LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

7 November 1988

Part VII: LEXICAL SCANNING

INTRODUCTION

In the last installment, I left you with a compiler that would ALMOST work, except that we were still limited to single-character tokens. The purpose of this session is to get rid of that restriction, once and for all. This means that we must deal with the concept of the lexical scanner.

Maybe I should mention why we need a lexical scanner at all ... after all, we've been able to manage all right without one, up till now, even when we provided for multi-character tokens.

The ONLY reason, really, has to do with keywords. It's a fact of computer life that the syntax for a keyword has the same form as that for any other identifier. We can't tell until we get the complete word whether or not it IS a keyword. For example, the variable IFILE and the keyword IF look just alike, until you get to the third character. In the examples to date, we were always able to make a decision based upon the first character of the token, but that's no longer possible when keywords are present. We need to know that a given string is a keyword BEFORE we begin to process it. And that's why we need a scanner.

In the last session, I also promised that we would be able to provide for normal tokens without making wholesale changes to what we have already done. I didn't lie ... we can, as you will see later. But every time I set out to install these elements of the software into the parser we have already built, I had bad feelings about it. The whole thing felt entirely too much like a band-aid. I finally figured out what was causing the problem: I was installing lexical scanning software without first explaining to you what scanning is all about, and what the alternatives are. Up till now, I have studiously avoided giving you a lot of theory, and certainly not alternatives. I generally don't respond well to the textbooks that give you twenty-five different ways to do something, but no clue as to which way best fits your needs. I've tried to avoid that pitfall by just showing you ONE method, that WORKS.

But this is an important area. While the lexical scanner is hardly the most exciting part of a compiler, it often has the most profound effect on the general "look & feel" of the language, since after all it's the part closest to the user. I have a particular structure in mind for the scanner to be used with KISS. It fits the look & feel that I want for that language. But it may not work at all for the language YOU'RE cooking up, so in this one case I feel that it's important for you to know your options.

So I'm going to depart, again, from my usual format. In this session we'll be getting much deeper than usual into the basic theory of languages and grammars. I'll also be talking about areas OTHER than compilers in which lexical scanning plays an important role. Finally, I will show you some alternatives for the structure of the lexical scanner. Then, and only then, will we get back to our parser from the last installment. Bear with me ... I think you'll find it's worth the wait. In fact, since scanners have many applications outside of compilers, you may well find this to be the most useful session for you.

LEXICAL SCANNING

Lexical scanning is the process of scanning the stream of input characters and separating it into strings called tokens. Most compiler texts start here, and devote several chapters to discussing various ways to build scanners. This approach has its place, but as you have already seen, there is a lot you can do without ever even addressing the issue, and in fact the scanner we'll end up with here won't look much like what the texts describe. The reason? Compiler theory and, consequently, the programs resulting from it, must deal with the most general kind of parsing rules. We don't. In the real world, it is possible to specify the language syntax in such a way that a pretty simple scanner will suffice. And as always, KISS is our motto.

Typically, lexical scanning is done in a separate part of the compiler, so that the parser per se sees only a stream of input tokens. Now, theoretically it is not necessary to separate this function from the rest of the parser. There is only one set of syntax equations that define the whole language, so in theory we could write the whole parser in one module.

Why the separation? The answer has both practical and theoretical bases.

In 1956, Noam Chomsky defined the "Chomsky Hierarchy" of grammars. They are:

- o Type 0: Unrestricted (e.g., English)
- o Type 1: Context-Sensitive
- o Type 2: Context-Free
- o Type 3: Regular

A few features of the typical programming language (particularly the older ones, such as FORTRAN) are Type 1, but for the most part all modern languages can be described using only the last two types, and those are all we'll be dealing with here.

The neat part about these two types is that there are very specific ways to parse them. It has been shown that any regular grammar can be parsed using a particular form of abstract machine called the state machine (finite automaton). We have already implemented state machines in some of our recognizers.

Similarly, Type 2 (context-free) grammars can always be parsed using a push-down automaton (a state machine augmented by a stack). We have also implemented these machines. Instead of implementing a literal stack, we have relied on the built-in stack associated with recursive coding to do the job, and that in fact is the preferred approach for top-down parsing.

Now, it happens that in real, practical grammars, the parts that qualify as regular expressions tend to be the lower-level parts, such as the definition of an identifier:

```
     <ident> ::= <letter> [ <letter> | <digit> ]*
```

Since it takes a different kind of abstract machine to parse the two types of grammars, it makes sense to separate these lower-level functions into a separate module, the lexical scanner, which is built around the idea of a state machine. The idea is to use the simplest parsing technique needed for the job.

There is another, more practical reason for separating scanner from parser. We like to think of the input source file as a stream of characters, which we process right to left without backtracking. In practice that isn't possible. Almost every language has certain keywords such as IF, WHILE, and END. As I mentioned earlier, we can't really know whether a given character string is a keyword, until we've reached the end of it, as defined by a space or other delimiter. So in that sense, we MUST save the string long enough to find out whether we have a keyword or not. That's a limited form of backtracking.

So the structure of a conventional compiler involves splitting up the functions of the lower-level and higher-level parsing. The lexical scanner deals with things at the character level, collecting characters into strings, etc., and passing them along to the parser proper as indivisible tokens. It's also considered normal to let the scanner have the job of identifying keywords.

STATE MACHINES AND ALTERNATIVES

I mentioned that the regular expressions can be parsed using a state machine. In most compiler texts, and indeed in most compilers as well, you will find this taken literally. There is typically a real implementation of the state machine, with integers used to define the current state, and a table of actions to take for each combination of current state and input character. If you write a compiler front end using the popular Unix tools LEX and YACC, that's what you'll get. The output of LEX is a state machine implemented in C, plus a table of actions corresponding to the input grammar given to LEX. The YACC output is similar ... a canned table-driven parser, plus the table corresponding to the language syntax.

That is not the only choice, though. In our previous installments, you have seen over and over that it is possible to implement parsers without dealing specifically with tables, stacks, or state variables. In fact, in Installment V I warned you that if you find yourself needing these things you might be doing something wrong, and not taking advantage of the power of Pascal. There are basically two ways to define a state machine's state: explicitly, with a state number or code, and implicitly, simply by virtue of the fact that I'm at a certain place in the code (if it's Tuesday, this must be Belgium). We've relied heavily on the implicit approaches before, and I think you'll find that they work well here, too.

In practice, it may not even be necessary to HAVE a well-defined lexical scanner. This isn't our first experience at dealing with multi-character tokens. In Installment III, we extended our parser to provide for them, and we didn't even NEED a lexical scanner. That was because in that narrow context, we could always tell, just by looking at the single lookahead character, whether we were dealing with a number, a variable, or an operator. In effect, we built a distributed lexical scanner, using procedures GetName and GetNum.

With keywords present, we can't know anymore what we're dealing with, until the entire token is read. This leads us to a more localized scanner; although, as you will see, the idea of a distributed scanner still has its merits.

SOME EXPERIMENTS IN SCANNING

Before getting back to our compiler, it will be useful to experiment a bit with the general concepts.

Let's begin with the two definitions most often seen in real programming languages:

```
     <ident> ::= <letter> [ <letter> | <digit> ]*
     <number ::= [<digit>]+
```

(Remember, the '*' indicates zero or more occurences of the terms in brackets, and the '+', one or more.)

We have already dealt with similar items in Installment III. Let's begin (as usual) with a bare cradle. Not surprisingly, we are going to need a new recognizer:

```
{--------------------------------------------------------------}
{ Recognize an Alphanumeric Character }

function IsAlNum(c: char): boolean;
begin
   IsAlNum := IsAlpha(c) or IsDigit(c);
end;
{--------------------------------------------------------------}
```

Using this let's write the following two routines, which are very similar to those we've used before:

```
{--------------------------------------------------------------}
{ Get an Identifier }

function GetName: string;
var x: string[8];
begin
   x := '';
   if not IsAlpha(Look) then Expected('Name');
   while IsAlNum(Look) do begin
     x := x + UpCase(Look);
     GetChar;
   end;
   GetName := x;
end;


{--------------------------------------------------------------}
{ Get a Number }

function GetNum: string;
var x: string[16];
begin
   x := '';
   if not IsDigit(Look) then Expected('Integer');
   while IsDigit(Look) do begin
     x := x + Look;
     GetChar;
   end;
   GetNum := x;
end;
{--------------------------------------------------------------}
```

(Notice that this version of GetNum returns a string, not an integer as before.)

You can easily verify that these routines work by calling them from the main program, as in

```
     WriteLn(GetName);
```

This program will print any legal name typed in (maximum eight characters, since that's what we told GetName). It will reject anything else.

Test the other routine similarly.

WHITE SPACE

We also have dealt with embedded white space before, using the two routines IsWhite and SkipWhite. Make sure that these routines are in your current version of the cradle, and add the the line

```
     SkipWhite;
```

at the end of both GetName and GetNum.

Now, let's define the new procedure:

```
{--------------------------------------------------------------}
```

```
{ Lexical Scanner }

Function Scan: string;
begin
   if IsAlpha(Look) then
      Scan := GetName
   else if IsDigit(Look) then
      Scan := GetNum
   else begin
      Scan := Look;
      GetChar;
   end;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

We can call this from the new main program:

```
{--------------------------------------------------------------}
{ Main Program }


begin
   Init;
   repeat
      Token := Scan;
      writeln(Token);
   until Token = CR;
end.
{--------------------------------------------------------------}
```

(You will have to add the declaration of the string Token at the beginning of the program. Make it any convenient length, say 16 characters.)

Now, run the program. Note how the input string is, indeed, separated into distinct tokens.

STATE MACHINES

For the record, a parse routine like GetName does indeed implement a state machine. The state is implicit in the current position in the code. A very useful trick for visualizing what's going on is the syntax diagram, or "railroad-track" diagram. It's a little difficult to draw one in this medium, so I'll use them very sparingly, but the figure below should give you the idea:

```
           |-----> Other---------------------------> Error
           |
   Start -------> Letter ---------------> Other -----> Finish
           ^                        V
           |                        |
           |<----- Letter <---------|
           |                        |
           |<----- Digit  <----------
```

As you can see, this diagram shows how the logic flows as characters are read. Things begin, of course, in the start state, and end when a character other than an alphanumeric is found. If the first character is not alpha, an error occurs.

Otherwise the machine will continue looping until the terminating delimiter is found.

Note that at any point in the flow, our position is entirely dependent on the past history of the input characters. At that point, the action to be taken depends only on the current state, plus the current input character. That's what make this a state machine.

Because of the difficulty of drawing railroad-track diagrams in this medium, I'll continue to stick to syntax equations from now on. But I highly recommend the diagrams to you for anything you do that involves parsing. After a little practice you can begin to see how to write a parser directly from the diagrams. Parallel paths get coded into guarded actions (guarded by IF's or CASE statements), serial paths into sequential calls. It's almost like working from a schematic.

We didn't even discuss SkipWhite, which was introduced earlier, but it also is a simple state machine, as is GetNum. So is their parent procedure, Scan. Little machines make big machines.

The neat thing that I'd like you to note is how painlessly this implicit approach creates these state machines. I personally prefer it a lot over the table-driven approach. It also results is a small, tight, and fast scanner.

NEWLINES

Moving right along, let's modify our scanner to handle more than one line. As I mentioned last time, the most straightforward way to do this is to simply treat the newline characters, carriage return and line feed, as white space. This is, in fact, the way the C standard library routine, iswhite, works. We didn't actually try this before. I'd like to do it now, so you can get a feel for the results.

To do this, simply modify the single executable line of IsWhite to read:

```
   IsWhite := c in [' ', TAB, CR, LF];
```

We need to give the main program a new stop condition, since it will never see a CR. Let's just use:

```
   until Token = '.';
```

OK, compile this program and run it. Try a couple of lines, terminated by the period. I used:

```
     now is the time
     for all good men.
```

Hey, what happened? When I tried it, I didn't get the last token, the period. The program didn't halt. What's more, when I pressed the 'enter' key a few times, I still didn't get the period.

If you're still stuck in your program, you'll find that typing a period on a new line will terminate it.

What's going on here? The answer is that we're hanging up in SkipWhite. A quick look at that routine will show that as long as we're typing null lines, we're going to just continue to loop. After SkipWhite encounters an LF, it tries to execute a GetChar. But since the input buffer is now empty, GetChar's read statement insists on having another line. Procedure Scan gets the terminating period, all right, but it calls SkipWhite to clean up, and SkipWhite won't return until it gets a non-null line.

This kind of behavior is not quite as bad as it seems. In a real compiler, we'd be reading from an input file instead of the console, and as long as we have some procedure for dealing with end-of-files, everything will come out OK. But for reading data from the console, the behavior is just too bizarre. The fact of the matter is that the C/Unix convention is just not compatible with the structure of our parser, which calls for a lookahead character. The code that the Bell wizards have implemented doesn't use that convention, which is why they need 'ungetc'.

OK, let's fix the problem. To do that, we need to go back to the old definition of IsWhite (delete the CR and LF characters) and make use of the procedure Fin that I introduced last time. If it's not in your current version of the cradle, put it there now.

Also, modify the main program to read:

```
{--------------------------------------------------------------}
{ Main Program }


begin
   Init;
   repeat
      Token := Scan;
      writeln(Token);
      if Token = CR then Fin;
   until Token = '.';
end.
{--------------------------------------------------------------}
```

Note the "guard" test preceding the call to Fin. That's what makes the whole thing work, and ensures that we don't try to read a line ahead.

Try the code now. I think you'll like it better.

If you refer to the code we did in the last installment, you'll find that I quietly sprinkled calls to Fin throughout the code, wherever a line break was appropriate. This is one of those areas that really affects the look & feel that I mentioned. At this point I would urge you to experiment with different arrangements and see how you like them. If you want your language to be truly free-field, then newlines should be transparent. In this case, the best approach is to put the following lines at the BEGINNING of Scan:

```
          while Look = CR do
             Fin;
```

If, on the other hand, you want a line-oriented language like Assembler, BASIC, or FORTRAN (or even Ada... note that it has comments terminated by newlines), then you'll need for Scan to return CR's as tokens. It must also eat the trailing LF. The best way to do that is to use this line, again at the beginning of Scan:

```
          if Look = LF then Fin;
```

For other conventions, you'll have to use other arrangements. In my example of the last session, I allowed newlines only at specific places, so I was somewhere in the middle ground. In the rest of these sessions, I'll be picking ways to handle newlines that I happen to like, but I want you to know how to choose other ways for yourselves.

OPERATORS

We could stop now and have a pretty useful scanner for our purposes. In the fragments of KISS that we've built so far, the only tokens that have multiple characters are the identifiers and numbers. All operators were single characters. The only exception I can think of is the relops <=, >=, and <>, but they could be dealt with as special cases.

Still, other languages have multi-character operators, such as the ':=' of Pascal or the '++' and '>>' of C. So while we may not need multi-character operators, it's nice to know how to get them if necessary.

Needless to say, we can handle operators very much the same way as the other tokens. Let's start with a recognizer:

```
{--------------------------------------------------------------}
{ Recognize Any Operator }

function IsOp(c: char): boolean;
begin
   IsOp := c in ['+', '-', '*', '/', '<', '>', ':', '='];
end;
{--------------------------------------------------------------}
```

It's important to note that we DON'T have to include every possible operator in this list. For example, the paretheses aren't included, nor is the terminating period. The current version of Scan handles single-character operators just fine as it is. The list above includes only those characters that can appear in multi-character operators. (For specific languages, of course, the list can always be edited.)

Now, let's modify Scan to read:

```
{--------------------------------------------------------------}
{ Lexical Scanner }

Function Scan: string;
begin
   while Look = CR do
      Fin;
   if IsAlpha(Look) then
      Scan := GetName
   else if IsDigit(Look) then
      Scan := GetNum
   else if IsOp(Look) then
      Scan := GetOp
   else begin
```

```
      Scan := Look;
      GetChar;
   end;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

Try the program now. You will find that any code fragments you care to throw at it will be neatly broken up into individual tokens.

LISTS, COMMAS AND COMMAND LINES

Before getting back to the main thrust of our study, I'd like to get on my soapbox for a moment.

How many times have you worked with a program or operating system that had rigid rules about how you must separate items in a list? (Try, the last time you used MSDOS!) Some programs require spaces as delimiters, and some require commas. Worst of all, some require both, in different places. Most are pretty unforgiving about violations of their rules.

I think this is inexcusable. It's too easy to write a parser that will handle both spaces and commas in a flexible way. Consider the following procedure:

```
{--------------------------------------------------------------}
{ Skip Over a Comma }

procedure SkipComma;
begin
   SkipWhite;
   if Look = ',' then begin
      GetChar;
      SkipWhite;
   end;
end;
{--------------------------------------------------------------}
```

This eight-line procedure will skip over a delimiter consisting of any number (including zero) of spaces, with zero or one comma embedded in the string.

TEMPORARILY, change the call to SkipWhite in Scan to a call to SkipComma, and try inputting some lists. Works nicely, eh? Don't you wish more software authors knew about SkipComma?

For the record, I found that adding the equivalent of SkipComma to my Z80 assembler-language programs took all of 6 (six) extra bytes of code. Even in a 64K machine, that's not a very high price to pay for user-friendliness!

I think you can see where I'm going here. Even if you never write a line of a compiler code in your life, there are places in every program where you can use the concepts of parsing. Any program that processes a command line needs them. In fact, if you think about it for a bit, you'll have to conclude that any time you write a program that processes user inputs, you're defining a language. People communicate with languages, and the syntax implicit in your program defines that language. The real question is: are you going to define it deliberately and explicitly, or just let it turn out to be whatever the program ends up parsing?

I claim that you'll have a better, more user-friendly program if you'll take the time to define the syntax explicitly. Write down the syntax equations or draw the railroad-track diagrams, and code the parser using the techniques I've shown you here. You'll end up with a better program, and it will be easier to write, to boot.

GETTING FANCY

OK, at this point we have a pretty nice lexical scanner that will break an input stream up into tokens. We could use it as it stands and have a servicable compiler. But there are some other aspects of lexical scanning that we need to cover.

The main consideration is <shudder> efficiency. Remember when we were dealing with single-character tokens, every test was a comparison of a single character, Look, with a byte constant. We also used the Case statement heavily.

With the multi-character tokens being returned by Scan, all those tests now become string comparisons. Much slower. And not only slower, but more awkward, since there is no string equivalent of the Case statement in Pascal. It seems especially wasteful to test for what used to be single characters ... the '=', '+', and other operators ... using string comparisons.

Using string comparison is not impossible ... Ron Cain used just that approach in writing Small C. Since we're sticking to the KISS principle here, we would be truly justified in settling for this approach. But then I would have failed to tell you about one of the key approaches used in "real" compilers.

You have to remember: the lexical scanner is going to be called a _LOT_! Once for every token in the whole source program, in fact. Experiments have indicated that the average compiler spends anywhere from 20% to 40% of its time in the scanner routines. If there were ever a place where efficiency deserves real consideration, this is it.

For this reason, most compiler writers ask the lexical scanner to do a little more work, by "tokenizing" the input stream. The idea is to match every token against a list of acceptable keywords and operators, and return unique codes for each one recognized. In the case of ordinary variable names or numbers, we just return a code that says what kind of token they are, and save the actual string somewhere else.

One of the first things we're going to need is a way to identify keywords. We can always do it with successive IF tests, but it surely would be nice if we had a general-purpose routine that could compare a given string with a table of keywords. (By the way, we're also going to need such a routine later, for dealing with symbol tables.) This usually presents a problem in Pascal, because standard Pascal doesn't allow for arrays of variable lengths. It's a real bother to have to declare a different search routine for every table. Standard Pascal also doesn't allow for initializing arrays, so you tend to see code like

```
     Table[1] := 'IF';
     Table[2] := 'ELSE';
     .
     .
     Table[n] := 'END';
```

which can get pretty old if there are many keywords.

Fortunately, Turbo Pascal 4.0 has extensions that eliminate both of these problems. Constant arrays can be declared using TP's "typed constant" facility, and the variable dimensions can be handled with its C-like extensions for pointers.

First, modify your declarations like this:

```
{--------------------------------------------------------------}
{ Type Declarations  }

type Symbol = string[8];

     SymTab = array[1..1000] of Symbol;

     TabPtr = ^SymTab;
{--------------------------------------------------------------}
```

(The dimension used in SymTab is not real ... no storage is allocated by the declaration itself, and the number need only be "big enough.")

Now, just beneath those declarations, add the following:

```
{--------------------------------------------------------------}
{ Definition of Keywords and Token Types }

const KWlist: array [1..4] of Symbol =
              ('IF', 'ELSE', 'ENDIF', 'END');

{--------------------------------------------------------------}
```

Next, insert the following new function:

```
{--------------------------------------------------------------}
{ Table Lookup }

{ If the input string matches a table entry, return the entry
  index.  If not, return a zero.  }

function Lookup(T: TabPtr; s: string; n: integer): integer;
var i: integer;
    found: boolean;
begin
   found := false;
   i := n;
   while (i > 0) and not found do
      if s = T^[i] then
         found := true
      else
         dec(i);
   Lookup := i;
end;
{--------------------------------------------------------------}
```

To test it, you can temporarily change the main program as follows:

```
{--------------------------------------------------------------}
```

```
{ Main Program }


begin
   ReadLn(Token);
   WriteLn(Lookup(Addr(KWList), Token, 4));
end.
{--------------------------------------------------------------}
```

Notice how Lookup is called: The Addr function sets up a pointer to KWList, which gets passed to Lookup.

OK, give this a try. Since we're bypassing Scan here, you'll have to type the keywords in upper case to get any matches.

Now that we can recognize keywords, the next thing is to arrange to return codes for them.

So what kind of code should we return? There are really only two reasonable choices. This seems like an ideal application for the Pascal enumerated type. For example, you can define something like

```
     SymType = (IfSym, ElseSym, EndifSym, EndSym, Ident, Number,
                    Operator);
```

and arrange to return a variable of this type. Let's give it a try. Insert the line above into your type definitions.

Now, add the two variable declarations:

```
    Token: Symtype;          { Current Token  }
    Value: String[16];       { String Token of Look }
```

Modify the scanner to read:

```
{--------------------------------------------------------------}
{ Lexical Scanner }

procedure Scan;
var k: integer;
begin
   while Look = CR do
      Fin;
   if IsAlpha(Look) then begin
      Value := GetName;
      k := Lookup(Addr(KWlist), Value, 4);
      if k = 0 then
         Token := Ident
      else
         Token := SymType(k - 1);
      end
   else if IsDigit(Look) then begin
      Value := GetNum;
      Token := Number;
      end
   else if IsOp(Look) then begin
      Value := GetOp;
      Token := Operator;
      end
   else begin
      Value := Look;
      Token := Operator;
```

```
      GetChar;
   end;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

(Notice that Scan is now a procedure, not a function.)

Finally, modify the main program to read:

```
{--------------------------------------------------------------}
{ Main Program }

begin
   Init;
   repeat
      Scan;
      case Token of
        Ident: write('Ident ');
        Number: Write('Number ');
        Operator: Write('Operator ');
        IfSym, ElseSym, EndifSym, EndSym: Write('Keyword ');
      end;
      Writeln(Value);
   until Token = EndSym;
end.
{--------------------------------------------------------------}
```

What we've done here is to replace the string Token used earlier with an enumerated type. Scan returns the type in variable Token, and returns the string itself in the new variable Value.

OK, compile this and give it a whirl. If everything goes right, you should see that we are now recognizing keywords.

What we have now is working right, and it was easy to generate from what we had earlier. However, it still seems a little "busy" to me. We can simplify things a bit by letting GetName, GetNum, GetOp, and Scan be procedures working with the global variables Token and Value, thereby eliminating the local copies. It also seems a little cleaner to move the table lookup into GetName. The new form for the four procedures is, then:

```
{--------------------------------------------------------------}
{ Get an Identifier }

procedure GetName;
var k: integer;
begin
   Value := '';
   if not IsAlpha(Look) then Expected('Name');
   while IsAlNum(Look) do begin
     Value := Value + UpCase(Look);
     GetChar;
   end;
   k := Lookup(Addr(KWlist), Value, 4);
   if k = 0 then
      Token := Ident
   else
      Token := SymType(k-1);
end;
```

```
{--------------------------------------------------------------}
{ Get a Number }

procedure GetNum;
begin
   Value := '';
   if not IsDigit(Look) then Expected('Integer');
   while IsDigit(Look) do begin
     Value := Value + Look;
     GetChar;
   end;
   Token := Number;
end;


{--------------------------------------------------------------}
{ Get an Operator }

procedure GetOp;
begin
   Value := '';
   if not IsOp(Look) then Expected('Operator');
   while IsOp(Look) do begin
     Value := Value + Look;
     GetChar;
   end;
   Token := Operator;
end;


{--------------------------------------------------------------}
{ Lexical Scanner }

procedure Scan;
var k: integer;
begin
   while Look = CR do
      Fin;
   if IsAlpha(Look) then
      GetName
   else if IsDigit(Look) then
      GetNum
   else if IsOp(Look) then
      GetOp
   else begin
      Value := Look;
      Token := Operator;
      GetChar;
   end;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

RETURNING A CHARACTER

Essentially every scanner I've ever seen that was written in Pascal used the mechanism of an enumerated type that I've just described. It is certainly a workable mechanism, but it doesn't seem the simplest approach to me.

For one thing, the list of possible symbol types can get pretty long. Here, I've used just one symbol, "Operator," to stand for all of the operators, but I've seen other designs that actually return different codes for each one.

There is, of course, another simple type that can be returned as a code: the character. Instead of returning the enumeration value 'Operator' for a '+' sign, what's wrong with just returning the character itself? A character is just as good a variable for encoding the different token types, it can be used in case statements easily, and it's sure a lot easier to type. What could be simpler?

Besides, we've already had experience with the idea of encoding keywords as single characters. Our previous programs are already written that way, so using this approach will minimize the changes to what we've already done.

Some of you may feel that this idea of returning character codes is too mickey-mouse. I must admit it gets a little awkward for multi-character operators like '<='. If you choose to stay with the enumerated type, fine. For the rest, I'd like to show you how to change what we've done above to support that approach.

First, you can delete the SymType declaration now ... we won't be needing that. And you can change the type of Token to char.

Next, to replace SymType, add the following constant string:

```
   const KWcode: string[5] = 'xilee';
```

(I'll be encoding all idents with the single character 'x'.)

Lastly, modify Scan and its relatives as follows:

```
{--------------------------------------------------------------}
{ Get an Identifier }

procedure GetName;
begin
   Value := '';
   if not IsAlpha(Look) then Expected('Name');
   while IsAlNum(Look) do begin
     Value := Value + UpCase(Look);
     GetChar;
   end;
   Token := KWcode[Lookup(Addr(KWlist), Value, 4) + 1];
end;


{--------------------------------------------------------------}
{ Get a Number }

procedure GetNum;
begin
   Value := '';
   if not IsDigit(Look) then Expected('Integer');
   while IsDigit(Look) do begin
     Value := Value + Look;
     GetChar;
   end;
   Token := '#';
end;


{--------------------------------------------------------------}
{ Get an Operator }

procedure GetOp;
```

```
begin
   Value := '';
   if not IsOp(Look) then Expected('Operator');
   while IsOp(Look) do begin
     Value := Value + Look;
     GetChar;
   end;
   if Length(Value) = 1 then
      Token := Value[1]
   else
      Token := '?';
end;


{--------------------------------------------------------------}
{ Lexical Scanner }

procedure Scan;
var k: integer;
begin
   while Look = CR do
      Fin;
   if IsAlpha(Look) then
      GetName
   else if IsDigit(Look) then
      GetNum
   else if IsOp(Look) then begin
      GetOp
   else begin
      Value := Look;
      Token := '?';
      GetChar;
   end;
   SkipWhite;
end;


{--------------------------------------------------------------}
{ Main Program }


begin
   Init;
   repeat
      Scan;
      case Token of
        'x': write('Ident ');
        '#': Write('Number ');
        'i', 'l', 'e': Write('Keyword ');
        else Write('Operator ');
      end;
      Writeln(Value);
   until Value = 'END';
end.
{--------------------------------------------------------------}
```

This program should work the same as the previous version. A minor difference in structure, maybe, but it seems more straightforward to me.

DISTRIBUTED vs CENTRALIZED SCANNERS

The structure for the lexical scanner that I've just shown you is very conventional, and about 99% of all compilers use something very close to it. This is not, however, the only possible structure, or even always the best one.

The problem with the conventional approach is that the scanner has no knowledge of context. For example, it can't distinguish between the assignment operator '=' and the relational operator '=' (perhaps that's why both C and Pascal use different strings for the two). All the scanner can do is to pass the operator along to the parser, which can hopefully tell from the context which operator is meant. Similarly, a keyword like 'IF' has no place in the middle of a math expression, but if one happens to appear there, the scanner will see no problem with it, and will return it to the parser, properly encoded as an 'IF'.

With this kind of approach, we are not really using all the information at our disposal. In the middle of an expression, for example, the parser "knows" that there is no need to look for keywords, but it has no way of telling the scanner that. So the scanner continues to do so. This, of course, slows down the compilation.

In real-world compilers, the designers often arrange for more information to be passed between parser and scanner, just to avoid this kind of problem. But that can get awkward, and certainly destroys a lot of the modularity of the structure.

The alternative is to seek some way to use the contextual information that comes from knowing where we are in the parser. This leads us back to the notion of a distributed scanner, in which various portions of the scanner are called depending upon the context.

In KISS, as in most languages, keywords ONLY appear at the beginning of a statement. In places like expressions, they are not allowed. Also, with one minor exception (the multi-character relops) that is easily handled, all operators are single characters, which means that we don't need GetOp at all.

So it turns out that even with multi-character tokens, we can still always tell from the current lookahead character exactly what kind of token is coming, except at the very beginning of a statement.

Even at that point, the ONLY kind of token we can accept is an identifier. We need only to determine if that identifier is a keyword or the target of an assignment statement.

We end up, then, still needing only GetName and GetNum, which are used very much as we've used them in earlier installments.

It may seem at first to you that this is a step backwards, and a rather primitive approach. In fact, it is an improvement over the classical scanner, since we're using the scanning routines only where they're really needed. In places where keywords are not allowed, we don't slow things down by looking for them.

MERGING SCANNER AND PARSER

Now that we've covered all of the theory and general aspects of lexical scanning that we'll be needing, I'm FINALLY ready to back up my claim that we can accomodate multi-character tokens with minimal change to our previous work. To keep things short and simple I will restrict myself here to a subset of what we've done before; I'm allowing only one control construct (the IF) and no Boolean expressions. That's enough to demonstrate the parsing of both keywords and expressions. The extension to the full set of constructs should be pretty apparent from what we've already done.

All the elements of the program to parse this subset, using single-character tokens, exist already in our previous programs. I built it by judicious copying of these files, but I wouldn't dare try to lead you through that process. Instead, to avoid any confusion, the whole program is shown below:

```
{--------------------------------------------------------------}
program KISS;

{--------------------------------------------------------------}
{ Constant Declarations }

const TAB = ^I;
      CR  = ^M;
      LF  = ^J;

{--------------------------------------------------------------}
{ Type Declarations  }

type Symbol = string[8];

     SymTab = array[1..1000] of Symbol;

     TabPtr = ^SymTab;


{--------------------------------------------------------------}
{ Variable Declarations }

var Look  : char;              { Lookahead Character }
    Lcount: integer;           { Label Counter       }


{--------------------------------------------------------------}
{ Read New Character From Input Stream }

procedure GetChar;
begin
   Read(Look);
end;


{--------------------------------------------------------------}
{ Report an Error }

procedure Error(s: string);
begin
   WriteLn;
   WriteLn(^G, 'Error: ', s, '.');
end;


{--------------------------------------------------------------}
{ Report Error and Halt }

procedure Abort(s: string);
begin
   Error(s);
   Halt;
end;


{--------------------------------------------------------------}
{ Report What Was Expected }
```

```
procedure Expected(s: string);
begin
   Abort(s + ' Expected');
end;

{--------------------------------------------------------------}
{ Recognize an Alpha Character }

function IsAlpha(c: char): boolean;
begin
   IsAlpha := UpCase(c) in ['A'..'Z'];
end;


{--------------------------------------------------------------}
{ Recognize a Decimal Digit }

function IsDigit(c: char): boolean;
begin
   IsDigit := c in ['0'..'9'];
end;


{--------------------------------------------------------------}
{ Recognize an AlphaNumeric Character }

function IsAlNum(c: char): boolean;
begin
   IsAlNum := IsAlpha(c) or IsDigit(c);
end;

{--------------------------------------------------------------}
{ Recognize an Addop }

function IsAddop(c: char): boolean;
begin
   IsAddop := c in ['+', '-'];
end;


{--------------------------------------------------------------}
{ Recognize a Mulop }

function IsMulop(c: char): boolean;
begin
   IsMulop := c in ['*', '/'];
end;


{--------------------------------------------------------------}
{ Recognize White Space }

function IsWhite(c: char): boolean;
begin
   IsWhite := c in [' ', TAB];
end;


{--------------------------------------------------------------}
{ Skip Over Leading White Space }

procedure SkipWhite;
begin
   while IsWhite(Look) do
      GetChar;
end;
```

```
{--------------------------------------------------------------}
{ Match a Specific Input Character }

procedure Match(x: char);
begin
   if Look <> x then Expected('''' + x + '''');
   GetChar;
   SkipWhite;
end;


{--------------------------------------------------------------}
{ Skip a CRLF }

procedure Fin;
begin
   if Look = CR then GetChar;
   if Look = LF then GetChar;
   SkipWhite;
end;


{--------------------------------------------------------------}
{ Get an Identifier }

function GetName: char;
begin
   while Look = CR do
      Fin;
   if not IsAlpha(Look) then Expected('Name');
   Getname := UpCase(Look);
   GetChar;
   SkipWhite;
end;


{--------------------------------------------------------------}
{ Get a Number }

function GetNum: char;
begin
   if not IsDigit(Look) then Expected('Integer');
   GetNum := Look;
   GetChar;
   SkipWhite;
end;


{--------------------------------------------------------------}
{ Generate a Unique Label }

function NewLabel: string;
var S: string;
begin
   Str(LCount, S);
   NewLabel := 'L' + S;
   Inc(LCount);
end;


{--------------------------------------------------------------}
{ Post a Label To Output }

procedure PostLabel(L: string);
begin
```

```
   WriteLn(L, ':');
end;


{--------------------------------------------------------------}
{ Output a String with Tab }

procedure Emit(s: string);
begin
   Write(TAB, s);
end;


{--------------------------------------------------------------}

{ Output a String with Tab and CRLF }

procedure EmitLn(s: string);
begin
   Emit(s);
   WriteLn;
end;


{---------------------------------------------------------------}
{ Parse and Translate an Identifier }

procedure Ident;
var Name: char;
begin
   Name := GetName;
   if Look = '(' then begin
      Match('(');
      Match(')');
      EmitLn('BSR ' + Name);
      end
   else
      EmitLn('MOVE ' + Name + '(PC),D0');
end;


{---------------------------------------------------------------}
{ Parse and Translate a Math Factor }

procedure Expression; Forward;

procedure Factor;
begin
   if Look = '(' then begin
      Match('(');
      Expression;
      Match(')');
      end
   else if IsAlpha(Look) then
      Ident
   else
      EmitLn('MOVE #' + GetNum + ',D0');
end;


{---------------------------------------------------------------}
{ Parse and Translate the First Math Factor }


procedure SignedFactor;
var s: boolean;
begin
```

```
   s := Look = '-';
   if IsAddop(Look) then begin
      GetChar;
      SkipWhite;
   end;
   Factor;
   if s then
      EmitLn('NEG D0');
end;


{---------------------------------------------------------------}
{ Recognize and Translate a Multiply }

procedure Multiply;
begin
   Match('*');
   Factor;
   EmitLn('MULS (SP)+,D0');
end;


{--------------------------------------------------------------}
{ Recognize and Translate a Divide }

procedure Divide;
begin
   Match('/');
   Factor;
   EmitLn('MOVE (SP)+,D1');
   EmitLn('EXS.L D0');
   EmitLn('DIVS D1,D0');
end;


{---------------------------------------------------------------}
{ Completion of Term Processing  (called by Term and FirstTerm }

procedure Term1;
begin
   while IsMulop(Look) do begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '*': Multiply;
       '/': Divide;
      end;
   end;
end;


{---------------------------------------------------------------}
{ Parse and Translate a Math Term }

procedure Term;
begin
   Factor;
   Term1;
end;


{---------------------------------------------------------------}
{ Parse and Translate a Math Term with Possible Leading Sign }

procedure FirstTerm;
begin
   SignedFactor;
   Term1;
```

```
end;


{--------------------------------------------------------------}
{ Recognize and Translate an Add }

procedure Add;
begin
   Match('+');
   Term;
   EmitLn('ADD (SP)+,D0');
end;


{--------------------------------------------------------------}
{ Recognize and Translate a Subtract }

procedure Subtract;
begin
   Match('-');
   Term;
   EmitLn('SUB (SP)+,D0');
   EmitLn('NEG D0');
end;


{--------------------------------------------------------------}
{ Parse and Translate an Expression }

procedure Expression;
begin
   FirstTerm;
   while IsAddop(Look) do begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '+': Add;
       '-': Subtract;
      end;
   end;
end;


{--------------------------------------------------------------}
{ Parse and Translate a Boolean Condition }
{ This version is a dummy }

Procedure Condition;
begin
   EmitLn('Condition');
end;


{--------------------------------------------------------------}
{ Recognize and Translate an IF Construct }

procedure Block;
 Forward;

procedure DoIf;
var L1, L2: string;
begin
   Match('i');
   Condition;
   L1 := NewLabel;
   L2 := L1;
   EmitLn('BEQ ' + L1);
   Block;
```

```
   if Look = 'l' then begin
      Match('l');
      L2 := NewLabel;
      EmitLn('BRA ' + L2);
      PostLabel(L1);
      Block;
   end;
   PostLabel(L2);
   Match('e');
end;


{--------------------------------------------------------------}
{ Parse and Translate an Assignment Statement }

procedure Assignment;
var Name: char;
begin
   Name := GetName;
   Match('=');
   Expression;
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)');
end;


{--------------------------------------------------------------}
{ Recognize and Translate a Statement Block }

procedure Block;
begin
   while not(Look in ['e', 'l']) do begin
      case Look of
       'i': DoIf;
       CR: while Look = CR do
              Fin;
       else Assignment;
      end;
   end;
end;


{--------------------------------------------------------------}
{ Parse and Translate a Program }

procedure DoProgram;
begin
   Block;
   if Look <> 'e' then Expected('END');
   EmitLn('END')
end;


{--------------------------------------------------------------}

{ Initialize }

procedure Init;
begin
   LCount := 0;
   GetChar;
end;


{--------------------------------------------------------------}
{ Main Program }
```

```
begin
   Init;
   DoProgram;
end.
{--------------------------------------------------------------}
```

A couple of comments:

(1) The form for the expression parser, using FirstTerm, etc., is a little different from what you've seen before. It's yet another variation on the same theme. Don't let it throw you ... the change is not required for what follows.

(2) Note that, as usual, I had to add calls to Fin at strategic spots to allow for multiple lines.

Before we proceed to adding the scanner, first copy this file and verify that it does indeed parse things correctly. Don't forget the "codes": 'i' for IF, 'l' for ELSE, and 'e' for END or ENDIF.

If the program works, then let's press on. In adding the scanner modules to the program, it helps to have a systematic plan. In all the parsers we've written to date, we've stuck to a convention that the current lookahead character should always be a non-blank character. We preload the lookahead character in Init, and keep the "pump primed" after that. To keep the thing working right at newlines, we had to modify this a bit and treat the newline as a legal token.

In the multi-character version, the rule is similar: The current lookahead character should always be left at the BEGINNING of the next token, or at a newline.

The multi-character version is shown next. To get it, I've made the following changes:

- o Added the variables Token and Value, and the type definitions needed by Lookup.
- o Added the definitions of KWList and KWcode.
- o Added Lookup.
- o Replaced GetName and GetNum by their multi-character versions. (Note that the call to Lookup has been moved out of GetName, so that it will not be executed for calls within an expression.)
- o Created a new, vestigial Scan that calls GetName, then scans for keywords.
- o Created a new procedure, MatchString, that looks for a specific keyword. Note that, unlike Match, MatchString does NOT read the next keyword.
- o Modified Block to call Scan.
- o Changed the calls to Fin a bit. Fin is now called within GetName.

Here is the program in its entirety:

```
{--------------------------------------------------------------}
program KISS;
```

```
{--------------------------------------------------------------}
{ Constant Declarations }

const TAB = ^I;
      CR  = ^M;
      LF  = ^J;

{--------------------------------------------------------------}
{ Type Declarations  }

type Symbol = string[8];

     SymTab = array[1..1000] of Symbol;

     TabPtr = ^SymTab;


{--------------------------------------------------------------}
{ Variable Declarations }

var Look  : char;              { Lookahead Character }
    Token : char;              { Encoded Token       }
    Value : string[16];        { Unencoded Token     }
    Lcount: integer;           { Label Counter       }


{--------------------------------------------------------------}
{ Definition of Keywords and Token Types }

const KWlist: array [1..4] of Symbol =
              ('IF', 'ELSE', 'ENDIF', 'END');

const KWcode: string[5] = 'xilee';


{--------------------------------------------------------------}
{ Read New Character From Input Stream }

procedure GetChar;
begin
   Read(Look);
end;

{--------------------------------------------------------------}
{ Report an Error }

procedure Error(s: string);
begin
   WriteLn;
   WriteLn(^G, 'Error: ', s, '.');
end;


{--------------------------------------------------------------}
{ Report Error and Halt }

procedure Abort(s: string);
begin
   Error(s);
   Halt;
end;


{--------------------------------------------------------------}
{ Report What Was Expected }
```

```
procedure Expected(s: string);
begin
   Abort(s + ' Expected');
end;

{--------------------------------------------------------------}
{ Recognize an Alpha Character }

function IsAlpha(c: char): boolean;
begin
   IsAlpha := UpCase(c) in ['A'..'Z'];
end;


{--------------------------------------------------------------}
{ Recognize a Decimal Digit }

function IsDigit(c: char): boolean;
begin
   IsDigit := c in ['0'..'9'];
end;


{--------------------------------------------------------------}
{ Recognize an AlphaNumeric Character }

function IsAlNum(c: char): boolean;
begin
   IsAlNum := IsAlpha(c) or IsDigit(c);
end;


{--------------------------------------------------------------}
{ Recognize an Addop }

function IsAddop(c: char): boolean;
begin
   IsAddop := c in ['+', '-'];
end;


{--------------------------------------------------------------}
{ Recognize a Mulop }

function IsMulop(c: char): boolean;
begin
   IsMulop := c in ['*', '/'];
end;


{--------------------------------------------------------------}
{ Recognize White Space }

function IsWhite(c: char): boolean;
begin
   IsWhite := c in [' ', TAB];
end;


{--------------------------------------------------------------}
{ Skip Over Leading White Space }

procedure SkipWhite;
begin
   while IsWhite(Look) do
      GetChar;
end;
```

```
{--------------------------------------------------------------}
{ Match a Specific Input Character }

procedure Match(x: char);
begin
   if Look <> x then Expected('''' + x + '''');
   GetChar;
   SkipWhite;
end;


{--------------------------------------------------------------}
{ Skip a CRLF }

procedure Fin;
begin
   if Look = CR then GetChar;
   if Look = LF then GetChar;
   SkipWhite;
end;


{--------------------------------------------------------------}
{ Table Lookup }

function Lookup(T: TabPtr; s: string; n: integer): integer;
var i: integer;
    found: boolean;
begin
   found := false;
   i := n;
   while (i > 0) and not found do
      if s = T^[i] then
         found := true
      else
         dec(i);
   Lookup := i;
end;


{--------------------------------------------------------------}
{ Get an Identifier }

procedure GetName;
begin
   while Look = CR do
      Fin;
   if not IsAlpha(Look) then Expected('Name');
   Value := '';
   while IsAlNum(Look) do begin
     Value := Value + UpCase(Look);
     GetChar;
   end;
   SkipWhite;
end;


{--------------------------------------------------------------}
{ Get a Number }

procedure GetNum;
begin
   if not IsDigit(Look) then Expected('Integer');
   Value := '';
   while IsDigit(Look) do begin
```

```
     Value := Value + Look;
     GetChar;
   end;
   Token := '#';
   SkipWhite;
end;


{--------------------------------------------------------------}
{ Get an Identifier and Scan it for Keywords }

procedure Scan;
begin
   GetName;
   Token := KWcode[Lookup(Addr(KWlist), Value, 4) + 1];
end;


{--------------------------------------------------------------}
{ Match a Specific Input String }

procedure MatchString(x: string);
begin
   if Value <> x then Expected('''' + x + '''');
end;


{--------------------------------------------------------------}
{ Generate a Unique Label }

function NewLabel: string;
var S: string;
begin
   Str(LCount, S);
   NewLabel := 'L' + S;
   Inc(LCount);
end;


{--------------------------------------------------------------}
{ Post a Label To Output }

procedure PostLabel(L: string);
begin
   WriteLn(L, ':');
end;


{--------------------------------------------------------------}
{ Output a String with Tab }

procedure Emit(s: string);
begin
   Write(TAB, s);
end;


{--------------------------------------------------------------}
{ Output a String with Tab and CRLF }

procedure EmitLn(s: string);
begin
   Emit(s);
   WriteLn;
end;
```

```
{--------------------------------------------------------------}
{ Parse and Translate an Identifier }

procedure Ident;
begin
   GetName;
   if Look = '(' then begin
      Match('(');
      Match(')');
      EmitLn('BSR ' + Value);
      end
   else
      EmitLn('MOVE ' + Value + '(PC),D0');
end;


{--------------------------------------------------------------}
{ Parse and Translate a Math Factor }

procedure Expression; Forward;

procedure Factor;
begin
   if Look = '(' then begin
      Match('(');
      Expression;
      Match(')');
      end
   else if IsAlpha(Look) then
      Ident
   else begin
      GetNum;
      EmitLn('MOVE #' + Value + ',D0');
   end;
end;


{--------------------------------------------------------------}
{ Parse and Translate the First Math Factor }

procedure SignedFactor;
var s: boolean;
begin
   s := Look = '-';
   if IsAddop(Look) then begin
      GetChar;
      SkipWhite;
   end;
   Factor;
   if s then
      EmitLn('NEG D0');
end;


{-------------------------------------------------------------}
{ Recognize and Translate a Multiply }

procedure Multiply;
begin
   Match('*');
   Factor;
   EmitLn('MULS (SP)+,D0');
end;


{------------------------------------------------------------}
{ Recognize and Translate a Divide }
```

```
procedure Divide;
begin
   Match('/');
   Factor;
   EmitLn('MOVE (SP)+,D1');
   EmitLn('EXS.L D0');
   EmitLn('DIVS D1,D0');
end;


{---------------------------------------------------------------}
{ Completion of Term Processing  (called by Term and FirstTerm }

procedure Term1;
begin
   while IsMulop(Look) do begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '*': Multiply;
       '/': Divide;
      end;
   end;
end;
{---------------------------------------------------------------}
{ Parse and Translate a Math Term }

procedure Term;
begin
   Factor;
   Term1;
end;


{---------------------------------------------------------------}
{ Parse and Translate a Math Term with Possible Leading Sign }

procedure FirstTerm;
begin
   SignedFactor;
   Term1;
end;


{---------------------------------------------------------------}
{ Recognize and Translate an Add }

procedure Add;
begin
   Match('+');
   Term;
   EmitLn('ADD (SP)+,D0');
end;


{---------------------------------------------------------------}
{ Recognize and Translate a Subtract }

procedure Subtract;
begin
   Match('-');
   Term;
   EmitLn('SUB (SP)+,D0');
   EmitLn('NEG D0');
end;
```

```
{--------------------------------------------------------------}
{ Parse and Translate an Expression }

procedure Expression;
begin
   FirstTerm;
   while IsAddop(Look) do begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '+': Add;
       '-': Subtract;
      end;
   end;
end;


{--------------------------------------------------------------}
{ Parse and Translate a Boolean Condition }
{ This version is a dummy }

Procedure Condition;
begin
   EmitLn('Condition');
end;


{--------------------------------------------------------------}
{ Recognize and Translate an IF Construct }

procedure Block; Forward;


procedure DoIf;
var L1, L2: string;
begin
   Condition;
   L1 := NewLabel;
   L2 := L1;
   EmitLn('BEQ ' + L1);
   Block;
   if Token = 'l' then begin
      L2 := NewLabel;
      EmitLn('BRA ' + L2);
      PostLabel(L1);
      Block;
   end;
   PostLabel(L2);
   MatchString('ENDIF');
end;


{-------------------------------------------------------------}
{ Parse and Translate an Assignment Statement }

procedure Assignment;
var Name: string;
begin
   Name := Value;
   Match('=');
   Expression;
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)');
end;


{-------------------------------------------------------------}
{ Recognize and Translate a Statement Block }
```

```
procedure Block;
begin
   Scan;
   while not (Token in ['e', 'l']) do begin
      case Token of
       'i': DoIf;
       else Assignment;
      end;
      Scan;
   end;
end;


{--------------------------------------------------------------}

{ Parse and Translate a Program }

procedure DoProgram;
begin
   Block;
   MatchString('END');
   EmitLn('END')
end;


{--------------------------------------------------------------}

{ Initialize }

procedure Init;
begin
   LCount := 0;
   GetChar;
end;


{--------------------------------------------------------------}
{ Main Program }

begin
   Init;
   DoProgram;
end.
{--------------------------------------------------------------}
```

Compare this program with its single-character counterpart. I think you will agree that the differences are minor.

CONCLUSION

At this point, you have learned how to parse and generate code for expressions, Boolean expressions, and control structures. You have now learned how to develop lexical scanners, and how to incorporate their elements into a translator. You have still not seen ALL the elements combined into one program, but on the basis of what we've done before you should find it a straightforward matter to extend our earlier programs to include scanners.

We are very close to having all the elements that we need to build a real, functional compiler. There are still a few things missing, notably procedure calls and type definitions. We will deal with those in the next few sessions. Before doing so, however, I thought it would be fun to turn the translator above into a true compiler. That's what we'll be doing in the next installment.

Up till now, we've taken a rather bottom-up approach to parsing, beginning with low-level constructs and working our way up. In the next installment, I'll also be taking a look from the top down, and we'll discuss how the structure of the translator is altered by changes in the language definition.

See you then.