GAWLIK, H. J. (1963). "MIRFAC: A Compiler based on Standard Notation and Plain English," *Comm. ACM*, Vol. 6, pp. 545–7.
GREMS, M., and POST, M. O. (1959). "A Symbol Coder for Automatic Documenting," *Comp. News*, Vol. 147, pp. 9–18, and Vol. 148, pp. 15–19.
HILDEBRAND, F. B. (1956). *Introduction to Numerical Analysis*, New York: McGraw-Hill Book Co.
KLERER, M., and MAY, J. (1963). "Algorithms for Analysis and Translation of a Special Set of Computable Mathematical Forms," Columbia U., Hudson Labs. Tech. Rept. No. 113.
KLERER, M., and MAY, J. (1964). "An Experiment in a User Oriented Computer System," *Comm. ACM*, Vol. 7, pp. 290–4.
LOS ALAMOS SCIENTIFIC LABORATORIES (1958). "Maniac II," *Comm. ACM*, Vol. 1, p. 26.
VANDERBURGH, A. (1958). "The Lincoln Keyboard—a Typewriter Keyboard Designed for Computers Input Flexibility," *Comm. ACM*, Vol. 1, p. 4.
WELLS, MARK B. (1961). "MADCAP: A Scientific Compiler for a Displayed Formula Textbook Language," *Comm. ACM*, Vol. 4, pp. 31–36.
WELLS, MARK B. (1963). "Recent Improvements in MADCAP," *Comm. ACM*, Vol. 6, pp. 674–8.

A special-purpose compiler

By C. B. Jones*

This paper describes how a particular computer application was tackled using a "Compiler" approach, and how the work is to be extended.

Computer applications give rise to many differing forms of files. These files are usually organized so that the basic record contains details of options within the item type. Thus in the case of an order file, a customer order record contains an item number and a set of options within that item. This paper will consider motor vehicle order files in which the *options coding* is both complicated and extensive: a complete vehicle, with its options, can be coded on to an 80-column card which will be referred to as a V.O.C. (*vehicle order card*).

On the V.O.C. each vehicle option is allocated a field (frequently only one column). The value in the field indicates which of a number of possibilities is required for the particular vehicle. The size limitation of an 80-column card necessitates that a number of conditions cannot be coded directly, and are, therefore, implied. Thus, in some cases, it is necessary to examine the contents of several fields before an option is found to be required.

The requirements for a particular vehicle are coded into the V.O.C. manually. The information contained on the cards is interpreted by a number of computer programs for a variety of reasons, e.g. data vetting, parts scheduling and model costing. One of the programs is a translation program which will convert the format and coding of these cards.

The translation program

The existence of more than one format for the V.O.C. posed a communication problem. These formats differed, not only in the position of the fields, but also in the way in which the basic information was coded. Information for a field in one format may have been inferred from a number of fields in another format.

Example: the following trivial translation problem will be used as an example throughout the paper.

Base coding Col. 21 is type of vehicle:

Codes
1 Car A
2 Estate A
3 Car B
4 Estate B

Col. 22 is number of doors:

Codes
2 Two doors
4 Four doors

Required coding Col.31 is body type:

Codes
1 Car
2 Estate

Col. 32 is car type with number of doors:

Codes
1 Car A, two doors
2 Car B, two doors
3 Car A, four doors
4 Car B, four doors.

The complete translation problem from the base to the required coding could be specified in a number of statements, which assign a value to a field of the new card subject to a set of tests on certain fields in the old card. Thus, a typical statement would be "put a 1 in column 31

* *Ford Motor Company Ltd., Central Office, Warley, Brentwood, Essex.*

of the output card, if there is a 1 in 21 or a 3 in 21 of the input card". We shall write this statement more concisely, i.e. "1, 31 IF 1, 21 or 3, 21". The statements which describe the above example are therefore:

1, 31 IF 1, 21 OR 3, 21
2, 31 IF 2, 21 OR 4, 21
1, 32 IF (1, 21 AND 2, 22) OR (2, 21 AND 2, 22)
2, 32 IF (3, 21 AND 2, 22) OR (4, 21 AND 2, 22)
3, 32 IF (1, 21 AND 4, 22) OR (2, 21 AND 4, 22)
4, 32 IF (3, 21 AND 4, 22) OR (4, 21 AND 4, 22)

It would have been possible to write a specific program to perform such a translation. However, it is clear that it would have taken a considerable time to write and test. The main objection was the problem of maintenance. The program would have undergone considerable modification when the basic formats changed. Each modification would need to be implemented quickly and accurately, as well as being fully documented.

Clearly, a generalized program was required, which could tackle any translation job of this kind, given some formal description of the problem. Similar problems had previously been tackled by an "interpretive" approach. Briefly, the technique is as follows. The statements which describe the translation would be held in a compact form as a table by the program. This table would be used by a master routine to set up and perform tests for various options. The table would also contain information as to what output was required if the tests found the conditions to be present. Whilst this technique is simple to program for a computer, it tends to be extremely slow, although any general program of this sort would solve the maintenance and documentation problems. It is also possible to tackle the problem using a compiler. The compiler would use the same basic information as the interpretive approach, i.e. the table of statements. Instead of using the table in this form, however, it would assemble actual machine-language instructions to perform the testing and field-set-up parts of each statement. Thus, the table would represent a computer program, written in a special language, which would be compiled into a machine-usable form. These instructions would be linked together as a complete program, so that no time is wasted in a control routine. The program which is compiled would then be completely self-contained and could be executed to perform the specified translation task in a separate run.

The machine to be used for the translation program was an IBM 1401 tape computer with a storage capacity of 16,000 digits. This is a variable word-length, decimal machine, and is ideal for this sort of task. The field testing was performed by a *compare* instruction, unless the field was only one digit in length, in which case the *branch if character equal* instruction was used. This instruction stores within itself the character for which it is testing, as well as the address of the position to be tested and the address of the next instruction if the test is successful. It thus provides a very compact way of sorting a test on a one-digit field.

Differences between methods

It is important to realize that the difference between the compiler approach and the interpretive approach is more than one of degree. In the interpretive approach the problem statements are stored in the memory of the computer as a table. As each V.O.C. is read, every part of every statement has to be examined and actioned. If the part under consideration is a test on a field, the test instruction has to be set up and performed; the result of the test (true or false) is then made known to the control routine and the instruction is destroyed. It is, therefore, necessary to test each item of the table and convert it into machine-code form for every V.O.C.

The compiler, on the other hand, works in a two-phase mode. In the first phase each item of the table is examined and its meaning determined, and appropriate machine instructions are formed into a self-linking program. Thus, the output from the compiler is a completely self-contained program tape which will perform the translation problem specified by the user. The machine instructions generated to perform a particular test will take more storage space than the concise representation of the test statement used in the table. With the size of problem we were considering, even the table approach required that the V.O.C.s were passed more than once (on work tapes). As is shown below, the input/output times required for the extra passes of the V.O.C.s proved to be insignificant.

It was also realized that the input language to the compiler could be more sophisticated, allowing a smaller and more efficient representation of the problem. This was possible because the instructions required to recognize and translate the extra features, were to be used only when compiling. In the interpretive approach these instructions would have been included in the table-decoding loop which was to be obeyed many times. This would mean that even with the more compact representation of the problem, the overall running times would have increased.

Syntax of the language

This description will refer to the statements which describe the task in two parts, the *test section* and the *object clause*. The test section will be the part of the statement which designates the required conditions. The object clause will describe the output required, if the tests are found to be true. A fixed format was used for ease of data reading and punching, the examples, however, will have the fields separated by , or .

Basic tests are those on single columns, which are written as the character to be tested for, followed by the number of the column in which the test is to be performed.

e.g. 1, 21 specifies a test for the character 1 in card, column 21.

Immediately adjacent conditions are taken to be in an *AND* relationship. Sets of conditions which are to be in an *OR* relationship are separated by an asterisk, which was less logically binding than the *AND* condition, e.g. our previous problem is expressed as follows:

1, 31 – 1, 21 * 3, 21.

2, 31 – 2, 21 * 4, 21.

1, 32 – 1, 21 . 2, 22 * 2, 21 . 2, 22 etc.

Tests on fields containing more than one character were specified as the string of characters to be tested for, enclosed by a special symbol @, followed by the position of the field. Thus the test section of line three of the above could be written:

@ 1 2 @, 22 * @ 22 @, 22.

The absence of a condition could be tested for, by writing *NOT* before the condition. This only affected the adjacent test.

It was possible to establish a logical flow within the program by an instruction which allowed the next statement to be other than the next sequential statement. This branch instruction referred to labels which could be interspersed with the other statements. Logical switches were incorporated to reduce the amount of work and to simplify parenthetical operations. This feature allowed a number of tests to be defined as a logical switch. Dependent on the result of these tests, the value of the switch was either true or false. When the switch was subsequently tested in another statement, one was effectively incorporating the original tests. An error object clause was included to allow data vetting in the object program. It was possible to intermingle the test and object clauses in any way: thus, a combination of simple tests, field tests and switch tests might be used to set up another switch. A number of other statements permitted one to copy and/or gang-punch information into the output of the object program.

The compiler

As mentioned above, the first version of the program was written for the IBM 1401 and this led to a severe limitation, in that no description of the input/output media could be processed.

The compiler works internally as a two-phase system. The first phase creates a piece of program in core which will perform the job of a complete statement. The machine instructions, as formed, will have special three-character symbolic labels, to which other instructions can refer. The second phase consists of resolving the symbolic labels to actual three-character machine addresses. In fact, only one pass is made as the addresses are resolved before the program tape is written. An optional feature allowed the compiled program to be listed, although it was not envisaged that this would normally be used in production work.

Having read a complete statement, which may contain up to two hundred tests, the machine-language instructions are formed. Error checking is done during the compilation, although when errors are found, the principle adopted is that *something* should be compiled to allow the object program to be tested.

When the whole statement has been analyzed a check is made to ascertain if this record will complete a core load. If so, instructions must be included which will write the V.O.C. on to a work tape for the next phase of the object program. The instructions formed at the beginning of the next core loading will cause the input to be taken from the work tape. The last phase will punch the final output cards. Branches or switch conditions, which overlap core loadings, also have to be dealt with using the work files.

Timing

Although this program was first written to facilitate V.O.C. translation, its application to many other projects was quickly realized. One existing job, on the 1401, had the same basic code recognition problem and used the interpretive method. This will be used for a comparison of the timings. The application dealt with breaking a vehicle down to the *assemblies* required to build it, dependent on the options specified in the V.O.C.s. This job had an analysis table of about six hundred statements which, on average, consisted of twenty conditions. The processing time of the interpretive program was about 75 minutes for 500 V.O.C.s. The time taken for the compilation was 10 minutes, but recompilation is only necessary when the table is changed. The compiled object program took only 10 minutes to process the 500 V.O.C.s. This speed increase is found despite the fact that the compiler approach requires ten "core-fulls" of object program, whilst the concise table occupied only four core-fulls, thus more than doubling the tape-handling time for the work tape. As these figures represent the processing of only one vehicle range, the overall savings of the project were considerable. This improvement has been made with no recoding effort for the table, which only uses the AND and OR features. Use of other features of the language would reduce running time still further.

Future work

It has been found that a large number of existing programs contain code recognition which could be described in the compiler language. However, many of them handle different records from a V.O.C. In particular most data-vetting problems fall into this category. Unfortunately, it is not possible to use the current compiler to tackle them. Having been written for an IBM 1401, the program is not able to include more than one input/output processing option, because of storage limitations. It is, therefore, necessary to maintain a version of the program to tackle each of the jobs mentioned above.

A larger and faster IBM 7010 with 80,000 positions of storage will shortly be available, on which it is hoped

to use a similar program to advantage. The implementation of the compiler on the 7010 would reduce the initial programming load and the maintenance problem, as well as leading to more efficient programs.

The above limitation will be tackled in two ways. The direct approach will allow input/output media to be described in formal statements, from which the compiler will be able to assemble data-handling instructions. Further generality would be provided by exploiting the features of OPSYS. OPSYS is the operating system software for the 7010, which includes FORTRAN and COBOL compilers and an Autocoder assembler. These are so written as to allow the sub-program concept to be used with various parts of the same program, having been written in different languages. Thus, the code recognition compiler could be put on the systems tape and use of *exits* would allow any further processing to be written in any language.

It is interesting to consider whether any changes should be made to utilize the different hardware features of the 7010. Although the calculation versus input/output speed ratio is higher, this must be considered against the increased storage. It is found that the calculation volume (as dictated by the amount of program which will fit in core) bears a similar relationship to the tape-pass time as that on the 1401, which indicates that the extra core used by the compiler is still justified. The 7010 has a powerful table look-up instruction which will be used when a statement consists of tests for many possibilities in the same field. Thus, in this circumstance, the compiler will generate one table look-up instruction, unpacking instructions and a series of fields. The extra tape units available should allow several tables to be run simultaneously using partial results of one another's tests, and producing their output on different tapes.

Conclusion

The writing of a special-purpose compiler for the 1401 has allowed a number of problems, stated in a problem-orientated fashion, to be translated to machine language without programmer intervention. This in turn has solved documentation and maintenance problems. It is hoped that a similar system, implemented on a large machine, would have even wider application.

Book Review

Language H (1962) *Manual*, 1964; 111 pages. (London: *The National Cash Register Co. Ltd.*, 42s.)

The attempt to float Basic English as a world language in the late forties had no chance of success, not because it was difficult for the non-English speaker to learn, but because it was virtually impossible for the English speaker to remember which of his ordinary words and turns of phrase constituted the sub-set, Basic English, and which did not. Within the thicket of his vocabulary, the boundary posts of Basic English were too undifferentiated to discern.

There is a lesson here for inventors of commercial computer languages. They assume that the language is easier to use if their executive functions are words in normal English usage. They encourage the use of English operands, and they supply linking words which serve no other purpose than to provide a spurious semblance of English. They have two aims. They want the layman to be able to read a computer program. But why should he? It is none of his business. They claim that a system flow-diagram can be written in their language, and lo!—programming is already accomplished. One such language has developed such an extensive vocabulary that the programmer is left groping for his boundaries, the programmer who would perhaps be happier if functions were given in Latin or Swahili.

Of course we do need computer languages which are not creatures of a particular machine or machine group. We need more of the simple basic ones, which will, it is to be hoped, drive out the prolix. This is why Language H should be welcome. It does, indeed, use English words rather than Erse. It employs fourteen words which are redundant or behave as punctuation "but help to amplify the meaning of the statement". Presumably these are included to appeal to this importunate layman. The considerations used in developing the language, however, give a clue to its character. To quote from the Preface:

> "the smallest possible number of effective phrases should be provided"
>
> "the strain on the memory of the user should be kept as low as possible"
>
> "The wealth of possible expressions, such as compound conditional phrases, has been deliberately reduced in order to make learning and accurate use easier"

Nobody is likely to quarrel with those aims.

Language H is expected to behave as a high-level language, but there is none of the tiresome indenting which goes with compound conditions. The program example given, if it were not for the linking words, smacks of a symbolic order code in three-address mode, using literals and interspersed with macro-instructions. Since the language is not machine-orientated, the resemblance is mainly on the surface, but the simple structure should enable relatively simple and efficient compilers to be developed for machines outside the National–Elliott range.

On the question of macro-instructions, input and output are adequately explained as GET and FILE, but I could find no reference in the manual to hierarchial sorting procedures, although SORT is used as a reserved word. Indeed, less than half of the reserved words appear in the index, and this is inconvenient.

The main text of the manual runs to thirty pages, the rest being taken up by thirteen appendices, some referring solely to the machines which already have compilers. A handy little pocket book is included as a programmer's *aide-mémoire*.

F. I. Musk