

# Write Your Own Compiler

Users of small computer systems have many projects for which standard computer languages are not well adapted. Yet writing a compiler for a new language has usually been a time-consuming task, even using a large computer. This paper describes a tool, Meta4, for writing compilers for small computers. It consists of a specialized interpreter and a specialized language for writing compilers. A language is completely specified, both syntactically and semantically, by a program in the Meta4 language.

Meta4 is an outgrowth of Meta II, first described in Schorre (Proceedings of 19th National Conference of the ACM, 1964), reprinted in this magazine in April 1980. As I looked for a name, I found articles describing a Meta 3 (Schneider and Johnson, Proceedings of 19th National Conference of the ACM, 1964), and a Meta 5 (Oppenheim and Haggerty, Proceedings of the 21st National Conference of ACM, pp. 465-468). Probably there was a Meta4 also, but I didn't find a description, and it's hard to pass up such a good word play.

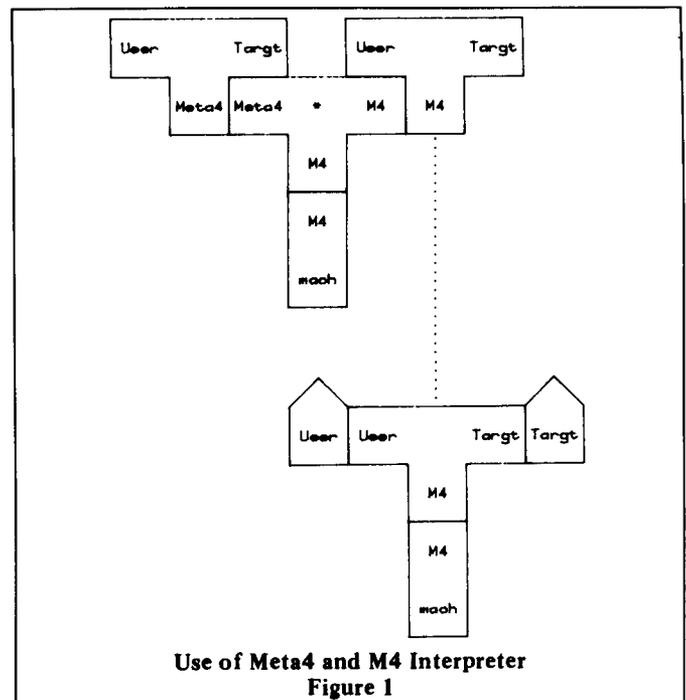
The reason that a 1964 compiler writing tool is interesting today is that today's microcomputers are comparable in size and speed to many of the larger 1964 computers. There has been progress in compiler writing tools, of course, so that better tools now exist. But the Meta's are both simple enough for a hobbyist to understand and powerful enough to be useful. An important point for me was that Meta4 was simple enough to use Pidgin to implement (see *DDJ* #57). If you want to pursue recent work, a good place to start is a four-article review contained in the August 1980 issue of the IEEE's *Computer*. It thus becomes part of a full boot strap operating system.

You won't need to know about Meta II to understand this article. But if you decide to implement Meta4, an understanding of Meta II will be useful. Meta4 keeps the ways that Meta II deals with lexical analysis, syntax analysis, and output. It provides better error handling, a symbolic memory for defining variables, arithmetic capabilities, memory stacks so code can be generated for machines without stacks, and facilities for generating machine code as extensions beyond Meta II.

The interpreter for Meta4 is written in Pidgin. The reason in a nutshell is portability. Pidgin, described in last month's issue, is a tiny language fairly easy to compile to machine code. In using Pidgin, I was most interested in cross-time portability for myself onto my next machine. But I also think that it is a fine means for describing to 8080, Z80, or 6800 processors the results of my work on a portable operating system.

An overview of how Meta4 works and what you will need to do in order to implement it will also serve to name the several languages and programs involved. A compiler involves three languages: the source language (what it translates from), the target language (what it translates to), and the coding language (what it itself is written in). To use Meta4, you need to write a compiler from the Userlang source language to the

Targetlang target language, in the Meta4 coding language. This program is shown by a "T" in the upper left of Figure 1, the diagram coming from a paper by Earley and Sturgis, "A Formalism for Translator Interactions," *CACM*, p. 607, 1970. A translator is shown with the source language in the left arm, the target language in the right arm, and the coding language in the base. An interpreter is shown with a coding language on the bottom of a rectangle and the interpreted language on the top. In the figures, programs provided here by listing have an asterisk in the center of their figure.



Now your compiler program is the input to a compiler with source language Meta4, a target language called M4, and which is coded in M4. This compiler program is provided by a listing accompanying this article, so you have that part. M4 is executable by an interpreter, for which the listing in Pidgin is provided. For now, suppose you have the interpreter in machine language. Then M4 is executable, and you can run the compiler from Meta4 to M4. When you input your compiler from Userlang to Targetlang written in Meta4, the output is a compiler from Userlang to Targetlang written in M4. But M4 is executable so you can now (as shown in the lower right) input your program, written in Userlang and get as output your program, written in Targetlang.

You can see that the M4 interpreter plays a key role in this process, because it is used once when you translate your compiler from Meta4 to M4, and again when you translate your program from Userlang to Targetlang. What you need to do to get Meta4 working is shown in Figure 2 - the listing of M4 interpreter in Pidgin is provided. You need a translator from Pidgin to your machine code that is executable on your machine, and that's what you got in last month's article.

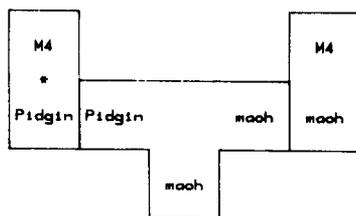
Since the key program in Meta4 is an interpreter, it is best termed a compiler generator interpreter. That is, it is an

**by William A. Gale**

© 1981 by William A. Gale. All Rights Reserved.

The author, who works in statistics and economics at Bell Telephone Laboratories, can be reached at 439 S. Orange Ave., South Orange, NJ 07079.

Producing an M4 Interpreter  
Figure 2



interpreter intended to both generate and run compilers.

To use Meta4 you have to understand the Meta4 language. So the next section gives you a start towards understanding it by going over the compiler from Meta4 to M4 written in Meta4. This is the first listing following the article (see page 22). It defines Meta4 exactly in terms of M4. The following section has some thoughts on *how* to use Meta4, for it can be used many ways. To implement Meta4, or to understand completely the definition of Meta4, you will need to understand M4, which is described in detail next. The article concludes with some help for the implementor.

### Meta4 Language

The listing of the compiler from Meta4 to M4 in Meta4 shows exactly what syntax constructions will be recognized, and how they will be compiled to M4. What I say here should be taken as a guide to that listing. It may sometimes be useful to compare parts of this listing with its translation (which is the listing of the Meta4 to M4 compiler in M4). The top line says to start processing with the subroutine called "COMPILE", which is at the bottom of the listing. COMPILE says to look first for the exact characters '.SYN', and then for an identifier (.ID, a letter followed by any number of letters or digits), then to generate a mark command (.OUT ('XM;')), a subroutine call to the identifier found (.OUT ('G'\*)), and a stop command (.OUT ('E')) so the translation of this top line is

```
XM;
GCOMPILE
E,
```

as can be seen in the Meta4 to M4 in M4 listing. The commands being output are M4 commands, explained in detail later. The syntax description then consists of zero or more "DECLARE's" (\$DECLARE) followed by zero or more "STATEMENT's" (\$STATEMENT). Let's check out the STATEMENTs, first, because they came from Meta II, and are the core of the parser.

Finding the subroutine STATEMENT just above COMPILE, you see that a STATEMENT is

- (1) An identifier, after which a label (.LAB \*) with the symbolic name of the identifier will be generated;
- (2) The exact character =
- (3) A PHRASE, and

- (4) The exact character ';' after which a return ('R') will be generated.

A PHRASE, in turn, has one of three alternative structures, which are written separated by the character '/'. It can be the exact characters '.PREP' followed by zero or more NOTSYN's, followed by an ALTS; or it can be just an ALTS; or it can be one NOTSYN followed by zero or more NOTSYN's. The first alternative that is satisfied will be accepted and others will not be checked. This is accomplished by generating a jump on true before each alternative as shown in ALTS. An ALTS is a SEQ followed by zero or more alternatives, each of which starts with the exact character '/', at which point we generate a jump on true to the label \*1, followed by another SEQ. After all the alternatives we define the label \*1 to which we generated jumps.

A SEQ, now, consists of a SYN followed immediately by zero or more SYN's or NOTSYN's. If the first SYN is not matched, we will just jump to the end of the SEQ (.OUT ('F/\*1)) but if any following SYN returns without matching, we have an error (.OUT ('X')). For following NOTSYN's we do not generate an error test.

A SYN is a basic syntactical item, for which it can be stated that one does or one does not occur next in the input. It is (1) a *new* identifier, which is compiled as a subroutine call, or (2) a single quoted string, which is compiled as a string to match exactly, or (3) the keyword '.ID' which is compiled as a lexical search for an identifier or (4) the keyword '.NUM' which is compiled as a lexical search for a number or (5) the keyword '.STR' followed by either a singly or doubly quoted string which is compiled as a lexical search for a string quoted by the first character of the string given, or (6) the keyword '.EMP' which compiles as a command to set the flag, or (7) a '\$' followed by a SYN, which is compiled as a request for zero or more of the specified SYN, or (8) the exact character '(' followed by a PHRASE followed by the exact character ')'. Any of these may be followed by an optional error message.

These subroutines define the parsing that Meta4 allows, and a few comments on their use may be worthwhile here. This is called recursive descent parsing, and it requires being able to tell which construct is to be chosen (at ALTS) by looking just at the next input symbol.

It works like this. A SYN always returns true or false, along with its specification. If the first SYN in a SEQ is false, we jump over the rest of the SEQ, still with false. If it is true, we go into the SEQ. If a SEQ returns false, the ALTS specifies trying the next alternative. If all alternatives return false, then an ALTS returns false. But if one returned true, then it returns true.

The order in which alternatives are listed is sometimes important. This is not good, but comes from choosing simple lexical analysis options. Notice that .EMP is always satisfied, so it must be last. Any alternatives beyond it will never be considered. If one keyword is contained in the left of another keyword (suppose you wanted to use REAL and REALTWO), then the longer one must be tested first. You will notice that all keywords in Meta4 start with '.'; this means they cannot be confused with identifiers so we don't have to put all keywords first. It is a good idea to put the most common alternatives first, and this gives the flexibility to do so.

A NOTSYN is an action not involving the input. In Meta II it consisted only of commands to generate commands

(.OUT) and of commands to generate labels (.LAB). A command to generate commands consists of the keyword '.OUT', followed by a '(' followed by zero or more AOUT's followed by a ')'. An AOUT is exactly the characters '\*1' or '\*2' in which case the first or second label numbers for the current subroutine will be concatenated to the output buffer (and generated if need be – see the discussion later), or the character '\*' in which case the input buffer will be concatenated to the output buffer, or a singly quoted string which will be concatenated to the output buffer or a RVALUE which will be concatenated to the output as a decimal number, or the character '.' for which the current input line number will be concatenated to the output. After the ')' is found, the output buffer will be dumped to the output file. A label is generated by the keyword '.LAB' followed by exactly one AOUT.

The most important NOTSYN that I have added is '.ACT'. This generates instructions to the compiler generated to do something. The instructions need to be in the M4 language. The instructions in an ACT are for the M4 interpreter. The instructions in an OUT are for the target machine. In this example the target is M4, but usually it is easier to tell them apart. Since Meta II didn't have anything to do except output, it didn't need this, but Meta4 needs to check symbol tables, assign storage, decide between alternatives, etc. An ACT is the keyword '.ACT', the symbol '(', and zero or more of {[a singly or doubly quoted string (the instruction code) followed by '\*1' or '\*2' or nothing] or [a comment which is a string quoted by '!']}. ACT essentially allows you to write compiler instructions in M4 machine code.

I've added two other kinds of output. '.CAT' is simply an OUT that does not dump the output buffer. It is useful for outputting part of a command, doing some more work, then finishing it. '.ERMS' is designed to allow putting error messages into a compiler so the users can know why they had an error. The basic error message that the M4 interpreter provides identifies the line and symbol number where it detected an error. In the context of Meta4, this is the line and symbol number of the first symbol in a statement that could not possibly be legal given the preceding symbols. This is quite useful, but any other help you can give yourself will be repaid.

The most frequent ACT is to transfer a number from one location to another. Meta4 makes these programmable in symbolic terms rather than having to use the machine instructions. The syntax for the transfers reflects the structure of M4 transfer commands with a requirement to first specify the value to transfer (RVALUE) as either a NAME identifier (previously defined by a declaration), or a CONSTSIMP or an EXPRESSION. If a NAME identifier is specified, it may also have appended to it ':' and a FIELD identifier. The value in the field of the symbolic memory indexed by the place named will be taken as the value. The transfer continues with the exact characters '=' and the NAME of a place optionally followed by ':' and a field identifier, which will be used to store the fetched value.

EXPRESSION, CONSTEXP, and CEXP all use a layering kind of structure that is most easily understood by studying CONSTEXP. The four subroutines CONSTEXP, CONSTTERM, CONSTFAC, and CONSTSIMP should be examined together. Start at the top line of CONSTSIMP, which says that a number is a constant factor, and to put it on the stack. Now pass up to CONSTTERM. It says to look for a CONSTFAC (which can

be satisfied by a CONSTSIMP), and then to look for an asterisk followed by another CONSTFAC. At this point two numbers are on the stack. The action is to pop (!) one of them and multiply (\*) that times what is on the stack. This will be done as many times as there are asterisks in the input. Now look at CONSTEXP. That says to look for a CONSTTERM (which has any number of multiplications of CONSTFACTORS) and then to look for a plus or a minus. If either is found, another CONSTTERM will be sought and then added to or subtracted from the first. The result of this layering is that '\*' binds more tightly than '+' or '-', so

$$a*b+c \text{ means } (a*b)+c \\ \text{not } a*(b+c).$$

And what if you want the second form? Well, look again at CONSTFAC. Besides CONSTSIMPS, which can be previously defined constant identifier (stack its value), a single quoted string (stack its first character), a minus sign and any other CONSTFAC (take the negative of the stack top), CONSTFAC can be an open parenthesis and any CONSTEXP then a close parenthesis. The last alternative lets you put in nested parentheses to get your meaning.

One point about EXPRESSION is worth discussing. Notice that it must be introduced by a '+' sign. This is a little ugly, but necessary to get the generality of expression wanted. The problem is that '(' has already been used in SYN to indicate a new PHRASE. Without the introductory '+', if you tried to start an expression with (a+b), then you would start compiling as a SYN rather than as an EXPRESSION. (Or, perhaps it would clobber your SYN's, depending on the order.) This was not obvious to me until it happened, and it represents a distinct limitation on the kinds of languages Meta4 can compile.

Another main NOTSYN is an IF statement which allows actions built of NOTSYN's to be done based on a test. In order not to foul up the recursive descent parsing, only NOTSYN's are allowed in the selected portions. Also, to keep the flag in shape for the parsing, it is set true by an IF after using it to make the branch. The IF statement also refers to a CEXP, a conditional expression. CEXP, CTERM, and CFAC form another layered definition, this time of the condition expression. CFAC shows that the low level entities are two RVALUE's connected by one of six relational operators. These can be qualified by '.NOT', joined by '.ANDIF' and '.ORIF', and grouped by parentheses.

NOTSYN's useful together with an IF statement are '.ERROR', which deliberately flags an error, and '.RETURN', which returns from a subroutine before the end. A widely useful NOTSYN is to define a string quoted by '!' as causing no action, so it becomes a comment. The most common transfer is to the stack, for which I introduced a special symbol ']'. Finally, the symbol '&' followed by a new identifier will be compiled as a subroutine call. This is the same action as the first SYN, but does not generate a test. It should be used if and only if the subroutine consists entirely of NOTSYN, but there is no test for this.

Finally, let's go back to the DECLARE's. These allow the user to define an identifier as a FIELD, a NAME, or a CONSTANT. FIELD's are only 0-3 for Meta4, but can be increased if needed in other compilers. NAME's are limited to ten registers, and the stacks. CONSTANT's can be any constant expression. The identifiers for these can be any length up

to the buffer sizes used in the M4 interpreter (80 bytes).

## Using Meta4

Schorre apparently conceived of using Meta II to generate code for an interpreter, not for direct machine execution. At first I thought this was a lousy idea, because interpreters are slower than machine code. So the M4 H, L, and OX instructions let it put out machine code. But after experience with the M4 interpreter, and with other interpreters derived from it, I see advantages to interpreters.

The big advantage is ease of debugging. Even though it is not interactive, the M4 code can be read easily with an edit program. If not understood, output commands of the interpreter can be inserted, and the bug chased down. This debugging ease comes in part from using only readable and partly meaningful symbols for M4. Somewhat more efficiency in storage and speed could be gained by sacrificing it, but don't be tempted. A microcomputer has little enough help for debugging, on which most of your time is spent.

A second advantage is that it is usually easier to write a compiler for an interpreter, because the interpreter does more work per instruction. It is also quite reasonable to build an interpreter with more advanced features, such as stack arithmetic and integers to make compiler writing easier.

Finally, (do I protest too much?) an interpreter is a particularly easy form of program to debug once the basic program loading and command fetching are done. This is because the interpreter exercises a command if and only if you tell it to. This narrows the scope for bugs, and leads to finding them faster.

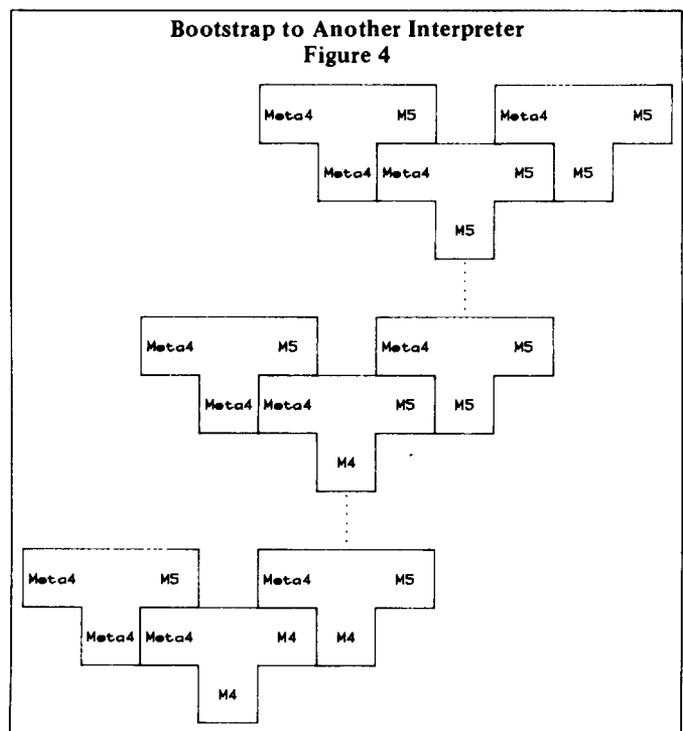
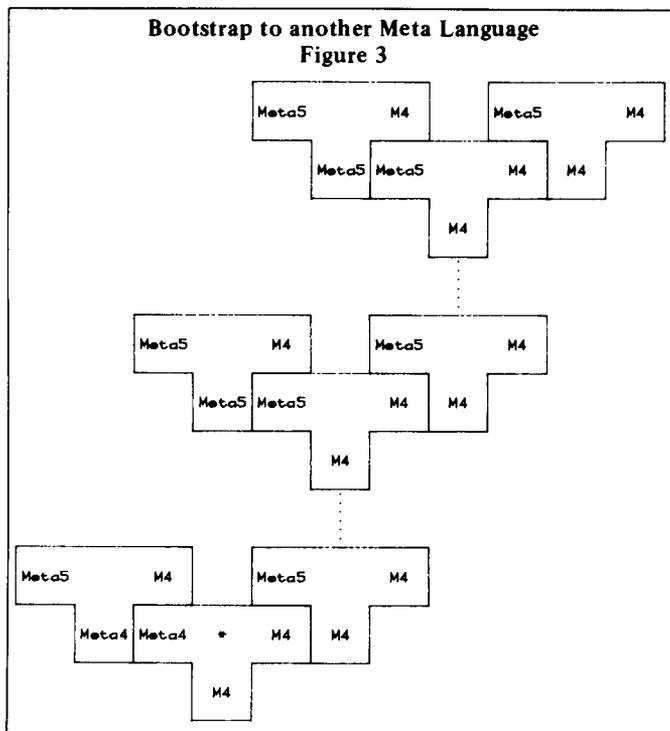
If you go the interpreter route, a second advantage to getting the M4 interpreter running is that it becomes a base you

can modify to make other interpreters pointed toward your own needs.

Meta4 makes a powerful system for defining a language. One of the useful features is that a Meta4 program defines not only the syntax of a language (the rules for what are legal statements), but also its semantics (what these statements mean). Admittedly the syntax is laid out a lot more clearly than the kinky, as it grew, means for dealing with semantics. Also, it is quite possible, when designing a compiler for a given language to add new Meta rules by augmenting Meta4. For example, a sequence of SYN's separated by given punctuation could be defined. Two-level means of defining languages have been receiving theoretical attention recently (Cleveland and Uzgalis, *Grammars for Programming Languages*, is a less abstract introduction). They are, needless to say, conceptually much cleaner, but operationally less clear.

Using Meta4 in a two-level manner, it becomes a fairly common operation to bootstrap a larger compiler. Figure 3 shows the operation. It can be done in two steps, but it's wise to make a third to be sure that the compiler is self-compiling. Start reading the figure at the lower left. You write a Meta 5 to M4 compiler in the Meta4 language. You compile this as usual to a Meta 5 to M4 compiler written in M4. Using this compiler you can translate another program you write (but usually similar to the first, and sometimes the same) which is the Meta 5 to M4 compiler written in Meta 5. You now generate a different Meta 5 to M4 compiler written in M4. You can use this with the same input as the second step and you should get the same output. Since that doesn't usually happen the first time, it's wise to check it.

Figure 4 shows a similar bootstrapping process if you want to improve the M4 language to an M5 language. The same



---

Meta4 to M5 compiler in Meta4 is input at all three stages, first producing a Meta4 to M5 compiler in M4, then a Meta4 to M5 compiler in M5. In the third step the coding language of the compiler running differs, but the source language and target language are the same, and the input is the same. Therefore the output should be the same. Check it.

Writing and initial debugging for a compiler written in Meta4 take about as long as other programs of comparable size in characters. But debugging the language – making sure that what you want to mean by a construct is what gets done – takes longer, because the compiler can only be tested by compiling and testing programs. Bugs in the compiled program may be due either to a bug in the program or in the compiler. I find that systematically exercising each feature in turn, by artificially short programs, flushes most of the bugs in a compiler, but a few will be left for your first programs to point out.

### The M4 Language

The three basic functions of lexical analysis, parsing, and output are handled very nearly the same in Meta4 as they are in Meta II. Table 1 gives the commands used for lexical analysis. The first five are very similar to Meta II, and each returns either “true” or “false” by setting or resetting a logical flag.

The lexical analysis commands are both powerful and restrictive: they each do a lot for one command, but only five

kinds of strings are recognized. Schorre thought out the kinds very well, however, and a wide class of languages can be recognized. I added only the recognition of a hexadecimal number. One dead end I followed was to make commands for simpler primitives, like an alphabetic symbol. Using them slowed the interpreter down a lot for little gain in flexibility. The use of whitespace is particularly likely to mismatch a pre-specified language definition, but is easy to change by changing the interpreter. Newlines always terminate strings, while blanks or tabs will also terminate identifiers or numbers. Otherwise these three whitespace characters are ignored, and can and should be used to improve readability of a Meta4 program.

The purpose of the fifth command is to allow a slight amount of backing up. After giving an LI command, especially, if an identifier has been found, the symbol table can be consulted. If the identifier found is not of a specified type (such as that representing a constant), it can be rejected at that point and tested again at a later point. It makes writing a syntax easier. The Meta4 to M4 compiler listed below has some examples (see IDNEW).

The commands used to do the recursive descent parsing explained before are just the flow of control statements given in Table 2.

The third core function, largely taken from Meta II, is output. The M4 interpreter has an output buffer to which

various strings are concatenated, and which is dumped to the output file on command. Table 3 describes the output options. Among them is the option to generate and use one or two labels per subroutine. The two are required, for instance, to build a loop construct, with one label at the top to repeat from, and one label at the bottom to exit to. It has turned out that only two labels has not been an impediment because it is

**Table 1**  
**Lexical Commands**

<b>LI</b>	An <i>Identifier</i> is a string starting with an alpha symbol and continuing with alpha or digits. The command returns false and does not change the input buffer if no identifier is in the input stream. Otherwise it returns true and moves the identifier to the input buffer. If the identifier is not in symbolic memory, it is entered. Register zero points to the memory cell.
<b>LN</b>	A <i>Number</i> is a sequence of digits. The command returns false if no digit is in the input stream. Otherwise it returns true and places the value of the number in register zero.
<b>LH</b>	A <i>Hex number</i> is a sequence of hexadecimal digits. The command returns false if no hexit is in the input stream. Otherwise it returns true and places the value of the number in register zero.
<b>LQchar</b>	A <i>Quoted string</i> is composed of a quoting character, any string of characters except newline or the quoting character, and a closing quoting character or newline. The command returns false if the input stream does not contain a string quoted by the character given. Otherwise it returns true and places the string, without its quotes, in the input buffer.
<b>LMchar . . . char</b>	A <i>Literal string</i> is any string of characters except a newline. The command returns true if and only if the input stream matches the literal string. The input buffer is not changed.
<b>L</b>	If the flag is set, the symbol found in the input stream is accepted. Otherwise, it is rejected and becomes the next string of characters in the input stream again. This modest 1 symbol backup makes different types of identifiers feasible.

**Table 2**  
**Control Commands**

<b>S</b>	Set the flag true.
<b>SF</b>	Set the flag false.
<b>SC</b>	Set the flag changed (true if it was false, and vice versa).
<b>.number</b> <b>.identifier</b>	Identify the next operational command as the place associated with this number or identifier.
<b>J/number</b>	Jump: Transfer control to the place associated with the number.
<b>F/number</b>	False jump: If the flag is false, transfer control to the place associated with the number, otherwise continue to the next command.
<b>T/number</b>	True jump: If the flag is true, transfer control to the place associated with the number, otherwise, continue to the next command.
<b>Gidentifier</b>	Go to a subroutine. Stack the next command location, and two zero label numbers, then transfer control to the place associated with the identifier.
<b>R</b>	Return from subroutine. Unstack two label numbers and a command location. Transfer control to the command location unstacked.
<b>E</b>	End. Stop processing.

two labels per subroutine call. Two labels are a marvelous encouragement to keep your subroutines small and intelligible, and to segregate concepts by level.

M4 has several ways to store numbers. The simplest are ten registers, each identified by a digit. The 'Y' stack allows stacking a number, reading its top entry, or unstacking a number. The 'Z' stack allows stacking or unstacking a number. Memory 'cells', described soon, can also store numbers. Table 4 shows the direct Fetches that are possible in M4.

Why a stack? Meta II can only compile code for a stack machine. By giving M4 a stack, operations can be stacked by the compiler, then output in the required order for a register machine. The second stack allows putting loop labels and if labels on different stacks, so that 'break' and 'continue' kinds of statements can be compiled. If the operations get hairy, with types for each variable, go to the stack of memory cells (Table 5).

The symbolic memory stores identifiers together with a memory *cell*. The cell consists of N=1 to 5 integers (known as 0 to N-1) and N byte (known as N to 2N-1) storage locations.

The symbolic memory is layered with all entries made in the top layer. The 'LI' command automatically searches the symbolic memory from the top layer down. If identifiers with the same name are in different levels, the top one will be found. This structure will allow defining variables with a local scope. Table 5 gives the commands for manipulating memory.

The last two commands use the space at the top of the symbolic memory vectors to contain a stack of memory cells.

The contents of a memory cell are fetched by an indirect Fetch. Indirect Fetches also allow getting the contents of the input buffer, and the registers. They are summarized in Table 6.

The direct and indirect fetches can be used for output with 'C' and 'L', and 'H' as shown in Table 3. There are several other disposal options shown in Table 7. The general form of a number manipulation is

[Fetch] [dispose]

while a fetch can be either

[directFetch]

or

[directFetch] [directFetch] [indirectFetch]

A final group of commands are useful for doing something about errors. Table 8 shows them. The 'XM' command essentially defines a place in the syntax to return to after finding an error. The 'XO' command is of some use for remarking which error was made, but of much greater use, I find, for debugging.

**Table 3**  
**Normal Output Options**

<b>O</b>	Out: Dump the output buffer to the output file with a newline, and reset it to the null string.
<b>OX</b>	Same as Out except no newline written.
<b>P specified string</b>	Concatenate the specified string to the output buffer.
<b>C</b>	Concatenate the input buffer to the output buffer.
<b>Fetch C</b>	(See Tables 4 and 6 for Fetches.) Convert the number fetched to a decimal string and concatenate to the output buffer.
<b>Fetch L</b>	Concatenate the low byte of the fetched to the output buffer.
<b>Fetch H</b>	Concatenate the low byte of the fetched to the output buffer as two hexadecimal digits.
<b>U</b>	If the first label number stacked is zero, substitute the symbol generator and then increment the symbol generator. Convert the first label number stacked to a digital string and concatenate to the output buffer. Also store the number in the zero register.
<b>V</b>	If the second label number stacked is zero, substitute the symbol generator and then increment the symbol generator. Convert the second label number stacked to a digital string and concatenate to the output buffer. Also store the number in the zero register.

It is easy to use an editor to insert output commands and an 'XO' command to trace the operation of a troublesome program so as to see what is going wrong. These are pretty crude aids by modern standards, but they let you skip over a bug you don't understand and work on others, because it will find a bug per statement.

### Implementing Meta4

The listing for the M4 interpreter in Pidgin gives detailed

**Table 4**  
**Direct Fetches**

<b>Y</b>	Fetch the number stacked on the Y stack, without popping.
<b>!</b>	Fetch the number stacked on the Y stack, and pop.
<b>Z</b>	Fetch the number stacked on the Z stack, and pop.
<b>H</b>	Fetch the high byte of the Y stack, without popping.
<b>Digit</b>	Fetch the contents of the register indexed.
<b>N/number</b>	Fetch the number specified.
<b>B</b>	Fetch the breadth of the input string.
<b>U</b>	Fetch the first number label without altering.
<b>V</b>	Fetch the second number label without altering.

**Table 5**  
**Memory Commands**

<b>MI digit</b>	Initialize memory, and set cell size to the digit (1-5); default size is 2.
<b>ME</b>	Enter the input buffer into the top layer of memory. Return the index pointing to the memory cell to register zero, and zero the memory cell.
<b>MS</b>	Stack memory – seal off the previous top layer and open a new layer.
<b>MP</b>	Pop memory – delete all entries in current top layer and reopen the next lower layer.
<b>MC</b>	Create a memory cell in the cell stack. Return the index to register zero, and zero the cell.
<b>MD</b>	Destroy the top memory cell on the cell stack. That is, pop this stack.
<b>MQ</b>	Query the entire memory using the input buffer. Return an index to the memory cell found (or zero) to register to zero.

**Table 6**  
**Indirect Fetches**

<b>M digit</b>	If digit is not in the range of memory cell indexes (0 to 2 N-1), use zero. Fetch the contents of the digit'th storage location of the cell indexed by the direct Fetch.
<b>S</b>	If the direct Fetch is less than the input string length, fetch the direct'th character from the input buffer. Default to the zeroth character.

**Table 7**  
**Non-Output Dispose Options**

<b>Y</b>	Stack the Fetched on the Y stack.
<b>Z</b>	Stack the Fetched on the Z stack.
<b>+</b>	Pop the Y stack, add the result to the fetched, and stack the result on Y.
<b>-</b>	Pop the Y stack, subtract the fetched from the popped, and stack the result on Y.
<b>*</b>	Pop the Y stack, multiply the result times the fetched, and stack the result on Y.
<b>&gt;</b>	Pop the Y stack, set the flag if the fetched was greater than the popped, else reset the flag.
<b>&lt;</b>	Pop the Y stack, set the flag if the fetched was less than the popped, else reset the flag.
<b>=</b>	Pop the Y stack, set the flag if the fetched was equal to the popped, else reset the flag.
<b>digit</b>	Store the fetched in the digit'th register.
<b>I[loc][direct]</b>	Store the fetched indirect to memory, in the loc'th location of the memory cell indexed by the result of this direct fetch.
<b>A</b>	Append low byte of fetched to input register.
<b>B</b>	Set length of string in input register (especially shorter) from the fetched.
<b>G</b>	Set the symbol generator from the fetched.
<b>U</b>	Store the fetched as the first label number.
<b>V</b>	Store the fetched as the second label number.
<b>D</b>	Dump the fetched -- do nothing.

**Table 8**  
**Error Handling Commands**

<b>XM character</b>	Save the current code position, stack depth and the mask character.
<b>X</b>	If flag is true, do nothing. If false print the line and symbol number; discard characters from the input until find the mask character or end of file. Reset the code pointer and stack depth to saved values.
<b>XN</b>	Copy the line number to the output buffer.
<b>XO</b>	Dump the output buffer to the terminal.

algorithms for implementing an M4 interpreter. You also need the Meta4 to M4 compiler in M4 to get started, once the M4 interpreter is debugged. A good test of your M4 interpreter is to recompile it from the Meta4 listing.

The M4 interpreter compiles into 9056 program bytes of 6502 code using the compiler described last month. It also uses memory for label index table, interpretive code, subroutine stack, and symbolic memory. I have set aside 2K, 10K, 1K, and 9K bytes respectively for these. To compile the Meta4 syntax only takes 1/2K, 3K, 1/10K, and 1/2K of this working space. It takes my Apple II about 60 seconds for this compilation.

The following brief description of the algorithms and methods of the M4 interpreter may also help in implementing. And don't overlook the comments within the program.

The first phase of the M4 interpreter is to initialize constants, etc., then to load and compact its program. The symbolic memory is used in this phase to compile the symbolic subroutine calls to show the location in the interpretive code referenced. Numbers preceded by a '/' are compiled into two-byte binary representations from a decimal representation. Number labels are entered in the numeric label table (they could also be compiled at this time and save the table space at a cost of more program to compile). In the interpretive code space each instruction is represented by a byte showing how many other bytes follow, then by the bytes of command, as compacted. This phase is accomplished about as fast as Apple can read the disk. At the end of this phase, the M4 interpreter notifies the user of how many instruction bytes were used, how many labels, and how many subroutines. For each subroutine call made to a subroutine never defined, a message is output. The name is not easily reconstructed at this point, but the last of the characters printed are the last characters of the name of subroutine called. Since the error is not necessarily fatal, the program continues. But it's usually a bad sign.

The main body of the M4 interpreter is a loop that fetches the next instruction, then chooses a sequence of actions as described before from the command characters. They are ordered so the most frequently used in the Meta4 to M4 compiler are first. When it is through it prints the maximum amount of symbolic memory that was used.

The registers and stack are integer vectors. Each stack operation is checked for overflowing or underflowing the space allocated. If you change a dimension statement, be sure to change the corresponding initialization statement.

The symbolic memory is constructed for digital searching (Knuth, *Art of Computer Programming*, Vol. III sorting and search, section 6.3). Knuth shows that this requires on average only  $\log_2 N$  search steps if the input data is random, so it is usually an efficient system. It is also easy to program. Two vectors are used to make a binary tree holding a level of the symbol table. One vector holds characters, the other integers which index an alternate place in the tree. In searching, the first character of memory is compared to the first character of the symbol. When a match is found, if it is a match on the end of string character (zero), the memory cell for the symbol begins at the next index. If a pair of characters matches, but is not the end of string character, the next character in memory will be compared to the next character in the symbol. Whenever the character in memory does not match, but there is another alternative (the alternative index is not zero), the cur-

rent character of the symbol will be compared to the alternative character in memory. If there is no alternative, then the search has failed - the symbol has not been entered into that level of memory.

Table 9 shows the representation from entering HAD, HAVE, HE, HER, in that order, and reserving memory cells of size 2.

### Availability in Machine Readable Form

With the listings of the M4 interpreter and Meta4 provided here, you can get Meta4 working. If you want to save yourself the typing, and can transfer files from an Apple near you, you may be interested in the following offer. A version of the M4 interpreter in an experimental operating system is available from the author. It includes the listings here, and machine language versions of the M4 interpreter, the Tincmp compiler, and an editor. It will run on an Apple, configured for 1 to 4 floppy disk drives. It requires a 16K memory card or an Apple II (not plus). Two disks and documentation are available for \$20. A third disk with Pidgin listings for the operating system programs is available for \$10. If you are reading this after 1981, better write to ask about availability. **DDJ**

**Table 9**  
**Representation in Symbolic Memory**

Index	Choice	Alt.	Notes
1	H	0	No alt to H
2	A	12	Alt to 'A' is 'E' starting at 12
3	D	7	Alt to 'D' is 'V' starting at 7
4	0	0	end of word HAD
5	M2	M0	Memory cell for HAD
6	M3	M1	
7	V	0	
8	E	0	
9	0	0	
10	M2	M0	Memory cell for HAVE
11	M3	M1	
12	E	0	No alt to 'HE'
13	0	16	But there is an alt to 'HE[end]'
14	M2	M0	Memory cell for HE
15	M3	M1	
16	R	0	
17	0	0	
18	M2	M0	Memory cell for HER
19	M3	M1	
20	Free memory starts		

## Listing 1

```

.SYN COMPILER
.FIELD VALUE 0 TYPE 3 ;
.CONSTANT NAME 1 FIELD 3 CONSTANT 2 UNDEFINED 0
.AVALIFIELDS 5
;
.NAME NUMBER 0 ' ID '0' SAVID '2' TEMP 'Z'
.STACK 'Y' UNSTACK 'I'
;
RVALUE =
IDNAME &CATNAME (: ' IDFIELD .CAT ('M' ID:VALUE)
/ .EMP )
/CONSTIMP
.CAT('N/' UNSTACK ' ' )
/ '+' EXPRESSION .CAT('I')
;
EXPRESSION = TERM $ ( '+'TERM .OUT ('I+')
/ '-' TERM .OUT ('I-')
)
;
TERM = FACTOR
$ ('**' FACTOR .OUT ('I**')
/'/' '256' .OUT ('HZ')
.OUT ('ID')
.OUT ('ZY')
/'%256'
.OUT ('HY')
.OUT ('N/' '256**')
.OUT ('I-')
)
;
FACTOR = (' EXPRESSION ' )
/CONSTIMP .OUT ('N/' UNSTACK 'Y')
/IDNAME &CATNAME (: ' IDFIELD
.CAT ('M' ID:VALUE)
/ .EMP ) .OUT ('Y')
/ '-' FACTOR .OUT ('Iz')
.OUT ('N/' '0Y')
.OUT ('Z-')
;
CATNAME = ID:VALUE=:STACK
.ACT ('IL') 'FACTS AS A ONE BYTE MACRO
;
TOPPLACE = IDNAME ID=:SAVID
(: ' IDFIELD .CAT ('I' ID:VALUE)
/ .EMP )
SAVID=:ID &CATNAME
;
DECLARE = '.FIELD' $ ( IDNEW ID=:SAVID
CONSTEXP &CHECKFRANGE NUMBER=:SAVID:VALUE
FIELD=:SAVID:TYPE
' )
;
/ ' .NAME' $ ( IDNEW ID=:SAVID
CONSTEXP
UNSTACK=:SAVID:VALUE
NAME=:SAVID:TYPE
' )
;
;
.OUT ('O')
/ .ACT ('S')
( .STR 'M' / .STR 'M' ) .ACT ('C')
('I' .ACT ('U'))
/'*2' .ACT ('V')
/ .EMP
.ACT ('O')
/ .STR 'I'
)
;
/ 'I' RVALUE .OUT ('Y')
/ RVALUE '=: TOPPLACE .OUT ()
/ 'IF' ( ' CEXP ' ) .OUT ('F/' *1) $ NOTSYN
( 'ELSE' .OUT ('J/' *2) .LAB *1 .OUT ('S')
/ .END' .LAB *1 .OUT ('S') )
/ .CAT
( 'SAOUT ' )
/ .CONDLAB
( '*1' .OUT ('UY')
.OUT ('0=')
.OUT ('T/' *1)
.OUT ('P... ' )
.OUT ('U')
/'*2' .OUT ('VY')
.OUT ('0=')
.OUT ('T/' *1)
.OUT ('P... ' )
.OUT ('V')
)
;
.OUT ('O')
.OUT ('S') !!!NOTE SIDE EFFECT!!!
.LAB *1
/ERRORMESSAGE
/ .ERROR' .OUT ('SF')
/ .SUCCEED' .OUT ('S')
/ .FAIL' .OUT ('SF')
/ '& IDNEW .OUT ('G' * )
/ .RETURN' .OUT ('R')
/ .STR 'I'
;
CRIGHT=
'=' .OUT ('Y')
RVALUE .OUT ('=' )
/ 'I' .OUT ('Y')
RVALUE .OUT ('=' ) .OUT ('SC')
/ '<' .OUT ('Y')
RVALUE .OUT ('<') .OUT ('SC')
/ '>' .OUT ('Y')
RVALUE .OUT ('>') .OUT ('SC')
/ '<' .OUT ('Y')
RVALUE .OUT ('>')
/ '>' .OUT ('Y')
RVALUE .OUT ('<')
;
CTERM = CFAC $ ( ' .ANDIF' .OUT ('F/' *2) CFAC )
/ .CONDLAB *2
;

```

```

/ ' .CONSTANT' S (IDNEW ID=:SAVID
  CONSTEXP
  CONSTANT=:SAVID:TYPE
  UNSTACK=:SAVID:VALUE
)
;
CONSTEXP = CONSTTERM $ ( '+' CONSTTERM .ACT('!+')
/ '- ' CONSTTERM .ACT('!-' )
)
;
CONSTTERM = CONSTFAC $ ( '*' CONSTFAC .ACT('!*')
)
;
CONSTSIMP = .NUM NUMBER=:STACK
/IDCONS ID:VALUE=:STACK
/.STR '"" .ACT('N/ØSY') !VALUE IS FIRST CHARACTER
/ ' .X' .HEXNUM !NUMBER
;
CONSTFAC =
CONSTSIMP
/ '- ' CONSTFAC .ACT('!Z' 'N/ØY' 'Z-' )
/ ' ' CONSTEXP ' )
;
IDTYPE = .ACT ('LI' !IS IT AN ID?
'F' *1 !NO, RETURN
'ME' !DEFINE OR FIND
'ØM3Y' 'Z=' !ID.TYPE==TEMP
'L' 'R' !TAKE AND RETURN
'!' *1 !NOW ACCEPT OR REWIND INPUT
'L' 'Z9' )
;
IDNAME = .PREP NAME=:TEMP IDTYPE
;
IDFIELD = .PREP FIELD=:TEMP IDTYPE
;
IDCONS = .PREP CONSTANT=:TEMP IDTYPE
;
IDNEW = .PREP UNDEFINED=:TEMP IDTYPE
;
CHECKFRANGE = . IF ( NUMBER<=AVAILFIELDS)
.RETURN
.ELSE
.ERMS ( NUMBER ' IS TOO BIG FOR A FIELD' )
.END
;
AOUT = '*1' .OUT ( 'U' )
/ '*2' .OUT ( 'V' )
/ '*3' .OUT ( 'C' )
/ ( .STR '"" / .STR '"" ) .OUT ( 'P' * )
/RVALUE .OUT ( 'C' )
/ 'H' RVALUE .OUT ( 'H' )
/ '!' .OUT ( 'XN' )
;
NOTSYN = ( ' .OUT ' ( ' $ AOUT ' )
/ ' .LAB' .OUT ( 'P....' ) AOUT
)

```

```

CFAC = RVALUE CRIGHT
/ ' .NOT' CFAC .OUT ( 'SC' )
/ ' ' CEXP ' )
/ ' & ' IDNEW .OUT ( 'G' * )
;
CEXP = CTERM $ ( ' .ORIP' .OUT ( 'T/' *1 ) CTERM )
.CONDLAB *1
;
ERRORMESSAGE = 'ERMS' ( ' .OUT ( 'T/' *1 ) SAOUT
' ) .OUT ( 'XO' ) .LAB *1
;
SYN = ( IDNEW .OUT ( 'G' * )
/ .STR '"" .OUT ( 'LM' * ) .OUT ( 'L' )
/ ' .ID' .OUT ( 'LI' )
.OOUT ( 'F/' *1 ) .OUT ( 'L' ) .OUT ( 'ME' ) .LAB *1
/ ' .NUM' .OUT ( 'LN' ) .OUT ( 'L' )
/ ' .STR' ( .STR '"" / .STR '"" ) .OUT ( 'LQ' * ) .OUT ( 'L' )
/ ' .EMP' .OUT ( 'S' )
/ ' $' .LAB *1 SYN .OUT ( 'T/' *1 ) .OUT ( 'S' )
/ ' ' PHRASE ' )
/ ' .HEXNUM' .OUT ( 'LH' ) .OUT ( 'L' )
)
(ERRORMESSAGE
/ .EMP )
;
SEQ = SYN .OUT ( 'F/' *1 )
$ ( SYN .OUT ( 'X' ) / NOTSYN )
.LAB *1
;
ALTS = SEQ $ ( ' / ' .OUT ( 'T/' *1 ) SEQ )
.CONDLAB *1
;
PHRASE = ' .PREP' $NOTSYN ALTS
/ALTS
/NOTSYN $NOTSYN
;
STATEMENT = .ID .LAB * '=' PHRASE ';' .OUT ( 'R' )
( .STR '!' / .EMP )
;
COMPILE = ' .SYN' .ID .OUT ( 'S'
.OOUT ( 'XM;' ) .OUT ( 'G' * ) .OUT ( 'E' )
.ACT ( 'XM;' ) $ DECLARE $ STATEMENT
;

```

(End Listing I. Listing II begins on next page.)

(Continued on top of page 22)



... CATNAME  
 ØMØY  
 !L  
 R ... TOPPLACE  
 ... GIDNAME  
 F/29  
 Ø2  
 LM:  
 L  
 F/3Ø  
 GIDFIELD  
 X  
 PI  
 ØMØC  
 ... 3Ø  
 T/31  
 S  
 F/32  
 ... 32  
 ... 31  
 X  
 2Ø  
 GCATNAME  
 ... 29  
 ... 33  
 R ... DECLARE  
 LM.FIELD  
 L  
 F/34  
 ... 35  
 GIDNEW  
 F/36  
 Ø2  
 GCØNSTEXP  
 X  
 GCHECKFRANGE  
 ØIØ2  
 N/3 I32  
 ... 36  
 ... 37  
 T/35  
 S  
 F/49  
 LM;  
 L  
 X  
 ... 34  
 T/38  
 LM-  
 LM.NAME  
 L  
 F/51  
 F/39  
 ... 4Ø  
 GIDNEW  
 F/41

Ø2  
 GCØNSTEXP  
 X  
 !IØ2  
 N/1 I32  
 ... 41  
 ... 42  
 T/4Ø  
 S  
 LM;  
 L  
 X  
 ... 39  
 T/38  
 LM.CONSTANT  
 L  
 F/43  
 ... 44  
 GIDNEW  
 F/45  
 Ø2  
 GCØNSTEXP  
 X  
 N/2 I32  
 !IØ2  
 ... 45  
 ... 46  
 T/44  
 S  
 X  
 LM;  
 L  
 ... 35  
 ... 38  
 R ... CONSTEXP  
 GCØNSTTERM  
 F/47  
 LM+  
 L  
 ... 48  
 F/49  
 GCØNSTTERM  
 X  
 i+  
 L  
 ... 49  
 T/5Ø  
 LM-  
 L  
 F/51  
 GCØNSTTERM  
 X  
 i-  
 ... 51

... 5Ø  
 T/48  
 S  
 X  
 ... 47  
 ... 52  
 R ... CONSTTERM  
 GCØNSTFAC  
 F/53  
 LM\*  
 L  
 F/55  
 GCØNSTFAC  
 X  
 i\*  
 ... 55  
 ... 56  
 T/54  
 S  
 X  
 ... 53  
 ... 57  
 R ... CONSTSIMP  
 LN  
 L  
 F/58  
 ØY  
 ... 58  
 T/59  
 GIDCONS  
 F/6Ø  
 ØMØY  
 ... 6Ø  
 T/59  
 LQ'  
 L  
 F/61  
 N/ØSY  
 ... 61  
 T/59  
 LM.X  
 L  
 F/62  
 LH  
 L  
 ... 49  
 T/5Ø  
 LM-  
 L  
 F/51  
 GCØNSTTERM  
 X  
 i-  
 ... 51

F/96  
 PO  
 O  
 ... 96  
 T/97  
 LM.ACT  
 L  
 F/98  
 LM(  
 L  
 X  
 ... 99  
 LQ'  
 L  
 F/1ØØ  
 ... 1ØØ  
 T/1Ø1  
 LQ\*  
 L  
 F/1Ø2  
 ... 1Ø2  
 ... 1Ø1  
 F/1Ø3  
 C  
 LM\*1  
 L  
 F/1Ø4  
 U  
 ... 1Ø4  
 T/1Ø5  
 LM\*2  
 L  
 F/1Ø6  
 V  
 ... 1Ø6  
 T/1Ø5  
 S  
 F/1Ø7  
 ... 1Ø7  
 ... 1Ø5  
 X  
 O  
 ... 1Ø3  
 T/1Ø8  
 LQ!  
 L  
 F/1Ø9  
 ... 1Ø9  
 ... 1Ø8  
 T/99  
 S  
 X  
 ... 62  
 ... 59  
 R  
 ... CONSTFAC  
 GCØNSTSIMP  
 F/63  
 ... 98

L  
 X  
 P...  
 V  
 O  
 ... 114  
 T/116  
 LM.END  
 L  
 F/117  
 P...  
 U  
 O  
 PS  
 O  
 ... 117  
 ... 116  
 X  
 ... 112  
 T/97  
 LM.CAT  
 L  
 F/118  
 LM(  
 L  
 X  
 ... 119  
 GAOUT  
 X  
 T/119  
 S  
 X  
 LM)  
 L  
 X  
 ... 118  
 T/97  
 GNØTSYN  
 LM.CONDLAB  
 L  
 F/12Ø  
 LM\*1  
 L  
 F/121  
 PUY  
 O  
 PØ=  
 O  
 PT/  
 U  
 O  
 PS  
 O  
 ... 115  
 GNØTSYN  
 T/115  
 O  
 S  
 X  
 LM.END  
 ... 121  
 T/122  
 LM\*2



```

V O ... 142
S ... 138
... 143
R ... CFAC
GRVALUE
F/144
GCRIGHT
X ... 144
T/145
LM.NOT
L ... 146
F/146
GCFAC
X
PSC
O ... 147
T/145
LM&
L
F/148
GIDNEW
X
PG
O ... 148
... 145
R ... CEXP
GCTERM
F/149
... 150
LM.ORIF
L
F/151
PT/
U
O
GCTERM
X

```

```

... 151
... 152
T/150
S ... 160
X X LM.ID
UY L
0= F/161
T/153 PLI
P.... O
U PF/
O U
O O
S PL
... 153
... 149
... 154
R O
... ERRORMESSAGE
LM.ERMS
L
F/155
LM(
L
T/145
PT/
U
O
... 156
GAOUT
T/156
S
X
LM)
L
X PXO
L
... 155
... 157
R ... SYN
GIDNEW
F/158
PG
C
O
... 158
T/159
LQ'
L
F/160
PLM
C
O

```

```

F/201
L
ME ... 201
X
PS
O
PXM;
O
PG
C
O
PE
O
XM;
... 202
GDECLARE
T/202
S

```

```

X ... 203
GSTATEMENT
T/203
S
X ... 200
... 204
R

```

End Listing II.

### Listing III

```

TOP;METATERP
;COPYRIGHT (C) 1981 W.A.GALE
HIMEM=36864;S0000
LOMEM=02048;S800
REGISTER=086
BYTE AA;ALL LOW DOUBLE LETTERS ARE TEMP VBLS
BYTE BB
BYTE BO(080);OUTPUT STRING
BYTE C0;NUMBER ZERO
BYTE C1;NUMBER ONE
BYTE C2;NUMBER TWO
BYTE C3;NUMBER THREE
BYTE C9;NUMBER NINE
BYTE CB;BLANK
BYTE CC
BYTE CD;'.' DOT
BYTE CE;'/' ESCAPE FOR NUMBERS
BYTE CG;'>'
BYTE CL;'<'
BYTE CM;'-'
BYTE CP;'+'
BYTE CQ;SINGLE QUOTE
BYTE CS;'*'
BYTE CT;TAB
BYTE CU;'='
BYTE CV;NUMBER 25
BYTE CX;'!'
BYTE DD
BYTE DS(010);DIGIT STACK FOR WRITING NUMBERS
BYTE EE
BYTE F1(00275);INPUT BUFFER
BYTE F2(00275);OUTPUT BUFFER
BYTE FL;FLAG FOR XRUE AND FALSE JUMPS
BYTE KS(10000);PROGRAM MEMORY SPACE
BYTE LI;LENGTH OF INSTRUCTION
BYTE LL;LINE LENGTH DURING LOADING
BYTE MC(03000);SYMBOLIC MEMORY CHARACTER VECTOR
BYTE MK;MEMORY CELL SIZE
BYTE MN;DIMENSION OF NS LESS ONE

```

# Write Your Own Compiler

(Listing III continued, text on pages 6-14)

```
BYTE ND;NUMBER OF DIGITS FOR WRITING NUMBERS
BYTE NL;NEWLINE !!! SYSTEM DEPENDENT
BYTE NS(080);CIRCULAR INPUTBUFFER
BYTE OS(080);INPUT STRING
BYTE PB;POINTER INTO BO
BYTE PI;INDEX INTO RI
BYTE PL;POINTER INTO NS
BYTE PM;BEGINNING IN NS OF INPUT NOT ACCEPTED
BYTE PN;COUNT OF SUBROUTINES AS LOADED
BYTE PO;POINTER INTO OS
BYTE QI;POINTER INTO RI
BYTE RC;COMMAND READ
BYTE RI(080);INSTRUCTION REGISTER
BYTE SD;STACK DIMENSIONS BOTH Y AND Z
BYTE WA;WORK IN PACK
BYTE WB;WORK IN PACK
BYTE X0;CHARACTER ZERO
BYTE X1
BYTE X2
BYTE X3
BYTE X9
BYTE XA;ALL X VARIABLES ARE CHARACTERS
BYTE XB
BYTE XC
BYTE XD
BYTE XE
BYTE XF
BYTE XG
BYTE XH
BYTE XI
BYTE XJ
BYTE XK
BYTE XL
BYTE XM
BYTE XN
BYTE XO
BYTE XP
BYTE XQ
BYTE XR
BYTE XS
BYTE XT
BYTE XU
BYTE XV
BYTE XW
BYTE XX
BYTE XY
BYTE XZ
BYTE YP;Y STACK POINTER
BYTE ZP;Z STACK POINTER
BYTE ZX;STORAGE FOR ERROR RECOVERY SYMBOL
INT I00;NUMBER ZERO
INT I01;NUMBER ONE
INT I03;NUMBER THREE
INT I10;NUMBER TEN
INT I16;NUMBER 16
INT IAA;WORKING
```

```
CHOOSE ON CC
CASE XM;MATCH SPECIFIC STRING
FL=+000
BB=+002
WHILE
AA=BB<!PI
ON AA
AA=RI(BB)
DD=NS(PL)
AA=AA!=DD
IF AA
GOTO 99;NO MATCH
ENDIF
BB++
GOSUB LA
ENDWHILE
FL=+001
GOSUB LB
CASE XI;ID TEST
FL=+000
CC=NS(PL)
PO=+000
GOSUB ZA
WHILE
ON AA
OS(PO)=CC
PO++
GOSUB LA
CC=NS(PL)
GOSUB ZA
DD=AA
GOSUB ZN
AA=DD?AA
ENDWHILE
AA=PO=C0
IF AA
GOTO 99
ENDIF
GOSUB MS;SEARCH ALL LEVELS
IPR(C0)=IAA
FL=+001
CASE XM; POSITIVE INTEGER TEST
FL=+000
IAA=I00
WHILE
CC=NS(PL)
GOSUB ZN
ON AA
FL=+001
IAA=IAA+I10
CC=CC-X0
IBB=CC
IAA=IAA+IBB
GOSUB LA
ENDWHILE
IPR(C0)=IAA
```

```

INT IBB;WORKING
INT IBK;BLOCK NUMBER FOR INPUT OR OUTPUT FILE
INT ICC;WORKING
INT IDD;WORKING
INT ILB;POINTER INTO ILT
INT ILN;LINE NUMBER OF INPUT
INT ILT(01000);LOCATION TABLE FOR NUMBER LABELS
INT IMB;MEMORY BASE INDEX FOR CURRENT LEVEL
INT IMD;DIMENSION OF MC AND IMI
INT IME;MEMORY FREE INDEX
INT IMI(03000);SYMBOLIC MEMORY INDEX VECTOR
INT IML;NUMBER OF LOCAL PARAMETERS PER MEM LEVEL
INT IMM;MAX MEM COUNTER
INT IMT;TOP OF FREE MEMORY, REST TAKEN BY CELL STACK
INT IMX;NULL MEMORY INDEX
INT IMZ;TEM VBL FOR MEM SEARCH
INT INL;NUMBER OF NUMERICAL LABELS
INT IPC;PROGRAM COUNTER, INDEX TO CURRENT INST
INT IPL;CODE POINTER WHILE LOADING
INT IPR(010);REGISTER VECTOR
INT IPT;POINTER INTO IST, SUBROUTINE STACK
INT IRN;NUMBER RETURNED FROM READ NUMBER ROUTINE
INT ISM;SYMBOL NUMBER OF INPUT
INT IST(00600);SUBROUTINE AND LABEL STACK
INT ITU;RESULT OF DIRECT FETCH
INT IUU;UNIQUE -- SYMBOL GENERATOR
INT IXX;WORK DURING NUMBER MANIP
INT IYS(080);Y STACK
INT IYY;WORK DURING NUMBER MANIP
INT IZC;ERROR PROGRAM COUNTER
INT IZS(080);ZSTACK
INT IZT;ERROR STACK POINTER
BEGINMAIN(AC,IAV)
NL=+010;!!!!!!
MS 'COPYRIGHT'
MS ' (C) 1981'
MS ' W.A.GALE'
WRITE NL
GOSUB IN
GOSUB IN
GOSUB RC
GOSUB LI;LEXICAL INITIALIZATION
IPC=+00000
LOC 00
GOSUB GI
CC=RI(C0)
CHOOSE ON CC
CASE XL;LEXICAL ANALYSIS COMMANDS
AA=PI=C1
IF AA
IF FL
GOSUB LW;SEEK WHITE SPACE AND MOVE UP BASE
ISM++;INCREMENT THE SYMBOL COUNT
ELSE
PL=PM;RESET THE LOOKAHEAD PINTER
ENDIF
ELSE;LONGER THAN ONE
CC=RI(C1)

```

```

CASE XH;HEX NUMBER TEST
FL=+000
IAA=+00000
WHILE
CC=NS(PL)
GOSUB ZH
ON AA
FL=+001
IAA=IAA*116
IBB=CC
IAA=IAA+IBB
GOSUB LA
ENDWHILE
IPR(C0)=IAA
DD=RI(C2)
CC=NS(PL)
PO=+000
AA=CC=DD
IF AA
GOSUB LA
WHILE
CC=NS(PL)
AA=CC!=NL
BB=CC!=DD
AA=AA&BB
ON AA
OS(PO)=CC
PO++
GOSUB LA
ENDWHILE
GOSUB LA
AA=CC=NL
IF AA
ILN++
ISM=I00
ENDIF
FL=+001
ELSE
FL=+000
ENDIF
DEFAULT
WRITE CC
MS ' NOT LEX!'
WRITE NL
ENDCHOOSE
ENDIF;TWO LETTER COMMANDS
CASE XF;FALSE JUMP
IF FL
ELSE
GOTO 20
ENDIF
CASE XP;PRINT STRING GIVEN
BB=+001
WHILE
AA=BB<!PI
ON AA

```

# Write Your Own Compiler

(Listing III continued, text on pages 6-14)

```
CC=RI (BB)
BO (PB) =CC
PB++
BB++
ENDWHILE
CASE XO;OUT
BB=+000
WHILE
AA=BB<!PB
ON AA
CC=BO (BB)
BB++
WRITE CC INTO F2
ENDWHILE
PB=C0
AA=PI=C1
IF AA
WRITE NL INTO F2
ENDIF
CASE XX;ERROR JUMP
AA=PI=C1
IF AA;JUST AN X
IF FL
ELSE
LOC 98
MS 'ERROR AT '
MS 'LINE NUM '
IAA=ILN
GOSUB PN
MS ' SYMBOL '
IAA=ISM
GOSUB PN
WRITE CB
WRITE NL
WHILE
CC=NS (PL)
AA=CC!ZX;AND COMPARE TO SPECIAL CHARACTER
BB=CC!C0;AND END OF FILE SIGNAL
AA=AA&BB
ON AA
AA=CC=NL
IF AA
ILN++
ISM=+00000
ENDIF
GOSUB LA;READ ONE MORE
GOSUB LB;AND BRING UP REAR
ENDWHILE;HAVE JUST READ SPECIAL CHARACTER
BB=CC=C0
IF BB;OR END OF FILE CHAR
MS 'END OF FILE'
GOTO 21
ENDIF
GOSUB LA;NOW MOVE BEYOND THE ZX
GOSUB LB
GOSUB LW;NOW EAT UP ANY WHITE SPACE AFTER IT

CC=RI (C1)
CHOOSE ON CC
CASE XF;SET FALSE
FL=+000
CASE XC;SET CHANGED
FL=C1-FL
DEFAULT
MS 'SET ERROR'
WRITE NL
ENDCHOOSE
ENDIF
CASE XU;UNIQUE NUMBER GENERATED AND STACKED
AA=PI=C1
IF AA;THEN DO A LOT OF WORK
IAA=IPT
IAA++;POINTS TO UN1
LOC 10
IBB=IST (IAA);THE CURRENT UNIQUE
AA=IBB<!I01;IE NEVER FILLED IN YET
IF AA
IUU++
IBB=IUU
IST (IAA)=IUU
ENDIF
IAA=IBB
IPR (C0)=IAA
GOSUB WN;WRITE NUMBER INTO BUFFER
ELSE;THIS IS A FETCH FROM U
GOTO 22
ENDIF
CASE XC;COPY INPUT
BB=+000
WHILE
AA=BB<!PO
ON AA
CC=OS (BB)
BO (PB)=CC
PB++
BB++
ENDWHILE
CASE XV;UNIQUE NUMBER 2
AA=PI=C1
IF AA;THEN THIS IS TO GENERATE A NUMBER
IAA=IPT
IAA++
GOTO 10;IAA POINTS TO SECOND UNIQUE
ELSE;JUST A FETCH
GOTO 22
```

```

IPC=IZC;RESTORE PROGRAM COUNTER
IPT=IZT;AND STACK POINTER
FL=+001;AND DECLARE TRUE
ENDIF
ELSE;A LONGER COMMAND
CC=RI(C1)
CHOOSE ON CC
CASE XN;WRITE LINE NUMBER INTO OUTBUF
IAA=ILN
GOSUB WN
CASE XO;WRITE OUTBUF TO ERROR OUTPUT
BB=+000
WHILE
AA=BB<IPB
ON AA
CC=BO(BB)
BB++
WRITE CC
ENDWHILE
WRITE NL
PB=C0
CASE XM;MARK FOR ERROR RETURN
IZC=IPC;SAVE CODE POSITION
IZT=IPT;SAVE STACK POSITION
ZX=RI(C2);AND SPECIAL CHARACTER TO READ THROUGH
DEFAULT
ENDCHOOSE
ENDIF;X + LETTER
CASE XT;TRUE JUMP
IF FL
GOTO 20
ENDIF
CASE XG;GOSUB LABEL MUST BE ALPHA
WA=RI(C1)
WB=RI(C2)
IPT=IPT+I03
IAA=+00597;STACK DEPTH-3
AA=IAA<=IPT
IF AA
MS 'STACK OVE'
MS 'R FLOW>>>'
GOTO 98
ENDIF
IST(IPT)=IPC
PACK(IPC,WA,WB)
IAA=IPT
IAA++
IST(IAA)=I00
IAA++
IST(IAA)=I00
CASE XR;RETURN
IPC=IST(IPT)
AA=IPT<I03
IF AA
MS 'STACK UND'
MS 'ERFLOW...'
GOTO 98

```

```

ENDIF
CASE XM;MEMORY OPERATIONS
CC=RI(C1)
CHOOSE ON CC
CASE XS;STACK MEMORY
GOSUB MH
CASE XP;POP MEMORY
GOSUB MP
CASE XE;DEFINE A CELL IN TOP
GOSUB ME
IPR(C0)=IAA
CASE XO;QUERY
GOSUB MS
IPR(C0)=IAA
CASE XC;CREATE CELL
GOSUB MC
IPR(C0)=IAA
CASE XD;DESTROY CELL
GOSUB MD
IPR(C0)=IAA
CASE XI;INITIALIZE
CC=RI(C2)
GOSUB ZN
IF AA
MK=CC-X0
ELSE
MK=C2
ENDIF
GOSUB MI
DEFAULT
MS 'ILG MEMOP'
WRITE NL
ENDCHOOSE
CASE XJ;JUMP UNCOND_ LABEL MUST BE NUMBER
LOC 20
AA=RI(C1)
BB=RI(C2)
PACK(ILB,AA,BB)
IPC=ILT(ILB)
CASE XE;END --STOP HERE
LOC 21
CLOSE F1
CLOSE F2
IAA=IMM;MAX MEM USED
GOSUB PN
MS 'MAXMEMUSE'
WRITE NL
STOP 0
DEFAULT;LOOK FOR FETCH AND STORE INSTRUCTION
LOC 22
QI=+000
GOSUB FT
GOSUB FI
GOSUB ST
ENDCHOOSE
GOTO 00
LOC 99

```

(Continued on top of page 30)

(Continued on next page)



```

MS 'INDEX TO '
MS 'MEM CELL '
BB=+0000
ENDIF
AA=BB<IMK
IF AA
IAA=BB
IAA=IAA+ITU
ITU=IMI(IAA)
RETURN
ELSE
BB=BB-MK
AA=BB<IMK
IF AA
IAA=BB
IAA=ITU+IAA
AA=MC(IAA)
ITU=AA
ELSE
GOTO 11
ENDIF
ENDIF
CASE XS;FETCH FROM STRING REGISTER
AA=ITU
BB=OS(AA)
ITU=BB
DEFAULT
OI--
ENDCHOOSE
ENDSUB
SUB FT;FETCHES DIRECT
CC=RI(OI)
CHOOSE ON CC
CASE XY;Y STACK
ITU=IYS(YP)
CASE CX;'I' POP Y
ITU=IYS(YP)
GOSUB PY
CASE XZ;Z STACK ALWAYS POP
ITU=IZS(ZP)
AA=ZP=C0
IF AA
MS 'Z STACKER '
WRITE NL
ZP=C1
FL=+000
ENDIF
ZP--
CASE XN;LITERAL FETCH OF NUMBER
OI++
AA=RI(OI)
OI++
BB=RI(OI)
PACK(ITU,AA,BB)
CASE XH;FETCH HIGH OF STACK
ITU=IYS(YP)
UNPACK(ITU,AA,BB)

```

```

XN='N'
XO='O'
XP='P'
XQ='Q'
XR='R'
XS='S'
XT='T'
XU='U'
XV='V'
XW='W'
XX='X'
XY='Y'
XZ='Z'
X0='0'
X1='1'
X2='2'
X3='3'
X9='9'
C9=+009
CV=+025
C0=+000
C1=+001
C2=+002
C3=+003
CB=' '
CX='!'
CS='*'
CM='-'
CP='+'
CG='>'
CU='='
CL='<'
SD=+080
CT='.'
CE='/'
CD='.'
CQ=':'
I00=+00000
I03=+00003
I01=+00001
I10=+00010
I16=+00016
MN=+079;DIMENSION OF NS LESS 1
IBK=IAY(C3)
CLOSE F2
OPEN F2 FOR XW AT IBK
FL=+000
PI=+000
PB=+000
IPC=+00000
PO=+000
IPT=+00000
ILB=+00000
PN=+000
IUU=+00000
ILN=+00001
ISM=+00000

```



```

ON AA
GOSUB LA
CC=NS(PL)
ENDWHILE
GOSUB LB
ENDSUB
SUB MC;CREATE CELL AT TOP
IAA=MK
IMT=IMT-IAA;LOWER TOP BY CELL SIZE
GOSUB MO;CHECK FOR OVERFLOW
IAA=IMT;POINT TO CELL
GOSUB MZ;AND ZERO
ENDSUB;MC
SUB MD;DESTROY CELL
IAA=MK
IMT=IMT+IAA;RAISE TOP BY CELL SIZE
AA=IMD<IMT
IF AA
MS 'DESTROY C'
MS 'ELL ERROR'
WRITE NL
ENDIF
IAA=IMT
ENDSUB;MD
SUB ME;ENTER INTO TOP LEVEL
;RETURNS IAA AS INDEX TO CELL
GOSUB ML;SEARCH TOP LEVEL
EE=IAA!=I00;FOUND IT HERE
IF EE
RETURN
ENDIF
IMI(IBB)=IMF;IBB FROM ML, STORE TO SHOW THIS OPTION
WHILE
MC(IMF)=CC;CC FROM ML ONCE
IMI(IMF)=IMX; NO ALTERNATIVES NOW
IMF++
GOSUB MO;CHECK OVERFLOW
AA=CC!=C0;NOT LAST CHAR
ON AA
BB++;BB FROM ML TO START
CC=OS(BB)
ENDWHILE
IAA=IMF
IAA=IMF
IDD=MK
IMF=IMF+IDD
GOSUB MO
GOSUB MZ
ENDSUB;ME
SUB MH;PUSHHH A NEW LEVEL
IMI(IMF)=IMB;POINT TO CURRENT BASE
IMB=IMF
IMF=IMF+IML;LOCAL PARAM SPACES
MC(IMF)=C0
IMI(IMF)=I00
ENDSUB
SUB MI;MEMORY INITIALIZE

```

```

IF AA
IMF=IMB
IMB=IMI(IMB)
ELSE
IMF=IML
MC(IMF)=C0
IMI(IMF)=I00
ENDIF
ENDSUB;MP
SUB MS;SEARCH ALL LEVELS
OS(PO)=C0
IMZ=IMB;SAVE CURRENT BASE
WHILE
GOSUB ML;SEARCH A LEVEL
EE=IAA==I00;NOT MATCHED
IMB=IMI(IMB);FOR NEXT LOWER
DD=IMB!=I00;NOT BOTTOM
CC=EE&DD
ON CC
ENDWHILE
IMB=IMZ;BACK TO TOP LEVEL
ENDSUB;MS
SUB MZ;ZERO A MEMORY CELL
BB=+000
IDD=IAA;PRESERVE POINTER TO CELL
WHILE
AA=BB<!MK
BB++
ON AA
MC(IDD)=C0
IMI(IDD)=I00
IDD++
ENDWHILE
ENDSUB;MZ
SUB PN;WRITE NUMBER IAA ON TERMINAL
GOSUB DS
WHILE
IBB=ND
AA=I00<!IBB
ON AA
ND--
AA=DS(ND)
WRITE AA
ENDWHILE
WRITE CB
ENDSUB
SUB PY;POP Y STACK
AA=YP=C0
IF AA
MS 'Y STACKER'
WRITE NL
YP=C1
FL=+000;AND TELL AN ERROR
ENDIF
YP--
ENDSUB

```

# Write Your Own Compiler

(Listing III continued, text on pages 6-14)

```

SUB RA;READ ALPHA LABEL
;FIRST COPY LABEL TO STRING INDEX
OS(C0)=CC
PO=+001
WHILE
  READ CC FROM F1
  GOSUB ZA
  DD=AA;SAVE THE RESULT
  GOSUB ZN;ALLOW EITHER ALPHA OR NUMERIC AFTER FIRST
  AA=AA?DD
ON AA
  OS(PO)=CC
  PO++
ENDWHILE
;NOW EAT UP LINE
AA=CC==NL
IF AA
  ELSE
  WHILE
    READ CC FROM F1
    AA=CC!NL
  ON AA
  ENDWHILE
ENDIF
ENDSUB
SUB RC;READ COMMANDS
CLOSE F1
IBK=IAV(C1)
OPEN F1 FOR XR AT IBK
GOSUB CK;CHECK OPENED OK
IPC=+0001
LL=+000
IPL=+00000
WHILE
  READ RC FROM F1
  AA=ER==C0
ON AA
  CHOOSE ON RC
  CASE CE;/' NUMBER FOLLOWS
    READ CC FROM F1
    AA=CC==CM
    IF AA;IF NEGATIVE SIGN
      READ CC FROM F1
      GOSUB RN;THEN READ NUMBER
      IRN=-IRN;AND INVERT
      GOTO 37
    ENDIF
  GOSUB ZN
  IF AA
    GOSUB RN;READ A NUMBER
  LOC 37
  UNPACK(IRN,AA,BB)
  KS(IPC)=AA
  IPC++
  LL++
ENDIF
DEFAULT;ACCEPT
LOC 35
KS(IPC)=RC
IPC++
LL++
ENDCHOOSE
ENDWHILE
;HAVE REACHED END
IPC--
AA=ER!C1
IF AA;NO, AN ERROR
MS 'CANT READ'
MS 'COMMANDS.'
WRITE NL
STOP 1
ENDIF
CLOSE F1
IBK=IAV(C2)
OPEN F1 FOR XR AT IBK
GOSUB CK
;NOW PATCH ALL THE GOSUBS THAT WERENT DEFINED
IAA=+00000
WHILE
  AA=IAA<IIPC
ON AA
  LL=KS(IAA)
  IBB=IAA+I01
  AA=KS(IBM);THE COMMAND
  AA=AA==NL
  IF AA;A GOSUB NEEDING PATCH
  KS(IBM)=XG;RESTORE THE RIGHT COMMAND
  IBB++
  AA=KS(IBM);GET THE INDEX TO DEFN
  IBB++
  BB=KS(IBM)
  PACK(IRN,AA,BB)
  BB=MC(IRN)
  AA=BB!C1
  IF AA;THE SUB WAS NEVER DEFINED. BUT WE DONT KNOW NAME
  WHILE
    AA=ICC<IIRN
    ON AA
      BB=MC(ICC)
      WRITE BB
      ICC++
    ENDWHILE
  WRITE CB
  MS 'SUB UNDEF'
  WRITE NL
ENDIF
ICC=IMI(IRN)
UNPACK(ICC,AA,BB);ICC IS THE REAL LOCATION OF THE SUB
KS(IBM)=BB;AND STORE IT
IBB--

```

```

KS(IPC)=BB
IPC++
LL++
RC=CC
AA=RC=CB
IF AA
ELSE
GOTO 33
ENDIF
ELSE
KS(IPC)=CE
IPC++
LL++
RC=CC
GOTO 33
ENDIF
CASE NL;END OF COMMAND
KS(IPL)=LL
IPL=IPC
IPC++
LL=C0
AA=LL=C0
IF AA
GOSUB RL
RC=NL
IPC--
GOTO 33
ELSE
GOTO 35
ENDIF
CASE XG;IF AT BEGINNING, A GOSUB TO COMPILE
IF AA
READ CC FROM F1
GOSUB RA;READ ALPHA LABEL
OS(PO)=C0
GOSUB ME;THIS WILL FIND OR DEFINE
IRN=IMI(IAA);DEFINED VALUE
AA=IRN=I00
IF AA
IRN=IAA;NOT DEFINED YET
KS(IPC)=NL;SET WARNING
ELSE
KS(IPC)=RC;WAS DEFINED SO WE HAVE FILLED VALUE
ENDIF
IPC++
UNPACK(IRN,AA,BB)
KS(IPC)=AA
IPC++
KS(IPC)=BB
IPC++
LL=C3
RC=NL
GOTO 33
ELSE
GOTO 35

```

```

KS(IBB)=AA
ENDIF
IBB=LL
IAA=IAA+IBB;THREAD WAY TO NEXT COMMAND
IAA=IAA+I01
ENDWHILE
GOSUB MP;POP MEMORY SO FORGET SUB NAMES
IAA=IPC
GOSUB PN
MS 'CMD BYTS '
IAA=INL
GOSUB PN
MS 'NUM LABS '
IAA=PN
GOSUB PN
MS 'SUBROUTIN'
WRITE NL
ENDSUB
SUB RL;READ LABEL
;FIRST DISCARD ANY NON ALPHANUMERIC
WHILE
READ CC FROM F1
GOSUB ZA
IF AA
GOTO 80;ALPHA LABEL
ENDIF
GOSUB ZN
IF AA
GOTO 85;NUMBERLABEL
ENDIF
AA=CC!NL;IF NO LABEL, NO ACTION
ON AA
ENDWHILE
RETURN
LOC 80;ALPHA LABEL
GOSUB RA;READ ALPHA LABEL
OS(PO)=C0
GOSUB ME;SEARCH MEMORY
IMI(IAA)=IPL;SAVE VALUE
MC(IAA)=C1;MARK DEFINED
PN++;COUNT THE SUBS
RETURN
LOC 85;NUMBER LABEL
INL++
GOSUB RN;READ NUMBER
ILT(IRN)=IPL
ENDSUB
SUB RN;READ NUMBER
IRN+=00000
WHILE
CC=CC-X0
IAA=CC
IRN=IRN+I10
IRN=IRN+IAA
READ CC FROM F1
GOSUB ZN
ON AA

```

# Write Your Own Compiler

(Listing III continued, text on pages 6-14)

```
ENDWHILE
ENDSUB
  SUB ST;STORE A NUMBER
  QI++
  CC=RI(QI)
  CHOOSE ON CC
  CASE XY;Y STACK
    YP++
    AA=SD<=YP
  IF AA
    MS 'Y OVERFLW'
    WRITE NL
    YP=SD
    FL=+000
  ENDIF
  IYS(YP)=ITU
  CASE XZ;Z STACK
    ZP++
    AA=SD<=ZP
  IF AA
    MS 'Z OVERFLW'
    WRITE NL
    ZP=SD
    FL=+000
  ENDIF
  IZS(ZP)=ITU
  CASE CP;'+', ADD TO S
    IAA=IYS(YP)
    IAA=IAA+ITU
    IYS(YP)=IAA
  CASE CM;'-', SUB FROM S TOP
    IAA=IYS(YP)
    IAA=IAA-ITU
    IYS(YP)=IAA
  CASE CS;'*', MULTIPLY
    IAA=IYS(YP)
    IAA=IAA*ITU
    IYS(YP)=IAA
  CASE CG; '>'
    IAA=IYS(YP)
    AA=IAA<!ITU
  LOC 12
  IF AA
    FL=+001
  ELSE
    FL=+000
  ENDIF
  GOSUB PY;POP Y STACK
  CASE CL;'<'
    IAA=IYS(YP)
    AA=ITU<!IAA
    GOTO 12
  CASE CU;'='
    IAA=IYS(YP)
    AA=ITU==IAA
    GOTO 12

CASE XV;SECOND STACK NUMBER
IAA=IPT
IAA++
GOTO 39
CASE XD;DUMP IT--NULL OP
CASE XH;WRITE LOW BYTE IN HEX
AA=ITU
ITU=AA
IAA=ITU/116;HIGH HEXIT
IBB=IAA*116
IBB=ITU-IBB;LOW HEXIT
CC=IAA
GOSUB WH
CC=IBB
GOSUB WH;WRITE HEXIT
DEFAULT;NUMBER OR ERROR
CC=RI(QI)
GOSUB ZN
IF AA
  AA=CC-X0
ELSE
  MS 'ILLG STOR'
  WRITE NL
  AA=C0
ENDIF
IPR(AA)=ITU
ENDCHOOSE
ENDSUB;ST
  SUB WH;WRITE HEXIT TO OUT BUF

AA=CC<=C9
IF AA
  CC=CC+X0
ELSE
  CC=CC+XA
  CC=CC-C9
  CC=CC-C1
ENDIF
BO(PB)=CC
PB++
ENDSUB;WH
  SUB WN;WRITE NUMBER INTO OUT BUFFER

GOSUB DS
WHILE
  IBB=ND
  AA=IBB<!IBB
ON AA
```

```

CASE XI;INDIRECT TO MEMORY
QI++
DD=RI(QI)
IBB=ITU
QI++
GOSUB FT
CC=DD
GOSUB ZN
IF AA
    BB=CC-X0
ELSE
    LOC 13
MS 'BAD INDIR'
MS 'ECT INDEX'
WRITE NL
BB+=000
ENDIF
AA=BB<!MK
IF AA
    IAA=BB
    IAA=IAA+ITU
    IMI(IAA)=IBB
    RETURN
ELSE
    BB=BB-MK
    AA=BB<!MK
    IF AA
        IAA=BB
        IAA=ITU+IAA
        AA=IBB
        MC(IAA)=AA
    ELSE
        GOTO 13
    ENDIF
ENDIF
CASE XC;CONVERT TO STRING AND CAT OUT BUF
IAA=ITU
GOSUB WN
CASE XI;WRITE LOW BYTE OF FETCHED INTO OUTBUF
AA=ITU
BO(PB)=AA
PB++
CASE XA;APPEND LOW BYTE OF FETCHED TO STRING BUFFER
AA=ITU
OS(PO)=AA
PO++
OS(PO)=C0
CASE XB;SET LENGTH OF STRING BUFFER
PO=ITU
OS(PO)=C0
CASE XG;SET GENERATOR
IUU=ITU
CASE XU;FIRST STACK NUMBER
IAA=IPT
LOC 39
IAA++
IST(IAA)=ITU
ND--
AA=DS(ND)
BO(PB)=AA
PB++
ENDWHILE
ENDSUB
SUB ZA;CC IZ ALPHA!!!!!!ASSUMES LINEAR
AA=CC-XA
BB=X2-CC
AA=AA<CV;25 (VINGT ET V)
BB=BB<=CV
AA=AA&BB
ENDSUB;ZA
SUB ZH;TEST AND CONVERT HEXIT
AA=X0<=CC
BB=CC<=X9
AA=AA&BB
IF AA
    CC=CC-X0
    RETURN
ENDIF
AA=XA<=CC
BB=CC<=XF
AA=AA&BB
IF AA
    CC=CC-XA
    BB+=010
    CC=CC+BB
    RETURN
ENDIF
;AA IS NOW FALSE
ENDSUB;CH
SUB ZN;CC IS NUMBER
AA=CC-X0
BB=X9-CC
AA=AA<=C9
BB=BB<=C9
AA=AA&BB
ENDSUB
SUB ZW;CC IS WHITE SPACE
AA=CC=CB;BLANK
BB=CC=CT;TAB
AA=AA?BB
BB=CC=NL;NEWLINE
AA=AA?BB
ENDSUB
BOTTOM

```