INFORMATION PROCESSING 68 - NORTH-HOLLAND PUBLISHING COMPANY - AMSTERDAM (1969)

THE ACME COMPILER

GARY Y. BREITBARD and GIO WIEDERHOLD
Stanford Computation Center. Stanford University, Stanford, California, USA

Described in this paper is an implementation of a true incremental compiler designed and executed at the Stanford Computation Center Real-Time Facility, located at the Stanford University Medical School. The compiler translates a powerful subset of PL/1 into machine language for use in a time-sharing system. The problems of efficiency versus flexibility are discussed, and some examples of the techniques used to cope with problems presented by the language and the environment are detailed.

1. INTRODUCTION

The ACME compiler is a true incremental compiler operating in a system providing real-time and time-sharing services in the Stanford Medical School and some other laboratories of Stanford University [1]. It provides the principal contact between the user and the computer in an integrated, one-language system in which both commands to the system and program statements are handled at the same level [2].

There has been much discussion concerning the desirability of incremental compiling [3, 4]. We were faced with a situation where the usual alternatives - interpretive systems or remote access to batch processing - were not adequate.

The system had to be powerful enough to support statistical calculations, data processing, and manuscript editing; and flexible enough to provide experiment control in a research environment.

A researcher needs to get his data from his instruments to the computer, have control over parameters and identifying information, have access to previous results for comparative evaluation, and have means of presenting the information to allow him to make decisions regarding the progress of his work. In conventional computing the elapsed time for one iteration through this loop is on the order of a week. Our system and, in particular, our compiler were designed to reduce this time to minutes.

2. CHOICE OF LANGUAGE

There were several reasons for choosing to implement a one-language system. Since our users are obliged to do their own programming, learning to program must be relatively easy [5]. The volume of systems code in a one-language system is less so that the compiler can be core resident, which increases operating efficiency. It is an achievable goal in a project with limited resources.

We wanted to avoid contributing another home-made product to the language explosion; neither did we want to augment an existing language beyond its scope. Subsetting a defined language would afford several advantages, primarily in the areas of compatibility and communication.

PL/1 offered a very powerful selection of language facilities from which to choose. It is reasonably rich in data types, has no reserved words, and relatively few arbitrary restrictions [6]. We selected a subset of PL/1 which seemed appropriate to a time-sharing environment and did not present insurmountable problems of implementation.

Judged not essential for the users which we are serving were such features as recursion, decimal arithmetic, and automatic sterling conversion. For implementation reasons we omitted block structure, in relation to the scope of variables. We will now introduce our notion of incremental compilation and explain this important decision.

3. INCREMENTAL COMPILING-STATEMENT LINKAGE

Statements are accepted by us as they are typed in by the user, one line at a time, any number of statements per line. For purposes of the discussion, we will assume that each line consists of exactly one statement. Each line is pre-

ceded by a number, called a "line number", which orders this statement within the program. The line is edited, with backspaces removed and the line number extracted from the beginning of the line. The line number tells the system where to enter this line into the user's current program file, and how to link this line into the currently compiled program. It is this linkage that allows us to compile one line at a time.

The statement is then translated fully into machine language code (fig. 1). This code is placed directly into the user's area of core memory, from where it can later be executed if he so wishes. At the end of every statement an indirect branch instruction is compiled. The target address of this branch is the beginning address of the statement with the next higher line number. This linking takes place in two steps:

a) Before beginning compilation of a statement, the branch address of the statement with the

```
10.        IF c=d|flag THEN DO;
11.                         j=j+1;
12.                         END;
```

The above statements would result in the generation of the following code (in System/360 Assembly language):

```
@10.    LE      0,c       c=d      Load floating register 0 with c
        CE      0,d                Compare reg. 0 with d
        LE      0,=E'1'            Load reg. 0 with 1 (TRUE)
        BALR    15,0               Establish base register
        BE      L1                 Test on the above compare
        LE      0,=E'0'            Load reg. 0 with 0 (FALSE)
L1      LTER    0,0       OR flag  Set up test on result (c=d)
        BALR    15,0               Establish base register
        BNZ     L2                 Test result
        LE      0,flag             If result is FALSE, result=flag
L2      LTER    0,0       THEN     Set up test on result (c=d|flag)
        L       15,A1....          Load branch register
        BZR     15                 If result Is FALSE, skip THEN unit
        BALR    15,0      DO       Establish base register
        B       L3                 Go around DO termination
A3      L       8,A2....           Branch around END
        BR      8                    statement
L3      L       8,@10....          Branch to statement with
        BALR    14,8                 next higher line number

@11.    LE      0,j       j=j+l    Load reg. 0 with j
        AE      0,=E'1'            Increment j by 1
        STE     0,j                Store result in j
        L       8,@11              Branch to statement with
        BALR    14,8                 next higher line number

012.    TM      TIMLOC    END      Test for time expired
        BO      TIMES_UP           If so, terminate time slice
        L       8,A3....           Branch back to DO in case
        BR      8                    of loop
A1,A2   L       8,@12              Branch to statement with
        BALR    14,8                 next higher line number

data region:

@10.... DC      A(@11.)   Current address of next statement
@11.... DC      A(@12.)
@12.... DC      A(@13.)

A1...... DC     A(A1)     Addresses filled In at run time
A2.... DC       A(A2)
A3.... DC       A(A3)
```

Fig. 1. Example: Source language and compiled code.

next lower line number is changed to point to the current program address.

b) When compilation of the statement is completed, a branch to the statement with the next higher line number is compiled.

This insures that even though the statements are compiled sequentially into core as they occur, they are executed in order of their line numbers. Statements with the same line number result in the earlier statement being bypassed. By means of this linkage branching at the end of every line, lines can freely be changed, deleted, inserted, or appended, imitating the flexibility of an interpretive system (fig. 2).

```
  19. DO i=1 to n;
  20.   a(i)=b(j);
  21. END;

?RUN!
* 114: SUBSCRIPT OUT OF BOUNDS.
   AT LINE   20.000 IN PROCEDURE example
?put data(i,j)!
 i       =     1,j       =     0;
?19.5   j=j+1;
?run 19!
```

Fig. 2. Run time error and correction.

4. DIFFICULTIES IN INCREMENTAL COMPILING

A number of difficulties arise if full flexibility is desired. Among these difficulties are:

a) the definition of attributes and scope of variables,

b) references to undefined labels,

c) statements which range over more than one line.

5. SCOPE OF VARIABLES

Suppose that variables have scope, that is, a given identifier has a certain meaning only within a certain part (block) of the program. Then it is not adequate to maintain merely one symbol table - the current one. The ability to change or insert a line anywhere in the program requires the maintenance of symbol tables defining the meanings of all identifiers at each place in the program. Known symbol table techniques for languages with scope of variables do not solve this problem. We decided to leave the problem for the next writer of an incremental compiler and did not include block structure in our language.

6. ATTRIBUTES OF VARIABLES

Another difficulty in maintaining flexibility of language, and perhaps the chief obstacle to incremental compiling, is that the attributes of a variable appearing in a statement must be known at the time the statement is compiled. We would like to discuss the extent of this problem and some of its solutions.

The first and most obvious solution to the problem of unknown variable type is to assign attributes by default. In PL/ACME, a variable which appears for the first time in a place where it is neither explicitly nor contextually declared, is given type REAL BINARY FLOAT. Currently, all of our arithmetic is done in this mode, which is represented internally as single precision hexadecimal floating point in IBM's System/360. If the user later uses the variable differently, e.g., he declares it to be type CHARACTER, he is informed in an error message that this is an inconsistent use of the variable. If he insists on the change, he may, after making other "illegal" modifications to his program, call for a recompilation of his entire program. This is one of the few cases where recompilation is required.

The first time a variable is encountered, the distinction between single variable, subscripted variable, and function call is made from the context. When a variable is followed by an expression or list of expressions in parentheses, it is assumed to be either a subscripted variable reference or a function call. The compiler handles these two possibilities identically. In either case a machine language CALL is compiled. In the case of a function call, the CALL is to the subroutine, as expected. If the reference is to a subscripted variable, the CALL will be to the array descriptor, which is executable code. To complete the compatibility, it is necessary for function calls to return the address of a location where the result is stored, since subscripted variable references must always return an address.

The desire to defer definition of the type of a variable to the last possible moment is very strong in a system providing interactive computing capabilities. Carrying out t. desire fully would lead to the compilation of calls to interpreting subroutines for every operation. This practice would compromise one of the goals of the compiler, which is to produce code that is fast enough to handle processing of experimental data in real time. Interpretive systems exist for those who desire complete flexibility in data declarations.

7. LABELS

References to labels, such as the statement "GO TO alpha", are compiled using an indirect branch. That is, in IBM System/360, a register is loaded with the contents of alpha, and then a branch on this register is executed. When "alpha" appears as a statement label, it is given a value of the absolute address of the statement. If "alpha" never appears as a label, it retains its initial value - the absolute address of an error routine which gives a diagnostic message of "Undefined Label".

8. STATEMENTS WHICH RANGE OVER MORE THAN ONE LINE

A major problem for the compiler are statements which range over more than one line. Such statements are PROCEDURE, DO, END, and IF statements when the THEN or ELSE units appear on different lines from the IF clause. It is not required that the program be syntactically correct at all times. In fact, in a line-by-line system, that would be virtually impossible, since the first time a PROCEDURE statement is entered, the corresponding END statement would be missing. We go further in PL/ACME and do not require that statements be meaningful at the time they are entered. For example, an END statement can be inserted anywhere, with no corresponding DO or PROCEDURE statement yet entered; a THEN unit may be entered without the preceding IF clause.

The way the compiler handles these statements is to compile the statement completely into machine language, leaving one address constant unresolved. It is this address constant that will connect the END to its DO statement, that will allow a jump across the THEN unit, etc. The resolution of these address constants takes place when the user requests that his program be run (fig. 1). It is a simple matter, at this point, to check through pointers to the lines for proper syntax, and fill in the connecting addresses.

The binary code is not saved beyond the terminal session, but a new and clean copy is compiled from the file when the program is used later. Our experience has shown that direct compilation of non-relocatable code can be nearly as fast as the loading of relocatable code containing many cross-referencing pointers [8].

9. ALLOCATION OF CORE MEMORY

The compiler works in a time-sharing environment, and so, one of the first problems that confronts it is how to allocate core memory. The system is designed to be totally core-resident. The users as well are completely resident - there is no core swapping. A user retains the memory space assigned to him until he logs off or frees certain parts of it with a FREE statement. His program is compiled for him into absolute locations in memory from where it can later be executed.

We have available a large addressable core memory, giving us 528 pages of one thousand words each. Of these, about 380 pages are available to the users. When a user signs on, he is assigned two pages of memory. After that, he may acquire additional pages virtually without limit, which, for those of you familiar with IBM's TSS, is quite different from acquiring virtual pages without limit.

In other words, our answer to the core fragmentation problem is to allow core to be fragmented. Users do not occupy, in general, contiguous portions of core larger or smaller than one page. The exceptional case is that of arrays larger than some fixed minimum. Storage for an array is automatically allocated the first time the array is referenced during execution. This allocation is of one contiguous section of core, obtained from the free core through the facilities of the operating system.

There are several different kinds of pages which a user may acquire. Two pages are obligatory - a "symbol table" page and a shared "program and immediate data" page. He may also later need one or more "execution data" pages. All of these storage blocks may grow without limit with the exception of immediate data pages. These are pages of data which require the attention of a base register, and they are limited to five in number.

10. THE SYMBOL TABLE

Each user has one or more pages of symbol table. This is a linked table containing the name, address, and attributes of every variable in the currently active program.

The form of the table is a binary tree [7]. The position of an entry in the tree depends upon its

algebraic value as a character string coded in binary. A new entry is compared against the top of the tree. If it is greater in value, it is sent to the right; if smaller, to the left. This comparison is made against the next element down the tree, and so on, until a match is found, or the end of a branch is encountered. In the latter case, the variable is added to the tip of the tree and pointed to by the entry one level up the tree. This algorithm has the advantages of giving a very fast search time when variable names appear in fairly random order, and producing a compact table. The primary disadvantage of this method is the two extra words per entry - the left and right pointers down the tree.

The entire symbol table remains in core, even when the program is being executed. This extravagance allows the user to modify his program at any time. Even when it is running, he can interrupt it to insert, delete, or change lines (fig. 2). We also take advantage of the presence of the symbol table to obtain execution time information not normally accessible to compiled code. For example, we can prompt out the names of variables when they are used in a GET DATA statement, which is used to demand input from a typewriter terminal. We can determine the type and size of a variable when it is being used in a record input/output statement. Furthermore, we can store the name of the variable along with the data, so that the information can later be retrieved by name rather than by position on the file.

11. COMPILATION OF EXPRESSIONS

Whenever an expression is encountered, the arithmetic processor is called to perform the compilation. An assignment statement is a special case of an expression which is handled by this processor.

The algorithm used is a one-pass, left-to-right scan designed by Keese and Huskey at the University of California at Berkeley [8]. It is based on two tables of operator ranks, called left rank and right rank. The rank to be used depends upon whether the operator is being compared from the left side or the right side. The code generated by this algorithm is reasonably efficient, although no further attempt at optimization is made. Execution speed of the generated code is about four times that of early versions of IBM's PL/1 compiler.

12. IMMEDIATE EXECUTION OF STATEMENTS

Most statements can be compiled and immediately executed by terminating the line with an exclamation mark instead of the usual semicolon. These statements, called commands, are not linked into the program. This provides a powerful debugging and experiment control facility.

A number of statements had to be added to the language for control purposes:

NAME	serves to identify the user. Two initials and six characters from his last name are extracted to provide the primary file qualification name.
PROJECT	identifies a group of files for a user. The response is used as a secondary qualification.
TERMINAL	informs the system-operator of the new activity.
PROGRAM	is used to call in a program file, qualified by the user name and project. If no such file exists yet for the user, a new program file is opened and initialized for him and he is prompted to write a procedure.
RUN	is the statement needed to initiate or restart execution.
MODIFY	gives the user control of line number prompting and contextual text editing facilities [9].
LOGOFF	ends a terminal session and returns all memory assigned the user to the pool. Accounting information is produced in terms of page-minutes, that is the time integral of memory used by him during the session.

Some other PL/1 statements have been expanded in scope to provide facilities such as listing and deleting of files.

13. EXTENSION OF INPUT/OUTPUT STATEMENTS

References to data arriving from or going to instruments are made using extensions of PL/1

record input/output statements. A PL/1 file name is considered to correspond to a user's analogue or digital line. In the ENVIRONMENT attribute of the file declaration, he provides information to the system such as method of sampling, data rate, and type of input line. The OPEN statement, when applied to an instrument file, has the effect of initializing a data acquisition program in a closely coupled auxiliary computer to which the data conversion and transmission equipment communicating with the user is connected. READ and WRITE statements control the actual transmission.

14. ON-CONDITIONS

To provide the program control over various error conditions and exceptions which may arise during execution of a program, PL/1 ON-conditions are provided. By means of these the user can define a block of code to be executed in case of a prescribed event. In addition to the PL/1 defined conditions, such as OVERFLOW, division by zero, END of FILE, there are locally defined conditions related to storage files and instrument file input/output.

The most general form of an ON statement is:

```
ON condition BEGIN;
        (on-condition block)
        END;
```

When the ON statement is encountered during execution, the block is bypassed, and two effects take place: the condition is said to be enabled, and this block becomes the current block for that on-condition. This is implemented by adding an entry to the on-condition chain for that user. The entry consists of the address of the above block, together with an integer which is the assigned number for this particular on-condition.

When an exception occurs during program execution, the on-condition chain for that user is checked to see if the occurring condition is enabled. If so, the current block for that on-condition is invoked as a procedure call. On normal completion of the block by passing through the END statement, his code is resumed at the point of interruption.

15. TIME SLICING

The compiler is designed specifically for a time-sharing mode of operation. One of its distinctive tasks is, for instance, the insertion of instructions that test whether a time slice is exhausted. These are generated at END, GO TO, and CALL statements, and drive the yielding mechanism of the ACME time-sharing system (fig. 1). Thus, control is never seized from a user. The object code is provided with "YIELD" points, which willingly give up control.

16. IMPLEMENTATION

The ACME compiler was written by one programmer working full time, and a few programmers working part time, in less than two years. This was made possible by realistic system design and by programming in a higher-level language [10]. The system was written almost entirely in FORTRAN IV (H-Level) of IBM's Operating System/360. There are numerous advantages to coding in a higher-level language. Besides the obvious advantages of ease of coding and checkout, writing in FORTRAN allowed us a system flexibility which machine language systems seldom achieve. Furthermore, the FORTRAN H-Level compiler provided a higher level of execution time optimization than can be found in machine language coding over such a large volume of code [11, 12]. Using a higher-level language also reduced the volume of documentation [13] necessary for the project.

17. CREDITS AND SUPPORT

Much credit for ideas and procedures used in this compiler goes to other computer installations; notably project MAC at M.I.T., the Allen Babcock Company in Los Angeles, System Development Corporation in Santa Monica, The Patient Care System at Massachusetts General Hospital, The MEDLAB project at the Latter Day Saints Hospital in Salt Lake City, the ARPA Project at the University of California at Berkeley, The Computer Center of the San Francisco Medical School, and, of course, the Computation Center and the Computer Science Department of Stanford.

ACKNOWLEDGEMENTS

This work was supported in the planning stages by the Josiah Macy Jr. Foundation and was implemented as part of the ACME system founded by The National Institutes of Health, grant number FR0311

REFERENCES

[1] W. J. Sanders, G. Breitbard, D. Cummins, R. Flexer, K. Holtz, J. Miller and G. Wiederhold, An Advanced Computer System for Medical Research, Proceedings of the FJCC (1967) p. 497.
[2] G. Wiederhold, A Control Language for an Interactive Time-sharing System, SHARE TSS/67 Control Language Committee (1966).
[3] J. D. Aron, Real-time Systems in Perspective. IBM Systems Journal, Vol. 61 (1967).
[4] K. Lock, Structuring Programs for Multi-program Time-sharing On-line Application, Preprints of the FJCC (1965).
[5] G. Wiederhold, How to Program in PL/ACME. Document No. 80-50-00, Stanford Computation Center, Stanford University (1967).
[6] H. Berg et al., Report of the SHARE Advanced Language Committee (1964).
[7] T. N. Hibbard, Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting. Journal of the ACM, Vol. 9 (1962) p. 13.
[8] W. Keese and H. D. Huskey, An Algorithm for the Translation of ALGOL Statements. Proceedings of the IFIP Congress (1962).
[9] J. H. Saltzer, TYPSET and RUNOFF Memorandum Editor. MAC Project Memo 193-2 (1965).
[10] H. D. Huskey and W. Wattenburg, A Basic Compiler for Arithmetic Expressions. Communication of the ACM (1961).
[11] R. Halliday, The design of the optimizing FORTRAN compiler. SHARE XXVI, San Diego (1966).
[12] IBM Corporation. FORTRAN H Program Logic Manual, Form No. Y28-6642.
[13] G. Y. Breitbard et al., ACME Notes, Internal Documentation, Stanford Computation Center, ACME Facility, Stanford University (1965), to be published.

DISCUSSION

Question by I. Pyle

You have done away with block structure to avoid local variables. What happens about formal parameters for procedures?

Answer

Procedures have what look like formal parameters, but the parameters are in fact global variables. Otherwise we would just have the same tools for dealing with block structure.

Question by R. S. Scowen

a) Is the version of PL/1 described in this paper a dialect or subset; and can programs be compiled with any other PL/1 compiler? b) Why do you assume a declaration for an entity which is used but not declared? If you ask for a declaration, it would be necessary to re-compile the whole program much more rarely.

Answer

a) Yes, it is a dialect, mainly because of our non-implementation of block structure. b) We do have default variables in PL/1 so we do not have to declare everything.

Question by C. Curtis

a) What about line overflow? b) Do you allow nested subscripts? c) What kind of editing and concatenation do you have? Can you retrieve chunks of code?

Answer

a) We do not allow line overflow, so a user can only have 240 character or less (if he is prepared to push the ball back by hand). b) Yes, nesting may take place to any level. c) We have extensive editing. We have a command 'MODIFY' which calls in an editor for the entire program file. The program is a data file, i.e. we cannot have several programs with common procedures.

Question by H. Hoffman

Do you think that an implementation of Full-PL/I for a more conventional machine configuration, let us say a System/360 with 256K bytes core storage, is feasible according to your scheme?

Answer

We would have had to restrict ourselves if we were implementing for a smaller machine. However, we can go for the Full-PL/1 language, but we do not know how good the compiler will be. I think we can have block structure if we develop the techniques. But we have to draw the line between efficiency and fullness of language.

Remark by R. Pengelly

We have had similar experience in developing an incremental compiler. We estimate that it is only $2\frac{1}{2}$ times slower than the normal compiler. This would imply that you could go for the full language without too big a penalty in time. I would further suggest that your language is little more than a powerful assembly language.

Answer

I would scarcely think that our implementa-

tion language was inadequate. We have dealt with 'Do' loops completely for example. The user has complete control from the terminal.

Question by W. Waite

Do you keep a complete text file available so that the programmer is able to clean up his text?

Answer

Yes, but this is kept on the disc. He can, for example, get a list of it.