

Storage Allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using COMMON blocks.

Chapter 12

The Curses and Terminfo Package

Chapter 12

THE CURSES AND TERMINFO PACKAGE

INTRODUCTION

This chapter is an introduction to **curses**(3X) and **terminfo**(4). It is intended for the programmer who must write a screen-oriented program using the **curses** package. Several example programs are discussed. The example programs can be found in Chapter 13. This chapter also documents each **curses** function. It is intended as a reference.

For **curses** to be able to produce terminal dependent output, it has to know what kind of terminal you have. The UNIX system convention for this is to put the name of the terminal in the variable TERM in the environment. Thus, a user on a DEC VT100 would set `TERM=vt100` when logging in. Curses uses this convention.

Output

A program using **curses** always starts by calling `initscr()`. (See Figure 12-1.) Other modes can then be set as needed by the program. Possible modes include `cbreak()`, and `idlok(stdscr, TRUE)`. These modes will be explained later. During the execution of the program, output to the screen is done with routines such as `addch(ch)` and `printw(fmt,args)`. (These routines behave just like `putchar` and `printf` except that they go through **curses**.) The cursor can be moved with the call `move(row,col)`. These routines only output to a data structure called a *window*, not to the actual screen. A window is a representation of a CRT screen, containing such things as an array of characters to be displayed on the screen, a cursor, a current set of video attributes, and various modes and options. You don't need to worry about windows unless you use more than one of them, except to realize that a window is buffering your requests to output to the screen.

To send all accumulated output, it is necessary to call `refresh()`.

(This can be thought of as a `flush`.) Finally, before the program exits, it should call `endwin()`, which restores all terminal settings and positions the cursor at the bottom of the screen.

```
#include <curses.h>
...
   initscr(); /* Initialization */

   cbreak();/* Various optional mode settings */
   nonl();
   noecho();
...
   while (!done) {/* Main body of program */
      ...
      /* Sample calls to draw on screen */
      move(row, col);
      addch(ch);
      printw("Formatted print with value %d\n", value)
      ...
      /* Flush output */
      refresh();
      ...
   }

   endwin();/* Clean up */
   exit(0);
```

Figure 12-1 - Framework of a Curses Program

See the program **scatter** in Chapter 13 for an example program. This program reads a file, and displays the file in a random order on the screen. Some programs assume all screens are 24 lines by 80 columns. It is important to understand that many are not. The variables `LINES` and `COLS` are defined by `initscr` with the current screen size. Programs should use them instead of assuming a 24x80 screen.

No output to the terminal actually happens until `refresh` is called. Instead, routines such as `move` and `addch` draw on a window data structure called `stdscr` (standard screen). **Curses** always keeps track of what is on the physical screen, as well as what is in `stdscr`.

When `refresh` is called, **curses** compares the two screen images and sends a stream of characters to the terminal that will turn the current screen into what is desired. Curses considers many different ways to do this, taking into account the various capabilities of the terminal, and similarities between what is on the screen and what is desired. It usually outputs as few characters as is possible. This function is called *cursor optimization* and is the source of the name of the **curses** package.

NOTE: Due to the hardware scrolling of terminals, writing to the lower righthand character position is impossible.

Input

Curses can do more than just draw on the screen. Functions are also provided for input from the keyboard. The primary function is `getch()` which waits for the user to type a character on the keyboard, and then returns that character. This function is like `getchar` except that it goes through **curses**. Its use is recommended for programs using the `cbreak()` or `noecho()` options, since several terminal or system dependent options become available that are not possible with `getchar`.

Options available with `getch` include `keypad` which allows extra keys such as arrow keys, function keys, and other special keys that transmit escape sequences, to be treated as just another key. (The values returned for these keys are listed below.) `KEY_LEFT` in `curses.h`. The values for these keys are over octal 400, so they should be stored in a variable larger than a **char**.) *nodelay* mode causes the value -1 to be returned if there is no input waiting. Normally, `getch` will wait until a character is typed. Finally, the routine `getstr(str)` can be called, allowing input of an entire line, up to a newline. This routine handles echoing and the erase and kill characters of the user. Examples of the use of these options are in later example programs.

The following function keys might be returned by **getch** if **keypa** has been enabled. Note that not all of these are currently supporte due to lack of definitions in **terminfo** or the terminal no transmitting a unique code when the key is pressed.

Name	*Value*	*Key name*
KEY_BREAK	0401	Break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	Backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 keys is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	Soft (partial) reset (unreliable)
KEY_RESET	0531	Reset or hard reset (unreliable)
KEY_PRINT	0532	Print or copy
KEY_LL	0533	Home down or bottom (lower left)

See the program **show** in Chapter 13 for an example use of **getch**. **Show** pages through a file, showing one screen full each time the user presses the space bar. By creating an input file for **show** made

up of 24 line pages, each segment varying slightly from the previous page, nearly any exercise for **curses** can be created. Such input files are called *show scripts*.

In the **show** program, `cbreak` is called so that the user can press the space bar without having to hit return. The `noecho` function is called to prevent the space from echoing in the middle of a `refresh`, messing up the screen. The `nonl` function is called to enable more screen optimization. The `idlok` function is called to allow insert and delete line, since many show scripts are constructed to duplicate bugs caused by that feature. The `clrtoeol` and `clrtobot` functions clear from the cursor to the end of the line and screen, respectively.

Highlighting

The function `addch` always draws two things on a window. In addition to the character itself, a set of *attributes* is associated with the character. These attributes cover various forms of highlighting of the character. For example, the character can be put in reverse video, bold, or be underlined. You can think of the attributes as the color of the ink used to draw the character.

A window always has a set of *current attributes* associated with it. The current attributes are associated with each character as it is written to the window. The current attributes can be changed with a call to `attrset(attrs)`. (Think of this as dipping the window's pen in a particular color ink.) The names of the attributes are `A_STANDOUT`, `A_REVERSE`, `A_BOLD`, `A_DIM`, `A_INVIS`, and `A_UNDERLINE`. For example, to put a word in bold, the code in Figure 12-2 might be used. The word "boldface" will be shown in bold.

```
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
...
refresh();
```

Figure 12-2 - Use of attributes.

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, **curses** will attempt to find a substitute attribute. If none is possible, the attribute is ignored.

One particular attribute is called *standout*. This attribute is used to make text attract the attention of the user. The particular hardware attribute used for standout varies from terminal to terminal, and is chosen to be the most visually pleasing attribute the terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or inverse video, but instead just need to highlight some text. For such applications, the `A_STANDOUT` attribute is recommended. Two convenient functions, `standout()` and `standend()` turn on and off this attribute.

Attributes can be turned on in combination. Thus, to turn on blinking bold text, use `attrset(A_BLINK|A_BOLD)`. Individual attributes can be turned on and off with `attron` and `attroff` without affecting other attributes.

For an example program using attributes, see **highlight**. The program takes a text file as input and allows embedded escape sequences to control attributes. In this example program, `\U` turns on underlining, `\B` turns on bold, and `\N` restores normal text. Note the initial call to `scrollok`. This allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw

past the bottom of the screen, **curses** will automatically scroll the terminal up a line and call `refresh`.

Highlight comes about as close to being a filter as is possible with **curses**. It is not a true filter, because **curses** must "take over" the CRT screen. In order to determine how to update the screen, it must know what is on the screen at all times. This requires **curses** to clear the screen in the first call to `refresh`, and to know the cursor position and screen contents at all times.

Multiple Windows

A window is a data structure representing all or part of the CRT screen. It has room for a two dimensional array of characters, attributes for each character (a total of 16 bits per character: 7 for text and 9 for attributes) a cursor, a set of current attributes, and a number of flags. Curses provides a full screen window, called `stdscr`, and a set of functions that use `stdscr`. Another window is provided called `curscr`, representing the physical screen.

It is important to understand that a window is only a data structure. Use of more than one window does not imply use of more than one terminal, nor does it involve more than one process. A window is merely an object which can be copied to all or part of the terminal screen. The current implementation of **curses** does not allow windows which are bigger than the screen.

The programmer can create additional windows with the function `newwin(lines, cols, begin_row, begin_col)` will return a pointer to a newly created window. The window will be `lines` by `cols`, and the upper left corner of the window will be at screen position `(begin_row, begin_col)`. All operations that affect `stdscr` have corresponding functions that affect an arbitrary named window. Generally, these functions have names formed by putting a "w" on the front of the `stdscr` function, and the window name is added as the first parameter. Thus, `waddch(mywin, c)` would write the character `c` to window `mywin`. The `wrefresh(win)` function is used to flush the contents of a window to the screen.

Windows are useful for maintaining several different screen images, and alternating the user among them. Also, it is possible to

subdivide the screen into several windows, refreshing each of them as desired. When windows overlap, the contents of the screen will be the more recently refreshed window.

In all cases, the non-w version of the function calls the w version of the function, using `stdscr` as the additional argument. Thus, a call to `addch(c)` results in a call to `waddch(stdscr, c)`.

The program **window** is an example of the use of multiple windows. The main display is kept in `stdscr`. When the user temporarily wants to put something else on the screen, a new window is created covering part of the screen. A call to `wrefresh` on that window causes the window to be written over `stdscr` on the screen. Calling `refresh` on `stdscr` results in the original window being redrawn on the screen. Note the calls to `touchwin` before writing out an overlapping window. These are necessary to defeat an optimization in **curses**. If you have trouble refreshing a new window which overlaps an old window, it may be necessary to call `touchwin` on the new window to get it completely written out.

For convenience, a set of "move" functions are also provided for most of the common functions. These result in a call to `move` before the other function. For example, `mvaddch(row, col, c)` is the same as `move(row, col); addch(c)`. Combinations, e.g. `mvwaddch(row, col, win, c)` also exist.

Multiple Terminals

Curses can produce output on more than one terminal at once. This is useful for single process programs that access a common database, such as multi-player games. Output to multiple terminals is a difficult business, and **curses** does not solve all the problems for the programmer. It is the responsibility of the program to determine the file name of each terminal line, and what kind of terminal is on each of those lines. The standard method, checking `$TERM` in the environment, does not work, since each process can only examine its own environment. Another problem that must be solved is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. Nonetheless, a program wishing to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons

would also make this inappropriate. However, for some applications, such as an inter-terminal communication program, or a program that takes over unused tty lines, it would be appropriate.) A typical solution requires the user logged in on each line to run a program that notifies the master program that the user is interested in joining the master program, telling it the notification program's process id, the name of the tty line and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program, and all programs exit.

Curses handles multiple terminals by always having a *current terminal*. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary **curses** routines.

References to terminals have type `struct screen *`. A new terminal is initialized by calling `newterm(type, fd)`. `newterm` returns a screen reference to the terminal being set up. `type` is a character string, naming the kind of terminal being used. `fd` is a stdio file descriptor to be used for input and output to the terminal. (If only output is needed, the file can be open for output only.) This call replaces the normal call to `initscr`, which calls `newterm(getenv(''TERM''), stdout)`.

To change the current terminal, call "`set_term(sp)`" where `sp` is the screen reference to be made current. `set_term` returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with `newterm`. Options such as `cbreak` and `noecho` must be set separately for each terminal. The functions `endwin` and `refresh` must be called separately for each terminal. See Figure 12-3 for a typical scenario to output a message to each terminal.

```
for (i=0; i<nterm; i++) {
    set_term(terms[i]);
    mvaddstr(0, 0, "Important message");
    refresh();
}
```

Figure 12-3 - Sending a message to several terminals

See the sample program **two** for a full example. This program pages through a file, showing one page to the first terminal and the next page to the second terminal. It then waits for a space to be typed on either terminal, and shows the next page to the terminal typing the space. Each terminal has to be separately put into nodelay mode. Since no standard multiplexor is available in current versions of the UNIX system, it is necessary to either busy wait, or call `sleep(1);`, between each check for keyboard input. This program sleeps for a second between checks.

The **two** program is just a simple example of two terminal **curses**. It does not handle notification, as described above, instead it requires the name and type of the second terminal on the command line. As written, the command `sleep 100000` must be typed on the second terminal to put it to sleep while the program runs, and the first user must have both read and write permission on the second terminal.

Low Level Terminfo Usage

Some programs need to use lower level primitives than those offered by **curses**. For such programs, the *terminfo level* interface is offered. This interface does not manage your CRT screen, but rather gives you access to strings and capabilities which you can use yourself to manipulate the terminal.

Programmers are discouraged from using this level. Whenever possible, the higher level **curses** routines should be used. This will make your program more portable to other UNIX systems and to a wider class of terminals. Curses takes care of all the glitches and

misfeatures present in physical terminals, but at the terminfo level you must deal with them yourself. Also, it cannot be guaranteed that this part of the interface will not change or be upward compatible with previous releases.

There are two circumstances when it is proper to use terminfo. The first is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. The second situation is when writing a filter. A typical filter does one transformation on the input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of terminfo is indicated.

A program writing at the terminfo level uses the framework shown in Figure 12-4.

```
#include <curses.h>
#include <term.h>
...
    setupterm(0, 1, 0);
    ...
    putp(clear_screen);
    ...
    reset_shell_mode();
    exit(0);
```

Figure 12-4 - Terminfo level framework

Initialization is done by calling `setupterm`. Passing the values 0, 1, and 0 invoke reasonable defaults. If `setupterm` can't figure out what kind of terminal you are on, it will print an error message and exit. The program should call `reset_shell_mode` before it exits.

Global variables with names like `clear_screen` and `cursor_address` are defined by the call to `setupterm`. They can

be output using **putp**, or also using **tputs**, which allows the programmer more control. These strings *should not* be directly output to the terminal using **printf** since they contain padding information. A program that directly outputs strings will fail on terminals that require padding, or that use the xon/xoff flow control protocol.

In the terminfo level, the higher level routines described previously are not available. It is up to the programmer to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo**(4).

The example program **termhl** shows simple use of terminfo. It is a version of **highlight** that uses terminfo instead of **curses**. This version can be used as a filter. The strings to enter bold and underline mode, and to turn off all attributes, are used.

This program is more complex than it need be in order to illustrate some properties of terminfo. The routine **vidattr** could have been used instead of directly outputting **enter_bold_mode**, **enter_underline_mode**, and **exit_attribute_mode**. In fact, the program would be more robust if it did since there are several ways to change video attribute modes. This program was written to illustrate typical use of terminfo.

The function **tputs(cap, affcnt, outc)** applies padding information. Some capabilities contain strings like $<20>, which means to pad for 20 milliseconds. **tputs** generates enough pad characters to delay for the appropriate time. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, **insert_line** may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention **affcnt** is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since **affcnt** is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, **affcnt** is always 1 and **outc** always just calls **putchar**. For these programs, the routine **putp(cap)** is

a convenient abbreviation. **termhl** could be simplified by using `putp`.

Note also the special check for the `underline_char` capability. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **termhl** keeps track of the current mode, and if the current character is supposed to be underlined, will output `underline_char` if necessary. Low level details such as this are precisely why the **curses** level is recommended over the terminfo level. Curses takes care of terminals with different methods of underlining and other CRT functions. Programs at the terminfo level must handle such details themselves.

A Larger Example

For a final example, see the program **editor**. This program is a very simple screen editor, patterned after the **vi** editor. The program illustrates how to use **curses** to write a screen editor. This editor keeps the buffer in `stdscr` to keep the program simple - obviously a real screen editor would keep a separate data structure. Many simplifications have been made here - no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. The routine to write out the file illustrates the use of the `mvinch` function, which returns the character in a window at a given position. The data structure used here does not have a provision for keeping track of the number of characters in a line, or the number of lines in the file, so trailing blanks are eliminated when the file is written out.

The program uses built-in **curses** functions `insch`, `delch`, `insertln`, and `deleteln`. These functions behave much as the similar functions on intelligent terminals behave, inserting and deleting a character or line.

The command interpreter accepts not only ASCII characters, but also special keys. This is important - a good program will accept both. (Some editors are *modeless*, using nonprinting characters for commands. This is largely a matter of taste - the point being made

here is that both arrow keys and ordinary ASCII characters should be handled.) It is important to handle special keys because this makes it easier for a new user to learn to use your program if he can use the arrow keys, instead of having to memorize that "h" means left, "j" means down, "k" means up, and "l" means right. On the other hand, not all terminals have arrow keys, so your program will be usable on a larger class of terminals if there is an ASCII character which is a synonym for each special key. Also, experienced users dislike having to move their hands from the "home row" position to use special keys, since they can work faster with alphabetic keys.

Note the call to `mvaddstr` in the input routine. `addstr` is roughly like the C `fputs` function, which writes out a string of characters. Like `fputs`, `addstr` does not add a trailing newline. It is the same as a series of calls to `addch` using the characters in the string. `mvaddstr` is the mv version of `addstr`, which moves to the given location in the window before writing.

The control-L command illustrates a feature most programs using **curses** should add. Often some program beyond the control of **curses** has written something to the screen, or some line noise has messed up the screen beyond what **curses** can keep track of. In this case, the user usually types control-L, causing the screen to be cleared and redrawn. This is done with the call to `clearok(curscr)`, which sets a flag causing the next `refresh` to first clear the screen. Then `refresh` is called to force the redraw.

Note also the call to `flash()`, which flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within earshot of the user. The routine `beep()` can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to `beep` will flash the screen.)

Another important point is that the input command is terminated by control-D, not escape. It is very tempting to use escape as a command, since escape is one of the few special keys which is available on every keyboard. (Return and break are the only others.) However, using escape as a separate key introduces an ambiguity.

Most terminals use sequences of characters beginning with escape ("escape sequences") to control the terminal, and have special keys that send escape sequences to the computer. If the computer sees an escape coming from the terminal, it cannot tell for sure whether the user pushed the escape key, or whether a special key was pressed. Curses handles the ambiguity by waiting for up to one second. If another character is received during this second, and if that character might be the beginning of a special key, more input is read (waiting for up to one second for each character) until either a full special key is read, one second passes, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape, then to type another key quickly, which causes **curses** to think a special key has been pressed. Also, there is a one second pause until the escape can be passed to the user program, resulting in slower response to the escape key. Many existing programs use escape as a fundamental command, which cannot be changed without infuriating a large class of users. Such programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a timeout solution. The moral is clear: when designing your program, avoid the escape key.

LIST OF ROUTINES

This section describes all the routines available to the programmer in the **curses** package. The routines are organized by function. For an alphabetical list, see **curses**(3X).

Structure

All programs using **curses** should include the file `<curses.h>`. This file defines several **curses** functions as macros, and defines several global variables and the datatype `WINDOW`. References to windows are always of type `WINDOW *`. Curses also defines `WINDOW *` constants `stdscr` (the standard screen, used as a default to routines expecting a window), and `curscr` (the current screen, used only for certain low level operations like clearing and redrawing a garbaged screen). Integer constants `LINES` and `COLS` are defined, containing the size of the screen. Constants `TRUE` and `FALSE` are defined, with values 1 and 0, respectively. Additional constants which are values returned from most **curses** functions are `ERR` and

OK. OK is returned if the function could be properly completed, and ERR is returned if there was some error, such as moving the cursor outside of a window.

The include file <curses.h> automatically includes <stdio.h> and an appropriate tty driver interface file, currently either <sgtty.h*> or <termio.h>. Including <stdio.h> again is harmless but wasteful, including <sgtty.h> again will usually result in a fatal error.

A program using **curses** should include the loader option -lcurses in the makefile. This is true for both the **terminfo** level and the **curses** level. The compilation flag -DMINICURSES can be included if you restrict your program to a small subset of **curses** concerned primarily with screen output and optimization. The routines possible with mini-curses are listed in " Mini-Curses" under " OPERATION DETAILS."

Initialization

These functions are called when initializing a program.

`initscr()`
The first function called should always be `initscr`. This will determine the terminal type and initialize **curses** data structures. `initscr` also arranges that the first call to `refresh` will clear the screen.

`endwin()`
A program should always call `endwin` before exiting. This function will restore tty modes, move the cursor to the lower left corner of the screen, reset the terminal into the proper non-visual mode, and tear down all appropriate data structures.

---

* The driver interface <sgtty.h> is a tty driver interface used in other versions of the UNIX system.

`newterm(type, fd)`
A program which outputs to more than one terminal should use `newterm` instead of `initscr`. `newterm` should be called once for each terminal. It returns a variable of type `SCREEN *` which should be saved as a reference to that terminal. The arguments are the type of the terminal (a string) and a stdio file descriptor (`FILE*`) for output to the terminal. The file descriptor should be open for both reading and writing if input from the terminal is desired. The program should also call `endwin` for each terminal being used (see `set_term` below). If an error occurs, the value `NULL` is returned.

`set_term(new)`
This function is used to switch to a different terminal. The screen reference `new` becomes the new current terminal. The previous terminal is returned by the function. All other calls affect only the current terminal.

`longname()`
This function returns a pointer to a static area containing a verbose description of the current terminal. It is defined only after a call to `initscr`, `newterm`, or `setupterm`.

Option Setting

These functions set options within **curses**. In each case, `win` is the window affected, and `bf` is a boolean flag with value `TRUE` or `FALSE` indicating whether to enable or disable the option. All options are initially `FALSE`. It is not necessary to turn these options off before calling `endwin`.

`clearok(win,bf)`
If set, the next call to `wrefresh` with this window will clear the screen and redraw the entire screen. If `win` is `curscr`, the next call to `wrefresh` with any window will cause the screen to be cleared. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

`idlok(win,bf)`
If enabled, **curses** will consider using the hardware insert/delete line feature of terminals so equipped. If disabled, **curses** will never use this feature. The insert/delete character feature is always

considered. Enable this option only if your application needs insert/delete line, for example, for a screen editor. It is disabled by default because insert/delete line tends to be visually annoying when used in applications where it isn't really needed. If insert/delete line cannot be used, **curses** will redraw the changed portions of all lines that do not match the desired line.

`keypad(win,bf)`
This option enables the keypad of the users terminal. If enabled, the user can press a function key (such as an arrow key) and `getch` will return a single value representing the function key. If disabled, **curses** will not treat function keys specially. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option will turn on the terminal keypad.

`leaveok(win,bf)`
Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

`meta(win,bf)`
If enabled, characters returned by `getch` are transmitted with all 8 bits, instead of stripping the highest bit. The value `OK` is returned if the request succeeded, the value `ERR` is returned if the terminal or system is not capable of 8-bit input.

Meta mode is useful for extending the non-text command set in applications where the terminal has a meta shift key. Curses takes whatever measures are necessary to arrange for 8-bit input. On other versions of UNIX systems, raw mode will be used. On our systems, the character size will be set to 8, parity checking disabled, and stripping of the 8th bit turned off.

Note that 8-bit input is a fragile mode. Many programs and networks only pass 7 bits. If any link in the chain from the terminal to the application program strips the 8th bit, 8-bit input is impossible.

`nodelay(win,bf)`
This option causes `getch` to be a non-blocking call. If no input is ready, `getch` will return -1. If disabled, `getch` will hang until a key is pressed.

`intrflush(win,bf)`
If this option is enabled when an interrupt key is pressed on the keyboard (interrupt, quit, suspend), all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt but causing **curses** to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default is for the option to be enabled. This option depends on support in the underlying teletype driver.

`typeahead(fd)`
Sets the file descriptor for typeahead check. **fd** should be an integer returned from `open` or `fileno`. Setting typeahead to -1 will disable typeahead check. By default, file descriptor 0 (stdin) is used. Typeahead is checked independently for each screen, and for multiple interactive terminals it should probably be set to the appropriate input for each screen. A call to `typeahead` always affects only the current screen.

`scrollok(win,bf)`
This option controls what happens when the cursor of a window is moved off the edge of the window, either from a newline on the bottom line, or typing the last character of the last line. If disabled, the cursor is left on the bottom line. If enabled, `wrefresh` is called on the window, and then the physical terminal and window are scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call `idlok`.

`setscrreg(t,b)`
`wsetscrreg(win,t,b)`
These functions allow the user to set a software scrolling region in a window `win` or `stdscr`. `t` and `b` are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok** are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in

the VT100. Only the text of the window is scrolled. If **idlok** is enabled and the terminal has either a scrolling region or insert/delete line capability, they will probably be used by the output routines.

Terminal Mode Setting

These functions are used to set modes in the tty driver. The initial mode usually depends on the setting when the program was called: the initial modes documented here represent the normal situation.

`cbreak()`
`nocbreak()`
These two functions put the terminal into and out of **CBREAK** mode. In this mode, characters typed by the user are immediately available to the program. When out of this mode, the teletype driver will buffer characters typed until newline is typed. Interrupt and flow control characters are unaffected by this mode. Initially the terminal is not in CBREAK mode. Most interactive programs using **curses** will set this mode.

`echo()`
`noecho()`
These functions control whether characters typed by the user are echoed as typed. Initially, characters typed are echoed by the teletype driver. Authors of many interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing.

`nl()`
`nonl()`
These functions control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations, **curses** is able to make better use of the linefeed capability, resulting in faster cursor motion.

`raw()`
`noraw()`
The terminal is placed into or out of raw mode. Raw mode is similar to cbreak mode in that characters typed are immediately passed

through to the user program. The differences are that in RAW mode, the interrupt, quit, and suspend characters are passed through uninterpreted instead of generating a signal. RAW mode also causes 8 bit input and output. The behavior of the BREAK key may be different on different systems.

`resetty()`
`savetty()`
These functions save and restore the state of the tty modes. `savetty` saves the current state in a buffer, `resetty` restores the state to what it was at the last call to `savetty`.

Window Manipulation

`newwin(num_lines, num_cols, beg_row, beg_col)`
Create a new window with the given number of lines and columns. The upper left corner of the window is at line `beg_row` column `beg_col`. If either `num_lines` or `num_cols` is zero, they will be defaulted to `LINES-beg_row` and `COLS-beg_col`. A new full-screen window is created by calling `newwin(0,0,0,0)`.

`newpad(num_lines, num_cols)`
Creates a new *pad* data structure. A pad is like a window, except that it is not restricted by the screen size, and is not associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing of input) do not occur. It is not legal to call `refresh` with a pad as an argument, the routines `prefresh` or `pnoutrefresh` should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

`subwin(orig, num_lines, num_cols, begy, begx)`
Create a new window with the given number of lines and columns. The window is at position (*begy, begx*) on the screen. (It is relative to the screen, not `orig`.) The window is made in the middle of the window `orig`, so that changes made to one window will affect both windows. When using this function, often it will be necessary to call `touchwin` before calling `wrefresh`

`delwin(win)`
Deletes the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

`mvwin(win, br, bc)`
Move the window so that the upper left corner will be at position `(br, bc)`. If the move would cause the window to be off the screen, it is an error and the window is not moved.

`touchwin(win)`
Throw away all optimization information about which parts of the window have been touched, by pretending the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change.

`overlay(win1, win2)`
`overwrite(win1, win2)`
These functions overlay `win1` on top of `win2`; that is, all text in `win1` is copied into `win2`. The difference is that `overlay` is nondestructive (blanks are not copied) while `overwrite` is destructive.

Causing Output to the Terminal

`refresh()`
`wrefresh(win)`
These functions must be called to get any output on the terminal, as other routines merely manipulate data structures. `wrefresh` copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. `refresh` is the same, using `stdscr` as a default screen. Unless **leaveok** has been enabled, the physical cursor of the terminal is left at the location of the window's cursor.

`doupdate()`
`wnoutrefresh(win)`
These two functions allow multiple updates with more efficiency than **wrefresh.** To use them, it is important to understand how **curses**

works. In addition to all the window structures, **curses** keeps two data structures representing the terminal screen: a *physical* screen, describing what is actually on the screen, and a *virtual* screen, describing what the programmer *wants* to have on the screen. `wrefresh` works by first copying the named window to the virtual screen (`wnoutrefresh`), and then calling the routine to update the screen (`doupdate`). If the programmer wishes to output several windows at once, a series of calls to `wrefresh` will result in alternating calls to `wnoutrefresh` and `doupdate`, causing several bursts of output to the screen. By calling `wnoutrefresh` for each window, it is then possible to call `doupdate` once, resulting in only one burst of output, with probably fewer total characters transmitted.

```
prefresh(pad,pminrow,pmincol,sminrow,
        smincol,smaxrow,smaxcol)
pnoutrefresh(pad,pminrow,pmincol,sminrow,
        smincol,smaxrow,smaxcol)
```

These routines are analogous to `wrefresh` and `wnoutrefresh` except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. `pminrow` and `pmincol` specify the upper left corner, in the pad, of the rectangle to be displayed. `sminrow`, `smincol`, `smaxrow`, and `smaxcol` specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures.

Writing on Window Structures

These routines are used to "draw" text on windows. In all cases, a missing `win` is taken to be `stdscr`. `y` and `x` are the row and column, respectively. The upper left corner is always (0,0), not (1,1). The `mv` functions imply a call to `move` before the call to the other function.

Moving the Cursor

```
move(y, x)
wmove(win, y, x)
```

The cursor associated with the window is moved to the given location. This does not move the physical cursor of the terminal until

`refresh` is called. The position specified is relative to the upper left corner of the window.

Writing One Character

```
addch(ch)
waddch(win, ch)
mvaddch(y, x, ch)
mvwaddch(win, y, x, ch)
```

The character `ch` is put in the window at the current cursor position of the window. If `ch` is a tab, newline, or backspace, the cursor will be moved appropriately in the window. If `ch` is a different control character, it will be drawn in the ^X notation. The position of the window cursor is advanced. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok** is enabled, the scrolling region will be scrolled up one line.

The `ch` parameter is actually an integer, not a character. Video attributes can be combined with a character by or-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another with `inch` and `addch`.)

Writing a String

```
addstr(str)
waddstr(win,str)
mvaddstr(y,x,str)
mvwaddstr(win,y,x,str)
```

These functions write all the characters of the null terminated character string `str` on the given window. They are identical to a series of calls to `addch`.

Clearing Areas of the Screen

```
erase()
werase(win)
```

These functions copy blanks to every position in the window.

```
clear()
wclear(win)
```
These functions are like `erase` and `werase` but they also call `clearok`, arranging that the screen will be cleared on the next call to `refresh` for that window.

```
clrtobot()
wclrtobot(win)
```
All lines below the cursor in this window are erased. Also, the current line to the right of the cursor is erased.

```
clrtoeol()
wclrtoeol(win)
```
The current line to the right of the cursor is erased.

Inserting and Deleting Text

```
delch()
wdelch(win)
mvdelch(y,x)
mvwdelch(win,y,x)
```
The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position. This does not imply use of the hardware delete character feature.

```
deleteln()
wdeleteln(win)
```
The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. This does not imply use of the hardware delete line feature.

```
insch(c)
winsch(win, c)
mvinsch(y,x,c)
mvwinsch(win,y,x,c)
```
The character `c` is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character on the line. This does not imply use of the hardware insert character feature.

```
insertln()
winsertln(win)
```
A blank line is inserted above the current line. The bottom line is lost. This does not imply use of the hardware insert line feature.

Formatted Output

```
printw(fmt, args)
wprintw(win, fmt, args)
mvprintw(y, x, fmt, args)
mvwprintw(win, y, x, fmt, args)
```
These functions correspond to `printf`. The characters which would be output by `printf` are instead output using `waddch` on the given window.

Miscellaneous

```
box(win, vert, hor)
```
A box is drawn around the edge of the window. `vert` and `hor` are the characters the box is to be drawn with.

```
scroll(win)
```
The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is `stdscr` and the scrolling region is the entire window, the physical screen will be scrolled at the same time.

Input from a Window

`getyx(win,y,x)`
The cursor position of the window is placed in the two integer variables `y` and `x`. Since this is a macro, no & is necessary.

`inch()`
`winch(win)`
`mvinch(y,x)`
`mvwinch(win,y,x)`
The character at the current position in the named window is returned. If any attributes are set for that position, their values will be or-ed into the value returned. The predefined constants `A_ATTRIBUTES` and `A_CHARTEXT` can be used with the & operator to extract the character or attributes alone.

Input from the Terminal

`getch()`
`wgetch(win)`
`mvgetch(y,x)`
`mvwgetch(win,y,x)`
A character is read from the terminal associated with the window. In nodelay mode, if there is no input waiting, the value -1 is returned. In delay mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**, this will be after one character, or after the first newline.

If **keypad** mode is enabled, and a function key is pressed, the code for that function key will be returned instead of the raw characters. Possible function keys are defined with integers beginning with 0401, whose names begin with `KEY_`. These are listed in "Input" under "INTRODUCTION." If a character is received that could be the beginning of a function key (such as escape), **curses** will set a 1-second timer. If the remainder of the sequence does not come in within 1 second, the character will be passed through, otherwise the function key value will be returned. For this reason, on many terminals, there will be a one second delay after a user presses the escape key. (Use by a programmer of the escape key for a single character function is discouraged.)

```
getstr(str)
wgetstr(win,str)
mvgetstr(y,x,str)
mvwgetstr(win,y,x,str)
```
A series of calls to `getch` is made, until a newline is received. The resulting value is placed in the area pointed at by the character pointer `str`. The users' erase and kill characters are interpreted.

```
scanw(fmt, args)
wscanw(win, fmt, args)
mvscanw(y, x, fmt, args)
mvwscanw(win, y, x, fmt, args)
```
This function corresponds to `scanf`. `wgetstr` is called on the window, and the resulting line is used as input for the scan.

Video Attributes

```
attroff(at)
wattroff(win, attrs)
attron(at)
wattron(win, attrs)
attrset(at)
wattrset(win, attrs)
standout()
standend()
wstandout(win)
wstandend(win)
```
These functions set the *current attributes* of the named window. These attributes can be any combination of `A_STANDOUT`, `A_REVERSE`, `A_BOLD`, `A_DIM`, `A_BLINK`, and `A_UNDERLINE`. These constants are defined in `<curses.h>` and can be combined with the C | (or) operator.

The current attributes of a window are applied to all characters that are written into the window with `waddch`. Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of characters put on the screen.

`attrset(at)` sets the current attributes of the given window to `at`. `attroff(at)` turns off the named attributes without affecting any other attributes. `attron(at)` turns on the named attributes without affecting any others. `standout` is the same as `attron(A_STANDOUT)` `standend` is the same as `attrset(0)`, that is, it turns off all attributes.

Bells and Flashing Lights

`beep()`
`flash()`
These functions are used to signal the programmer. `beep` will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. `flash` will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible nothing will happen. Nearly all terminals have an audible signal (bell or beep) but only some can flash the screen.

Portability Functions

These functions do not directly involve terminal dependent character output but tend to be needed by programs that use **curses**. Unfortunately, their implementation varies from one version of UNIX† to another. They have been included here to enhance the portability of programs using **curses**.

`baudrate()`
`baudrate` returns the output speed of the terminal. The number returned is the integer baud rate, for example, 9600, rather than a table index such as `B9600`.

`erasechar()`
The erase character chosen by the user is returned. This is the character typed by the user to erase the character just typed.

---

* Trademark of AT&T Bell Laboratories

`killchar()`
The line kill character chosen by the user is returned. This is the character typed by the user to forget the entire line being typed.

`flushinp()`
`flushinp` throws away any typeahead that has been typed by the user and has not yet been read by the program.

Delays

These functions are highly unportable, but are often needed by programs that use **curses**, especially real time response programs. Some of these functions require a particular operating system or a modification to the operating system to work. In all cases, the routine will compile and return an error status if the requested action is not possible. It is recommended that programmers avoid use of these functions if possible.

`draino`(ms) The program is suspended until the output queue has drained enough to complete in `ms` additional milliseconds. Thus, `draino`(50) at 1200 baud would pause until there are no more than 6 characters in the output queue, because it would take 50 milliseconds to output the additional 6 characters. The purpose of this routine is to keep the program (and thus the keyboard) from getting ahead of the screen. If the operating system does not support the ioctls needed to implement `draino`, the value ERR is returned; otherwise, OK is returned.

`napms`(ms) This function suspends the program for `ms` milliseconds. It is similar to `sleep` except with higher resolution. The resolution actually provided will vary with the facilities available in the operating system, and often a change to the operating system will be necessary to produce good results. If resolution of at least .1 second is not possible, the routine will round to the next higher second, call sleep, and return ERR. Otherwise, the value OK is returned. Often the resolution provided is 1/60th second.

Lower Level Functions

These functions are provided for programs not needing the screen optimization capabilities of **curses**. Programs are discouraged from working at this level, since they must handle various glitches in certain terminals. However, a program can be smaller if it only brings in the low level routines.

Cursor Motion

`mvcur(oldrow, oldcol, newrow, newcol)`
This routine optimally moves the cursor from (oldrow, oldcol) to (newrow, newcol). The user program is expected to keep track of the current cursor position. Note that unless a full screen image is kept, **curses** will have to make pessimistic assumptions, sometimes resulting in less than optimal cursor motion. For example, moving the cursor a few spaces to the right can be done by transmitting the characters being moved over, but if **curses** does not have access to the screen image, it doesn't know what these characters are.

Terminfo Level

These routines are called by low level programs that need access to specific capabilities of **terminfo**. A program working at this level should include both `<curses.h>` and `<term.h>` in that order. After a call to `setupterm`, the capabilities will be available with macro names defined in `<term.h>`. See **terminfo**(4) for a detailed description of the capabilities.

Boolean valued capabilities will have the value 1 if the capability is present, 0 if it is not. Numeric capabilities have the value -1 if the capability is missing, and have a value at least 0 if it is present. String capabilities (both those with and without parameters) have the value `NULL` if the capability is missing, and otherwise have type `char *` and point to a character string containing the capability. The special character codes involving the \ and ^ characters (such as \r for return, or ^A for control A) are translated into the appropriate ASCII characters. Padding information (of the form $<time>) and parameter information (beginning with %) are left uninterpreted at this stage. The routine `tputs` interprets padding information, and `tparm` interprets parameter information.

If the program only needs to handle one terminal, the definition **-DSINGLE** can be passed to the C compiler, resulting in static references to capabilities instead of dynamic references. This can result in smaller code, but prevents use of more than one terminal at a time. Very few programs use more than one terminal, so almost all programs can use this flag.

setupterm(term, filenum, errret)
This routine is called to initialize a terminal. **term** is the character string representing the name of the terminal being used. **filenum** is the UNIX file descriptor of the terminal being used for output. **errret** is a pointer to an integer, in which a success or failure indication is returned. The values returned can be 1 (all is well), 0 (no such terminal), or -1 (some problem locating the **terminfo** database).

The value of **term** can be given as 0, which will cause the value of TERM in the environment to be used. The **errret** pointer can also be given as 0, meaning no error code is wanted. If **errret** is defaulted, and something goes wrong, **setupterm** will print an appropriate error message and exit, rather than returning. Thus, a simple program can call **setupterm(0, 1, 0)** and not worry about initialization errors.

If the variable TERMINFO is set in the environment to a path name, **setupterm** will check for a compiled **terminfo** description of the terminal under that path, before checking /etc/term. Otherwise, only /etc/term is checked.

setupterm will check the tty driver mode bits, using **filenum**, and change any that might prevent the correct operation of other low level routines. Currently, the mode that expands tabs into spaces is disabled, because the tab character is sometimes used for different functions by different terminals. (Some terminals use it to move right one space. Others use it to address the cursor to row or column 9.) If the system is expanding tabs, **setupterm** will remove the definition of the **tab** and **backtab** functions, making the assumption that since the user is not using hardware tabs, they may not be properly set in the terminal. Other system dependent changes, such as disabling a virtual terminal driver, may be made here.

As a side effect, **setupterm** initializes the global variable **ttytype**, which is an array of characters, to the value of the list of names for the terminal. This list comes from the beginning of the **terminfo** description.

After the call to **setupterm**, the global variable **cur_term** is set to point to the current structure of terminal capabilities. By calling **setupterm** for each terminal, and saving and restoring **cur_term**, it is possible for a program to use two or more terminals at once.

The mode that turns newlines into CRLF on output is not disabled. Programs that use **cursor_down** or **scroll_forward** should avoid these capabilities if their value is linefeed unless they disable this mode. **setupterm** calls **reset_prog_mode** after any changes it makes.

reset_prog_mode()
reset_shell_mode()
def_prog_mode()
def_shell_mode()
These routines can be used to change the tty modes between the two states: *shell* (the mode they were in before the program was started) and *program* (the mode needed by the program). **def_prog_mode** saves the current terminal mode as program mode. **setupterm** and **initscr** call def_shell_mode automatically. **reset_prog_mode** puts the terminal into program mode, and **reset_shell_mode** puts the terminal into normal mode.

A typical calling sequence is for a program to call **initscr** (or **setupterm** if a **terminfo** level program), then to set the desired program mode by calling routines such as **cbreak** and **noecho**, then to call **def_prog_mode** to save the current state. Before a shell escape or control-Z suspension, the program should call **reset_shell_mode**, to restore normal mode for the shell. Then, when the program resumes, it should call **reset_prog_mode**. Also, all programs must call **reset_shell_mode** before they exit. (The higher level routine **endwin** automatically calls **reset_shell_mode**.)

Normal mode is stored in **cur_term->Ottyb**, and program mode is in **cur_term->Nttyb**. These structures are both of type **SGTTYB**

(which varies depending on the system). Currently the possible types are `struct sgttyb` (on some other systems) and `struct termio` (on this version of the UNIX system). `def_prog_mode` should be called to save the current state in `Nttyb`.

`vidputs(newmode, putc)`
`newmode` is any combination of attributes, defined in <`curses.h`>. `putc` is a putchar-like function. The proper string to put the terminal in the given video mode is output. The previous mode is remembered by this routine. The result characters are passed through `putc`.

`vidattr(newmode)`
The proper string to put the terminal in the given video mode is output to **stdout**.

`tparm(instring, p1, p2, p3, p4, p5, p6, p7, p8, p9)`
`tparm` is used to instantiate a parameterized string. The character string returned has the given parameters applied, and is suitable for `tputs`. Up to 9 parameters can be passed, in addition to the parameterized string.

`tputs(cp, affcnt, outc)`
A string capability, possibly containing padding information, is processed. Enough padding characters to delay for the specified time replace the padding specification, and the resulting string is passed, one character at a time, to the routine `outc`, which should expect one character parameter. (This routine often just calls `putchar`.) `cp` is the capability string. `affcnt` is the number of units affected by the capability, which varies with the particular capability. (For example, the `affcnt` for `insert_line` is the number of lines below the inserted line on the screen, that is, the number of lines that will have to be moved by the terminal.) `affcnt` is used by the padding information of some terminals as a multiplication factor. If the capability does not have a factor, the value 1 should be passed.

`putp(str)`
This is a convenient function to output a capability with no `affcnt`. The string is output to `putchar` with an `affcnt` of 1. It can be used in simple applications that do not need to process the output of `tputs`.

`delay_output(ms)`
A delay is inserted into the output stream for the given number of milliseconds. The current implementation inserts sufficient pad characters for the delay. This should not be used in place of a high resolution sleep, but rather for delay effects in the output. Due to buffering in the system, it is unlikely that this call will result in the process actually sleeping. Since large numbers of pad characters can be output, it is recommended that `ms` not exceed 500.

OPERATION DETAILS

These paragraphs describe many of the details of how the **curses** and terminfo package operates.

Insert and Delete Line and Character

The algorithm used by **curses** takes into account insert and delete line and character functions, if available, in the terminal. Calling the routine

```
idlok(stdscr, TRUE);
```

will enable insert/delete line. By default, **curses** will not use insert/delete line. This was not done for performance reasons, since there is no speed penalty involved. Rather, experience has shown that some programs do not need this facility, and that if **curses** uses insert/delete line, the result on the screen can be visually annoying. Since many simple programs using **curses** do not need this, the default is to avoid insert/delete line. Insert/delete character is always considered.

Additional Terminals

Curses will work even if absolute cursor addressing is not possible, a long as the cursor can be moved from any location to any othe location. It considers local motions, parameterized motions, hom and carriage return.

Curses is aimed at full duplex, alphanumeric, video terminals. N attempt is made to handle half-duplex, synchronous, hard copy, o bitmapped terminals. Bitmapped terminals can be handled b programming the bitmapped terminal to emulate an ordinar alphanumeric terminal. This does not take advantage of the bitma capabilities, but it is the fundamental nature of **curses** to deal wit alphanumeric terminals.

The **curses** handles terminals with the "magic cookie glitch" in thei video attributes. The term "magic cookie" means that a change i video attributes is implemented by storing a "magic cookie" in location on the screen. This "cookie" takes up a space, preventing a exact implementation of what the programmer wanted. Curses take the extra space into account, and moves part of the line to the right as necessary. In some cases, this will unavoidably result in losin text from the right hand edge of the screen. Advantage is taken o existing spaces.

Multiple Terminals

Some applications need to display text on more than one terminal controlled by the same process. Even if the terminals are of differen types, **curses** can handle this.

All information about the current terminal is kept in a globa variable

```
struct screen *SP;
```

Although the screen structure is hidden from the user, the C compile will accept declarations of variables which are pointers. The use

rogram should declare one screen pointer variable for each terminal t wishes to handle. The routine

```
struct screen *
newterm(type, fd)
```

ill set up a new terminal of the given terminal type which does utput on file descriptor fd. A call to **initscr** is essentially **ewterm(getenv(''TERM''),stdout)**. A program wishing to se more than one terminal should use **newterm** for each terminal nd save the value returned as a reference to that terminal.

o switch to a different terminal, call

```
set_term(term)
```

he old value of SP will be returned. The programmer should not ssign directly to SP because certain other global variables must also e changed.

ll **curses** routines always affect the current terminal. To handle everal terminals, switch to each one in turn with **set_term**, and hen access it. Each terminal must be set up with **newterm**, and losed down with **endwin**.

ideo Attributes

ideo attributes can be displayed in any combination on terminals vith this capability. They are treated as an extension of the tandout capability, which is still present.

ach character position on the screen has 16 bits of information ssociated with it. Seven of these bits are the character to be lisplayed, leaving separate bits for nine video attributes. These bits re used for standout, underline, reverse video, blink, dim, bold, lank, protect, and alternate character set. Standout is taken to be vhatever highlighting works best on the terminal, and should be used y any program that does not need specific or combined attributes. nderlining, reverse video, blink, dim, and bold are the usual video

attributes. Blank means that the character is displayed as a space, for security reasons. Protected and alternate character set depend on the particular terminal. The use of these last three bits is subject to change and not recommended. Note also that not all terminals implement all attributes - in particular, no current terminal implements both dim and bold.

The routines to use these attributes include

```
attrset(attrs)                  wattrset(win, attrs)
attron(attrs)                   wattron(win, attrs)
attroff(attrs)                  wattroff(win, attrs)
standout()                      wstandout(win)
standend()                      wstandend(win)
```

Attributes, if given, can be any combination of `A_STANDOUT`, `A_UNDERLINE`, `A_REVERSE`, `A_BLINK`, `A_DIM`, `A_BOLD`, `A_INVIS`, `A_PROTECT`, and `A_ALTCHARSET`. These constants, defined in **curses**.h, can be combined with the C | (or) operator to get multiple attributes. `attrset` sets the current attributes to the given `attrs`; `attron` turns on the given `attrs` in addition to any attributes that are already on; `attroff` turns off the given attributes, without affecting any others. `standout` and `standend` are equivalent to `attron(A_STANDOUT)` and `attrset(A_NORMAL)`.

If the particular terminal does not have the particular attribute or combination requested, **curses** will attempt to use some other attribute in its place. If the terminal has no highlighting at all, all attributes will be ignored.

Special Keys

Many terminals have special keys, such as arrow keys, keys to erase the screen, insert or delete text, and keys intended for user functions. The particular sequences these terminals send differs from terminal to terminal. **Curses** allows the programmer to handle these keys.

A program using special keys should turn on the keypad by calling

```
keypad(stdscr, TRUE)
```

at initialization. This will cause special characters to be passed through to the program by the function `getch`. These keys have constants which are listed in " Input" under " INTRODUCTION." They have values starting at 0401, so they should not be stored in a **char** variable, as significant bits will be lost.

A program using special keys should avoid using the **escape** key, since most sequences start with escape, creating an ambiguity. **Curses** will set a one second alarm to deal with this ambiguity, which will cause delayed response to the escape key. It is a good idea to avoid escape in any case, since there is eventually pressure for nearly *any* screen oriented program to accept arrow key input.

Scrolling Region

There is a programmer accessible scrolling region. Normally, the scrolling region is set to the entire window, but the calls

```
setscrreg(top, bot)
wsetscrreg(win, top, bot)
```

set the scrolling region for `stdscr` or the given window to any combination of top and bottom margins. When scrolling past the bottom margin of the scrolling region, the lines in the region will move up one line, destroying the top line of the region. If scrolling has been enabled with `scrollok`, scrolling will take place only within that window. Note that the scrolling region is a software feature, and only causes a window data structure to scroll. This may or may not translate to use of the hardware scrolling region feature of a terminal, or insert/delete line.

Mini-Curses

Curses copies from the current window to an internal screen image for every call to `refresh`. If the programmer is only interested in screen output optimization, and does not want the windowing or input functions, an interface to the lower level routines is available. This will make the program somewhat smaller and faster. The interface is a subset of full **curses**, so that conversion between the levels is not necessary to switch from mini-curses to full **curses**.

The following functions of **curses** and terminfo are available to the user of minicurses:

addch(ch)	addstr(str)	attroff(at)	attron(at)
attrset(at)	clear()	erase()	initscr
move(y, x)	mvaddch(y,x,ch)	mvaddstr(y,x,str)	newterm
refresh()	standend()	standout()	

The following functions of **curses** and terminfo are *not* available to the user of minicurses:

box	clrtobot	clrtoeol	delch
deleteln	delwin	getch	getstr
inch	insch	insertln	longname
makenew	mvdelch	mvgetch	mvgetstr
mvinch	mvinsch	mvprintw	mvscanw
mvwaddch	mvwaddstr	mvwdelch	mvwgetch
mvwgetstr	mvwin	mvwinch	mvwinsch
mvwprintw	mvwscanw	newwin	overlay
overwrite	printw	putp	scanw
scroll	setscrreg	subwin	touchwin
vidattr	waddch	waddstr	wclear
wclrtobot	wclrtoeol	wdelch	wdeleteln
werase	wgetch	wgetstr	winsch
winsertln	wmove	wprintw	wrefresh
wscanw	wsetscrreg		

The subset mainly requires the programmer to avoid use of more than the one window `stdscr`. Thus, all functions beginning with "w" are generally undefined. Certain high level functions that are convenient but not essential are also not available, including `printw` and `scanw`. Also, the input routine `getch` cannot be used with

nini-curses. Features implemented at a low level, such as use of ıardware insert/delete line and video attributes, are available in both ersions. Also, mode setting routines such as `crmode` and `noecho` ıre allowed.

To access mini-curses, add `-DMINICURSES` to the CFLAGS in the nakefile. If routines are requested that are not in the subset. the oader will print error messages such as

```
Undefined:
m_getch
m_waddch
```

to tell you that the routines `getch` and `waddch` were used but are not available in the subset. Since the preprocessor is involved in the implementation of mini-curses, the entire program must be recompiled when changing from one version to the other.

TTY Mode Functions

In addition to the save/restore routines `savetty()` and `resetty()`, standard routines are available for going into and out of normal tty mode. These routines are `resetterm()`, which puts the terminal back in the mode it was in when **curses** was started; `fixterm()`, which undoes the effects of `resetterm`, that is, restores the "current **curses** mode"; and `saveterm()`, which saves the current state to be used by `fixterm()`. `endwin` automatically calls `resetterm`, and the routine to handle control-Z (on other systems that have process control) also uses `resetterm` and `fixterm`. Programmers should use these routines before and after shell escapes, and also if they write their own routine to handle control-Z. These routines are also available at the *terminfo* level.

Typeahead Check

If the user types something during an update, the update will stop, pending a future update. This is useful when the user hits several keys, each of which causes a good deal of output. For example, in a screen editor, if the user presses the "forward screen" key, which draws the next screen full of text, several times rapidly, rather than drawing several screens of text, the updates will be cut short, and

only the last screen full will actually be displayed. This feature is automatic and cannot be disabled. The feature only works on versions of the UNIX system with the necessary support in the operating system.

getstr

No matter what the setting of *echo* is, strings typed in here are echoed at the current cursor location. The users erase and kill characters are understood and handled. This makes it unnecessary for an interactive program to deal with erase, kill, and echoing when the user is typing a line of text.

longname

The `longname` function does not need any arguments. It returns a pointer to a static area containing the actual long name of the terminal.

Nodelay Mode

The call

```
nodelay(stdscr, TRUE)
```

will put the terminal in "nodelay mode". While in this mode, any call to `getch` will return -1 if there is nothing waiting to be read immediately. This is useful for writing programs requiring "real time" behavior where the users watch action on the screen and press a key when they want something to happen. For example, the cursor can be moving across the screen, in real time. When it reaches a certain point, the user can press an arrow key to change direction at that point.

Portability

Several useful routines are provided to improve portability. The implementation of these routines is different from system to system, and the differences can be isolated from the user program by including them in **curses**.

Functions `erasechar()` and `killchar()` return the characters which erase one character, and kill the entire input line, respectively. The function `baudrate()` will return the current baud rate, as an integer. (For example, at 9600 baud, the integer 9600 will be returned, not the value `B9600` from `<sgtty.h>`.) The routine `flushinp()` will cause all typeahead to be thrown away.

Chapter 13

Curses Examples

Chapter 13

CURSES EXAMPLES

The following examples are provided to demonstrate uses of **curses**. They are for illustration purposes only. A good programmer would expand the programs presented here before using them.

EXAMPLE PROGRAM 'editor'

```
/*
 * editor: A screen-oriented editor.  The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr itself to simplify
 * the program.
 */

#include <curses.h>

#define CTRL(c) ('c' & 037)

main(argc, argv)
char **argv;
{
    int i, n, l;
    int c;
    FILE *fd;

    if (argc != 2) {
        fprintf(stderr, " Usage: edit file0);
        exit(1);
    }

    fd = fopen(argv[1], " r" );
    if (fd == NULL) {
        perror(argv[1]);
        exit(2);
    }

    initscr();
    cbreak();
    nonl();
```

```
    noecho();
    idlok(stdscr, TRUE);
    keypad(stdscr, TRUE);

    /* Read in the file */
    while ((c = getc(fd)) != EOF)
        addch(c);
    fclose(fd);

    move(0,0);
    refresh();
    edit();

    /* Write out the file */
    fd = fopen(argv[1], " w" );
    for (l=0; l<23; l++) {
        n = len(l);
        for (i=0; i<n; i++)
            putc(mvinch(l, i), fd);
        putc('0, fd);
    }
    fclose(fd);

    endwin();
    exit(0);
}

len(lineno)
int lineno;
{
    int linelen = COLS-1;

    while (linelen >=0 && mvinch(lineno, linelen) == ' ')
        linelen--;
    return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
    int c;
```

```
for (;;) {
    move(row, col);
    refresh();
    c = getch();
    switch (c) {    /* Editor commands */

    /* hjkl and arrow keys: move cursor */
    /* in direction indiated */
    case 'h':
    case KEY_LEFT:
        if (col > 0)
            col--;
        break;

    case 'j':
    case KEY_DOWN:
        if (row < LINES-1)
            row++;
        break;

    case 'k':
    case KEY_UP:
        if (row > 0)
            row--;
        break;

    case 'l':
    case KEY_RIGHT:
        if (col < COLS-1)
            col++;
        break;

    /* i: enter input mode */
    case KEY_IC:
    case 'i':
        input();
        break;

    /* x: delete current character */
    case KEY_DC:
    case 'x':
        delch();
        break;
```

```
        /* o: open up a new line and enter input mode */
        case KEY_IL:
        case 'o':
            move(++row, col=0);
            insertln();
            input();
            break;

        /* d: delete current line */
        case KEY_DL:
        case 'd':
            deleteln();
            break;

        /* ^L: redraw screen */
        case KEY_CLEAR:
        case CTRL(L):
            clearok(curscr);
            refresh();
            break;

        /* w: write and quit */
        case 'w':
            return;

        /* q: quit without writing */
        case 'q':
            endwin();
            exit(1);
        default:
            flash();
            break;
        }
    }
}

/*
 * Insert mode: accept characters and insert them.
 * End with ^D or EIC
 */
input()
{
    int c;
```

```
standout();
mvaddstr(LINES-1, COLS-20, " INPUT MODE" );
standend();
move(row, col);
refresh();
for (;;) {
     c = getch();
     if (c == CTRL(D) || c == KEY_EIC)
          break;
     insch(c);
     move(row, ++col);
     refresh();
}
move(LINES-1, COLS-20);
clrtoeol();
move(row, col);
refresh();
```

EXAMPLE PROGRAM 'highlight'

```
/*
 * highlight: a program to turn U, B, and
 * N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */
#include <curses.h>

main(argc, argv)
char **argv;
{
    FILE *fd;
    int c, c2;

    if (argc != 2) {
        fprintf(stderr, " Usage: highlight file0);
        exit(1);
    }

    fd = fopen(argv[1], " r" );
    if (fd == NULL) {
        perror(argv[1]);
        exit(2);
    }

    initscr();
    scrollok(stdscr, TRUE);

    for (;;) {
        c = getc(fd);
        if (c == EOF)
            break;
        if (c == '\') {
            c2 = getc(fd);
            switch (c2) {
            case 'B':
                attrset(A_BOLD);
                continue;
            case 'U':
                attrset(A_UNDERLINE);
                continue;
            case 'N':
```

```
                    attrset(0);
                    continue;
                }
                addch(c);
                addch(c2);
            }
            else
                addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

EXAMPLE PROGRAM 'scatter'

```
/*
 *   SCATTER. This program takes the first
 *   23 lines from the standard
 *   input and displays them on the
 *   VDU screen, in a random manner.
 */

#include <curses.h>

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS];/* Screen Array */

main()
{
    register int row=0,col=0;
    register char c;
    int char_count=0;
    long t;
    char buf[BUFSIZ];

    initscr();
    for(row=0;row<MAXLINES;row++)
        for(col=0;col<MAXCOLS;col++)
            s[row][col]=' ';

    row = 0;
    /* Read screen in */
    while( (c=getchar()) != EOF && row < LINES ) {
        if(c != '0') {
            /* Place char in screen array */
            s[row][col++] = c;
            if(c != ' ')
                char_count++;
        } else {
            col=0;
            row++;
        }
    }

    time(&t);/* Seed the random number generator */
    srand((int)(t&0177777L));
```

```
    while(char_count) {
        row=rand() % LINES;
        col=(rand()>>2) % COLS;
        if(s[row][col] != ' ')
        {
            move(row, col);
            addch(s[row][col]);
            s[row][col]=EOF;
            char_count--;
            refresh();
        }
    }
    endwin();
    exit(0);
}
```

EXAMPLE PROGRAM 'show'

```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fd;
    char linebuf[BUFSIZ];
    int line;
    void done(), perror(), exit();

    if(argc != 2)
    {
        fprintf(stderr," usage: %s file0, argv[0]);
        exit(1);
    }
    if((fd=fopen(argv[1]," r" )) == NULL)
    {
        perror(argv[1]);
        exit(2);
    }
    signal(SIGINT, done);

    initscr();
    noecho();
    cbreak();
    nonl();
    idlok(stdscr, TRUE);

    while(1)
    {
        move(0,0);
        for(line=0; line<LINES; line++)
        {
            if(fgets(linebuf, sizeof linebuf, fd) == NULL)
            {
                clrtobot();
                done();
            }
            move(line, 0);
            printw(" %s" , linebuf);
```

```
            }
            refresh();
            if(getch() == 'q')
                done();
        }
}

void
done()
{
    move(LINES-1, 0);
    clrtoeol();
    refresh();
    endwin();
    exit(0);
}
```

EXAMPLE PROGRAM 'termhl'

```
/*
 * A terminfo level version of highlight.
 */
#include <curses.h>
#include <term.h>

int ulmode = 0;          /* Currently underlining */

main(argc, argv)
char **argv;
{
    FILE *fd;
    int c, c2;
    int outch();

    if (argc > 2) {
        fprintf(stderr, " Usage: termhl [file]0);
        exit(1);
    }

    if (argc == 2) {
        fd = fopen(argv[1], " r" );
        if (fd == NULL) {
            perror(argv[1]);
            exit(2);
        }
    } else {
        fd = stdin;
    }

    setupterm(0, 1, 0);

    for (;;) {
        c = getc(fd);
        if (c == EOF)
            break;
        if (c == '\') {
            c2 = getc(fd);
            switch (c2) {
            case 'B':
                tputs(enter_bold_mode, 1, outch);
                continue;
```

```
                case 'U':
                    tputs(enter_underline_mode, 1, outch);
                    ulmode = 1;
                    continue;
                case 'N':
                    tputs(exit_attribute_mode, 1, outch);
                    ulmode = 0;
                    continue;
                }
                putch(c);
                putch(c2);
            }
            else
                putch(c);
        }
        fclose(fd);
        fflush(stdout);
        resetterm();
        exit(0);
}

/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
int c;
{
        outch(c);
        if (ulmode && underline_char) {
            outch('
)
;
            tputs(underline_char, 1, outch);
        }
}

/*
 * Outchar is a function version of putchar that can be passed to
 * tputs as a routine to call.
 */
outch(c)
int c;
{
        putchar(c);
}
```

EXAMPLE PROGRAM 'two'

```
#include <curses.h>
#include <signal.h>

struct screen *me, *you;
struct screen *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
char **argv;
{
    int done();
    int c;

    if (argc != 4) {
        fprintf(stderr, " Usage: two othertty otherttytype inputfile0);
        exit(1);
    }

    fd = fopen(argv[3], " r" );
    fdyou = fopen(argv[1], " w+" );
    signal(SIGINT, done);  /* die gracefully */

    me = newterm(getenv(" TERM" ), stdout);/* initialize my tty */
    you = newterm(argv[2], fdyou);/* Initialize his terminal */

    set_term(me);           /* Set modes for my terminal */
    noecho();           /* turn off tty echo */
    cbreak();           /* enter cbreak mode */
    nonl();                 /* Allow linefeed */
    nodelay(stdscr,TRUE); /* No hang on input */

    set_term(you);          /* Set modes for other terminal */
    noecho();
    cbreak();
    nonl();
    nodelay(stdscr,TRUE);

    /* Dump first screen full on my terminal */
    dump_page(me);
```

```
    /* Dump second screen full on his terminal */
    dump_page(you);

    for (;;) {              /* for each screen full */
        set_term(me);
        c = getch();
        if (c == 'q')       /* wait for user to read it */
            done();
        if (c == ' ')
            dump_page(me);

        set_term(you);
        c = getch();
        if (c == 'q')       /* wait for user to read it */
            done();
        if (c == ' ')
            dump_page(you);
        sleep(1);
    }
}

dump_page(term)
struct screen *term;
{
    int line;

    set_term(term);
    move(0, 0);
    for (line=0; line<LINES-1; line++) {
        if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
            clrtobot();
            done();
        }
        mvprintw(line, 0, "%s", linebuf);
    }
    standout();
    mvprintw(LINES-1, 0, "--More--");
    standend();
    refresh();      /* sync screen */
}

/*
 * Clean up and exit.
```

```
*/
done()
{
    /* Clean up first terminal */
    set_term(you);
    move(LINES-1,0); /* to lower left corner */
    clrtoeol();          /* clear bottom line */
    refresh();     /* flush out everything */
    endwin();      /* curses cleanup */

    /* Clean up second terminal */
    set_term(me);
    move(LINES-1,0); /* to lower left corner */
    clrtoeol();          /* clear bottom line */
    refresh();     /* flush out everything */
    endwin();      /* curses cleanup */

    exit(0);
}
```

EXAMPLE PROGRAM 'window'

```
#include <curses.h>

WINDOW *cmdwin;

main()
{
    int i, c;
    char buf[120];

    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(3, COLS, 0, 0);/* top 3 lines */
    for (i=0; i<LINES; i++)
        mvprintw(i, 0, " This is line %d of stdscr" , i);

    for (;;) {
        refresh();
        c = getch();
        switch (c) {
        case 'c':  /* Enter command from keyboard */
            werase(cmdwin);
            wprintw(cmdwin, " Enter command:" );
            wmove(cmdwin, 2, 0);
            for (i=0; i<COLS; i++)
                waddch(cmdwin, '-');
            wmove(cmdwin, 1, 0);
            touchwin(cmdwin);
            wrefresh(cmdwin);
            wgetstr(cmdwin, buf);
            touchwin(stdscr);
            /*
             * The command is now in buf.
             * It should be processed here.
```

```
                */
                break;
        case 'q':
                endwin();
                exit(0);
        }
    }
}
```

Chapter 14

SHELL INTRODUCTION

The chapters behind the Shell tab are intended to provide information on how to use the **shell**. The *Using Shell Commands* chapter builds on the *UNIX System User Guide* or the "hands-on" experience some have acquired. It is intended for those users who have some basic familiarity with **shell** but desire more detailed information. The *Shell Programming* chapter provides information for programming with **shell**. Those users that intend to do **shell** programming should read the *Using Shell Commands* chapter as well as the *Shell Programming* chapter. The *Examples of Shell Procedures* chapter contains examples of **shell** programs.

Knowledge of another programming language is not required when reading this document.

It is important to note a few things about **shell**. The **shell** functions as a

- Command language—The **shell** reads command lines entered at a terminal and interprets the lines as requests to execute other programs.

- Programming language—The **shell** is a programming language just like BASIC, COBOL, Fortran, and other languages. The **shell** is a high-level programming language that is easy to learn. The programs written using the **shell** programming language are called **shell** scripts, procedures, or commands. These programs are stored in files and executed just like commands. The **shell** provides variables, conditional constructs, and iterative constructs.

- Working environment—The **shell** also provides an environment that can be tailored to an individual's or group's needs by manipulating environment variables.

---

* Trademark of AT&T Bell Laboratories

Throughout this section, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *Superuser Reference Manual* book. Each reference of the form **name**(1) and **name**(6) refers to entries in the *User Reference Manual* book. All other references to entries of the form **name**(N), where possibly followed by a letter, refer to entry **name** in section N of the *Programmer Reference Manual* book.

All command names in this document are in **bold** font.

Normally when the system is ready for a command from a terminal, a prompt is displayed on the terminal (**?** by default). With certain commands, the system expects more than one line of terminal input. When this is the case, a secondary prompt is displayed (> by default). To avoid confusion with what the system displays and what the user types, this document does not show prompts displayed by the system unless noted otherwise.

---

* Trademark of AT&T Technologies.

Chapter 15

Using Shell Commands

Chapter 15

USING SHELL COMMANDS

INTRODUCTION

This chapter provides information to enhance uses of the **shell**. Most information should be useful to both the programmer and nonprogrammer alike. Some information may be of more use to the more advanced user. It is assumed that the user has been introduced to the UNIX system and understands such basics as how to log in, set the terminal baud rate, etc.

EXECUTING SIMPLE SHELL COMMANDS

A simple **shell** command consists of the command name possibly followed by some arguments such as

```
cmd arg1 arg2 arg3 ...
```

where **cmd** is the command name consisting of a sequence of letters, digits, or underscores beginning with a letter or underscore. For example, the **shell** command

```
ls
```

prints a list of files in the current directory.

INPUT/OUTPUT REDIRECTION

Most commands produce output to a terminal. Output can be redirected to a file in two different ways. First, standard output may be redirected to a file by the notation " >", thus

```
ls -l > tempfile
```

causes the **shell** to redirect the output of the command **ls** to be put in *tempfile*. If there is no file *tempfile*, one is created by the **shell**. Any previous contents of *tempfile* are destroyed.

Standard output may be appended to the end of a file by the notation " >>" thus

```
ls -l >> tempfile
```

causes the **shell** to append the output of the command **ls** to the end of the contents of *tempfile*. If *tempfile* does not already exist, it is created.

Although input is normally from a terminal, it can also be redirected by the " <" notation. Thus

```
wc < tempfile
```

would send the contents of *tempfile* to the **wc** command which would give a character, word, and line count of *tempfile*. Another modification of input is possible with the " <<" notation. The form

```
cmd <<word
```

would send standard input to the specified command until a line the same as *word* is input. As an example

```
sort <<finished
```

would send all the standard input to **sort** until **finished** is input. Then the input would be sorted and output to the terminal. If the notation " <<-" is used, then all leading tabs would be stripped. As an example, the following is entered at the terminal (note that the primary system prompt $ and the secondary system prompt > provided by the system are shown in this example)

```
$sort <<end
>no one does anything about it
>everyone talks about the weather but
>end
```

and the following would be returned

```
everyone talks about the weather but
no one does anything about it
```

PIPELINES AND FILTERS

The standard output of one command may be connected to the standard input of another by using the pipe (|) operator between commands as in

```
ls -l | wc
```

A sequence of one or more commands connected in this way constitutes a pipeline, and the overall effect is the same as

```
ls -l > file; wc < file
```

except no file is used. Instead the two processes are connected together by a pipe [see **pipe**(2)] and are run in parallel. Each command is run as a separate process.

Pipes allow one to execute several commands sequentially from left to right with the standard output from each command becoming the standard input of the next command. This prevents creating temporary files and is faster than not using pipes. Pipes are unidirectional. Synchronization is achieved by halting **wc** when there is nothing to read and halting **ls** when the pipe is full.

A filter is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, **grep**(1), selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines that contain the string " old" . Another filter is the **sort**(1) command that gives alphabetical listings.

PERMISSION MODES

All UNIX system files have three independent attributes (often called "permissions"), read, write, and execute (rwx). These three permissions are assigned to three different levels of users. The first level is the owner level. Normally, the creator of the file is the owner. This ownership can be changed with the **chown**(1) command. The second level is the group level. The third level is the others level. The permission for each level must be set to allow reading, writing, or executing a file.

The **ls** command will display among other things the permissions for a file when used as follows

```
ls -l filename
```

The general format of the permissions is

```
-rwxrwxrwx
```

where the first character will be a dash if it is an ordinary file. The second, third and fourth characters (the first **rwx**) indicate the permission modes for the owner. The fifth, sixth, and seventh characters (the second **rwx**) indicate the permission modes of the group. And the eighth, ninth, and tenth characters (the last **rwx**) indicate the permission modes of others. A dash in any permission mode position indicates that the mode is not allowed.

For example, the input

```
ls -l wg
```

displays the permissions of *wg* as follows

```
-rwxr-x---  1 abc    UNIX          66 May  4 09:25 wg
```

In this case, the owner has read (r), write (w), and execute (x) permission, the group has read and execute permission, and all others are denied (-) permission to *wg*.

The **chmod**(1) command is used by the owner to change the permission modes of a file. To change the permissions of *wg* so that everyone could execute the procedure, enter the following command

```
chmod 751 wg
```

which would result in a permission mode of **rwxr-x--x**. The **7** assigns the owner read, write, and execute permission [4 (read) + 2 (write) + 1 (execute) = 7]. The **5** assigns the group read and execute permission [4 (read) + 1 (execute) = 5]. The **1** assigns others execute permission.

The **chmod** command could also be entered as

```
chmod +x wg
```

which would add execute permission for owner, group, and all others.

FILE NAME GENERATION

The **shell** provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates as arguments to **ls**(1) all file names in the current directory that end in **.c**. The character "*" is a pattern that will match any string including the null string. In general, patterns are specified as follows

*	Matches any string of characters including the null string.
?	Matches any single character.
[...]	Matches any character enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
ls -l [a-z]*
```

matches all names in the current directory beginning with letters *a* through *z*. The input

```
ls -l /usr/fred/test/?
```

matches all names in the directory */usr/fred/test* that consist of a single character. This mechanism is useful both to save typing and to select names according to some pattern.

There is one exception to the general rules given for patterns. The character "." at the start of a file name must be explicitly matched. The input

```
echo *
```

prints all file names in the current directory not beginning with ".". The input

```
echo .*
```

prints all those file names that begin with ".". This avoids inadvertently matching the names "." and ".." that mean "the current directory" and "the parent directory," respectively. [Notice that **ls**(1) suppresses information for the files "." and "..".]

QUOTING

Characters that have a special meaning to the **shell**, such as

```
< > * ? ¦ & $ ; \ " ' ' [ ]
```

are called metacharacters.

The **shell** can be inhibited from interpreting and acting upon the special meaning assigned metacharacters by preceding them with a backslash (\). Any character preceded by a \ loses its special meaning. For example

```
echo *
```

prints all the file names in the current directory. To echo an asterisk, enter

```
echo \*
```

The backslash turns off any special meaning of a metacharacter.

To allow long strings to be continued over more than one line, the sequence **newline** (or RETURN) is ignored. The \ is convenient for quoting single characters. When more than one character needs quoting, the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. All characters enclosed between a pair of single quote marks are quoted except for a single quote. For example,

```
echo xx'****'xx
```

will print

```
xx****xx
```

The quoted string may not contain a single quote but may contain new lines that are preserved. This quoting mechanism is the

simplest and is recommended for casual use.

EXECUTING COMMANDS IN THE BACKGROUND

To execute a command, the **shell** normally creates a new process and waits for it to finish. A command may be run without waiting for it to finish. Executing commands in the background enables the terminal to be used for other tasks. Adding an ampersand (&) at the end of a command line before the RETURN starts the execution of a command and immediately returns to the **shell** command level. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing "&" is an operator that instructs the **shell** not to wait for the command to finish. To help keep track of such a process, the **shell** reports its process number following its creation. This means the system will respond with a process number followed by the primary **shell** prompt.

Determining Completion of Background Commands

When a command is executed in the background, a prompt is not received when the command completes execution. The only way to see that the command is either in process or complete is to request process status. The status of all active processes assigned to a user can be represented as follows

```
ps -u ulist
```

where "ulist" is the login name. If the process number and associated command name are output by the **ps** command, then the command is running in the background. If the process number and associated command name are not output by the **ps** command, then the command has finished executing.

Terminating Background Commands

Once a command starts in the background, it will run until it is finished or is stopped. The BREAK, RUBOUT, DELETE, or other keys will not stop a command running in the background. Instead, the process must be " killed" with the **kill**(1) command as follows

```
kill PID
```

where " PID" is the process identification number. The **shell** variable $! contains the " PID" of the last process run in the background and can be obtained as follows

```
echo $!
```

All nonessential background processes can be stopped by executing the following command

```
kill 0
```

Some processes can ignore the software termination signal. To stop these processes, enter the following

```
kill -9 PID
```

A process running in the background is automatically killed when the user logs out. The **nohup**(1) command can be used to continue the process after logging off or hanging up. For example,

```
nohup nroff text &
```

would continue the formatting of the file *text* using the **nroff**(1) formatter even if one logged off or the telephone line to the computer went down. The system responds with the lines

```
28096
$ Sending output to nohup.out
```

The **28096** is the process id number. A file *nohup.out* is created by the **nohup** command, and all output of the process is directed to this file. To redirect the output to a particular file, use the redirect command as follows

```
nohup nroff text & > formatted
```

to direct the output to the file *formatted.*

SHELL VARIABLES

A variable is a name representing a string value. (Loosely defined, a string is a combination of one or more alphanumeric characters or symbols.) Variables that are normally set on a command line are called parameters. There are two types of parameters in the **shell**—positional and keyword.

Positional Parameters

When a **shell** procedure is invoked, the **shell** implicitly creates *positional parameters.* The **shell** assigns the positional parameters as follows

```
${0} ${1} ${2} ${3} ... ${9}
```

Since the general form of a simple command is

```
cmd arg1 arg2 arg3 ...
```

then the values of the positional parameters are

```
cmd  arg1 arg2 arg3 ... arg9
${0} ${1} ${2} ${3} ... ${9}
```

For instance, if the following command is entered

```
cmd temp1 temp2 temp3
```

then the positional parameter ${1} would have the value **temp1**. Notice that the command procedure name is always assigned to ${0}.

The positional parameters are used often in **shell** programs. If a **shell** program, **wg**, contained

```
who ¦ grep $1
```

then the call to run the program

```
sh wg fred
```

is equivalent to

```
who ¦ grep fred
```

The variable $* is a special **shell** parameter used to substitute for all positional parameters except $0. Certain other similar variables are used by the **shell**. The following are set by the **shell**:

$? — The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise, a nonzero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.

$# — The number of positional parameters in decimal.

$$ — The process number of this **shell** in decimal. Since process numbers are different from all other existing processes, this string is frequently used to generate temporary file names. For example,

```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```

$! The process number of the last process run in the background (in decimal).

$– The current **shell** flags, such as **–x** and **–v**.

Keyword Parameters

The **shell** uses certain variables known as keyword parameters for specific purposes. The following variables are discussed in this portion of the document:

HOME
PATH
CDPATH
MAIL
MAILCHECK
MAILPATH
PS1
PS2
IFS
SHACCT
SHELL.

HOME

The variable *HOME* is used by the **shell** as the default value for the **cd**(1) command. Entering

```
cd
```

is equivalent to entering

```
cd $HOME
```

where the value of *HOME* is substituted by the **shell**. If $*HOME=/d3/abc/def*, then each of the above two entries would be equivalent to

```
cd /d3/abc/def
```

Normally, *HOME* is initialized by **login**(1) to the login directory. The value of *HOME* can be changed to */d3/abc/ghi* by entering the following

```
HOME=/d3/abc/ghi
```

No spaces are permitted. The change of the variable will have no effect unless the value is **exported** [see **export** in Chapter 3 under "Special Commands" and in **sh**(1)]. All variables (with their associated values) that are known to a command at the beginning of execution of that command constitute its environment. To change the environment to a new variable setting, the following must be entered

```
export variable-name
```

For instance, if **HOME** has been modified, then the command

```
export HOME
```

will cause the environment to be modified accordingly. The variable **HOME** need be exported only once. At login the next time, the original variable settings will be reestablished. A change to the *.profile* would modify the environment for each new login.

PATH

The variable *PATH* is used by the **shell** to specify the directories to be searched to find commands. Each directory entry in the *PATH* variable is separated by a colon (:). Several directories can be specified in the *PATH* variable but each directory before the command is found consumes processor time. Obviously, the directories that contain the most often used commands should be

specified first to reduce searching time. The following is the default *PATH* value

PATH=:/bin:/usr/bin

Since no value precedes the first :, then the current directory is the first directory searched. Then directory */bin* is searched followed by */usr/bin*. To change the *PATH* variable, simply enter *PATH=* followed by the directories to be searched. Each directory should be separated by a colon. As when changing all variables, no spaces are allowed before or after the =.

CDPATH

The variable *CDPATH* specifies where the **shell** is to look when it is searching for the argument of the **cd** command if that argument is not null and does not begin with ../, ./, or /. For example, if the **CDPATH** variable were

CDPATH=:/d3/abc/def:/d3/abc

then the command

cd ghi

would cause the current directory, */d3/abc/def* directory, and */d3/abc* directory to be searched for the subdirectory *ghi*. If found in the */d3/abc/def* directory, the full pathname of the subdirectory would be printed and the current working directory would be changed to */d3/abc/def/ghi*.

MAIL, MAILCHECK, MAILPATH

When the *MAILPATH* variable is set, the **shell** informs the user of modifications to any of the files specified by the *MAILPATH* variable. The *MAIL* variable, if set, is ignored. When the *MAILPATH* variable is not set, the **shell** looks at the file specified by the *MAIL* variable and informs the user if there are any modifications.

If *MAILPATH=/d3/abc/def/mailfile*, then a change to *mailfile* would cause the message

```
You have mail
$
```

to be displayed when a check is made. Note that the prompt appears on the line after the message. To display a customized message, follow the file name with a % and the message. For example

```
MAILPATH=/d3/abc/def/mailfile%" Mailfile modified"
```

would cause the following message to be displayed after *mailfile* is modified

```
Mailfile modified
$
```

Several files can be checked by adding them to *MAILPATH*. For instance

```
MAILPATH=/usr/mail/def:/d3/abc/def/mailfile%" Mailfile
modified" :/d3/abc/othermail%" Othermail modified"
```

would check for modifications to the three specified files. The standard mailfile is specified. Otherwise, the user would not be notified of the reception of standard mail except at login.

The *MAILCHECK* variable specifies how often (in seconds) the **shell** will check for mail. The default value is 600 seconds (10 minutes). If set to zero, the **shell** will check before each prompt. To set the *MAILCHECK* variable to zero, enter the following

```
MAILCHECK=0
```

The presence of mail in the standard mail file (*/usr/mail/*loginname) is announced at login regardless of the setting of *MAIL* or *MAILPATH* variables. Otherwise, to be notified of the arrival of

mail, either the *MAIL* or *MAILPATH* variable must be set.

PS1

The variable *PS1* is used by the **shell** to specify the primary **shell** prompt. This is displayed at a terminal whenever the **shell** is awaiting a command input. The default primary prompt is $. To change the prompt to <>, for example, the following is entered

```
PS1=" <>"
```

PS2

The variable *PS2* is used by the **shell** to specify the secondary **shell** prompt. This is displayed whenever the **shell** receives a newline in its input but more is expected. The default value of *PS2* is >. To change the prompt to <**more**> for example, the following is entered

```
PS2=" <more>"
```

IFS

The variable *IFS* is used by the **shell** to specify the internal field separators. Normally, the *space*, *tab*, and *newline* characters are used. After parameter and command substitution, internal field separators are used to split the results of substitution into distinct arguments where such characters are found. Explicit null arguments (" " and ' ') are retained.

SHACCT

The variable *SHACCT* is used by the **shell** to specify a file for storing **shell** (as opposed to process) accounting records. Whenever a **shell** procedure is executed and the variable *SHACCT* is set, the **shell** will write an accounting record to the file specified by *SHACCT*. This file must be writable by the user. The accounting records can be analyzed by accounting routines such as **acctcom**(1) and **acctcms**(1M).

User Defined Variables

A user variable can be defined using an assignment statement of the form *name=value*. The *name* must begin with a letter or underscore and may then consist of any sequence of letters, digits, or underscores. The *name* is the variable. Positional parameters cannot be in the *name*.

The **shell** provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. Variables may be given values by entering

```
user=fred box=m000 acct=mh000
```

to assign values to the variables *user, box,* and *acct*. A variable may be set to the null string by entering

```
null=
```

The value of a variable is substituted by preceding its name with $; for example,

```
echo $user
```

will print *fred.*

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv file $b
```

moves the *file* from the current directory to the directory */usr/fred/bin*. A more general notation is available for parameter (or variable) substitution as in

```
echo ${user}
```

This is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >$tmpa
```

directs the output of **ps**(1) to the file */tmp/psa*, whereas,

```
ps a >$tmpa
```

causes the value of the variable *tmpa* to be substituted.

SPECIAL COMMANDS

The following special commands are used in writing **shell** procedures. Many of the commands are only needed when programming. Others have nonprogramming uses.

:	read
.	readonly
break	return
continue	set
cd	shift
echo	test
eval	times
exec	trap
exit	type
export	ulimit
hash	umask
newgrp	unset
pwd	wait

The ones that are useful to the casual (nonprogramming) user are described below.

cd

The **cd** command is used to change the current working directory as follows

```
cd [arg]
```

where *arg* specifies the new directory desired. For instance,

```
cd /d3/abc/ghi
```

moves the user from anywhere in the file system to the directory */d3/abc/ghi*. The full directory pathname must be specified to be used in this way. Execute permissions must be set in the desired directory.

If only the desired directory name is specified and the *CDPATH* variable is not set, then the current directory is searched for a subdirectory by that name. For instance, if the current directory */d3/abc* contains a subdirectory *subdir*, then the command

```
cd subdir
```

changes the current working directory to */d3/abc/subdir*. If the argument begins with *../*, the current working directory is changed relative to its parent directory. If the argument begins with *./*, the current directory value precedes additional arguments. For instance, if the current working directory is */d3/abc*, the following command

```
cd ./ghi
```

changes the current directory to */d3/abc/ghi*.

If the variable *CDPATH* is set, the **shell** searches each directory specified in *CDPATH* for the directory specified by the **cd** command. If the directory is present, the directory becomes the new working directory. (See " **CDPATH**" under " Keyword Parameters" .)

exec

The command

```
exec [arg ...]
```

causes the command specified by *arg* to be executed in place of the **shell** without creating a new process. Input/output arguments may appear and, if no other arguments are given, cause the **shell** input/output to be modified.

hash

When a command is executed, it is entered into a special **hash** table. This table keeps track of what commands have been used, where they were located, and how much directory searching is involved in locating the command. Since this table is the first place that the **shell** looks, the amount of time used to search for a previously used command is reduced. Note that if a command is created and a command by the same name has been previously used, the **hash** table will contain only the location of the previously used command. The **hash** table is reinitialized upon each login session. The **hash** table can be cleared by entering

```
hash -r
```

To display the contents of the **hash** table, the following is entered

```
hash
```

The following is an example of a **hash** table:

hits	cost	command
3	5	/d3/abc/progbin/l
1	2	/bin/ed
1	7	/d3/abc/def/busy
1	2	/bin/date
2	2	/bin/who
1	2	/bin/ls

The **hits** column displays the number of times a command has been called. The **cost** column displays the number of nodes (i.e., **PATH**=node:node:node) searched to find the command. The **command** column displays the full pathname of the command.

An asterisk (*) displayed beside the **hits** information indicates that the command location may be reevaluated when the working directory is changed and the command is re-executed.

If a command *name* is entered with **hash**, the location of the command is determined and stored in the **hash** table without executing the command.

newgrp

By issuing the command **newgrp**(1), the user is assigned a new group identification. The command is of the form

```
newgrp [-] [group]
```

All access permissions are then evaluated with the new group. This allows access to files with different group ID permissions.

Entering **newgrp** with no argument changes the group identification back to the original group. When a – is entered, the environment is changed to the login environment.

pwd

The **pwd** command prints the full pathname of the current working directory. This command is especially useful when working directories are changed often.

set

The **set** command provides the capability of altering several aspects of the behavior of the **shell** by setting certain **shell** *flags*. Some of the more useful flags for the nonprogrammer and their meanings are:

- -a Mark variables that are modified or created for export.
- -f Disable file name generation.
- -v Print lines as they are read by the **shell**. The commands on each input line are executed after that input line is printed.
- -x Print commands and their arguments as they are executed. This causes a trace of only those commands that are actually executed.

To set the **x** flag for example, enter

```
set -x
```

To turn the **x** flag off for example, enter

```
set +x
```

These commands are especially useful for troubleshooting within **shell** procedures.

The **set** command entered with no arguments will display the values of variables in the environment.

type

The **type** command indicates how a specified command would be interpreted if used as a command name. The form of the command is

```
type [command-name]
```

For example, if the interpretation of the **cd** command is desired, enter

```
type cd
```

which returns

```
cd is a shell builtin
```

ulimit

The **ulimit** command has the form

```
ulimit [-f] [n]
```

When the option *-f* is used or if no option is specified, this command imposes a limit of *n* blocks on the size of files written by the **shell** and its child processes. Any size files may be read. If *n* is omitted, the current value of this limit is printed. The default value for *n* varies from one installation to another.

umask

The **umask** command has the form

```
umask [nnn]
```

The user file creation mask is set to *nnn*. This mask is used to determine the permission modes set on a file when it is created. For instance,

```
umask 033
```

causes a newly created file to be assigned the permission set of 744. (See " PERMISSION MODES" .)

unset

The **unset** command has the form

```
unset [name ...]
```

For each variable *name*, the **shell** removes the corresponding variable or function. (This is not the same as making a variable null; removing a variable makes it nonexistent.) The variables *PATH, PS1, PS2, MAILCHECK,* and *IFS* cannot be *unset.*

RESTRICTED SHELL

A restricted **shell** is also available with the UNIX system. This restricted version of **shell** is used to create an environment that controls and limits the capabilities. The actions of **rsh** are identical to that of **sh**, except that the following are disallowed:

- Changing directory
- Setting the value of *PATH* variable
- Specifying path or command names containing /
- Redirecting output (> and >>).

The system administrator often sets up a directory of commands that can be safely invoked by **rsh**. A restricted editor may also be provided.

Chapter 16

Shell Programming

Chapter 16

SHELL PROGRAMMING

INTRODUCTION

This chapter describes **shell** as a programming language and builds upon the information provided in the *Using Shell Commands* chapter. It is expected that the reader has read the *Using Shell Commands* chapter and has experience with UNIX system commands.

INVOKING THE SHELL

The **shell** is an ordinary command and may be invoked in the same way as other commands:

sh *proc* [*arg...*]	A new instance of the **shell** is explicitly invoked to read *proc*.
sh **−v** *proc* [*arg ...*]	This is equivalent to putting **set** **−v** at the beginning of *proc*. Similarly for other set flags including **x**, **e**, **u**, and **n** flags.
proc [*arg ...*]	If *proc* is marked executable, and is not a compiled, executable program, the effect is similar to that of the **sh** *proc* [*args ...*] command. An advantage of this form is that *proc* may be found by the search procedure.

INPUT/OUTPUT

Unless redirected by a command inside the program, a **shell** program uses the input and output connections of the **shell** program. A redirection on a command changes redirection for that command only.

Single Line

The following could be used to print a line from a program

```
echo The date is:
date
```

and would result in

```
The date is:
Tue May 21 16:13:38 EDT 1984
```

Printing Error Messages

Normally, error messages are associated with file descriptor 2 and are sent to standard error. Error messages can be redirected to a file with the following command

```
sample 2>ERROR
```

If an error message is produced when running the program **sample**, the error output is redirected to the file *ERROR*.

Multiline Input (Here Documents)

One way to input several lines to programs is with what is referred to as " Here Documents" . The general form is

```
cmd arg1 arg2 ... <<word
```

where everything entered at this command is accepted until *word* is entered on a line by itself. For example

```
sort <<finish
```

sends all the standard input to **sort** until **finish** is inputted. Then the input would be sorted and output to the terminal. For example

```
$ sort <<finish
> def
> abc
> finish
abc
def
```

Note that the primary system prompt ($) and the secondary system prompt (>) are shown. The final two lines are returned by the system.

The command

```
sort <<-word
```

removes all leading spaces or tabs.

SHELL VARIABLES

The **shell** has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are usually referred to as *parameters*. *Parameters* are the variables normally set only on a command line. There are also *positional parameters* and *keyword parameters*. Other variables are simply names to which the user or the **shell** itself may assign string values.

Positional Parameters: When a **shell** procedure is invoked, the **shell** implicitly creates *positional parameters*. The argument in position zero on the command line (the name of the **shell** procedure itself) is called **$0**, the first argument is called **$1**, etc. The **shift**

command may be used to access arguments in positions numbered higher than nine.

One can explicitly force values into these positional parameters by using the **set** command

```
set abc def ghi
```

which assigns "abc" to the first positional parameter (**$1**), "def" to the second (**$2**), and "ghi" to the third (**$3**). For this example, **set** also *unsets* **$4**, **$5**, etc. even if they were previously set. Positional parameter **$0** may not be assigned a value so that it always refers to the name of the **shell** procedure or to the name of the **shell** (in the login **shell**).

For instance,

```
set abc def ghi
echo $3 $2 $1
```

prints

```
ghi def abc
```

User-defined Variables: The **shell** also recognizes alphanumeric variables to which string values may be assigned. Positional parameters may not appear on the left-hand side of an assignment statement. Positional parameters can only be set as described in "Positional Parameters". A simple assignment is of the form

name=string

Thereafter, $*name* yields the value "string". A *name* is a sequence of letters, digits, and underscores that begins with a letter or an underscore. Note that no spaces surround the = in an assignment statement.

More than one assignment may appear in an assignment statement, but beware since *the* **shell** *performs the assignments from right to left.* The following command line results in the variable *a* acquiring the value " abc"

```
a=$b b=abc
```

The following are examples of simple assignments. *Double* quotes around *the right-hand side* allow blanks, tabs, semicolons, and newlines to be included in " string", while also allowing *variable substitution* (also known as *parameter substitution*) to occur. In *parameter substitution,* references to positional parameters and other variable names that are prefaced by $ are replaced by the corresponding values, if any. *Single* quotes inhibit variable substitution. Some examples follow

```
MAIL=/usr/mail/gas
var=" $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

The variable *var* has as its value the string consisting of the values of the first four positional parameters, separated by blanks. No quotes are needed around the string of asterisks being assigned to **stars** because pattern matching (expansion of *, ?, [. . .]) does *not* apply in this context. Note that the value of **$asterisks** is the literal string " $stars", *not* the string " *****", because the single quotes inhibit substitution.

In assignments, blanks are not reinterpreted after variable substitution, so that the following example results in **$first** and **$second** having the same value

```
first='a string with embedded blanks'
second=$first
```

In accessing the value of a variable, one may enclose the variable's name (or the digit designating the positional parameter) in braces {} to delimit the variable name from any following string. In particular, if the character immediately following the name is a

letter, digit, or underscore (digit only for positional parameters), the the braces are *required*

```
a='This is a string'
echo "${a}ent test"
```

returns the following message

```
This is a stringent test
```

Command Substitution: Any command line can be placed withi grave accents (‘ . . .‘) to capture the output of the command. Thi concept is known as *command substitution*. The command c commands enclosed between grave accents are first executed by th **shell** and then their output replaces the whole expression, grav accents and all. This feature is often combined with **shell** variable so that

```
today=‘date‘
```

assigns the string representing the current date to the variable *toda* (e.g., **Tue Nov 27 16:01:09 EST 1984**). The command

```
users=‘who | wc -l‘
```

saves the number of logged-in users in the variable *users*. An command that writes to the standard output can be enclosed in grav accents. Grave accents may be nested. The inside sets must b escaped with \. For example

```
logmsg=‘echo Your login directory is \‘pwd\‘“
```

Shell variables can also be given values indirectly by using the **she** builtin command **read**. The **read** command takes a line from th standard input (usually the terminal) and assigns consecutive wor on that line to any variables named

read first init last

ill take an input line of the form

A. A. Smith

nd has the same effect as if

first=A. init=A. last=Smith

ad been typed.

he **read** command assigns any excess "words" to the last variable.

redefined Special Variables: Several variables have special ieanings. The following are set *only* by the **shell**:

$# records the number of *positional* arguments passed to the **shell**, not counting the name of the **shell** procedure itself. The variable $# yields the number of the highest-numbered positional parameter that is set. Thus, **sh x a b c** sets $# to 3. One of its primary uses is in checking for the presence of the required number of arguments

```
if test $# -lt 2
then
   echo 'two or more args required'; exit
fi
```

$? is the exit status (also referred to as *return code, exit code,* or *value*) of the last command executed. Its value is a decimal string. Most UNIX system commands return **0** to indicate successful completion. The **shell** itself returns the current value of **$?** as *its* exit status.

$$ is the process number of the current process. Since process numbers are unique among all existing processes, this string of up to five digits is often used to generate unique names for temporary files. The UNIX system provides no mechanism for the automatic creation and deletion of temporary files. A file exists until it is explicitly removed. Temporary files are generally undesirable. The UNIX system pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur. The following example also illustrates the recommended practice of creating temporary files in a directory used only for that purpose

```
temp=$HOME/temp/$$
ls > $temp
commands, some of which use $temp, go here
rm $temp
```

$! is the process number of the last process run in the background. Again, this is a string of up to five digits.

$– is a string consisting of names of execution flags currently turned on in the **shell**. The $– variable has the value **xv** when tracing output.

CONDITIONAL SUBSTITUTION

Normally, the **shell** replaces occurrences of $*variable* by the string value assigned to *variable*, if any. However, there exists a special notation to allow conditional substitution depending upon whether the variable is set and/or not null. By definition, a variable is *set* if it has *ever* been assigned a value. The value of a variable can be the null string which may be assigned to a variable in any one of the following ways

```
A=
bcd=" "
Ef_g="
set " " "
```

The first three of these examples assign the null string to each of the corresponding *shell variables*. The last example sets the first and second *positional parameters* to the null string and *unsets* all other positional parameters.

The following conditional expressions depend upon whether a variable is *set and not null*. (Note that, in these expressions, *variable* refers to either a digit or a variable name.

${*variable*:-*string*} If *variable* is set and is non-null, then substitute the value $*variable* in place of this expression. Otherwise, replace the expression with *string*. Note that the value of *variable* is *not* changed by the evaluation of this expression.

${*variable*:=*string*} If *variable* is set and is non-null, then substitute the value $*variable* in place of this expression. Otherwise, set *variable* to *string*, and then substitute the value $*variable* in place of this expression. Positional parameters may not be assigned values in this fashion.

${*variable*:?*string*} If *variable* is set and is non-null, then substitute the value of *variable* for the expression. Otherwise, print a message of the form

variable: *string*

and exit from the current **shell**. (If the **shell** is the login **shell**, it is not exited.) If *string* is omitted in this form, then the message

variable: parameter null or not set

is printed instead.

${*variable*:+*string*} If *variable* is set and is non-null, then substitute *string* for this expression; otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.

These expressions may also be used without the colon (:). In this case, the **shell** does *not* check whether *variable* is null or not. It only checks whether *variable* has *ever* been set.

The two examples below illustrate the use of this facility:

1. If *PATH* has ever been set and is not null, then keep its current value. Otherwise, set it to the string *:/bin:/usr/bin.* Note that one needs an explicit assignment to set *PATH* in this form

```
PATH=${PATH:-':/bin:/usr/bin'}
```

2. If *HOME* is set and is not null, then change directory to it; otherwise, set it to */usr/gas* and change directory to it. Note that *HOME* is automatically assigned a value in this case

```
cd ${HOME:='/usr/gas'}
```

CONTROL COMMANDS

The **shell** provides several commands that are useful in creating **shell** procedures. A few definitions are needed before explaining the commands.

A *simple command* is defined as a sequence of nonblank arguments separated by blanks or tabs. The first argument usually specifies the name of the command to be executed. Any remaining arguments, with a few exceptions, are passed to the command. Input/output redirection arguments can appear in a simple command line and are passed to the **shell**, *not* to the command.

A *command* is a simple command or any of the **shell** commands described below. A *pipeline* is a sequence of one or more commands separated by |. (For historical reasons, ^ is a synonym for | in this context.) The standard output of each command but the last in a pipeline is connected [by a *pipe(2)]* to the standard input of the next command. Each command in a pipeline is run separately. The **shell** waits for the last command to finish. If no exit status argument is specified, the exit status is that of the last command executed (an end-of-file will also cause the **shell** to exit.

A *command list* is a sequence of one or more pipelines separated by ;, &, &&, or ||, and optionally terminated by ; or &. A semicolon (;) causes sequential execution of the previous pipeline (i.e., the **shell** waits for the pipeline to finish before reading the next pipeline), while & causes asynchronous execution of the preceding pipeline. Both sequential and asynchronous execution are thus allowed. An asynchronous pipeline continues execution until it terminates voluntarily or until its processes are **kill**ed.

More typical uses of & include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, typing

```
nohup cc prog.c&
```

allows one to continue working while the C compiler runs in the background. A command line ending with & is immune to interrupts and quits, but it is wise to make it immune to hang-ups as well. The **nohup** command is used for this purpose. Without **nohup**, if one hangs up while **cc** in the above example is still executing, **cc** will be killed and the output will disappear.

The && and || operators, which are of equal precedence (but lower than & and |), cause conditional execution of pipelines. In **cmd1** || **cmd2**, **cmd1** is executed and its exit status examined. Only if **cmd1** fails (i.e., has a nonzero exit status) is **cmd2** executed. This is thus a more terse notation for

```
if  cmd1
      test $? != 0
then
    cmd2
fi
```

The **&&** operator yields the complementary test: in **cmd1 && cmd2,** the second command is executed only if the first succeeds (has a zero exit status). In the sequence below, each command is executed in order until one fails

```
cmd1 && cmd2 && cmd3 && ... && cmdn
```

A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line prints *two* separate documents

```
{ nroff -cm  text1; nroff -cm  text2; } | col
```

Programming Constructs

Several control flow commands are provided in the **shell** that are especially useful in programming. These are referred to as programming constructs and are described below.

A command often used with programming constructs is the **test**(1) command. An example of the use of the **test** command is

```
test -f  file
```

This command returns zero exit status (true) if *file* exists and nonzero exit status otherwise. In general, **test** evaluates a predicate and returns the result as its exit status. Some of the more frequently used **test** arguments are given below [see **test**(1) and " Test" under " SPECIAL COMMANDS" for more information].

test s	true if the argument *s* is not the null string
test -f file	true if *file* exists
test -r file	true if *file* is readable
test -w file	true if *file* is writable
test -d file	true if *file* is a directory.

Control Flow—while

The actions of the **for** loop and the **case** branch are determined by data available to the **shell**. A **while** or **until** loop and an **if then else** branch are also provided whose actions are determined by the exit status returned by commands. A **while** loop has the general form

```
while command-list1
do
    command-list2
done
```

The value tested by the **while** command is the exit status of the last simple command following **while**. Each time around the loop *command-list1* is executed. If a zero exit status is returned, then *command-list2* is executed; otherwise, the loop stops. For example,

```
while test $1
do
    ...
    shift
done
```

The **shift** command is a **shell** command that renames the positional parameters **$2**, **$3**, ... as **$1**, **$2**, ... and loses **$1**.

Another use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop, the termination condition is reversed. For example,

```
until test -f  file
do
    sleep 300
done
commands
```

will loop until *file* exists. Each time round the loop, it waits for 5 minutes (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

A file **print** could be written to use **while** and **test** as follows

```
while test $# != 0
do
    echo " $1 being submitted"
    lp -dprtd42  -c  -o12  -w  -tuser1  $1
    shift
done
lpstat -oprtd42
```

Control Flow—if

Also available is a general conditional branch of the form,

```
if command-list
then
  command-list
else
  command-list
fi
```

that tests the value returned by the last simple command following **if**. If a zero exit status is returned, the command-list following the **then** is executed. If a zero exit status is not returned, the command-list following the **else** is executed.

The **if** command may be used with the **test** command to test for the existence of a file as in

```
if test -f  file
then
        process file
else
        do something else
fi
```

A multiple test **if** command of the form

```
if ...
then
        ...
else
        if ...
        then
                ...
        else
                if ...
                ...
                fi
        fi
fi
```

may be written using an extension of the **if** notation as,

```
if ...
then
        ...
elif ...
then
        ...
elif ...
...
fi
```

A file could be written to include the use of **if** and **test** as follows

```
if test $# = 0
then
    echo " enter a filename after $0"
else
    if [ ! -f  $1 ]
    then
        echo " $1 does not exist"
        echo " Enter a filename that exists"  ; exit
    else
    echo " $1 being submitted"
    lp -dprtd42  -c  -o12  -w  -tuser1  $*
    lpstat -oprtd42
    fi
fi
```

The [...] is shorthand for **test**. The **if** [**!** **-f** **$1**] means **if** the file $1 does not exist **then** do this.

The sequence

```
if command1
then
      command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes **command2** only if **command1** fails. In each case, the value returned is that of the last simple command executed.

Control Flow—for

A frequent use of **shell** procedures is to loop through the arguments (**$1, $2, ...**) executing commands once for each argument. An example of such a procedure is *tel* that searches the file */usr/lib/telnos* that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

```
for i
do
    grep $i /usr/lib/telnos
done
```

The command

```
tel fred
```

prints those lines in */usr/lib/telnos* that contain the string "fred".

The command

```
tel fred bert
```

prints those lines containing "fred" followed by those for "bert".

The **for** loop notation is recognized by the **shell** and has the general form

```
for name in words
do
    command-list
done
```

A ***command-list*** is a sequence of one or more simple commands separated or ended by a newline or a semicolon. A *name* is a **shell** variable that is set to ords... in turn each time the ***command-list*** following **do** is executed. If "words ..." is omitted, then the loop is executed once for each positional parameter; that is, **in $*** is assumed. Execution ends when there are no more words in the list.

An example of the use of the **for** loop is the **create** command whose text is

```
for i do >$i; done
```

The command

```
create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation *>file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before **done**.

The **for** can also be used in a program. Assume a document is formatted and stored in chapters (files) that begin with the letters "ch" (ch1, ch2, ch3, and chtoc). A program can be written to send the document to the line printer. The program contains

```
for i in ch*
do
    lp -dprtd42 -c -o12 -w -tuser1 $i
done
    lpstat -oprtd42
```

This will send each chapter as a separate job. Notice that **$i** is used instead of **$***.

Control Flow—case

A multiple way (choice) branch is provided for by the **case** notation. For example,

```
case $# in
    1) cat >>$1 ;;
    2) cat >>$2 <$1 ;;
    *) echo 'usage: append [ from ] to' ;;
esac
```

is an append command. (Note the use of semicolons to delimit the cases.) When called with one argument as in

```
append file
```

$# is the string "1", and the standard input is appended (copied) onto the end of *file* using the **cat**(1) command.

```
append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to append is other than 1 or 2, then a message is printed indicating proper usage.

The general form of the **case** command is

```
case word in
    pattern1|pattern) command-list ;;
    ...
esac
```

The **shell** attempts to match word with each pattern in the order that the patterns appear. If a match is found, the associated ***command-list*** is executed; and execution of the **case** is complete. Since * is the pattern that matches any string, it can be used for the default case.

Caution: No check is made to ensure that only one pattern matches the case argument.

The first match found defines the set of commands to be executed. In the example below, the commands following the second "*" will never be executed since the first "*" executes everything it receives.

```
case $# in
    *) ... ;;
    *) ... ;;
esac
```

A program **print** can be used to send a document to different line printers. Assume there are two line printers named " prtd42" and " prtd43" . Send a document to " prtd42" as follows

```
print 42 files
```

Send a document to " prtd43" as follows

```
print 43 files
```

The **print** program contains the following

```
case $1 in
42) shift;lp -dprtd42 -c -o12 -w -tuser1 $*;lpstat -oprtd42;;
43) shift;lp -dprtd43 -c -o12 -w -tuser1 $*;lpstat -oprtd43;;
 *) echo " line printer does not exist" ;;
esac
```

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a **cc**(1) command.

```
for i
do
    case $i in
        -[ocs]) ... ;;
        -*)     echo 'unknown flag $i' ;;
        *.c)    /lib/c0 $i ... ;;
        *)      echo 'unexpected argument $i' ;;
    esac
done
```

To allow the same commands to be associated with more than one pattern, the **case** command provides for alternative patterns separated by à ¦ . For example,

```
case $i in
    -x¦-y)...
esac
```

is equivalent to

```
case $i in
    -[xy])...
esac
```

The usual quoting conventions apply so that

```
case $i in
    \?)...
```

will match the character **?**.

Functions

Functions may be defined and used in the current **shell**. The general form of a function is

name **()** {*list*}

A space or a newline is required after the beginning brace ({). A semicolon or newline is required before the terminating brace (}). As an example, a function could be defined to see how many people are currently on the system as follows

```
busy () { who | wc -l;}
```

The **wc** **-l** command returns a count of the lines returned by the **who** command. Notice that a space is placed after the beginning brace ({) and a semicolon is placed before the terminating brace (}). This function is called by its name as follows

```
busy
```

which returns

```
12
```

if 12 people are logged into the system.

The same function could be defined using multiple lines as follows

```
busy ()
{
who | wc -l
}
```

Positional parameters can be used to pass information to a function. For instance,

```
present () { who | grep $1 ; }
```

searches the output of the **who** command for the value of the positional parameter **$1** and prints all lines containing the value. For instance, the entry

```
present abc
```

returns

```
abc      tty09      May 7 09:31
```

if abc is logged in on tty09.

The current **shell** contains the function definition. A different **shell** would not be able to execute the function until it is defined in that **shell**. To display the functions defined in the current **shell**, enter

```
set
```

The value for all variable names will be displayed including the functions defined.

SPECIAL COMMANDS

There are several special commands that are *internal* to the **shell** (some of which have already been mentioned). These commands should be used in preference to other UNIX system commands whenever possible because they are faster and more efficient. The **shell** does not fork to execute these commands, so no additional processes are spawned.

Many of these special commands were described in the *Using Shell Commands* chapter. These commands include:

cd
exec
hash
newgrp
pwd
set
type
ulimit
umask
unset.

Descriptions of the remaining special commands follow. These commands include:

:
.
break
continue
echo
eval
exit
export
read
readonly
return
shift
test
times
trap
wait.

: (Colon)

The : command is the null command. This command can be used to return a zero (true) exit status.

. (Period)

The . command has the form

```
. file
```

This command reads and executes commands from **file** and returns. The search path specified by *PATH* is used to find the directory containing *file*. If the file **command1** contained the following

```
echo Today is:
date
```

then the command

```
. command1
```

returns

```
Today is:
Thu Sep 22 14:40:04 EDT 1984
```

Any currently defined variable can be used in the **shell** procedure called.

break

This command has the form

```
break [n]
```

This command is used to exit from the enclosing **for**, **until**, or **while** loop. If *n* is specified, then exit *n* levels. An example of **break** is as follows

```
# This procedure is interactive; the 'break'
# command is used to allow
# the user to control data entry.
while true
do
        echo " Please enter data"
        read response
        case " $response" in
            " done" )       break       # no more data
                        ;;
            *)
                        process the data here
                        ;;
        esac
done
```

continue

This command has the form

```
continue [n]
```

This command causes the resumption of an enclosing **for**, **until**, or **while** loop. If *n* is specified, then it resumes at the *n*-th enclosing loop.

echo

The form of the echo command is

```
echo [arg ...]
```

The **echo** command writes its arguments separated by blanks and terminated by a newline on the standard output. For instance, the input

```
echo Message to be printed.
```

returns

Message to be printed.

The following escapes can be used with **echo**:

\b backspace
\c print line without new-line
\f new-line
\r carriage return
\t tab
\\ backslash
\n the 8-bit character whose ASCII code is the 1-, 2-, or 3-digit octal number, which must start with a zero.
\v vertical tab

For example:

```
echo " The current date is \c"
date
```

would return

```
The current date is Tue May 16 08:00:30 EDT 1984
```

eval

Sometimes, one builds command lines inside a **shell** procedure. In this case, one might want to have the **shell** rescan the command line after all the initial substitutions and expansions are done. The special command **eval** is available for this purpose. The form of this command is

```
eval [arg ...]
```

The **eval** command takes a command line as its argument and simply rescans the line performing any variable or command substitutions that are specified. Consider the following situation:

```
command=who
output='|wc -l'
eval $command $output
```

This segment of code results in the pipeline **who|wc −l** being executed.

The uses of **eval** can be nested.

exit

A **shell** program may be terminated at any place by using the **exit** command. The form of the **exit** command is

```
exit [n]
```

The **exit** command can also be used to pass a return code (*n*) to the **shell**. By convention, a **0** return code means **true** and a **1** to **255** return code means **false**. The return code can be found by **$?**. For instance, if the executable procedure **testexit** contained

```
exit 5
```

then

```
testexit
```

would execute **testexit**. The command

```
echo $?
```

would return

```
5
```

export

The form of the **export** command is

```
export [name ...]
```

The **export** command places the named variables in the environments of both the **shell** and all its future child processes. Normally, all variables are local to the **shell** program. Commands executed from within the **shell** program do not have access to the local variables. If a variable is **exported**, then the commands within the **shell** program will be able to access the variable.

To export variables, the following command is used

```
export variable1 variable2 ...
```

To obtain a list of variables **exported**, the following command is entered

```
export
```

read

A variable may also be set using the **read** command. The **read** command reads one line from the standard input of the **shell** procedure and puts that line in the variables which are its arguments. Leading spaces and tabs are stripped off. The general form of the command is

```
read variable1 variable2 ...
```

The last variable gets what is left over. For example, if **testread** contains the following

```
echo 'Please type your first and last name:\c'
read first_name last_name
echo Your name is ${first_name} ${last_name}
```

then when the program is run the first line would be printed

```
Please type your first and last name:
```

and would wait for the input. (The input would appear on the same line.) Assuming the name is **Jane Doe**, after the input, the following line would be printed

```
Your name is Jane Doe
```

readonly

Variables can be made **readonly**. After becoming readonly, a variable cannot receive a new value. The general form of the command is

```
readonly variable-name variable-name ...
```

To print the names of variables that are readonly, enter

```
readonly
```

return

The return command causes a function to exit with a specified return value. The form of the command is

```
return [n]
```

where *n* is the desired return value. When *n* is omitted, the return status of the last command executed is displayed.

shift

The **shift**[**sh**(1)] command reassigns the positional parameters. Positional parameter **$1** would receive the value of **$2**, **$2** would receive the value of **$3**, etc. Notice that **$0** (the procedure name) is unchanged and that the number of positional parameters (**$#**) is decremented.

If the executable program **shifter** contains the following

```
echo ${#} positional parameters
echo ${*}
echo Now shift
shift
echo ${#} positional parameters
echo ${*}
```

then the command

```
shifter first second third
```

would result in

```
3 positional parameters
first second third
Now shift
2 positional parameters
second third
```

test

The **test**(1) command evaluates the expression specified by its arguments and, if the expression is true, returns a zero exit status. Otherwise, a nonzero (false) exit status is returned. The **test** command also returns a nonzero exit status if it has no arguments. Often it is convenient to use the **test** command as the first command in the *command list* following an **if** or a **while**. Shell variables used in **test** expressions should be enclosed in double quotes if there is any chance of their being null or not set.

The square brackets ([]) may be used as an alias for **test**; e.g., [*expression*] has the same effect as **test** *expression*.

The following is a partial list of the primaries that can be used to construct a conditional expression:

-r *file*	*true* if the named file exists and is readable by the user.
-w *file*	*true* if the named file exists and is writable by the user.
-x *file*	*true* if the named file exists and is executable by the user.
-s *file*	*true* if the named file exists and has a size greater than zero.
-d *file*	*true* if the named file exists and is a directory.
-f *file*	*true* if the named file exists and is an ordinary file.
-p *file*	*true* if the named file exists and is a named pipe (*fifo*).
-z *s1*	*true* if the length of string " s1" is zero.
-n *s1*	*true* if the length of the string " s1" is nonzero.
-t *fildes*	*true* if the open file whose file descriptor number is *fildes* is associated with a terminal device. If *fildes* is not specified, file descriptor **1** is used by default.
s1 = *s2*	*true* if strings " s1" and " s2" are identical.

s1 **!=** *s2*	*true* if strings " s1" and " s2" are *not* identical.
s1	*true* if " s1" is *not* the null string.
n1 **−eq** *n2*	*true* if the integers *n1* and *n2* are algebraically equal. Other algebraic comparisons are indicated by **−ne**, **−gt**, **−ge**, **−lt**, and **−le**.

These primaries may be combined with the following operators:

!	unary negation operator.
−a	binary logical *and* operator.
−o	binary logical *or* operator. The **−o** has lower precedence than **−a**.
(*expr* **)**	parentheses for grouping; they must be escaped to remove their significance to the **shell**. When parentheses are absent, the evaluation proceeds from left to right.

Note that all primaries, operators, file names, etc. are separate arguments to **test**.

For example, the procedure **nametest**

```
if test -d $1
   then echo $1 is a directory
elif test -f $1
   then echo $1 is a file
else echo $1 does not exist
fi
```

then if the file bucket existed, then

bucket is a file

would be returned.

times

The **times** command prints the accumulated user and system times for processes run from the **shell**. The **times** command is entered on a line by itself. For example, the command

times

returns

0m3s 0m10s

trap

A **shell** program may handle interrupts by using the trap command. The trap command interfaces with the underlying UNIX operating system mechanism for handling interupts.

The UNIX operating system provides signals that tell a program when some unusual condition has occurred. These signals may be from the keyboard or from other programs.

By default, if a program receives a signal, the program will terminate. However, these signals may be caught, the program suspended, the interrupt routine run, and the program restarted at the point it was suspended. Or these signals may be ignored.

trap *arg signal-list*

is the form of the **trap** command, where *arg* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers [as described in **signal**(2)].

The following signals are used in the UNIX system:

01	hangup
02	interrupt
03	quit
04	illegal instruction
05	trace trap
06	IOT instruction
07	EMT instruction
08	floating point exception
09	kill
10	bus error
11	segmentation violation
12	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination signal
16	user defined signal 1
17	user defined signal 2
18	death of a child
19	power fail.

The commands in *arg* are scanned at least once when the **shell** first encounters the **trap** command. Because of this, it is usually wise to use single rather than double quotes to surround these commands. The single quotes inhibit immediate command and variable substitution. This becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the trap command is first read by the **shell**. The following procedure will print the name of the current directory on the file **errdirect** when it is interrupted, thus giving the user information as to how much of the job was done

```
trap 'echo 'pwd' >errdirect' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
        cd $i
          commands to be executed in directory $i here
done
```

while the same procedure with double (rather than single) quotes (**trap " echo 'pwd' >errdirect" 2 3 15**) will, instead, print the name of the directory from which the procedure was executed.

Signal 11 (SEGMENTATION VIOLATION) may never be trapped because the **shell** itself needs to catch it to deal with memory allocation. Zero is not a UNIX system signal. Zero is effectively interpreted by the **trap** command as a signal generated by exiting from a **shell** (either via an **exit** command or by "falling through" the end of a procedure). If *arg* is not specified, then the action taken upon receipt of any of the signals in *signal-list* is reset to the default system action. If *arg* is an explicit null string (" or ""), then the signals in *signal-list* are *ignored* by the **shell**.

The most frequent use of **trap** is to assure removal of temporary files upon termination of a procedure. The second example of "Predefined Special Variables" in subpart "D. Shell Variables" would be written more typically as follows

```
temp=$HOME/temp/$$
trap 'rm $temp; trap 0; exit' 0 1 2 3 15
ls > $temp
    commands, some of which use $temp, go here
```

In this example whenever signals 1 (HANGUP), 2 (INTERRUPT), 3 (QUIT), or 15 (SOFTWARE TERMINATION) are received by the **shell** procedure or whenever the **shell** procedure is about to exit, the commands enclosed between the single quotes will be executed. The **exit** command must be included or else the **shell** continues reading commands where it left off when the signal was received. The **trap 0** turns off the original trap on exits from the **shell** so that the **exit** command does not reactivate the execution of the trap commands.

Sometimes it is useful to take advantage of the fact that the **shell** continues reading commands after executing the trap commands. The following procedure takes each directory in the current directory, changes to it, prompts with its name, and executes commands typed at the terminal until an end-of-file (*control-d*) or an interrupt is received. An end-of-file causes the **read** command to return a nonzero exit status, thus terminating the **while** loop and restarting the cycle for the next directory. The entire procedure is terminated

if interrupted when waiting for input; but during the execution of a command, an interrupt terminates *only* that command.

```
dir='pwd'
for i in *
do
        if test -d  $dir/$i
        then
            cd $dir/$i
            while echo " $i:"
                trap exit 2
                read x
            do
                trap : 2  # ignore interrupts
                eval $x
            done
        fi
done
```

Several **traps** may be in effect at the same time. If multiple signals are received simultaneously, they are serviced in ascending order. To check what traps are currently set, type

```
trap
```

It is important to understand some things about the way the **shell** implements the **trap** command in order not to be surprised. When a signal (other than 11) is received by the **shell**, it is passed on to whatever child processes are currently executing. When those (synchronous) processes terminate, normally or abnormally, the **shell** *then* polls any traps that happen to be set and executes the appropriate **trap** commands. This process is straightforward except in the case of traps set at the command (outermost or login) level. In this case, it is possible that no child process is running, so the **shell** waits for the termination of the first process spawned *after* the signal is received before it polls the traps.

For internal commands, the **shell** normally polls traps on completion of the command. An exception to this rule is made for the **read**, **hash**, and **echo** commands.

wait

The **wait** command has the following form

```
wait [n]
```

With this command, the **shell** waits for the child process whose process number is *n* to terminate. The exit status of the wait command is that of the process waited on. If *n* is omitted or is not a child of the current **shell**, then *all* currently active processes are waited for and the return code of the **wait** command is zero. For example, the executable program **format**

```
while test " $1"  != " "
nroff $1>>junk&
shift
wait $!
done
echo ***nroff complete***
```

envokes the nroff formatter for each file specified and informs the user when it is finished. If the files *chapter1* and *chapter2* required formatting, the entry

```
format chapter1 chapter2
```

would format the two chapters and when they are finished return

```
***nroff complete***
```

COMMAND GROUPING

Commands may be grouped in two ways

```
{ command-list ; }
```

and

(*command-list*)

The first form, *command-list,* is simply executed. The second form executes *command-list* as a separate process. If a list of commands is enclosed in a pair of parentheses, the list is executed as a subshell. The subshell inherits the environment of the main **shell**. The subshell does not change the environment of the main **shell**. For example,

(cd x; rm junk)

executes *rm junk* in the directory *x* without changing the current directory of the invoking **shell**.

The commands

cd x; rm junk

have the same effect but leave the invoking **shell** in the directory *x*.

A COMMAND'S ENVIRONMENT

All the variables (with their associated values) known to a command at the beginning of execution of that command constitute its *environment.* This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a **shell** passes to its child processes are those that have been named as arguments to the **export** command. The **export** command places the named variables in the environments of both the **shell** *and* its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command

line. Such variables are placed in the environment of the procedure being invoked. For example

```
#       key_command
echo $a $b
```

is a simple procedure that **echo**es the values of two variables. If it is invoked as

```
a=key1 b=key2 key_command
```

then the output is

```
key1 key2
```

A procedure's keyword parameters are *not* included in the argument count $#.

A procedure may access the value of any variable in its environment. However, if changes are made to the value of a variable, these changes are *not* reflected in the environment. The changes are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the **export** command within that procedure. To obtain a list of variables that have been made **export**able from the current **shell**, type

```
export
```

To get a list of name-value pairs in the current environment, type

```
env
```

DEBUGGING SHELL PROCEDURES

The **shell** provides two tracing mechanisms to help when debugging **shell** procedures. The first is invoked within the procedure as

```
set -v
```

(*v* for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without changing the procedure by entering

```
sh -v proc ...
```

where *proc* is the name of the **shell** procedure. This flag may be used with the **−n** flag to prevent execution of later commands. (Note that typing "**set −n**" at a terminal will render the terminal useless until an end-of-file is typed.)

The command

```
set -x
```

will produce an execution trace with flag **−x**. Following parameter substitution, each command is printed as it is executed. (Try the above at the terminal to see the effect it has.) Both flags may be turned off by typing

```
set -
```

and the current setting of the **shell** flags is available as $−.

Chapter 17

EXAMPLES OF SHELL PROCEDURES

Chapter 17

EXAMPLES OF SHELL PROCEDURES

Some examples in this subpart are quite difficult for beginners. For ease of reference, the examples are arranged alphabetically by name, rather than by degree of difficulty.

copypairs

```
#       usage: copypairs file1 file2 ...
#       copy file1 to file2, file3 to file4, ...
while test " $2"  != " "
do
        cp $1 $2
        shift; shift
done
if test " $1"  != " "
then
    echo " $0: odd number of arguments"
fi
```

Note: This procedure illustrates the use of a **while** loop to process a list of positional parameters that are somehow related to one another. Here a **while** loop is much better than a **for** loop because you can adjust the positional parameters via **shift** to handle related arguments.

copyto

```
#       usage: copyto dir file ...
#       copy argument files to 'dir',
#       making sure that at least
#       two arguments exist and that 'dir'
#       is a directory
if test $# -lt  2
then
      echo " $0: usage: copyto directory file ..."
elif test ! -d  $1
then
      echo " $0: $1 is not a directory" ;
else
      dir=$1; shift
  for eachfile
  do
        cp $eachfile $dir
  done
fi
```

Note: This procedure uses an **if** command with two tests in order to screen out improper usage. The **for** loop at the end of the procedure loops over all of the arguments to **copyto** but the first. The original **$1** is shifted off.

distinct

```
#       usage: distinct
#       reads standard input and reports
#       list of alphanumeric strings
#       that differ only in case,
#       giving lower-case form of each
tr -cs  '[A-Z][a-z][0-9]'   '[\012*]' | sort -u |
      tr '[A-Z]'  '[a-z]'  | sort | uniq -d
```

Note: This procedure is an example of the kind of process that is created by the left-to-right construction of a long

pipeline. It may not be immediately obvious how this works. [See **tr**(1), **sort**(1), and **uniq**(1) if you are completely unfamiliar with these commands.] The **tr** translates all characters except letters and digits into newline characters and then squeezes out repeated newline characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The **sort** command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next **tr** converts everything to lowercase so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The **uniq** **-d** prints (once) only those lines that occur more than once yielding the desired list.

The process of building such a pipeline uses the fact that pipes and files can usually be interchanged. The two lines below are equivalent assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3
cmd1>temp1;cmd2<temp1>temp2;cmd3<temp2;rm temp[12]
```

Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output. As an exercise, try to mimic **distinct** with such a step-by-step process using a file of test data containing:

```
ABC:DEF/DEF
ABC1 ABC
Abc abc
```

Although pipelines can give a concise notation for complex processes, exercise some restraint lest you succumb to the "one-line syndrome" sometimes found among users of especially concise languages. This

syndrome often yields incomprehensible code.

draft

```
# usage: draft file(s)
# prints the draft (-rC3) of a document on a DASI 450
# terminal in 12-pitch using memorandum macros (MM).
nroff -rC3 -T450-12 -cm $*
```

Note: Users often write this kind of procedure for convenience in dealing with commands that require the use of many distinct flags. These flags cannot be given default values that are reasonable for all (or even most) users.

edfind

```
# usage: edfind file arg
# find the last occurrence in 'file' of a line whose
# beginning matches 'arg', then print 3 lines (the one
# before, the line itself, and the one after)
ed - $1 <<!
H
?^$2?;-,+p
!
```

Note: This procedure illustrates the practice of using editor (**ed**) inline input scripts into which the **shell** can substitute the values of variables. It is a good idea to turn on the **H** option of **ed** when embedding an **ed** script in a **shell** procedure [see **ed**(1)].

edlast

```
# usage: edlast file
# prints the last line of file, then deletes that line
ed - $1 <<-\eof # no variable substitutions in " ed" script
        H
        $p
        $d
        w
        q
eof
echo Done.
```

Note: This procedure contains an in-line input document or script; it also illustrates the effect of inhibiting substitution by escaping a character in the *eofstring* (here, **eof**) of the input redirection. If this had not been done, **$p** and **$d** would have been treated as **shell** variables.

fsplit

```
# usage: fsplit file1 file2
# read standard input and divide it into three parts:
# append any line containing at least one letter
# to file1, any line containing at least one digit
# but no letters to file2, and throw the rest away
total=0 lost=0
while read next
do
        total=" `expr $total + 1`"
        case " $next" in
        *[A-Za-z]*)
                echo " $next" >> $1 ;;
        *[0-9]*)
                echo " $next" >> $2 ;;
        *)
                lost=" `expr $lost + 1`"
        esac
done
echo " $total lines read, $lost thrown away"
```

Note: In this procedure, each iteration of the **while** loop reads a line from the input and analyzes it. The loop terminates only when **read** encounters an end-of-file.

Do not use the **shell** to read a line at a time unless you must — it can be grotesquely slow.

initvars

```
#       usage: . initvars
#       use carriage return to indicate " no change"
echo " initializations? \c"
read response
if test " $response"  = y
then
        echo " PS1=\c" ; read temp
                PS1=${temp:-$PS1}
        echo " PS2=\c" ; read temp
                PS2=${temp:-$PS2}
        echo " PATH=\c" ; read temp
                PATH=${temp:-$PATH}
        echo " TERM=\c" ; read temp
                TERM=${temp:-$TERM}
fi
```

Note: This procedure would be invoked by a user at the terminal or as part of a *file. The assignments are effective even when the procedure is finished because the* **dot** command is used to invoke it. To better understand the **dot** command, invoke **initvars** as indicated above and check the values of **PS1**, **PS2**, **PATH**, and **TERM**; then make **initvars** executable, type **initvars**, assign different values to the three variables, and check again the values of these three **shell** variables after **initvars** terminates. It is assumed that **PS1**, **PS2**, **PATH**, and **TERM** have been **export**ed, presumably by your *.profile*.

```
#       usage:  merge src1 src2 [ dest ]
#       merge two files, every other line:
#       the first argument starts off the merge,
#       excess lines of the longer
#       file are appended to
#       the end of the resultant file
exec 4<$1 5<$2
dest=${3-$1.m}# default destination file is named $1.m
while true
do
            # alternate reading from the files;
            # 'more' represents the file descriptor
            # of the longer file
        line <&4 >>$dest || { more=5; break ;}
        line <&5 >>$dest || { more=4; break ;}
done
            # delete the last line of destination
            # file, because it is blank.
ed - $dest <<\eof
        H
        $d
        w
        q
eof
while line <&$more >> $dest
do :; done # read the remainder of the longer
          # file—the body of the 'while' loop
          # does nothing; the work of the loop
          # is done in the command list following
          # 'while'
```

Note: This procedure illustrates a technique for reading sequential lines from a file or files without creating any subshells to do so. When the file descriptor is used to access a file, the effect is that of opening the file and moving a file pointer along until the end of the file is read. If the input redirections used **src1** and **src2** explicitly rather than the associated file descriptors, this procedure would never terminate because the *first* line of each file would be read over and over again.

mkfiles

```
# usage: mkfiles pref [ quantity ]
# makes 'quantity' (default = 5) files,
# named pref1, pref2, ...
quantity=${2-5}
i=1
while test " $i"  -le  " $quantity"
do
        > $1$i
        i=" `expr $i + 1`"
done
```

Note: This procedure uses input/output redirection to create zero-length files. The **expr** command is used for counting iterations of the **while** loop. Compare this procedure with procedure **null** below.

```
if test " $#"  = 0; then cat <<\!
Usage: " mmt [ options ] files"  where " options"  are:
-a      => output to terminal
-e      => preprocess input with eqn
-t      => preprocess input with tbl
-Tst    => output to STARE phototypesetter by Honeywell
-T4014  => output to 4014 manufactured by Tektronix
-Tvp    => output to printer manufactured by Versatec
-  => use instead of " files"  when mmt used inside a pipeline.
Other options as required by TROFF and the MM macros.
!
        exit 1
fi
PATH='/bin:/usr/bin'; O='-g';  o='lgcat -ph';
#  Assumes typesetter is accessed via gcat(1)
#  If typesetter is on-line, use O=''; o=''
while test -n  " $1"  -a  ! -r  " $1"
do
   case " $1"  in
    -a)          O='-a';         o='' ;;
    -Tst)        O='-g';         o='lgcat -st';;
#           Above line for STARE only
    -T4014)      O='-t';         o='ltc';;
    -Tvp)        O='-t';         o='lvpr -t';;
    -e)          e='eqn';;
    -t)          f='tbl';;
    -)           break;;
    *)             a=" $a $1" ;;
   esac
   shift
done
if test -z  " $1"
then
                 echo 'mmt: no input file'
                 exit 1
fi
if test " $O"  = '-g'
then
                 x=" -f$1"
fi
d=" $*"
```

```
if test " $d"  = '-'
then
                    shift
                    x="
                    d="
fi
if test -n  " $f"
then
                    f=" tbl $*f"
                    d="
fi
if test -n  " $e"
then
                if test -n  " $f"
                   then e='eqnf'
                   else e=" eqn $*f"
                   d="
                fi
fi
eval " $f $e troff $O -cm  $a $d $o $x" ;    exit 0
```

Note: This is a slightly simplified version of an actual UNIX system command. It uses many of the features available in the **shell**. If you can follow through it without getting lost, you have a good understanding of **shell** programming. Pay particular attention to the process of building a command line from **shell** variables and then using **eval** to execute it.

null

```
#       usage: null file
#       create each of the named files
#       as an empty file
for eachfile
do
      > $eachfile
done
```

Note: This procedure uses the fact that output redirection

creates the (empty) output file if that file does not already exist. Compare this procedure with procedure **mkfiles** above.

phone

```
#       usage: phone initials
#       prints the phone number(s) of person
#       with given initials
echo 'inits      ext       home'
grep "^$1" <<\!
abc       1234       999-2345
def       2234       583-2245
ghi       3342       988-1010
xyz       4567       555-1234
!
```

Note: This procedure is an example of using an inline input document or *script* to maintain a *small* data base.

writemail

```
#   usage: writemail message user
#   if user is logged in, write message on terminal;
#   otherwise, mail it to user
echo "$1" | { write "$2" || mail "$2" ;}
```

Note: This procedure illustrates command grouping. The message specified by **$1** is piped to the **write** command and, if **write** fails, to the **mail** command.