A General-Purpose Macro Processor as a Poor Man's Compiler-Compiler

ANDREW S. TANENBAUM, MEMBER, IEEE

***Abstract*—A method for quickly producing compilers for high level languages is described. The technique consists of feeding a description of the language to be translated to a general-purpose macro processor. Used in this way, the macro processor functions as a compiler-compiler, providing automatic parsing, lexical scanning, symbol table operations, and handling of syntactic errors. A complete syntactic and semantic description of a WHILE statement (except for Boolean expression processing) is given in only seven lines, as an example. A system programming language implemented by this method is discussed in order to illustrate the main ideas. The compiler produced for this language is compared to other compilers produced by conventional methods.**

***Index Terms*—Compiler, compiler-compiler, implementation methodology, macro processor, systems programming language.**

I. Introduction

SEVERAL techniques for producing compilers are in use. One method, which we call the "from scratch" method, consists of simply sitting down and writing the compiler in some general-purpose programming language, such as Pascal or PL/I. A second method is to use a compiler-compiler, a program that accepts a syntactic and semantic description of the programming language and produces as output a compiler for that language. This technique requires far less work on the part of the compiler writer, but has the disadvantage of requiring a suitable compiler-compiler before it can be used. In as much as few good compiler-compilers exist, and even fewer are portable, their use, however desirable, is all too often simply not practical. This paper suggests that a general-purpose macro processor [1]-[4] has many of the desirable characteristics of a compiler-compiler, and in addition, the advantage of high portability.

The idea of using macro processors to implement programming languages is not new. Halpern [5] and Brown [6], [7] have described similar techniques. Nevertheless, in practice, the use of macro processors to implement full scale production compilers which are then heavily used is still exceedingly rare. In this paper we discuss the method and give a case study of its use for implementing a general-purpose systems programming language (Sal). The Sal compiler has been in heavy use for nearly two years, and has been used, among other things, to implement a timesharing system.

This investigation grew out of a project to design and implement a general-purpose virtual memory timesharing system on a PDP-11/45. A language was needed in which to write the operating system. The languages provided by the manufacturer (Fortran, Basic, and Assembler) were deemed unsatisfactory. No compiler-compiler was available on the PDP-11. Furthermore, a working compiler for the systems programming language was needed within two months (in order to allow students taking the author's operating system course to participate in the construction of the system).

The design goals for the systems programming language to be implemented using the compiler-compiler were the following.

1) It must be possible to implement the compiler very quickly (2 months).

2) The language must be easy to learn and use (i.e., similar to Pascal and Algol).

3) The language must encourage well-structured programs (e.g., adequate control structures so that GOTO's will not be needed).

4) It must be possible to do machine dependent operations needed in operating systems (e.g., I/O, and changing the virtual memory map).

5) There must be powerful debugging facilities.

Because there was a severe time constraint, it was decided to write the compiler in a general-purpose macro language, ML/I [1], because it appeared to offer some of the facilities provided by compiler-compilers, and furthermore it was available.

In the following section we describe the systems programming language implemented using this method. We do this in order to demonstrate that it is a substantial language, and not merely an extension of assembly language. Many people are unaware of the fact that macros are a suitable technique for producing true high level languages. First, we will show that Sal is a high level language, similar in notation to languages like Pascal and Bliss [8]. In the following section we will discuss the implementation method. In the last section we will compare an ML/I based compiler for Sal to some conventionally produced compilers.

II. Description of Sal

In this section we describe the systems programming language (Sal) implemented using ML/I. Since there are no "standards" in this area, we succumbed to the temptation to design our own language. The overall structure of a Sal program is similar to that of a program in other high level languages, such as Pascal, Bliss, and Algol 68. As an example, Dijkstra's [9] divisionless prime number algorithm is given below. This algorithm is given in Pascal in Wirth [10, p. 141]; a quick glance at the Pascal version will demonstrate the similarity between Pascal and Sal.

Manuscript received February 6, 1975; revised November 13, 1975.
The author is with the Vrije Universiteit, Amsterdam, The Netherlands.

```
EQUATE N = 100   //DEFINITION OF A MANIFEST CONSTANT;
EQUATE M = 10
PROC PRIME   //PROCEDURE DECLARATION;
LOCAL X,SQUARE,I,K,LIM,PRIM,P[1:N], V[1:M]   //DECLARA-
  TION OF VARIABLES;
LET P[1] = 2   //ASSIGNMENT TO FIRST ELEMENT OF P;
PRINT 2   //OUTPUT A LINE CONTAINING THE NUMBER 2;
LET X = 1
LET LIM = 1
LET SQUARE = 4
FOR I FROM 2 TO N   //LOOP. I TAKES ON 2, 3, ··· N;
DO //THE BODY OF THE FOR LOOP IS ENCLOSED BY DO ··· OD;
   REPEAT //STOP WHEN "UNTIL" CONDITION IS TRUE;
      LET X = X + 2
      IF SQUARE <= X
        THEN LET V[LIM] = SQUARE
             LET LIM = LIM + 1
             LET SQUARE = P[LIM]*P[LIM]
      FI   //FI CLOSES THE IF STATEMENT;
      LET K = 2
      LET PRIM = TRUE
      WHILE PRIM AND K < LIM
      DO IF V[K] < X THEN LET V[K] = V[K] + P[K] FI
         LET PRIM = X /= V[K]   //PRIM IS ASSIGNED
                                  TRUE OR FALSE;
         LET K = K + 1
      OD
   UNTIL PRIM LITNU   //THIS LINE CLOSES THE REPEAT;
   LET P[I] = X
   PRINT X
OD
END
```

The principal statement types are indicated below.

1) LET ⟨lhs⟩ = ⟨expression⟩ (assignment statement)
2) IF ⟨condition⟩ THEN ⟨statements⟩ ELSE ⟨statements⟩ FI
3) WHILE ⟨condition⟩ DO ⟨statements⟩ OD
4) REPEAT ⟨statements⟩ UNTIL ⟨condition⟩ LITNU
5) FOR ⟨variable⟩ FROM ⟨expression⟩ TO ⟨expression⟩ BY ⟨expression⟩ DO ⟨statements⟩ OD
6) CASE ⟨expression⟩ IN ⟨clause list⟩ ESAC
7) DO FOREVER ⟨statements⟩ OD
8) CALL ⟨procedure name⟩ ⟨optional parameter list⟩
9) PRINT ⟨item list⟩
10) RETURN ⟨optional expression⟩
11) EXITLOOP

In addition there are a variety of statements for debugging and miscellaneous use, as well as some declaration statements. Statements may extend over as many lines as necessary, like high level languages, and unlike assembly languages. Extra blanks may be used for paragraphing, as in the above example. // and ; delimit comments.

The usual variations are allowed; for example, ELSE parts in IF statements and BY parts in FOR statements are optional. Assignment statements begin with LET to aid parsing (every statement begins with a unique key word). The use of Algol 68 style explicit closing delimiters (FI, OD, LITNU, and ESAC) solves the dangling ELSE and analogous problems, and eliminates the need for DO ··· END or BEGIN ··· END to delimit compound statements.

The basic data types are machine words, arrays of machine words, character strings, bit fields, and tables. A machine word is an untyped aggregate of bits (16 in the case of the PDP-11) used for counting, indexing, and other purposes. A bit field is a named subfield of a word, allowing any consecutive sequence of 1 to 16 bits to be accessed symbolically. Tables consist of one-dimensional arrays of entries. Each entry contains one or more scalar or array fields accessed by name. This feature is somewhat similar to structures in Algol 68 or PL/I, or records in Pascal.

Sal provides a wide variety of debugging and performance monitoring aids. These include: the ability to "single cycle" at the source level, having selected (or all) source statements typed out before their execution; procedure call/return tracing; variable tracing, assertion checking [11]; a symbolic interactive debugger; and automatic collection of procedure call frequency statistics.

We summarize this section by reiterating that Sal is a high level language rather than an assembly-like language. In the following section we shall discuss the use of ML/I for implementing high level languages.

III. Use of ML/I as a Compiler-Compiler

At this point we turn our attention toward the production of compilers (for languages like Sal) using ML/I. There are four major aids provided to the compiler writer by ML/I: automatic parsing, lexical scanning, symbol table operations, and syntactic error handling. There are also several minor aids. These will now be discussed in turn.

The most important facility ML/I provides to the compiler writer is automatic parsing. Rather than providing a context-free grammar to define the language to be translated, the user provides a set of prototype statements (macros) that drive the parser. Each prototype statement is a syntactic skeleton of the statement to be recognized. A prototype statement begins with a reserved word or token, and ends with a reserved word or token (which may be end-of-line). These are called delimiters. Furthermore, intermediate delimiters may also be provided. For example,

FOR FROM TO DO OD

represents a possible prototype for a FOR statement.

The above prototype statement has five delimiters—FOR, FROM, TO, DO, and OD. It directs the resulting compiler to scan the program to be compiled for the word FOR, and upon finding it, to search for the other four delimiters. The text detected between FOR and FROM is called the first actual parameter; we denote it by ⟨a1⟩. Similarly, the text between FROM and TO, between TO and DO, and between DO and OD will be denoted by ⟨a2⟩, ⟨a3⟩, and ⟨a4⟩, respectively. If the statement

```
FOR I FROM 0 TO 2 * N + 1
DO LET J = SQRT(I)
    IF I < K
        THEN LET A[I] = FUNCT1(J)
        ELSE  LET A[I] = FUNCT2(J)
    FI
OD
```

is encountered in a program to be translated, the actual parameters are

```
⟨a1⟩ = I
⟨a2⟩ = 0
⟨a3⟩ = 2 * N + 1
⟨a4⟩ = LET J = SQRT(I)
       IF I < K
          THEN LET A[I] = FUNCT1(J)
          ELSE LET A[I] = FUNCT2(J)
       FI
```

Note that an actual parameter may contain several lines, with imbedded carriage returns, as illustrated in ⟨a4⟩.

To make clear how the parsing and code generation work, we will examine the following simple prototype statement. In our system, ML/I key words and internal macros are in lower case to avoid conflicting with user chosen identifiers, which are all in upper case.

```
mcdef  WHILE DO OD    /*DEFINE A NEW PROTOTYPE
                        STATEMENT */
as[L⟨t2⟩:             /*OUTPUT A LABEL, E.G. L4: */
expr(⟨a1⟩, X⟨t2⟩)     /*TRANSLATE AND OUTPUT THE
                        CONDITION */
⟨a2⟩                  /*EXPAND THE SECOND PARAM-
                        ETER (BETWEEN DO OD) */
JMP L⟨t2⟩             /*GENERATE A JUMP TO THE TOP
                        OF THE LOOP */
X⟨t2⟩:                /*LABEL TO JUMP TO WHEN CON-
                        DITION FALSE */
]
```

In this example, comments are enclosed between /* and */. ⟨a1⟩ is the part between WHILE and DO (i.e., the condition) and ⟨a2⟩ is the part between DO and OD (i.e., the statements in the loop). The symbol ⟨t2⟩ is a temporary variable initialized to the number of macro calls to date. Thus, L⟨t2⟩ is a unique label. expr is itself a macro with two parameters, the first being a Boolean expression to be analyzed, and the second being a label. expr must expand into assembly code to evaluate the expression and jump to the label when it is false.

The line containing ⟨a2⟩ simply outputs the second actual parameter, causing it to be parsed and its statements to be translated. L⟨t2⟩: and X⟨t2⟩: are the labels for repeating the loop and exiting the loop, respectively. The square brackets [ ] are used (in this example) to define the extent of the translation rules.

It should be emphasized that except for the expression processing macro, expr, the seven lines above represent the complete syntax and semantics for processing a WHILE statement. Although expr is longer, it is also called in IF, CASE, REPEAT, assignment, and everywhere else where expressions are allowed, so only one such macro is needed for the entire compiler.

Prototype statements may be more complicated than indicated in this example. One useful feature is the ability to specify sets of alternative delimiters. For example, an IF statement consists of the keywords IF and THEN followed either by FI (no else part) or ELSE FI.

Automatic parsing is not the only compiler-compiler facility provided by ML/I; another one is lexical scanning. Most compilers have one or more procedures, or sometimes even a complete pass, to read source programs and break them up into basic symbols. Such basic symbols are identifiers (e.g., variable names), punctuation marks, or certain combinations of these. For example in Algol, := is regarded as a single basic symbol, the assignment symbol. When using ML/I, the compiler writer need not worry about the recognition of the basic symbols; it is all done automatically.

Another typical compiler-compiler facility provided by ML/I is symbol table handling. This can be most easily exploited by using a compile-time array of integer variables to store the properties of declared variables, data structures, named constants, etc. Such properties may include access mechanism (e.g., local, parameter, or global), type, run-time address, dimensionality, and size. The name is not stored explicitly.

Instead, whenever a name is declared, it is defined as a parameterless macro that expands into an underline (or some other easily recognizable character) followed by three or four digits. These digits comprise an index into the property array where the attributes of the name is stored. In effect, the name is redefined as a "pointer" to the properties of the name.

Whenever a declared name is encountered during the code generation phase, it is automatically replaced by an index into the property array. This replacement is part of ML/I's normal macro expansion, and requires no work on the part of the compiler writer. If the code generation phase of the expression macro checks for actual parameters that begin with an underline, the location of the corresponding properties is immediately known, with no hashing or searching needed. Of course, ML/I itself must maintain hash tables in order to look up macro names and replace them by character strings, but how ML/I performs these conversions is its business, and not the compiler writer's. To him it is automatic.

The compilers produced by ML/I have the ability to detect certain syntactic errors and give error messages, without the compiler writer having to explicitly program this. In particular, if an ML/I generated compiler detects the keyword that begins a statement, but fails to find one of the intermediate delimiters or the closing delimiter, it automatically issues a message to that effect, telling what was missing and where. The automatic error message facility applies only to syntactic errors, and not to semantic errors, such as undefined variables.

Inasmuch as compilation to assembly language may be viewed as a problem in string manipulation, namely, replacing the input strings (source statements) by output strings (assembly code), character handling facilities are useful. ML/I

TABLE I

	Compiler: Assembler Object Program Speed Ratio				Compiler: Assembler Object Program Size Ratio			
Program	Sal	Pascal	Algol 68	Algol 60	Sal	Pascal	Algol 68	Algol 60
towers of Hanoi	1.6	1.1	2.7	8.7	1.8	1.8	2.3	4.0
dot product	3.1	2.6	5.2	5.4	2.0	4.2	6.2	7.6
interchange sort	2.5	3.1	6.1	8.0	2.0	4.2	6.3	6.7
finding primes	1.1	1.3	1.6	2.1	1.4	3.2	4.1	3.7
Average	2.1	2.0	3.9	6.0	1.8	3.4	4.7	5.5

provides simple methods for extracting substrings, concatenating strings, and comparing strings or substrings.

ML/I's macro facilities are useful for purposes in addition to parsing source statements. A major use is to provide functions whose outputs are character strings. In this sense they are similar to function procedures in conventional programming languages. Macros may yield the null string as output. They are also useful for tasks such as printing messages and setting up predefined constants such as TRUE and FALSE.

Furthermore, since ML/I provides extensive macro facilities, it is easy to include macro facilities in the source language at no extra cost.

ML/I can also produce a listing of the translated code, saving the compiler writer the trouble of providing a routine. When viewed as a compiler-compiler, ML/I first reads in a file containing the prototype statements, i.e., the macro definitions. Then ML/I is "Frozen" by writing an absolute core image onto another file. This core image is the compiler. When executed, this compiler reads an input file containing the source program, and produces an output file containing the assembly code. This is precisely what one expects of a compiler.

A noteworthy side effect of using ML/I as a compiler-compiler is that once translation macros have been written and debugged, they can be immediately used on any other computer having an ML/I macro processor. Furthermore, ML/I has been intentionally designed to be easy to implement on a new machine. Our experience is that a good student can get ML/I running in under two weeks.

IV. Discussion

The Sal translator consists of 57 macros, totaling just over 1450 lines of code. (A line of ML/I averages about 20 characters.) About 750 of these lines are involved with the optimization of arithmetic and Boolean expressions. Another 100 lines are concerned with detecting semantic errors and issuing error messages. The remaining 600 lines handle the complete parsing and code generation for 13 kinds of executable statements and more than 20 kinds of declarations and miscellaneous statements (e.g., do not generate a run-time table for the interactive debugger).

To give an idea of the compactness of the translator, the total number of lines of ML/I used in some executable statements are given below. These sizes include the full syntax and semantics, code generation (with optimization), checking for semantic errors, printing error messages, and debugging features, but not expression translation.

LET	6	IF	16
RETURN	10	PRINT	33
CALL	13	ASSERTION	36
REPEAT	15	CASE	37
WHILE	16	FOR	56

Two man months were required to produce the initial version of the compiler. This was done by the author and his student assistants, none of whom were familiar with ML/I at the outset. The initial version did no optimizing, provided no debugging facilities, and simply ignored all semantic errors. However, it was definitely usable, and allowed work to begin that would otherwise have been held up.

Six months later, a much more polished version was begun, including considerable optimizing, a variety of debugging facilities, and much better semantic error messages. An additional four months was required to write the second version, for a total of about six man months altogether.

One diffficulty with the use of ML/I is that the resulting compilers tend to be quite slow, although the object programs they generate can be fast. Work is still underway attempting to speed up the compilation process, primarily by modifying ML/I itself. At present the compilation rate is about 1-2 lines of source code per second. This is more than an order of magnitude slower than conventionally produced compilers. However, much of the compilation time is spent optimizing arithmetic expressions. Eliminating this would increase compilation speed by perhaps a factor of 2. The Sal compiler itself occupies 40K bytes of memory, which is reasonable. Although the compilation process is slow, the quality of the generated object programs is quite good. Since we had no comparable compilers available on the PDP-11 to measure the Sal compiler against, the following machine independent test was devised. A sample program is written both in a high level language and in assembly language. The execution speed ratio of the compiler generated object program to the carefully hand coded program is an "absolute" measure of the compiler quality.

This measure is machine independent because both the compiler and assembly language programmer are using the same architecture, instruction set, etc. It simply measures how close the compiler comes to producing assembly language quality code. To compare compilers for different computers, the test programs must be separately coded in the assembly language for each machine, so that each compiler is "competing" against its own assembler.

The Sal compiler was compared using this method to three other high level languages on the CDC Cyber 73: Pascal, Algol 60, and Algol 68 in terms of both execution speed and object program size. The results are shown in Table I. The Sal compiler produced the most compact code of all (partly due to language design, since there are only two scope levels, local and global). It was also almost as good as the Pascal compiler in producing fast object programs. The Sal compiler was better than both the Algol 68 and Algol 60 compilers for speed and size on all four tests. All compilers ran in their most optimizing mode (e.g., no subscript checking).

ACKNOWLEDGMENT

I would like to thank E. Keizer and S. Mullender for helping with this project in a variety of ways.

REFERENCES

[1] P. J. Brown, "The ML/I macro processor," *Commun. Ass. Comput. Mach.*, vol. 10, pp. 618–623, Oct. 1967.
[2] W. M. Waite, "The mobile programming system: STAGE2," *Commun. Ass. Comput. Mach.*, vol. 13, pp. 415–421, July 1970.
[3] C. N. Mooers, "TRAC-A procedure describing language for the reactive typewriter," *Commun. Ass. Comput. Mach.*, vol. 9, pp. 215–219, Mar. 1966.
[4] C. Strachey, "A general purpose macrogenerator," *Comput. J.*, vol. 8, pp. 225–241, Oct. 1965.
[5] M. I. Halpern, "Towards a general processor for programming languages," *Commun. Ass. Comput. Mach.*, vol. 11, pp. 15–25, Jan. 1968.
[6] P. J. Brown, "Macro processors and software implementation," *Comput. J.*, vol. 12, pp. 327–331, Nov. 1969.
[7] —, "Levels of language for portable software," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 1059–1062, Dec. 1972.
[8] W. Wulf, D. Russell, and A. Habermann, "BLISS: A language for systems programming," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 780–790, Dec. 1971.
[9] E. W. Dijkstra, "Structured programming," Rep. EWD249, Eindhoven Tech. Univ., Eindhoven, The Netherlands, 1969, (quoted in [10]).
[10] Niklaus Wirth, *Systematic Programming: An Introduction*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[11] R. W. Floyd, "Assigning meanings to programs," in *Mathematical Aspects of Computer Science: Proceedings*, vol. 19, J. T. Schwartz (Ed.). Providence, RI: Amer. Math. Soc., 1967, pp. 19–32.

Andrew S. Tanenbaum (M'75) was born in New York City, NY, in 1944. He received the B.S. degree from the Massachusetts Institute of Technology, Cambridge, and the Ph.D. degree from the University of California, Berkeley.

In 1971 he went to The Netherlands to work on Algol 68 compilation and high level machine architecture at the Mathematical Centre. He is presently engaged in teaching and research in the Computer Science Group at the Vrije Universiteit, Amsterdam, The Netherlands. His current interests include high level machines, operating systems, distributed systems, and software engineering. He is the author of a recent book, *Structured Computer Organization* (Englewood Cliffs, NJ: Prentice-Hall, 1976).

Dr. Tanenbaum is a member of the Association for Computing Machinery, the IEEE Computer Society, and Sigma Xi.