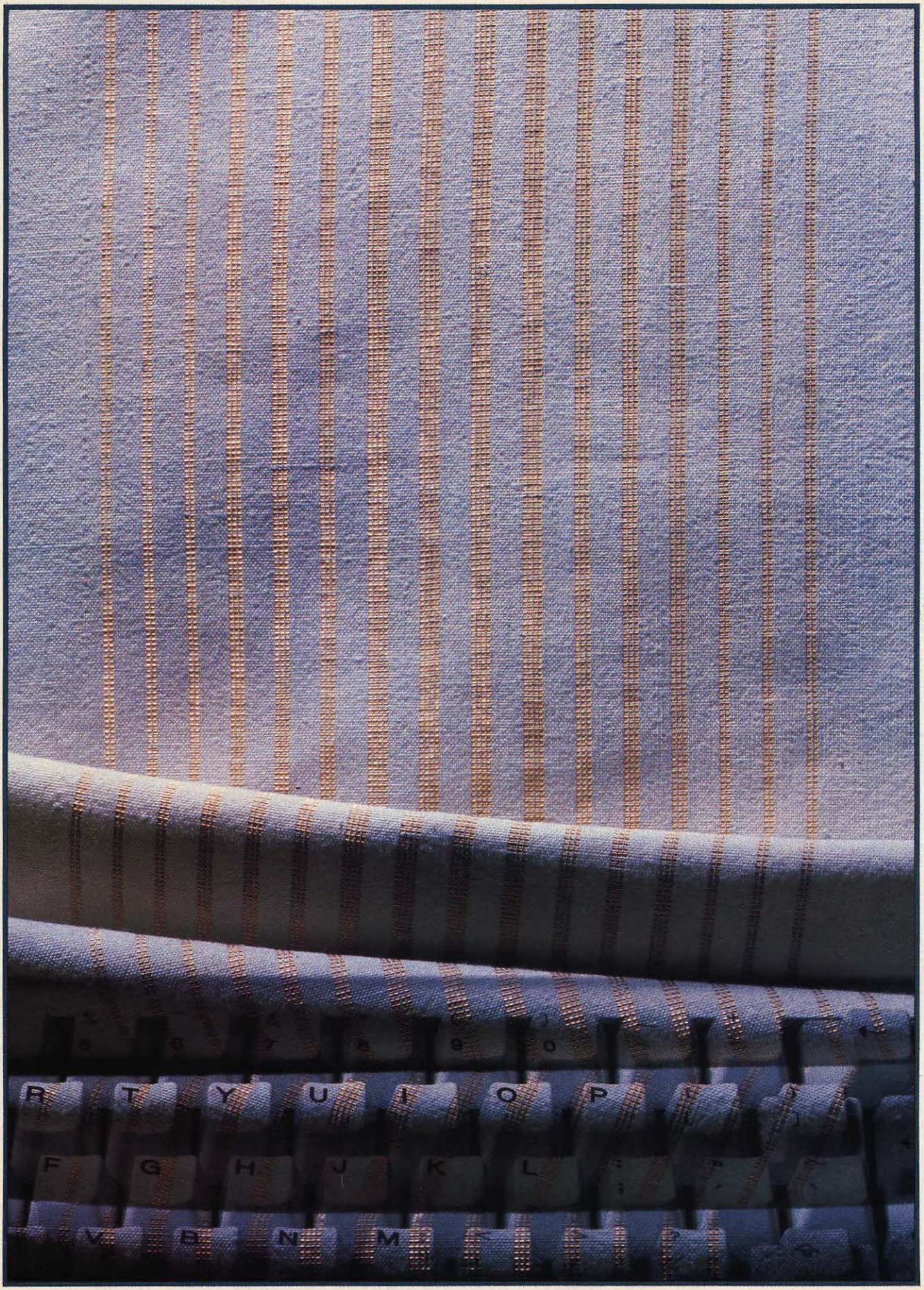
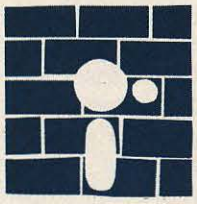

R T Y U I O P
F G H J K L
V B N M

AN ASSEMBLER FOR VM2

BY JONATHAN AMSTERDAM

This assembler can ease the pain of machine-language programming



Last month I discussed my design for the VM2 virtual machine. The VM2 interpreter and a simple monitor program are available on BYTEnet Listings. (Call (617) 861-9774 before November 1; thereafter, call (617) 861-9764.)

This month I will present an assembler that will make VM2 easier to use and will provide a stepping-stone to my ultimate project—the construction of a high-level-language compiler. The assembler is written in Modula-2 and can also be downloaded from BYTEnet Listings.

You may recall that VM2 is a stack machine, which means that most of its instructions expect their arguments or place their results on the computer's stack. ADD, for example, takes the top two values off the stack, adds them, and pushes the result back on the stack. Some instructions expect their arguments to follow them directly in memory; an example is PUSHC, which pushes a constant on the stack.

As it stands now, the only way to get VM2 to do anything is to use the monitor program I described last month, which allows you to poke instructions and data one word at a time into memory. The assembler I am presenting this month will let you write programs using instruction mnemonics as well as symbolic names for data. This tool will also automatically translate these programs into a sequence of numbers that can then be loaded into VM2's memory and run.

WHAT IS AN ASSEMBLER?

An assembler is a computer program that translates other programs from one form to another. The input to the assembler is source code—human-readable text that uses symbolic names for instructions and data. The output is object code—a sequence of bit patterns (or numbers) that can be loaded directly into the machine's memory and executed. Figure 1 is a graphic description of the assembler's function.

The assembler's job is fairly easy because the level of the source code is close to the computer's actual instruction set. Consider the following VM2 program listing that adds 5 and 6, leaves the result on the top of the stack, and then stops.

```
PUSHC 5
PUSHC 6
ADD
HALT
```

Each of the assembler's instructions—

(continued)

Jonathan Amsterdam is a graduate student at the Massachussetts Institute of Technology Artificial Intelligence Laboratory. He can be reached at 1643 *Cambridge St. #34, Cambridge,* MA 02138.

ILLUSTRATED BY DAVID JOHANSON

PUSHC, ADD, and HALT—is an actual VM2 instruction, represented in the computer as some word-long pattern of bits called an op code. Say, for example, that the op code for PUSHC is the number 1, for ADD is 2, and for HALT is 3. The object code for the short program listing just shown would then be the following:

1 5 1 6 2 3

Based on this example, you might think that something like the algorithm shown in figure 2 would serve for the assembler. It just keeps reading items from the input and translating them.

Such a program would certainly be useful, but you can improve it in two ways. First, you might want to provide some error checking. In its present form, this program will cheerfully assemble the sequence PUSHC HALT, even though PUSHC is supposed to be followed by an argument.

Second, you might want to be able to supply your own symbolic names for memory locations. This means you can name locations in which you want to store data, and it allows you to label points in your program to which you might want to branch. Listing 1 shows both of these uses of symbolic names, or labels, as they are usually called. This program counts down from 10 to 0. First the variable COUNT—that is, the contents of the memory location to which the label COUNT refers—is pushed on the stack. The BREQL instruction pops this value off the stack and, if it's 0, branches to DONE—that is, to the memory location to which the label DONE refers. If COUNT isn't 0, it is decremented and you go around the loop again.

Listing 1 also illustrates the syntax I used in my assembler. Assembly-language programmers will find it familiar; certain of its aspects are universal. The syntax can be described as follows:

- Labels appear at the beginning of the line and are followed by a colon.
- An instruction mnemonic may appear next. If the instruction takes an argument, the argument immediately follows the mnemonic.
- An argument may be a number or a label name.
- A mnemonic need not follow a label. Instead, an argument—that is, a number or label name—may follow a label (as with COUNT, above).
- Anything between a semicolon and the end of a line is a comment and is ignored by the assembler.

How are labels assembled? Well, whenever a label is defined (that is, whenever it appears at the beginning of a line and is followed by a colon), you need to associate the label name with the current memory location in some table. When the label is used (as an argument, for example), you retrieve the value from the table and output it. You can keep track of the current memory location by setting a counter to 0 before you start assembling and by incrementing it every time you assemble an item.

A revised version of my original algorithm is shown in figure 3. It keeps track of the current memory location and records labels and their values in a data structure called the label table. It also checks to see if arguments are provided for the instructions that require them.

This algorithm is closer to what I want, but it's not exactly right; it actually fails to work in some cases. In fact, it won't work in listing 1. I'll explain why in the next section.

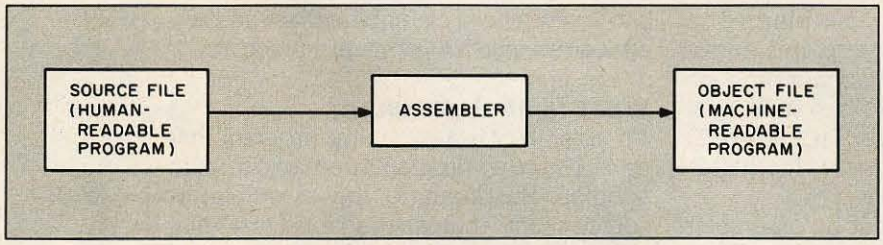


Figure 1: *The function of an assembler.*

```
Repeat until the end of input is reached:
  Read the next item from the input;
  If it is a mnemonic, find its op code and output it;
  If it is a number, just output the number.
```

Figure 2: A *first try at an assembler.*

Listing 1: A VM2 *program with labels.*

```
LOOP:   PUSH    COUNT
        BREQL   DONE    ; stop when COUNT = 0
        PUSH    COUNT
        PUSHC   1
        SUB
        POPC    COUNT   ; COUNT := COUNT - 1
        BRANCH  LOOP
DONE:   HALT
COUNT:  10
```

Backpatching Forward References

The second algorithm (figure 3) can't handle a situation where a label is used before it is defined. These so-called forward references are quite common. They occur twice in listing

(continued)

Lexical analysis is the process of organizing a sequence of characters into meaningful chunks.

1: Both COUNT and DONE are used as arguments before they are defined. In listing 1 you could avoid the problem by moving the label definition to the beginning, but sometimes it's impossible.

This is a serious problem in the second algorithm. When the assembler reaches the item COUNT in the first line of listing 1, it doesn't know what the label's value is. Yet it has to output the correct value if it is to produce a proper translation of the program.

One possible solution is to read the input file twice. The first time through you just gather label definitions. The next time around you do the actual assembling. This method is reasonable, but it's time-consuming.

Instead, you could choose to reread the output file. You assemble as before, except that when you find an undefined label you output a special marker. When the input is finished, go back and read the output file, substituting the label definitions for the markers. You could read the output file faster than you could read the input file, because it has a simpler structure, but this method is still slow.

I used a third algorithm, shown in figure 4, that trades time for space. With this algorithm you don't need to reread any files, but you do need to keep the assembled program in memory. Instead of outputting an assembled value, you save it in a table I'll the program array, which is indexed by VM2 memory address. Not until you have read the whole input file do you output the program array. When you first see an undefined label, you insert it in the label table with an indication that it is undefined. Each time you encounter the label, you add the current location to a forward reference list. When you finally find the label's definition, you go *back* through the program array, *patching* each location in the forward-reference list with the correct value. This technique is called backpatching.

The algorithm shown in figure 4 is the same as the second one but it is augmented with backpatching. There are a few slight additions I want to make to this algorithm. It would be nice to be able to specify characters as well as integers so that a single quote followed by a character denotes the ASCII value of that character. For example, the instruction PUSHC '? would result in the ASCII value for a question mark being pushed on the stack. I also want to let a sequence of characters enclosed in double quotes indicate a string, so I could write "BYTE" instead of 'B 'Y 'T 'E. I also want to add a facility for assembler directives, that is, commands to the assembler that can be embedded in the source code. I'll say that a dot followed by a symbol indicates a directive. The only directive I'm going to include is .BLOCK, which reserves a chunk of memory. It's useful for specifying buffers, arrays, and tables. For example, I could write NED: .BLOCK 30 to reserve 30 words of memory and label the first word NED.

```
Main program:
  Set the current location to zero.
  Repeat until the end of input is reached:
    Read the next item from the input;
    If it is a comment, ignore it;
    If it is a label definition, record it along with the current location
                in the label table;
    If it is a mnemonic, find and output its op code, and increment
                the current location;
      If the instruction takes an argument,
            Read the next item from the input and treat it as an argument.
    If it is not a mnemonic, treat it like an argument.

Handling arguments:
  If the item is a number, output it and increment the current location;
  If it is a label name, then
    If the label is defined, then get its value from the label table,
        output the value, and increment the current location;
    If the label is not defined, signal an error;
  If it is anything else (e.g., a mnemonic), signal an error.
```

Figure 3: *The revised assembler algorithm.*

THE DETAILS

Three key sections of my assembler program need to be explained more fully: scanning the input, the organization of the label table, and backpatching.

LEXICAL ANALYSIS

The three algorithms I've presented ask us to read the next item from the input. But what constitutes an item? Obviously, the items should be chunks of input that are meaningful to the program. For the assembler, a label would be an item, as would an integer. Unfortunately, most programming languages can't help much with this job: Usually, they allow you to read the input only a character at a time. The process of organizing the sequence of characters into meaningful chunks is called lexical analysis, and usually the chunks are called tokens, not items. Successive calls on a lexical analyzer's getToken routine return successive tokens in the input stream.

If you pick up any textbook on compilers, you can find out more about lexical analysis than you probably want to know. For my purposes, a

rather simple scheme will suffice. This scheme requires that you back up over the input occasionally (which is something the fancy lexical analyzers never have to do—that's why they're faster). The first task is to provide low-level routines that allow you to read characters from the input and put them back onto the input when you need to. Since you'll never have to put back more than one character at a time, this is not difficult to implement. These two procedures, which I call getChar and ungetChar, are used by the rest of the lexical analyzer in building up tokens. The lexical analyzer consists of several different procedures, each designed to read a different kind of token. Typically, a procedure keeps getting characters until it finds one that does not belong in the token it is constructing; it then "ungets" that character and returns the token. For example, the procedure for reading an integer reads characters from the input until a nondigit is encountered, "ungets" the nondigit, and returns the integer represented by the string of digits it has read.

When called on to get a token, the lexical analyzer uses the next character of the input to decide which of these token-building procedures to call. If, for example, the next character is a digit, the number routine is called. This decision is most elegantly implemented by a dispatch table, which is an array of procedures indexed by character. The dispatch routine needs merely to get the next character, use it to index into the table and call the associated procedure. You can use this technique only if your programming language has procedure variables, as do C and Modula-2. If not, you will have to resort to a case statement.

Although the procedures using getChar are under the impression that only one character at a time is being read from the input, getChar itself doesn't have to work that way. In particular, the input can be read a line at a time, with getChar doling out characters in the line one by one. The advantage of this scheme is that the whole line is available for printing out when an error is detected, to give you some idea of where in the program the error occurred.

A lexical analyzer is a handy tool to have on your software workbench. Unfortunately, every program has its own way of carving up the input. Still, much of the guts of the lexical analyzer can be separated from the program-specific details, providing a general-purpose toolkit for building lexical analyzers. One of the modules for this project, LexAnStuff, is just such a toolkit. It provides getChar and ungetChar, a dispatch mechanism, an error-display procedure, and some useful reading routines that construct strings and integers. To build a lexical analyzer with LexAnStuff, you write your program-specific reading routines and install them in the character table. The dispatch routine does the rest.

My assembler's lexical analyzer recognizes eight different kinds of

(continued)

```
Main program:
  Set the current location to zero.
  Repeat until the end of the input is reached:
    Read the next item from the input;
    If it is a comment, ignore it.
    If it is a label definition, then
      Look it up in the label table;
      If it is not present, enter it along with the current location;
      If it is present but undefined, backpatch using the forward-
        reference list for the label;
      If it is present and defined, signal an error.
    If the item is a mnemonic, find its op code, place the op code
        in the program array at the current location, and
        increment the current location;
      If the instruction takes an argument, then
          Read the next item from the input and treat it as an argument.
    If the item is a directive, then
      If the directive is BLOCK, then
        Read its argument, which should be an integer;
        Increment the current location by the argument;
      If the directive is not BLOCK, signal an error.
    If the item is none of the above, treat it like an argument.
  When the end of the input is reached,
      Check for labels that were used but not defined;
      Output the program array.

Handling arguments:
  If the item is a number, place it in the program array and increment the
      current location.
  If it is a character, place it in the program array and increment
      the current location.
  If it is a string, treat it as if it were a list of characters.
  If it is a label name, then
    If the label is defined, then
        Get its value from the label table;
        Place the value in the program array and increment the
          current location;
    If the label is not present in the label table,
      Enter it into the table;
      Start the forward-reference list by making the label-table
        entry point to the current location and placing a
        NIL in the program array at the current location.
    If the label is present in the label table but is undefined,
      Add the current location to the forward-reference list.
  If the item is anything else (e.g. a mnemonic), signal an error.
```

Figure 4: *The final assembler algorithm.*

The label table provides some sort of mapping between label names and values.

tokens: identifiers (strings of alphanumeric characters), label definitions (identifiers followed by a colon), directives (identifiers preceded by a dot), integers; character constants (a single quote followed by a character), string constants (a list of characters enclosed by double quotes), and end-of-line and end-of-file indicators. As all good lexical analyzers should do, it skips over comments, never mentioning their presence. So the callers of the lexical analyzer need not even consider comments—they can treat the program as comment-free. However, it does not distinguish mnemonics from uses of a label, although it easily could. Instead, it considers both to be identifiers, leaving the job of separating the two to higher-level functions. This was an arbitrary decision; you could do it either way.

The Label Table

The label table provides some sort of mapping between label names and their values (the addresses at which they were defined). The simplest way to implement this mapping scheme would be to create a list of <label, value> pairs—used, perhaps, as an array or linked list of records. Searching this list for a label could take a long time, because there would be as many comparisons as there are labels. And since redefining labels is not allowed, you would need to search the whole list every time you want to insert a label to make sure it hasn't already been defined. Other schemes involving sorted lists or trees are also possible, but it is probably best in this case to use some form of hashing.

Hashing is based on the observation that, ideally, you'd like to be able to get the value of a label by just indexing into an array, using the label itself as an index. That would be extremely fast—and more importantly, the time it would take would be independent of the number of labels in the table. But such an array would have to be absurdly large, requiring one entry for every possible label; and even if you just consider, for example, seven-letter labels, there are over 8 billion possibilities.

But of course, no program has anywhere near this number of labels. Could you get by with a much smaller array by somehow compressing the labels into a smaller range of indexes? Perhaps you can take a label—indeed, any string of characters—and hash it up, turning it into a number small enough to index into your array. That's exactly the purpose of a hash function—given a string as input, it produces a number as output. The number serves as an index into an array called, appropriately enough, a hash table. There are several good choices for a hash function; one of the simplest and most effective is to add up the ASCII values of the characters in the string, divide the sum by the size of the hash table and output the remainder.

It is possible that two different strings will hash to the same spot in the table. My implementation handles these collisions by linear probing: If a label hashes to location n in the hash table and n is occupied, then locations $n+1$, $n+2$, . . . are examined until an empty one is found and the string is placed there. If I run off the bottom of the hash table, I continue searching from the top. When looking for a label in the table, I use the same technique, only now when I reach an empty hash-table entry (or when I have searched the entire table) I know that the label is not present.

The Advantage of Backpatching

It may seem that backpatching requires space not only for the assembled program but also for the forward-reference list. But what endears backpatching to me is the fact that you need no additional storage for forward-reference lists. The forward references for a label form a linked list that is kept in the program array itself, in the very locations that would have been used to store the label values had the label been defined.

When the assembler encounters for the first time a label that hasn't been defined, it inserts that label into the label table with the current location as its value and puts a special value NIL in the program array at the current location. The value cell of the label-table record acts as a pointer into the program array, indicating the beginning of a linked list of forward references. A NIL terminates the linked list. Figure 5a illustrates this situation.

When a second occurrence of an undefined label is encountered, you link the current location onto the front of the forward-reference list. It is easy to do this: You put the contents of the label's value cell in the program array at the current location and replace the contents of the value cell with the current location. This situation is illustrated in figure 5b.

When you finally find the definition for the label, you update the label-table record to reflect this and step through the forward-reference list putting the correct value of the label into the indicated locations in the program array. You destroy the forward-reference list in the process, but so what? Its purpose has been served. Figure 5c shows what memory looks like after you have finished backpatching.

What value should NIL be? Anything that's not a valid VM2 address will do. But what, if any, word-length bit pattern can be a valid address? There's one value that can't be confused with the address of a forward reference; I leave its discovery to you.

Interface to VM2

My assembler reads in a VM2 assembly-language program and outputs a file that can be loaded by the VM2 monitor, a program I supplied with the VM2 machine simulator. The output file is actually a text file containing integers instead of some specially formatted file; this means you can read it with your friendly text editor.

Not that you'd want to do that too often—it's a little like reading punched paper tape—but I found it useful for debugging.

The loader always starts loading at location 0 and always loads consecutive memory locations. This affects the implementation of the .BLOCK directive. Basically, if a program says .BLOCK 100, you've got to output 100 zeroes (or some other value). This can make for large output files. It would be reasonable to extend the loader so that a special character in the file followed by a number *n* would result in the rest of the file being loaded from location *n*. This would solve the .BLOCK problem and also make it possible to start loading programs from a location other than 0. But be careful: The assembler assumes that the program will be loaded at a particular memory location—whatever the initial value of the current location variable is—and loading it into any other place will screw up all the program's references to memory.

EXTENSIONS

You can spiff up your assembler in a number of ways. Good assemblers allow the programmer to define symbolic constants with numeric values. So, for example, you could write

```
BUFSIZE = 30
```

at the beginning of your program and from then on use the more readable BUFSIZE instead of 30. Often you write arithmetic expressions that the assembler will evaluate for you. So if an array consists of 20 elements of two words apiece, you could write

```
NELEMENTS   = 20
ELSIZE      = 2
ARRAYSIZE   = NELEMENTS
  * ELSIZE
```

Another common addition is a macro facility. A macro instruction is a mnemonic that stands for several actual assembly-language instructions. You define the macro in your program by means of an assembler directive, and when you use it, the macro is replaced by the instructions that constitute it. Macro assemblers often have powerful features, but even a simple one will let you write macros with arguments. For example, say you are sick of writing

```
PUSH    A
PUSHC   1
ADD
POPC    A
```

every time you want to increment a variable. You'd like to just write INC A. Of course, you don't want to make the macro specific to A; any other label name should work in its place. The following macro will do the trick:

```
   .MACRO  INC  %1
; the % indicates an argument
   PUSH  %1
; argument is substituted here. . .
   PUSHC  1
   ADD
   POPC  %1    ; . . .and here
   .ENDM
; the "end macro" directive
```

All these extensions make it easier to write assembly-language code. But, except for a few test programs I wrote, I don't plan to do assembly coding. Instead, next month I'm going to tell you how to program in a high-level language and compile it into VM2 assembly language. ■

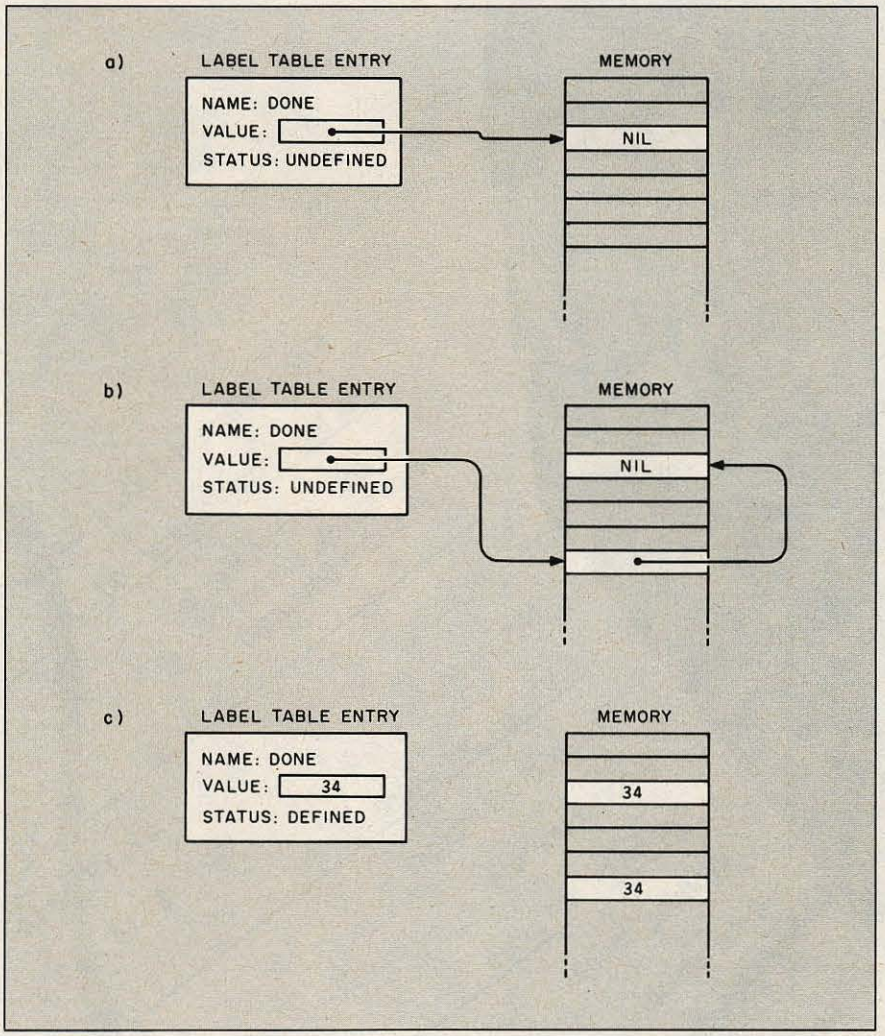


Figure 5: *The backpatching algorithm. (a) An undefined label is first encountered. (b) A second occurrence of the label is seen. (c) The label is defined at location 34.*