Some Techniques Used in the ALCOR ILLINOIS 7090

D. Gries, M. Paul and H. R. Wiehle
Rechenzentrum der Technischen Hochschule, München and University of Illinois, Urbana, Illinois†*

An ALGOL compiler has been written by the ALCOR group for the IBM 7090. Some little known but significant techniques in compiler writing, together with organizational details of this compiler, are described. Timing estimates and an indication of compiler requirements are also given.

1. Introduction

1.1. *The project.* ALCOR ILLINOIS is an Algol compiler written for the IBM 7090. Work on the compiler was begun at the Digital Computer Laboratory, University of Illinois, in June 1962 as a joint project by this University, the Mainz Institute for Applied Mathematics, and the Rechenzentrum, Munich Institute of Technology. The directors of these institutes felt a responsibility to make Algol available as a practical language to their students and to a wider public and, therefore, initiated the task since, at that time, the manufacturer was not going to provide an Algol compiler for the 7090. In January 1963 the work was partly transferred to the Rechenzentrum, Munich Institute of Technology, and by May 1963 the main parts of the compiler had been programmed and checked. The field test phase was entered in November 1963, and in July 1964 the compiler was given a final release.

The language of the ALCOR ILLINOIS 7090 is Algol with the exception of **own**. Input and output are temporarily controlled by standard input and output procedures under a Fortran-like format, described in [5]. As an introduction to Algol (with the exception of input-output) and as a manual, [11] can be used.

1.2. *Historical background.* The ALCOR (meaning ALgol COnverteR) group, formed in 1958, is a cooperative group of computing centers and computing manufacturers, interested in automatic translating. From its beginning, it has based its language decoding (text analysis) on a sequential translation technique using the well-known cellar principle and state transition matrix developed by Bauer and Samelson [4], and has based its run-time organization on principles of dynamic storage allocation and subroutine

* This project was partly supported by the Deutsche Forschungsgemeinschaft.

† Department of Computer Science. The 7090/1401 System at the University of Illinois used in this development was partially supported by the National Science Foundation under grant No. NSF-GP-700.

linkage introduced mainly by Rutishauser [10]. This group has built over 10 ALGOL compilers, the ALCOR ILLINOIS 7090 being one of the latest and most modern ones. This group is continually doing research in improving and developing techniques for formal language translation and in the development of programming languages.

1.3. *Aims.* In contrast to the other compilers built by the ALCOR group, this was to be the first one built for a very fast machine with a big and homogeneous storage. The authors could therefore concentrate on refinements of the translation techniques used. Restrictions on the language earlier imposed by slow, small machines were no longer necessary, and methods were developed to implement all features. There was no limitation on the size of the compiler itself. Other problems, however, presented themselves, one being the necessity for a compiler which could easily be imbedded in the numerous 7090 operating systems.

Basically, the successful techniques developed by the ALCOR group, which emphasize a fast object program, a good syntax checker and a short translation time, are used.

1.4. *Organization of the compiler.* Text analysis is preferably performed using the cellar principle and a state-transition automata with a one-sided tape (the cellar). A complete syntax check can be accomplished through the use of a complete transition matrix, including the necessary means for synchronization in case of syntactical errors. On the other hand, a system of priorities of the states and incoming symbols [2, 9] is simpler and more lucid. However, it requires a correct program. The solution is as follows:

Pass 1 performs the usual mapping of the source program symbols into an internal representation and generates a few lists needed by the syntax checker.

Pass 2, using a complete transition matrix, performs a complete syntax check without actually translating. However, it creates lists of all necessary information needed for the actual translation, and transfers control to the next pass only if the source program is correct.

Pass 3, the actual translator, working only on correct programs and using the lists already produced, can use the priority method and devote much space and effort to produce efficient object programs.

Pass 4 does final address calculation and relocation.

1.5. *Use of storage space.* One of the ways in which translation time can be held down is to keep tape movement to a minimum. Since the source program must be read, processed, and modified by each pass, it should be kept in memory as much as possible. With program buffers of 1000 words (each word containing 5 ALGOL symbols), a program of 150 to 200 cards needs no intermediary tape; while compiling a program of 500 cards, at most 5 or 6 records are written and read.

One problem with a compiler is the number of lists to be stored. In order to minimize translation time, these lists must be kept in memory, even if this limits the size of programs which can be translated. This is no real restriction since a program can be split up into code procedures. With an available storage of 32K words, the ALCOR ILLINOIS 7090 limits the number of different identifiers within an ALGOL program to 3000, **for** loops to 400, blocks to 500, and the number of constants to 700. These limits are large enough to accomodate most programs and small enough so that no list (except the ALGOL program itself) has to be put on tape. Many lists become obsolete from pass to pass and the space is then used for other purposes. Due to this, the final object program buffer has 4000 words. The compiler itself has a total of 26,000 instructions and uses $70{,}000_8$ locations in memory.

1.6. *Remarks.* Since the main compiling techniques of the ALCOR group have become well known and widely used, this discussion is limited to a few significant, but little known, points. Questions of run time organization are mentioned in connection with Pass 3 (the object program generator), the most important being the use of linear address incrementing, whenever possible, in **for** loops.

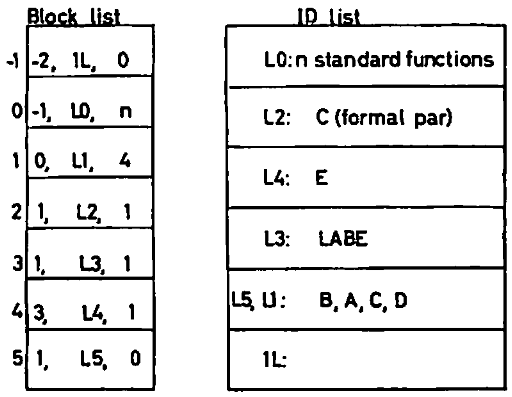
2. Pass 1

The first pass has two main objectives. The first is to change all ALGOL symbols into a 7-bit representation while discarding blanks and comments (except in strings). The other is to produce a list, called the ID-list, of all formal parameters, labels and variables. This list is necessary for a good, fast syntax check. It must be in a form which is easy to search, since (1) every identifier can occur before it is declared, and (2) for each occurrence of an identifier the syntax checker must find it in the ID-list and check it. At the end of pass 2 this list will contain all information about each identifier (e.g., type, kind, number of subscripts, formal parameter or not). In order to prevent duplication of effort between pass 1 and pass 2, pass 1 inserts into the ID-list only the name and, implicitly, the block number. This has the additional advantage that the pass 1 state-transition matrix is very small, since only block structure and declarations must be analyzed. For instance, expressions may be entirely skipped; no states are necessary for "+", "*", and so on.

Experience has shown that a very efficient way to organize the ID-list is by "block number," where a block has block number n if it is the nth block to be statically opened. For reasons explained later, each **for** loop and each procedure is also technically counted as a block (with the formal parameters being the only identifiers belonging to the artificially created procedure block). Two lists are actually produced. The first, called the block list, contains in the ith location (corresponding to block i) the triple (surrounding block number, address of ID-list for block i, number of identifiers in block i).

In addition, blocks 0 and -1 exist and surround all blocks. Block 0 contains all standard procedures, with complete information about them. The compiler can be changed easily to add or delete standard procedures by changing this list in pass 1. Block -1 is empty and will be used by pass 2 for a list of all undeclared identifiers, to aid in continuing after errors. The second list, the ID-list itself, contains for each block the identifiers and labels declared in that block. Figure 1 illustrates the formation of the lists for the following program.

```
begin integer procedure B(C);
    integer C;  B := C*A;
  integer A, C, D;
  A := C := 2;
  for D := 1 step 1 until C do LABE:
    begin integer E;
      E := B(D) + A;  A := E;
    end;
  for D := 1 step 1 until C do print (D, A*D);
end
```



	Block list
-1	-2, 1L, 0
0	-1, L0, n
1	0, L1, 4
2	1, L2, 1
3	1, L3, 1
4	3, L4, 1
5	1, L5, 0

ID list	
L0:	n standard functions
L2:	C (formal par)
L4:	E
L3:	LABE
L5, L1:	B, A, C, D
1L:	

FIG. 1

Notice that the identifier list is not ordered by block number, but by the order in which the blocks are closed. This makes no difference to pass 2 and is easier to generate.

3. Pass 2

Pass 2 performs a complete syntax check. The method (state transition matrix with one stack as in pass 1) is not discussed here. Practical and theoretical explanations can be found in [1] and [4].

There are some jobs which pass 2 can easily do in order to simplify pass 3 and help optimize the object program.

3.1. *ID and block lists.* One important job is to replace each identifier in the program by the entry number of the identifier in the ID-list. (The nth identifier in the list has entry number n). Pass 3 uses this number directly in order to find an identifier in the list; no search of the list is required. The identifier list can then be shortened at the end of pass 2 by deleting the name of each identifier, leaving more room in memory for other lists (especially the object program to be produced). Another operation easily performed is the calculation of addresses of variables for the object program. Since all information for each identifier will then be in the ID-list, pass 2 deletes all specifications and all declarations except procedures, switch and array declarations from the source program.

Returning to the discussion of the ID-list, one sees that its form allows efficient searching. The subroutine LUIL (Look Up in ID-List), given in Figure 2 with the identifier and block number B as parameters, performs this search. This one subroutine can be used for all occurrences of identifiers except in declarations and specifications. It also automatically tests for jumps into a **for** loop from outside. Such a use of a label causes it to be undefined since a **for** loop is technically treated as a block. Declarations and specifications can be handled by the addition of another entry point, LUIL 1, which searches only in block number B, due to the treatment of procedure declarations as blocks. The only other problem is the checking of variables occurring in expressions, E_i, of an array declaration

$$\textbf{array}\ A[E_1:E_2,\ \cdots,\ E_{2n-1}:E_{2n}].$$

These variables must be declared in surrounding blocks. This check is accomplished by the following:

(a) When [of an array declaration is pushed into the cellar: (1) the current block number is saved in an auxiliary location; (2) the surrounding block number is put into the current block number location.

(b) At the end of the array declaration ( ] encounters the [in cellar): (1) the current block number is restored in the current block number location.

The need for counters and pointers to indicate which variables have been used in which blocks as array limits has been eliminated.

Another consequence of the block structured ID-list is the simplicity of the "block entry" and "block close" subroutines. They consist merely of pushing the actual block number into the cellar and inserting the new one, and restoring the old block number respectively. This method is simpler than those used by some other authors [3].

3.2. *Linear address incrementing.* Linear address incrementing is a term used for a special type of optimization of array element address calculation in **for** loops.

Let a loop be

$$\textbf{for}\ I := E1\ \textbf{step}\ E2\ \textbf{until}\ E3\ \textbf{do}\ \Sigma;$$

and $A[E]$ be an array element occurring in the statement Σ. If E is linear in the loop variable I, then $A[E]$ can be

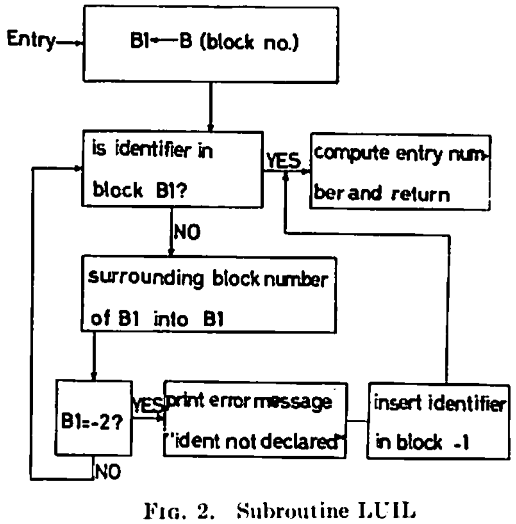

Entry
B1←B (block no.)
is identifier in block B1?
YES
compute entry number and return
NO
surrounding block number of B1 into B1
B1=-2?
YES
print error message "ident not declared"
insert identifier in block -1
NO

FIG. 2. Subroutine LUIL

relocatable since the storage space is dynamically allocated. If a procedure call takes place in block b, $\langle DFSL_b \rangle$ are given as a parameter to the procedure. The procedure, having m locations in its *FFS*, takes locations $\langle DFSL_b \rangle$ to $\langle DFSL_b \rangle + m - 1$ for its *FFS*, and inserts the value $\langle DFSL_b \rangle + m$ in its own *DFSL* location. Every variable and location in the *FFS* of a hierarchy is then defined by the pair (A, STR), where A is its absolute position in the *FFS*, and *STR* (stack reference) is the relocatable beginning of the *FFS*. Upon entry to a procedure from block b, *STR* is clearly equal to $\langle DFSL_b \rangle$.

It is obvious that all procedures can be made recursive if every piece of information which depends upon the call is saved in the *FFS*. Each time the procedure is entered, space is again reserved for the *FFS* of the procedure.

We do not wish to go into details of just which information must be saved. In order to indicate how to make this as efficient as possible, we give a few practical techniques. When in a certain procedure, an index register can be used to contain its stack reference. The relocation is performed once and for all at the beginning of the procedure by loading the index register with the stack reference, and using this index register as a tag to each instruction referencing the *FFS*. Hierarchy 0—the main program—cannot be recursive and must always be in memory. It is therefore given absolute locations for its *FFS*. A detailed description can be found in [7].

5. Pass 4

This pass does final relocation and puts the program onto a tape in the equivalent form of relocatable binary cards. It also does final address calculation, in order to handle forward references to labels, switches and procedures. Error messages are also printed by this pass, from a list containing all necessary information (card number, error number, state of the matrix when the error occurred, etc.) made up by pass 2.

6. Parameterization of the Compiler

The existence of numerous systems used on the IBM 7090 made it worthwhile to parameterize the ALCOR ILLINOIS 7090. The compiler has about 50 parameters, representing most connections between the compiler and a system. Examples are: tape numbers, locations to be used by the compiler, and positioning of the system tape after compilation. An installation fills in the list of parameters according to its needs [6], punches them on cards, and inserts them in the front of each pass. The passes are then assembled, producing absolute binary cards ready for use.

7. Concluding Remarks

As of this date, the ALCOR ILLINOIS 7090 has been distributed to over 10 installations and imbedded in 3 different systems. Wider use is expected. A confirmation of the efficiency of the techniques used is the speed of translation. A program of 20–50 cards compiles in less than 10 seconds (including the 2.7 seconds overhead to read the compiler from tape). One program of 2700 cards, produced by duplicating a single program many times and surrounding the result by a **begin** and **end**, took 75 seconds to compile. In the few tests made, object programs were 1.2 to 2.2 times slower than a corresponding FORTRAN program, the higher of the factors resulting from a program in which mostly integers were used, the lower in the calculation of the determinant of a 50×50 matrix. Barth, Institute of Technology Darmstadt, programmed and tested a mathematical algorithm in FORTRAN and ALGOL. The program was about 550 cards long. Results are listed in Figure 4.

	ALCOR ILLINOIS 7090		FORTRAN II	SHARE 7090 ALGOL
	A	B		
Compilation time plus loading time (sec)	37	37	260	330
Factor (using ALCOR ILLINOIS as 1)	1	1	7.0	8.9
execution time (sec)	320	420	190	630
Factor	1	1.3	.59	1.9
total time (sec)	357	457	450	960
Factor	1	1.4	1.4	2.7

A with linear address incrementing
B without linear address incrementing

FIG. 4

RECEIVED FEBRUARY, 1965

REFERENCES

1. EICKEL, J., PAUL, M., BAUER. F. L., AND SAMELSON, K. A syntax controlled generator of formal language processors. *Comm. ACM 6* (Aug. 1963), 451–455.
2. FLOYD, R. W. Syntactic analysis and operator precedence. *J. ACM 10* (July 1963), 316–333.
3. LIETZKE, M. P. A method of syntax-checking ALGOL 60. *Comm. ACM. 7* (Aug. 1964), 475–478.
4. SAMELSON, K., AND BAUER, F. L. Sequential formula translation. *Comm. ACM 3* (Feb. 1960), 76–83.
5. BAYER, R., MURPHREE JR., E., GRIES, D. User's manual for the ALCOR ILLINOIS 7090 ALGOL-60 translator, 2nd ed. U. of Illinois, Sept. 1964.
6. GRIES, D., PAUL, M., AND WIEHLE, H. R. ALCOR ILLINOIS 7090—An ALGOL compiler for the IBM 7090. Rep. no. 6415, Rechenzentrum der Tech. Hochsch., München, 1964.
7. ——. The object program produced by the ALCOR ILLINOIS 7090 compiler. Rep. no. 6412, Rechenzentrum der Tech. Hochsch., München, 1964.
8. HILL, U., LANGMAACK, H., SCHWARZ, H. R., AND SEEGMÜLLER, G. Efficient handling of subscripted variables in ALGOL 60 compilers. *Proc. 1962 Rome Symposium on Symbolic Languages in Data Processing*, Gordon and Breach, New York, 1962, 311–340.
9. PAUL, M. A general processor for certain formal languages. *Proc. 1962 Rome Symposium on Symbolic Languages in Data Processing*, Gordon and Breach, New York, 1962, 65–74.
10. WALDBURGER, H. Gebrauchsanleitung für die ERMETH. Institut für angew. Mathematik der ETH, Zürich, 1959.
11. BAUMANN, R., FELICIANO, M., BAUER, F. L., SAMELSON, K. *Introduction to ALGOL*. Prentice Hall, Englewood Cliffs, N.J., 1964.

written as $A[aI+b]$ where a and b are (integer) expressions which do not change in the loop, and

$$\text{address} \quad (A[a(I+E2)+b]) = \text{address} \quad (A[a(I)+b]) + a*E2.$$

Upon entering the loop, address $(A[a*E1+b])$ and the value $a*E2$ are calculated and stored. Then each time the loop is repeated, $a*E2$ is added to the address.

Due to a questionable generality in ALGOL 60, the program must be checked thoroughly to make sure that I and $E2$ do not change during execution of the loop and to determine just which array elements are linear in the loop variable. Further time and storage space in the object program can be saved by making identity checks of array elements (e.g., with respect to equal initial addresses and address increments) in pass 2. More information is contained in [4, 8]. For an exact description of its use in the ALCOR ILLINOIS 7090, see [5, 7].

In order that pass 3 can perform this optimization in the object program, pass 2 must produce a list of **for** loops in which linear address incrementing is used and must test each array element for linearity in the loop variable. This can complicate pass 2 if incorporated directly into it, since, at each step of analyzation of an expression, a test must be made to see if it is in a subscript expression, and if so, a jump to a subroutine which tests for linearity must be executed.

It would therefore seem logical to have an extra pass (linearity checker) to check for linear address incrementing, but this would increase the translation time, since the source program would be read and processed again.

The solution to the problem is to run both parts—the syntax checker and the linearity checker—*separately, but simultaneously*. Each one has its own set of subroutines and can be programmed separately, but they use common lists and locations. The syntax checker processes a source program symbol and passes it on to the linearity checker. When it is finished with one symbol it reads the next symbol and jumps to the syntax checker. This technique could be used to run any number of separate passes which can run parallel. What is saved is the bookkeeping which must accompany each pass, and the corresponding time involved.

Figure 3 gives a simplified flowchart for running three passes together.

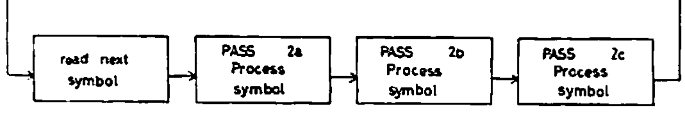


FIG. 3

In order to save time, the linearity checker is in action only when necessary. This requires connections between the two parts at the following points:

(1) **for** is the symbol just read
(2) $a[$ (beginning of an array element)
(3) $a($ (procedure or function call)
(4) $a :=$ (where a is of type integer)
(5) end of a **for** loop

4. Pass 3

Pass 3, operating only with correct programs, uses a list of priorities assigned to the ALGOL symbols, together with 2 cellars. This method, described in [2, 9], is not discussed here. The optimization through the use of linear address incrementing also needs no further discussion, although it is the most effective type of optimization [4, 5, 7, 8]. Instead, we discuss the use of dynamic storage allocation during execution and its relation to recursive procedures.

4.1. *Storage for blocks and arrays.* In ALGOL, arrays must be handled dynamically since the bounds of an array may only be determined at run time. Therefore, the program must have a mechanism for reserving and releasing storage. The usual method is to think of free storage as a stack. There exists one location, *DFSL* (Dynamic Free Storage Location), which always contains the address of the first (lowest) free location. When a new block is entered, the block saves the contents of *DFSL* in an auxiliary location. An array declaration needing n locations causes locations $\langle DFSL\rangle$ through $\langle DFSL\rangle + n - 1$ to be reserved for it and $\langle DFSL\rangle$ to be changed to $\langle DFSL\rangle + n$. ($\langle A\rangle$ means contents of location A). The problem is that at every exit from the block, the old value of *DFSL* which was saved upon entry must be restored in order to save storage space, since the arrays declared in the block are no longer needed. This causes complications in the compiler and in the object program, especially if an exit leaves more than one block at a time.

A simple device, introduced by K. Samelson, eliminates the need for any reinitialization when leaving a block. Define the outermost block to be block number 0. An inner block has block number n if the immediately surrounding block has block number $n-1$. The block number is then the level of nesting within other blocks. To each block number b there exists a location $DFSL_b$.

Upon entering a block with number b, the instructions $\langle DFSL_{b-1}\rangle \rightarrow DFSL_b$ are executed. An array declaration declared in block b, needing n locations, causes the locations $\langle DFSL_b\rangle$ to $\langle DFSL_b\rangle + n - 1$ to be reserved, and $\langle DFSL_b\rangle$ to be increased by n. It is obvious that no extra work is required when leaving a block, since each block with number b uses only $DFSL_b$ to indicate storage reserved for it, and not a universal *DFSL*.

4.2. *Procedures.* Let the main program be called hierarchy 0, and each procedure hierarchy $n+1$, if it is declared in a hierarchy of order n. The hierarchy number is then the level of nesting of procedures. Each procedure has a fixed number of locations (called *FFS*, for free fixed storage) associated with it—simple variables declared in it, formal parameters, auxiliary locations, etc.

In order to save space, it is desirable that locations for an *FFS* be reserved only when the procedure is being executed. This means, however, that the variables be easily