EDP ANALYZER

SEPTEMBER, 1975
VOL. 13, NO. 9

PROGRESS TOWARD EASIER PROGRAMMING

Today's so-called high level programming languages (such as COBOL and PL/1) are really just steps in an evolutionary process of making the coding part of programming easier. The terminology is unfortunate, because "high level" implies that a language plateau has been reached. In order to categorize further developments, one is forced to use unsatisfactory terms like "higher level" or "very high level." While the data processing field has moved relatively slowly up these steps—from absolute binary to symbolic to assembly languages to compiler languages—progress is continuing in making the coding function easier. In this report, we discuss several such developments (languages that are one or more steps above COBOL, PL/1, etc., as far as ease of use is concerned) and some user experiences with them.

In this report, we will be concerned with some developments that are aimed at making the programmer/computer interface somewhat less demanding for the programmer. We are using the term "programmer" in its broadest sense, to apply to anyone who creates new procedures for the computer to follow. Not only is the interface being made easier for professional programmers, but it is also being improved so that non-programmers can create procedures for computers.

In a business-type organization, there is a spectrum of people who might create procedures for the computer. Listed in the order of decreasing amount of programming training required, they are:

- System software programmers
- Application programmers
- Analyst/programmers
- Professionals using computers in their work
- Business system analysts
- User department personnel
- User department managers
- Executives

The goal of the developments we will be discussing is to make it easier for all of these types to use the computer. This does not imply that all will be dealing with the same complexity of procedures; rather, each will deal with procedures that are appropriate to his or her job.

The approach taken with most of these developments has been that of higher level languages. As Leavenworth and Sammet (Reference 1) say, the term "high level" is relative. They trace the evolution of languages away from the need to specify *how* something is done and toward the specifying of *what* should be done. But the state-of-the-art is not at the point yet where just the *what* need be specified.

What are the characteristics of today's higher level languages, as compared with the norm such as COBOL and PL/1? Here is how we see it.

CHARACTERISTICS OF HLL

As compared with COBOL or PL/1, a HLL:

1. Has more powerful commands that do more than typical conventional commands.
2. Is not necessarily as rich nor as flexible a language, but is still a "complete" language that can be used for pro-

gramming a complete application system.

3. Can be learned more quickly and easily; as a byproduct of this characteristic, finding good programmers in the language should not be as difficult as finding good COBOL or PL/1 programmers.
4. Helps to structure the program design, by assisting in handling complexity.
5. Has default options for housekeeping functions, such as open and close files, relieving the programmer of these routine duties.
6. Eventually will be largely nonprocedural, wherein the user specifies *what* is to be accomplished rather than *how* it will be done.

There are several types of computer languages that we are *not* including within our discussion in this report. These are:

- Shorthand processors for conventional languages
- Data definition and data manipulation languages
- Job control languages
- Small system languages, such as RPG and RPG II
- Query languages

The criterion we used in selection was: can the language largely or completely replace COBOL or PL/1 in a medium to large size installation? Some come close; RPG II has supplanted COBOL in small to medium size installations, and query languages have almost completely replaced COBOL for creating ad hoc reports at many places. But we chose to draw the line according to the criterion just given.

Want to change languages?

If a user organization is considering changing its "standard" programming language, in order to make programming easier, several options are available. For this discussion, we will assume that the organization is currrently using ANS COBOL. Here are the major options, as we see them:

CURRENT OPTIONS IN HLL

1. Change to a COBOL-based HLL; preprocessor used to convert the language statements to ANS COBOL.
2. Change to a structured generator language that is directly compiled into object code, without going through COBOL.
3. Supplement COBOL with one or more specialized languages that are appropriate for particular applications.

We have not included among the options the choice of converting from COBOL to, say, PL/1. As far as we are concerned, PL/1 is essentially at the same programming level as COBOL (without getting into a discussion of the merits of each language). It is for this same reason that we did not include shorthand processors for today's conventional languages; the level remains the same but the verbiage is reduced. (For readers who *are* interested in this type of change, Reference 2 has a discussion of the problems involved; for instance, Harold discusses the COBOL to PL/1 change, but the remarks could apply to any change of programming languages.)

A main factor in the choice among the various options, of course, is the type of user who will use the language. For the purposes of this report, we will concentrate on languages designed for application programmers, analyst/programmers, business system analysts, and end users.

Here, then, are user experiences with several higher level languages.

COBOL-based higher level languages

As mentioned above, COBOL-based HLLS are translated (by a preprocessor) into ANS COBOL source code. The COBOL code must then be compiled into object code in a conventional manner. The "higher" level of the language is achieved by either the use of COBOL macro-instructions or by the use of a very different language that is translated into COBOL.

Included in this category are both program and application system *generators.* With generators, the user provides values of parameters; the generators then generate the COBOL source code (in this category, at least).

We will discuss user experiences with four systems in this category: METACOBOL, WORK TEN, CL*IV, and GENASYS.

MetaCOBOL

Metropolitan Life Insurance Company, with headquarters in New York City, has assets of almost $33 billion and employs over 45,000 people. The company has four data processing centers in the U. S., in New York City; Scranton, Pennsylvania; Wichita, Kansas; and Greenville, South Carolina. Metropolitan Life uses multiple vendors, having both IBM 370 and Honeywell equipment installed and an H-66/20 on order.

Essentially all applications programming on the IBM equipment is done in COBOL, and the plan for the future is to become an all-COBOL shop. In order to provide a better degree of pro-

gram transferability between its computer systems, the company has developed its own standard subset of COBOL.

In 1972, Metropolitan Life acquired METACOBOL. One reason for choosing METACOBOL was to provide efficient COBOL macro-instructions for frequently occurring functions. Another reason was to use the METACOBOL preprocessor as a COBOL standards enforcement tool.

The COBOL macros are developed by the company's system programmers, who are quite careful about this implementation. They want to avoid conflicts between different macros, and they want to achieve as high a degree of efficiency as they can in the macros. One group in systems programming is responsible for the coordination of the development and maintenance of macros to avoid or eliminate conflicts.

Once macros have been developed, they are entered into a coordinated macro library for use by programmers. Some macros are called out by application programmers in the programs they write. Other macros analyze the application programs during the METACOBOL preprocessing step for compliance with programming standards. Still others have been used for converting existing programs from, say, GE COBOL to IBM COBOL.

Currently, Metropolitan has been using using METACOBOL to convert its GE435 COBOL programs to IBM COBOL as well as to develop an automatic test data generator. When completed, the TDG will require little if any input from a programmer, but, rather, will analyze a program to locate and examine all conditional instructions and then automatically create test cases for testing all paths. Metropolitan Life already has plans to use the facilities of METACOBOL for other purposes in its computing environment.

METACOBOL is a generalized macro preprocessor operating under DOS or OS on IBM 360 and 370 systems. Applied Data Research, Inc., developers of METACOBOL, has continued to announce improvements and enhancements to it.

There are a number of elements of the METACOBOL system. The METACOBOL *translator* includes the macro translator and library facilities for a shorthand, a standards enforcer, a COBOL source program optimizer, and a major logic generator. The major logic generator generates COBOL source code for file matching, input editing, and report writing. It also includes conversion facilities for converting from the earlier IBM 360, Honeywell D and H, and RCA Spectra COBOL source code. A *test data generator* creates test data for debugging programs. The *run-time debugging aid* reports the location and cause of one or more clerical errors and the contents of input, output and intermediate fields during program testing. And a COBOL *Performance Monitor* analyzes and reports on program activity by paragraph—isolating critical processing and input/output activity and counting paragraph entries.

The METACOBOL translator makes use of a program development macro library. Users can define higher level verbs for frequently used functions, and express those verbs in COBOL source code. When the translator encounters such a verb, it replaces it with the appropriate COBOL source code. METACOBOL can also be used to interface COBOL programs to data management systems such as IMS or TOTAL or to monitor programs such as CICS.

ADR has also developed structured programming facilities for the COBOL language. This facility includes a modular discipline, under which each module of a program has a standard format. This format includes an identification section, a data section (for locally defined data), and a procedure section.

Several language changes have been made for supporting structured programming. The GOTO verb has been eliminated. The IF statement has been changed. A COBOL IF statement requires a period to terminate it—but a period terminates all outstanding IF statements. With COBOL, it is hard to nest a hierarchy of IF statements without using GOTO. So METACOBOL has added ENDIF to the IF statement (IF ... ELSE ... ENDIF) which terminates only the most recent IF.

DOWHILE is one of the basic structured programming control structures. The procedure is performed if the test is true, and is exited if the test is false.

METACOBOL has been in use since 1970, There are now some 275 organizations worldwide using it.

For more information on METACOBOL, see References 3 and 6.

WORK TEN

National Steel and Shipbuilding Company (NASSCO), with headquarters in San Diego, Cali-

fornia, constructs and repairs ocean-going ships. Annual sales are in the order of $200 million and the company employs some 5,000 people. Their data processing is performed on an IBM 370/145 with a 384K memory, operating under DOS/VS. They have eight programmers, two analysts, and two system programmers. Languages used are COBOL, ALC, DYL 260, and WORK TEN.

In 1971, NASSCO learned that other companies around the country were successfully using WORK TEN. NASSCO was not too happy with the verbosity of COBOL and began looking into available packages. They selected WORK TEN and have used it for essentially all batch programs since that time. They had not been using it for application programs operating in a teleprocessing mode; those programs were written in COBOL. One-time jobs, such as ad hoc reports, have been written in DYL 260.

WORK TEN is a COBOL source program generator. Coding is done via a set of structured forms. The WORK TEN preprocessor generates ANS COBOL source coding, using symbolic data names. The COBOL source code is then compiled to give the object code.

NASSCO has found that, on the average, WORK TEN requires about 60–65% of the time required to develop and debug the same program in COBOL. A programmer who knows COBOL can use WORK TEN better than one who does not. The quality of the object code—in terms of amount of memory used and execution time—is probably competitive with COBOL, everything considered, in their opinion. It might be possible to make more efficient use of resources using COBOL, but it would require more skilled programmers, they believe. WORK TEN tends to enforce discipline; for instance, it is easier for the programmer to use the file library than it is to bypass it—so the result is more consistent data definitions. WORK TEN is very reliable and relatively easy to learn; experienced COBOL programmers are in productive use of it after two weeks of training and beginning use.

One typical job involved the batch updating of an inventory file. The program made use of two input and two output files, all of which had been previously defined in the file library. It required only four man-days to code and test the program according to specifications. The object program used about 45K bytes of memory.

NASSCO recently reassessed their use of WORK TEN. They wondered if they should move to another language, which they could use for both batch and on-line application systems. They decided that they very much liked what WORK TEN was doing for them and that they would consider using it for the on-line programs.

WORK TEN is marketed and supported by National Computing Industries of Atlanta, Georgia. As indicated above, it is a COBOL generative file management system. As NCI says, "WORK TEN automatically generates the necessary logic to perform the mechanical, repetitive tasks of programming." These tasks include opening and closing files, reading transactions, reading master records, matching transactions to master records, rolling and clearing control break accumulators, writing master records, printer control, etc. It runs on an IBM 360/370 with a minimum of 65K characters of memory.

Four structured forms are used in programming via WORK TEN. The *work sheet* is used to record the analyst's narrative description of the program. The *field/record sheet* is used to specify the COBOL identification and environment division entries, plus file and record definitions, but in a much more concise form than COBOL. The *storage form* corresponds to the COBOL working storage definition, but again is expressed in more concise form. Finally, the *procedure form* corresponds to the COBOL Procedure Division. A special set of verbs, similar to COBOL verbs, are used for specifying the procedures.

NCI claims WORK TEN to be an easy discipline for implementing structured programming. They call their approach ASPT (Automatic Structured Programming Technique). ASPT actually, produces a top down design automatically, without the use of special verbs or rules to follow. Using ASPT, the programmer performs the coding of program modules or stubs. The modules have one entry point and one exit point. The use of GO TO is restricted to referencing another line of code within the same module or referencing the exit point of the module. The programmer is prevented from invoking any coding at the current or higher level by ASPT. The main line control logic, plus the logic to perform the mechanical tasks described above, are automatically generated by WORK TEN in a hierarchical manner.

For more information on WORK TEN, see Ref-

erences 3 and 7, at the end of this report.

*CL*IV*

The Arizona Bank, with headquarters in Phoenix, has assets of over $900 million and 70 branches in Arizona. For their data processing, they use an IBM 370/158 with 1M bytes of memory. They are in the process of converting from DOS to VS1. They employ about 25 programmers, and the languages used include COBOL, BAL, CULPRIT (for ad hoc reports) and CL*IV.

The management policy at Arizona Bank is to purchase as much of their application software as possible, rather than develop it in-house. If a software package does not quite meet their needs, however, and if they cannot arrange for the vendor to make the needed changes, they will make the changes in house. Most of the work of their programmers is in maintaining and extending the application systems that have been purchased.

In 1971, Arizona Bank entered into a contract with GSI, Inc., a Phoenix-based software company, to develop a portion of a time deposit accounting system. GSI had some ideas for a new language, CL*IV, and they felt they could develop it as a part of the contract. Some of the principals of GSI had previously developed a similar language and so had experience with this type of software. GSI suggested that they develop the new language and write the application system in it, and the Bank would have a permanent license to use the language and preprocessor. After going over the plans for the language and preprocessor, the Bank agreed to the proposal.

Previous to 1971, the Bank had obtained an application system that was written in one of the commercial higher level languages that produced COBOL source programs. In that case, the Bank did not obtain the preprocessor or the HLL programs; instead, it received just the COBOL source code that had been generated by the preprocessor. The Bank found this code hard to maintain, due to the way the data names had been assigned by the HLL and due to the difficulty of changing the file matching code generated by the HLL. If they had received the programs in HLL form, along with the preprocessor, this maintenance would have been no problem. But with only the COBOL source code, maintenance was difficult. So, with CL*IV, they insisted on getting the preprocessor and the programs in HLL form.

(In November 1972, GSI sold all rights in CL*IV to Informatics MARK IV Systems Co., which markets the well-known MARK IV file management system.)

Two of the five sub-systems of the time deposit accounting system were written in CL*IV under the contract. The two sub-systems went into operation in 1973, and the Bank has been maintaining them since that time. They have found CL*IV easy to use. CL*IV generates ANS COBOL, which is then compiled to produce the object code. While the Bank could maintain the programs from the COBOL source code, they prefer to maintain it via CL*IV. The data names that are generated are very understandable. All programs are structured the same way, so the programmers know just where to look for the code that is to be changed. Essentially all debugging is done at the CL*IV source level; seldom do the programmers look at the COBOL source.

When the Bank started to convert from DOS to VS1, they selected the CL*IV programs for the first conversion. As far as the application programs were concerned, all the Bank had to do was change one card in each CL*IV source deck and then rerun that deck through the preprocessor. It required only three man-months to convert 51 programs, including creating the JCL, testing, and documentation.

CL*IV is marketed and supported by Informatics MARK IV Systems Co., of Canoga Park, California, and 14 other locations in the U. S., Canada, Europe, and Japan. CL*IV is an enhancement to ANS COBOL. It operates on the IBM 360/370 under DOS, DOS/VS, OS, VS1, and VS2.

When using CL*IV, the application logic is written in ANS COBOL. A simple syntax is used for the identification, environment, and data division entries. COBOL source code is generated for routine procedural functions, from parameters supplied by the programmer. These include: the main line control structure; file management functions such as opening and closing files, reading and writing records, record matching, detecting end-of-file, etc.; setting up print lines; accumulator management; and control break logic.

CL*IV adds two powerful verbs as an enhancement to the COBOL verbs. One is the PRINT verb. It eliminates having to write a lot of MOVE operations and having to set up lines of print. When-

ever the PRINT verb is compiled, it creates a sample report format so that the programmer can check what the report will look like. The other is the INITIALIZE verb. It creates new occurrences of records and sets each field in the record to spaces or zeros, depending on the elementary picture. The INITIALIZE verb is also used for clearing any group or elementary item, such as tables, hold areas, etc., defined in the program.

In addition, users can pre-define data definitions or paragraphs and sections of procedural code and store these in the source library. These areas of code can be inserted in CL*IV programs by means of a COPYH command.

CL*IV imposes no artificial restrictions on ANS COBOL. All access methods and storage devices allowed by ANS COBOL are supported by CL*IV.

For more information on CL*IV, see Reference 8.

GENASYS

The *University of California's* Information Systems Division, which is part of the Office of the President, is located in Berkeley, Calif. ISD handles the development of computer-based systems for administrative functions of the university. For the data processing in Berkeley, they have an IBM 360/65 and a 360/30.

In 1972, one of the University Extension programs wanted ISD to develop an application system on an urgent basis. ISD just did not have the resources to meet the tight time schedule and suggested that the system might be developed by an outside contractor. A contract was awarded to study the requirements and develop the specifications for the new system. The programming of the system was to be done under another contract, where the bids were based on the system specifications.

It was at this bidding stage for the programming that things got interesting, we were told. Bids were requested from about a dozen contractors and three bids were received. Of these, the lowest was from International Computer Trading Corporation, in San Francisco. It was about one-half the price of the next lowest bid, and furthermore, it was a fixed price bid. It was based on ICT's use of their GENASYS service. ICT was given the contract, and delivered on schedule.

GENASYS is a set of generators for generating application system programs, in source code form. The source code may be ANS COBOL or PL/1. In the University's case, they are a PL/1 shop and so they requested that the programs be deliverd in PL/1 code.

Based on this successful use of GENASYS, the Information Systems Division made sure it was considered in two other cases. One of these—a treasurer's information system—had the system specifications developed by an outside contractor before bids for the programming were requested. In this instance, only two bids were received—and again, ICT was the low bidder with a bid price almost one-half of the other. In the other case, the ISD people developed the system specifications in-house and put them out to bid—and ICT was the only bidder. In both of these latter instances, the bids from ICT were fixed price. But the treasurer's information system contract had to be renegotiated because the specifications that had been developed by another outside contractor proved to be incomplete. So additional time and cost were needed for completing the specifications. Except for this delay, the programs were delivered about on schedule.

What about the maintenace of the programs generated by GENASYS, we asked. First of all, we were told, if any bugs are uncovered that were caused by GENASYS, ICT corrects these. But for modifications and enhancements, the programs have proved to be quite maintainable. The programs have a common structure, so the programmer knows where to look for the code to be changed. And the naming standards become clear upon study. GENASYS does make a different use of resources than their own programmers might, we were told, and additional time was required during final debugging stages than might be needed if the work was being done in-house. But GENASYS did get the jobs done for the University.

GENASYS is based on the concept of standard system architectures, expressed in terms of system generator software. ICT people believe, based on their experiences to date, that these standard system architectures can meet from 80% to 90% of the needs of any given business application system. The other 10% to 20% of the needs must be met by custom programming. ICT has developed four standard system designs. The ICT analyst selects the design that best meets the need of the application system, in any given case.

Input for GENASYS is developed by means of a specification workbook, with questionnaire-like forms, plus copies of all input forms and reports. Master files are defined and requirements for tailor-made procedures and/or reports are identified. At this point, the ICT production unit takes over. Using the collected material, plus GENASYS (in the design mode), the production unit produces the first version of a design manual. This manual contains extensive documentation about the system.

At this point, the ICT analyst on the job sits down with the customer's representative, to see if the design is on the right track. ICT finds that an average of about two weeks have elapsed to this point, since the time that the contract was signed. Any changes are noted in the design manual, which is then returned to the ICT production unit. Using the change data, the next version of the design manual is created. This version is more detailed than the first, and default options are spelled out. The design is reviewed with the ICT analyst on the job, and any necessary corrections are made.

Then, using GENASYS in the generate mode, source programs are created in either ANS COBOL or PL/1. The programs are compiled and any syntactical errors are corrected. Then they recompile the programs, produce the necessary JCL code, and produce the revised documentation manual.

During this same period, the analyst on the job has been working with the customer to develop acceptance data. ICT finds they need this test data typically by the third week of the project. Two options are provided for testing: the testing may be done either at ICT or on the customer's machine. ICT prefers the former case. Out of this testing comes the need for more changes. The ICT production unit makes the changes and regenerates the source code. The source programs are then generally ready to turn over to the customer. Average total elapsed time—30 to 35 working days.

In addition, ICT conducts a training class to acquaint the customer with the application system—naming conventions, logic of all programs, etc. The customer must be in a position to maintain the system (which is in ANS COBOL or PL/1) from that time onward.

To date, ICT has concentrated on selling the GENASYS service, as discussed. However, arrangements can be made for leasing the GENASYS package, for those organizations that would want to make extensive use of it.

For more information on GENASYS, see Reference 9.

Structured generator languages

This category of HLL probably should be subtitled: "Ones that make a complete break with COBOL." It is possible to have structured generator languages that produce ANS COBOL; we discussed two such (WORK TEN and GENASYS) above. But it is also possible for them to produce object code directly. This is the type of structured generator language we will be discussing in this section.

Structured generator languages have been called by various names in years past. "File management systems," "file maintenance generators," and "report program generators" are terms that have been applied to this type of language. Typically, these languages have used structured forms that can be filled in with a fraction of the writing effort of, say, a COBOL program. For the file definitions, for instance, the form might ask: Standard labels? Non-standard labels? No labels? If the programmer checks that standard labels are to be used, pre-defined routines for creating and handling the standard labels will automatically be embedded in the program. Also, for queries and conditional expressions, a standard format is laid out on the form: Field No. 1, Relational Operator (Equals, Not Equals, Less Than, etc.), Field No. 2 (or literal expression), and action to take. Complex conditions, where a series of relations are joined by Ands and Ors, are simply expressed on a series of lines on the form.

Included in this category are the MARK IV and ASI-ST file management systems. We have discussed both of these systems in previous issues, particularly the April 1970 report, and will not repeat the discussion here. Both are very popular. MARK IV is used at some 800 installations, and ASI-ST is used at over 100 installations.

Another popular structured generator system is the Adpac system, which is in use at over 150 installations. We have not previously discussed Adpac, and so will include it in this report.

Also, an interesting enhancement has been made to ASI-ST, to promote its use by the end user market. This is an on-line version, called Conver-

sational ASI-ST. We will discuss it, too.

It should be mentioned that some users of this type of language have reported to us that their programmers resist using the language. Perhaps the ease of filling in the forms implies a downgrading of their programming talents, in their minds. But where data processing managements have seen the value of such languages and have firmly supported their use, the benefits have been impressive.

Adpac

Matson Navigation Company, with headquarters in San Francisco, California, is a pioneer and leader in the transportation of containerized ocean freight. The company operates 14 vessels between the United States Pacific Coast, Hawaii, and Guam. The company has almost 2,000 employees. For their data processing, they use an IBM 370/135 with 192K bytes of memory, operating under DOS/VS. The installation not only processes local batch work but also on-line services for terminals located in several facilities along the West Coast of the U.S. and Canada. They have nine programmers and three system analysts. About 85% of their 2,000 active programs have been written in Adpac, about 10% in assembly language, and the remainder in COBOL. COBOL is being phased out.

Matson purchased Adpac in 1965 to help lessen a potential major conversion effort from IBM 1400-series equipment to the then new 360/30s. Since Adpac programs are machine independent, code written for the 1400 computer was directly usable by the 360 version with only trivial code changes. Thus, with the conversion in the offing, Matson experimented with new programs in Adpac so that the conversion could be eased. They continued to use Adpac as one of their programming tools until 1969, although most of the programming was done in COBOL and Assembler Language after the 360/30 and the 360/40 were installed.

In 1969, Matson was faced with the need to program a complicated freight documentation, operations and statistics system. They estimated the programming job at well over 50 man-years. With their project team of ten programmers and two systems analysts, this just looked like too big an effort to implement in COBOL with the target completion date. Since they had used Adpac and were quite impressed with it, they considered it. It looked like the whole job might be done in about 32 man-years—still a big effort, but substantially below that of their estimated COBOL time.

By 1971, the programming on the freight system was going so well that Matson decided to standardize on Adpac. Since that time, essentially all new batch programs have been written in Adpac. COBOL has been used only for maintaining the old programs, and these have been almost completely phased out.

Adpac uses structured forms, of the type described earlier. A one-pass compiler generates relocatable object code for the IBM 360/370. Compiling time is about at card read speed; a 500 card program can be compiled in less than one minute.

Adpac generates either user defined or standard structures for its programs, and Matson finds that this makes program maintenance much easier than it is with free-form programs in other languages. The programmers know just where to look in the program structure for the code that is going to be changed. In fact, most of Matson's program maintenance time (for "fire-fighting") is now going into the 100 or so COBOL programs that are still active.

Compared with COBOL, Matson says that many fewer input cards are required in order to create the same program, by a factor of 2 or 3 to 1. While they have not tried to parallel program a total system concurrently with both Adpac and COBOL, based upon recoding many programs in Adpac, they have found that Adpac is saving significant programmer time (for design, code, test, and documentation) in the same proportion as the reduction in input cards. The code generated by Adpac is at least as good as COBOL in efficiency—that is, in memory usage and in execution time.

As an example of use, Matson quoted the time required to create a simple program for calculating and printing some shipping tariff rates. All input files were already in existence. It took 35 minutes to write the Adpac program, which resulted in 205 cards. Compile time was a total of one minute and seven seconds—of which about 25 seconds was the compile time and the rest was link edit time. Two test shots were required, and then the program was ready to go. The complexity of problems programmed have ranged from simple to complex, where the latter might

have over 2,000 coded cards, with from one to six instructions per card.

Matson feels that any user who is considering this type of system should give it a fair test, for at least two to three months. It takes this long to learn to "think" in Adpac and get the most out of it. Experienced assembly language programmers can learn it in less time.

Adpac is a structured language, with defined logic for the routine portions of a program. These portions include file open and close, read and write records, record matching, and so on. A CALL facility is available for using object modules in the program library, and a COPY facility can be used for copying data descriptions, procedures, and even whole programs from the source library. Recent changes improve Adpac's performance in a virtual storage environment.

Each Adpac program has the same structure, with four divisions in the coding forms. These four divisions are: input-output, data, process control, and instructions. The process control division defines the flow of control in the program. Each division has a specific coding form, with (in general) a fixed format. The exception is that, for algebraic equations, Adpac uses FORTRAN-like free-form coding.

There are five major components of Adpac II software. The *program generator-compiler* is a one-pass processor that translates the Adpac input into object code. For the routine processes just mentioned, it generates the code from relatively few parameters from the input forms. Adpac also includes a *text editor, a source library management system, an Adpac-to-COBOL translator, and a program specifications writer.*

In Adpac, each program is stand-alone. All data names are unique to that program. When data definitions are copied from the library, they are redefined for the new program. Within a program, the file names are one character in length, and data names and sub-routine names are three characters in length. (Matson told us that these restrictions imposed no particular problems on them.)

For more information about Adpac, see References 3 and 10.

Conversational ASI-ST

Combustion Engineering, Inc., with headquarters in Stamford, Connecticut, is a major manufacturer of steam generating and other industrial equipment. Its sales in 1974 were in excess of $1.4 billion, and the company has over 40,000 employees.

For its main data processing, Combustion Engineering uses an IBM 370/158 and a 370/168 located at Windsor, Connecticut, near Hartford. The main programming languages used are COBOL, PL/1, FORTRAN, and ASI-ST. There are about 200 employees with job titles of programmer, an almost equal number of system analysts, and a large number of end users who desire to interact directly with the computer. We discussed Combustion Engineering's use of ASI-ST, TOTAL, TSO, and INTERCOMM in our June 1973 report.

The company has been using ASI-ST since 1969. They have used it not only for one-time jobs (such as ad hoc reports) but also for programming complete application systems where appropriate. ASI-ST is a preprogrammed data management system that uses basic structures for input validation, update and file maintenance, reporting, and such. In the regular use of ASI-ST, the users fill out structured forms, from which key punching is done, or they enter the data on cassettes via Datapoint terminals. And when Combustion Engineering used ASI-ST with TSO, as we discussed in our earlier report, the user essentially filled out the ASI-ST forms and used the terminal much as a keypunch, entering characters representing desired options in specific character positions on a line.

But this interface—regular ASI-ST under TSO—was not a convenient one for end users to use. They had to fill out the structured forms and simulate keypunching at the terminal. And there was some resistance from programmers to filling out such forms, perhaps on the basis that such activity did not use their programming talents sufficiently.

Applications Software Inc., developers and marketers of ASI-ST, introduced Conversational ASI-ST in 1973. It is a front-end package for ASI-ST. Combustion Engineering installed Conversational ASI-ST in 1974, still operating under TSO. It is a high level language for use at a terminal. It provides a free form (unstructured) language capability and can prompt the user for the needed parameter values. The front-end makes diagnostic checks, indicates errors to the user, and passes correct statements on to ASI-ST. The language translator can also be used in the batch made for

those who want to use the free form language.

Combustion Engineering uses ASI-ST for essentially all ad hoc reports. But training in the use of Conversational ASI-ST is in the build-up phase, so that only a fraction of the ad hoc reports as yet are requested through it. There is some evidence that system analysts and programmers are more willing to use the free form language of Conversational ASI-ST than they were to filling out regular ASI-ST coding forms. An ad hoc report request generally takes less than one hour, from the time the request is written until the report is returned to the user.

Further, the people at Combustion Engineering find that it is possible to program a fairly extensive application with ASI-ST or Conversational ASI-ST in a fraction of the time it would take to do the same job with a conventional programming language. Man-months of effort can be shortened to man-weeks, in such cases.

While it is hard to compare the coding efforts required with different languages, Combustion Engineering estimates a 10 to 1, or better, reduction in the number of lines of code needed with ASI-ST or Conversational ASI-ST, as compared wtih COBOL.

As mentioned, Conversational ASI-ST was developed and is marketed and supported by Applications Software Inc. of Torrance, California. It is an enhancement to ASI-ST; a user would need ASI-ST, to which the conversational package can be added. We discussed ASI-ST in the April 1970 report.

Conversational ASI-ST is designed to run on the IBM 360/370 under TSO, IMS DC, and CICS. It consists of five main elements: ASI-ST, the Conversational ASI-ST Language, a preprocessor, an extended interactive facility, and a macro language option. Using a terminal, the user enters a statement in a free but somewhat coded form. The preprocessor scans the statement, checks for syntax errors, and notifies the user of errors. After corrections have been made, the preprocessor translates the statement into a fixed-field ASI-ST statement. When the complete request, or a set of requests, has been entered, they are transmitted to batch ASI-ST for execution.

The extended interactive facility extends the system to a communications environment. The command language uses simple words (change, delete, find, etc.), an on-line text editor, and a library management facility. With the latter, users may save files from session to session; the facility also provides accounting control information and the monitoring of file passwords.

With the macro language option, programmers may produce their own commands and subcommands. Also, they can write interactive application programs that can communicate with users, using terms that are familiar to those users. Moreover, it can be used to develop terminal oriented dialog which prompts users for data that is required by any program, regardless of whether ASI-ST is involved in the process. The option includes the macro language and a macro language processor.

ASI has recently announced ASI/INQUIRY, an interactive query system that runs under IBM's IMS DB/DC (data base/data communications). It employs an easy to use language, has rapid access to on-line IMS data bases, and provides both manual and automatic searches on an interactive basis.

For more information on ASI-ST, see References 3 and 11. Details on Conversational ASI-ST can be obtained from ASI, Reference 11.

Supplemental languages

Another way of providing computer services to a broader range of "programmers" is to supplement a conventional language (such as COBOL) with one or more higher level languages. The goal in such a procedure would be to provide end users with an easier way to retrieve and process data—for answering inquiries, for preparing ad hoc reports, and for performing special analyses.

Conversational ASI-ST falls into this category, as do the commercial query and reporting packages. As mentioned earlier, we are not discussing the query and reporting packages in this report becasue of their inability to handle validation functions and updating functions.

We have singled out two languages to illustrate this category of supplemental languages. These are RAMIS and APL. Some people might object to this classification on the grounds that either might be considered a "complete" language, capable of replacing conventional high level languages. But for business data processing, we believe that languages such as RAMIS and APL will be used in addition to the conventional languages, as we hope the following discussion will explain.

RAMIS

Esmark, Inc., with headquarters in Chicago, Illinois, is a holding company formed in April 1973 which has major interests in food, chemicals, industrial products, energy, insurance, and business and financial services. It has four sub-holding companies, one of which is Swift & Company, a diversified food complex; each sub-holding company owns several subsidiary companies. Esmark's annual sales are in the order of $4.6 billion and it employs some 33,500 people.

In 1971, the people at Swift & Co. saw a need for a financial planning model. After considering a number of alternative ways to obtain the model, they awarded a consulting contract to MATHEMATICA, Inc., of Princeton, New Jersey. In addition to bringing model-building talent to the job, Mathematica also made available its RAMIS system for both programming and running the model. RAMIS is a data management, information retrieval, and reporting system that can be run in either a batch or an on-line mode. RAMIS can be licensed or leased from Mathematica or RAMIS service can be obtained from National CSS Inc. timesharing. The availability of RAMIS was one of the factors in Esmark's selection of Mathematica. They purchased RAMIS so that it could be run on the computers of their data processing subsidiary, Cogna Systems Corporation.

The characteristics of the financial planning model give some idea of how Esmark has used RAMIS. The Esmark planning cycle occurs twice a year. In the spring of each year, the strategic, 5-year plans are developed. Lots of "what if" questions are analyzed—"what would be the effect on our financial statement if project X was started two years later?" and so on. Alternative courses of action are developed. During the summer months, these alternatives are studied, revised, sharpened. Then in the fall, the management plan is developed. This is the 5-year plan for the overall corporation, with greatest emphasis placed on the first year. After it has been developed, the management plan is presented to the Esmark Board of Directors. After the Board has approved (and perhaps revised) the plan, it is used by the various components of Esmark for developing their monthly operating budgets.

The model was developed by Esmark (Swift & Co.) staff members working with the Mathematica consultant. The consultant then programmed the model on RAMIS. That first year, we were told, was a real challenge. The development and programming of the model was a problem, of course—but the real challenge came in getting the financial data in shape in a one-week period. Validating the data requires that the data be analyzed in several different but related ways, to make sure that it is accurate and consistent. The data, which had been supplied by the sub-holding companies, was put into a RAMIS file and all validation tests were performed on it in that file.

Validating the financial planning data continues to be a challenge, so Esmark has adopted a change in procedure. A terminal has been installed at each of the sub-holding companies, for input to the Cogna computer, on a remote batch basis. Each sub-holding company develops its own planning data file and validates it, using RAMIS. When the validations have been completed, Cogna consolidates the files of the four sub-holding companies and transfers the resultant file to National CSS. The people at Esmark headquarters use National CSS for developing the planning reports because of the excellent turnaround that National CSS provides.

In addition, a number of other uses of RAMIS have developed within Esmark headquarters and within the several sub-holding companies.

We asked about the training required in order to use RAMIS. There are really two types of users within Esmark, we were told. A number of them simply retrieve data from RAMIS files. Training for this function is relatively simple, they say—perhaps one hour of training in order to get useful output, plus more training and experience to learn various ways to format reports. The other type of user must validate input data and write programs for updating RAMIS files. While the people who perform these functions are not classified as programmers at Esmark, they have many characteristics of trained programmers. But a good RAMIS programmer is different from a good conventional programmer, we were told; RAMIS has high level languages and the programmer must learn to program within the constraints of these high level languages. It takes at least two months of training and job experience before useful applications system output is obtained for this type of RAMIS use, say the people at Esmark.

In the main, RAMIS is a user-oriented information and data retrieveal system. Actually, it is

more than that, as its main components will indicate. *Languages:* RAMIS has an English-like report request language plus a high level programming language for file maintenance, updating, and simple input validation functions. RAMIS also interfaces other languages such as COBOL, FORTRAN, and PL/1. Generally, complex input validation and mathematical routines are programmed in these other languages. *Data files:* RAMIS provides hierarchical and network structures for data. *System software:* the RAMIS system software catalogs requests, retrieves requests from the catalog, and supervised the sequence of activities in the running of RAMIS jobs. *Equipment:* RAMIS is designed to operate on IBM 360/370 computers under all versions of OS. It is also available via National CSS Inc.

For more information on RAMIS, see References 3 and 12.

APL

Some readers may wonder why we are including a mathematically-oriented language such as APL in a discussion of business languages. The reason is that we have found APL being used as a supplemental language in a growing number of business environments. It is becoming sufficiently important in this respect, as a matter of fact, that we hope to devote a complete report to it in the not-distant future. The following discussion will only treat some of the high points of its use.

The British Steel Corporation, with headquarters in London, England, is the nationalized consolidation of the major U.K. steel companies. British Steel produces about 25 million metric tons of steel per year, and employs some 230,000 people. The corporation does not have a computer installed at its headquarters but uses installations at its plants, at its research centers, or uses outside services, depending upon the application.

The Planning Department of British Steel, among its other responsibilities, produces an economic plan on a year-by-year basis, for ten years in the future. This economic plan includes a forecast of sales, prices, the need for additional facilities, and so on. An economic planning model has been created for this function, written in FORTRAN.

In addition, the department performs specific planning studies that go into more detail than the 10-year economic plan. One such study, begun in 1973, was concerend with selecting the most profitable strategy for the production of billets. A billet is a piece of partially completed steel, about four inches square and from 20 to 60 feet long. The study involved forecasting the sales of billets over a ten year period. A number of alternate strategies were defined which allocated production to the plants over a multiple year time period. The goal was to insure efficient overall production as well as to provide a basis for costing and evaluating the profitability of the strategies. The problem was reasonably complex, with some 60 different types of billet, 15 plants with differing production characteristics, and 12 different strategies. In adddition, each plant was potentially capable of being expanded or closed. The approach which was selected made use of two models, one to perform the allocation and the other to evaluate the strategies.

One member of the Planning Department had seen a demonstration of the APL service offered in London by I. P. Sharp Associates Limited, of Toronto, Canada. He was impressed with how well APL handled arrays, and felt that British Steel should give it a try. So the strategy evaluation sub-model was selected as the pilot test of APL within British Steel. The model would not use linear programming, but instead would use a straight-forward logic, easily understandable by the people in the plants. If the model were programmed in FORTRAN, they already knew the difficulties they would face when they wanted to get at the data or change the model. With APL, it looked as though the economists themselves, working at a terminal, could easily retrieve desired data, perform analyses, and make changes in the model.

British Steel gave a contract to I. P. Sharp Associates to write the model in APL. The work was done during a two month period in early 1974 by Keith Iverson (the son of Dr. Kenneth Iverson, the creator of APL) who works for Sharp. British Steel was delighted with the results. APL in fact did give the Planning Department a much easier interface with the computer.

Based on the results of this pilot test, the Planning Department asked British Steel's Operations Research Department to write the allocation submodel in APL. The OR people had no difficulty in learning to use APL, and in fact, found it a very

logical language as compared to FORTRAN. So British Steel expects to expand their use of APL in the future.

A discussion of the history and characteristics of APL can be found in References 1 and 4. The language was created by Dr. Kenneth Iverson in the early 1960s, while working at Harvard University and later at an IBM Research Center. It was described in an early book, *A Programming Language*, by Iverson (Wiley, New York, 1962). But APL did not catch on as a programming language in a batch environment. It was not until the late 1960s, when it was implemented as an interactive language, that it began to become popular.

APL uses a very concise, symbolic notation. The notation tends to scare off some programmers from the use of APL. The notation is so powerful that programmers can write meaningful one-line programs. The intent of such "one liners" is often not clear to another programmer—or to the programmer who created it at a later point in time. (In fact, we understand that a game has been made of this, where one programmer shows a one-liner to another programmer and says, "Bet you can't tell me what this does.") However, the intent of the language is to provide powerful operators, not to write one-liner programs whose function is unclear. In fact, in Reference 4, it is reported that APL has been successfully taught at the seconday school level.

Becasue of the power of the APL operators and the conciseness of the language, it is reported that coding time can be reduced up to 90%.

As mentioned earlier, we plan to return to the subject of APL in a not-distant report. For more information on APL, see References 13a and 13b.

The future—non-procedural languages?

Philippakis, in Reference 5, reports the results of a survey he conducted on the use of various programming languages in business environments. He sent out 390 questionnaires and received 164 responses, from a variety of types and sizes of companies. While he cautions against attaching too much precision to the results, the figures give a general impression of language usage. He computed a "usage index" as the product of the percent of users using a language and the average percentage of the time they used it. The results were: COBOL, 59; assembler, 20; report generator type language, 6; FORTRAN, 5; PL/1, 4; other, 3; Basic, 1; APL, 0.

His results are not surprising. COBOL clearly shows up as the leader in business data processing, with some form of assembly language second. Apparently the structured generator languages are used more than either FORTRAN or PL/1, and that is a point of interest (certainly considering IBM's efforts to push PL/1 for much of the past decade). While two companies reported using APL, the usage was so small that his usage index was zero for this language.

One point should be emphasized. The amount of use was measured as the approximate man-hours of programming in each language per one hundred man-hours of programming. Higher level languages, such as we have discussed in this issue, would therefore be penalized in multi-language shops. Conventional languages, such as COBOL, FORTRAN, and assemblers, would get higher usage indexes because they require more man-hours of use to accomplish a job, as compared with the higher level languages.

A follow-on study in another two years or so would be interesting, to see what the shifts in usage might be. We would expect to see a measurable shift toward higher level languages. But again, the figures might be misleading if amount of usage is measured in man-hours.

It is clear, of course, that the bulk of the programming being done today uses procedural languages. Non-procedural functions—such as open and close files, record reading and writing, file matching, etc.—have appeared in some of the higher level languages. Macros are being used to eliminate the need of rewriting detailed procedures for frequently performed actions. APL has introduced powerful operators, perhaps not unlike macros for the handling of arrays, to eliminate the need for some detailed procedures. So progress is being made in the direction of non-procedural programming. But the fact remains that all of the languages discussed in this report still require procedural programming.

What will happen in the next few years? Our guess is that users will attempt to reduce development and maintenance costs by moving toward the higher level languages, such as the ones we have discussed. These languages may well be enhanced over what is available today, by the addition of more non-procedural functions. But in general, the languages will look very much as

they do today. Custom-programmed detailed procedures will still be needed by users for those activities that are not covered, or not covered adequately, in the non-procedural portions of the languages.

When will the "true" non-procedural languages appear, where one need only specify *what* is desired rather than *how* to achieve it? Based on the progress of the past ten years or so, we cannot yet foresee the arrival of the "true" non-procedural language.

But maybe the "true" non-procedural language is an ideal that will never be reached. Maybe it is important only as a goal to strive for, not as something that the field should be eagerly awaiting. In that case, we would expect to see significant progress in the next five years. The languages that we have discussed in this report have, in general, been well enough received in the marketplace that their suppliers will continue to improve them. We have discussed how several of them have been enhanced over the past several years, and there is every reason to expect that the improvements will continue.

What does all of this mean to you? As we see it, the picture of higher level languages for the next five years or so is reasonably clear. They will look very much like today's higher level languages. If you are interested in gaining the advantages of such higher level languages, there seems to be no good reason for waiting. Select which basic course of action you prefer—COBOL-based HLL, structured generator language, or supplemental language—and start using it. During the next five years, you will be able to obtain a series of improvements to the language of your choice, in all likelihood.

And by the end of this decade, you might be quite surprised how "non-procedural" your programming has become, as compared with today's conventional languages.

REFERENCES

1. *Proceedings of a symposium on very high level languages,* SIGPLAN Notices, April 1974; order from ACM (1133 Avenue of the Americas, New York, N.Y. 10036), price $15 prepaid.
2. *Commercial Language Systems,* Infotech State of the Art Report 19, Infotech Information Limited (Nicholson House, High Street, Maidenhead, Berkshire, England), price £40 ($95).
3. *Datapro 70,* Datapro Research Corporation (1805 Underwood Boulevard, Delran, New Jersey 08075), price $250 per year.
4. *EDP In-Depth Reports* (P.O. Box 1127, Station B., Weston, Ontario M9L 2R8, Canada); January and February 1974 issues on APL; price $4.50 per copy.
5. Philippakis, A. S., "Programming language usage," *Datamation* (1801 S. La Cienega Blvd., Los Angeles, Calif. 90035), October 1973, p. 109, 110, 114.
6. For more information on METACOBOL, write Applied Data Research, Inc., Route 206 Center, Princeton, New Jersey 08540.
7. For more information on WORK TEN, write National Computing Industries, 6075 Roswell Road N.E., Atlanta, Georgia 30328.
8. For more information on CL•IV, write Informatics MARK IV Systems Company, 21050 Vanowen Street, Canoga Park, Calif 91304.
9. For more information on GENASYS, write to International Computer Trading Corporation, 465 California Street, Suite 318, San Francisco, Calif. 94104.
10. For more information on Adpac, write to Adpac Computing Languages Corp., 101 Howard Street, San Francisco, Calif 94105.
11. For more information on Conversational ASI-ST, write to Applications Software Inc., 21515 Hawthorne Blvd., Torrance, Calif. 90503.
12. For more inforamtion on RAMIS, write MATHEMATICA, Inc., P.O. Box 2392, Princeton, New Jersey 08540.
13. For more information on APL:
    a) Write I. P. Sharp Associates Ltd., Suite 1400, 145 King Street West, Toronto, Ontario, Canada.
    b) Datapro Feature Report, "All about remote computing services," (address above), February 1975, price $10; lists a number of services that offer APL.

SUBJECTS COVERED BY EDP ANALYZER IN PRIOR YEARS

1972 (Volume 10)

Number

1. Computer Security: Backup and Recovery Methods
2. Here Comes Remote Batch
3. The Debate on Data Base Management
4. Intelligent Terminals
5. COBOL Aid Packages
6. On-Line Development of COBOL Programs
7. Modular COBOL Programming
8. New Training in System Analysis/Design
9. Savings from Performance Monitoring
10. That Maintenance "Iceberg"
11. The "Data Administrator" Function
12. The Mini-Computer's Quiet Revolution

1973 (Volume 11)

Number

1. The Emerging Computer Networks
2. Distributed Intelligence in Data Communications
3. Developments in Data Transmission
4. Computer Progress in Japan
5. A Structure for EDP Projects
6. The Cautious Path to a Data Base
7. Long Term Data Retention
8. In Your Future: Distributed Systems?
9. Computer Fraud and Embezzlement
10. The Psychology of Mixed Installations
11. The Effects of Charge-Back Policies
12. Protecting Valuable Data—Part 1

1974 (Volume 12)

Number

1. Protecting Valuable Data—Part 2
2. The Current Status of Data Management
3. Problem Areas in Data Management
4. Issues in Programming Management
5. The Search for Software Reliability
6. The Advent of Structured Programming
7. Charging for Computer Services
8. Structures for Future Systems
9. The Upgrading of Computer Operators
10. What's Happening with CODASYL-type DBMS?
11. The Data Dictionary/Directory Function
12. Improve the System Building Process

1975 (Volume 13)

Number

1. Progress Toward International Data Networks
2. Soon: Public Packet Switched Networks
3. The Internal Auditor and the Computer
4. Improvements in Man/Machine Interfacing
5. "Are We Doing the Right Things?"
6. "Are We Doing Things Right?"
7. "Do We Have the Right Resources?"
8. The Benefits of Standard Practices
9. Progress Toward Easier Programming

(List of subjects prior to 1972 sent upon request)

PRICE SCHEDULE

The annual subscription price for EDP ANALYZER is $48. The two year price is $88 and the three year price is $120; postpaid surface delivery to the U.S., Canada, and Mexico. (Optional air mail delivery to Canada and Mexico available at extra cost.)

Subscriptions to other countries are: One year $60, two years $112, and three years $156. These prices include AIR MAIL postage. All prices in U.S. dollars.

Attractive binders for holding 12 issues of EDP ANALYZER are available at $4.75. Californians please add 29¢ sales tax.

Because of the continuing demand for back issues, all previous reports are available. Price: $6 each (for U.S., Canada, and Mexico), and $7 elsewhere; includes air mail postage.

Reduced rates are in effect for multiple subscriptions and for multiple copies of back issues. Please write for rates.

Subscription agency orders limited to single copy, one-, two-, and three-year subscriptions only.

Send your order and check to:
EDP ANALYZER
Subscription Office
925 Anza Avenue
Vista, California 92083
Phone: (714) 724-3233

Send editorial correspondence to:
EDP ANALYZER
Editorial Office
925 Anza Avenue
Vista, California 92083
Phone: (714) 724-5900

Name_____

Company_____

Address_____

City, State, ZIP Code_____