LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

3 June 1989

Part XI: LEXICAL SCAN REVISITED

INTRODUCTION

I've got some good news and some bad news. The bad news is that this installment is not the one I promised last time. What's more, the one after this one won't be, either.

The good news is the reason for this installment: I've found a way to simplify and improve the lexical scanning part of the compiler. Let me explain.

BACKGROUND

If you'll remember, we talked at length about the subject of lexical scanners in Part VII, and I left you with a design for a distributed scanner that I felt was about as simple as I could make it ... more than most that I've seen elsewhere. We used that idea in Part X. The compiler structure that resulted was simple, and it got the job done.

Recently, though, I've begun to have problems, and they're the kind that send a message that you might be doing something wrong.

The whole thing came to a head when I tried to address the issue of semicolons. Several people have asked me about them, and whether or not KISS will have them separating the statements. My intention has been NOT to use semicolons, simply because I don't like them and, as you can see, they have not proved necessary.

But I know that many of you, like me, have gotten used to them, and so I set out to write a short installment to show you how they could easily be added, if you were so inclined.

Well, it turned out that they weren't easy to add at all. In fact it was darned difficult.

I guess I should have realized that something was wrong, because of the issue of newlines. In the last couple of installments we've addressed that issue, and I've shown you how to deal with newlines with a procedure called, appropriately enough, NewLine. In TINY Version 1.0, I sprinkled calls to this procedure in strategic spots in the code.

It seems that every time I've addressed the issue of newlines, though, I've found it to be tricky, and the resulting parser turned out to be quite fragile ... one addition or deletion here or there and things tended to go to pot. Looking back on it, I realize that there was a message in this that I just wasn't paying attention to.

When I tried to add semicolons on top of the newlines, that was the last straw. I ended up with much too complex a solution. I began to realize that something fundamental had to change.

So, in a way this installment will cause us to backtrack a bit and revisit the issue of scanning all over again. Sorry about that. That's the price you pay for watching me do this in real time. But the new version is definitely an improvement, and will serve us well for what is to come.

As I said, the scanner we used in Part X was about as simple as one can get. But anything can be improved. The new scanner is more like the classical scanner, and not as simple as before. But the overall compiler structure is even simpler than before. It's also more robust, and easier to add to and/or modify. I think that's worth the time spent in this digression. So in this installment, I'll be showing you the new structure. No doubt you'll be happy to know that, while the changes affect many procedures, they aren't very profound and so we lose very little of what's been done so far.

Ironically, the new scanner is much more conventional than the old one, and is very much like the more generic scanner I showed you earlier in Part VII. Then I started trying to get clever, and I almost clevered myself clean out of business. You'd think one day I'd learn: K-I-S-S!

THE PROBLEM

The problem begins to show itself in procedure Block, which I've reproduced below:

```
{--------------------------------------------------------------}
{ Parse and Translate a Block of Statements }
```

```
procedure Block;
begin
   Scan;
   while not(Token in ['e', 'l']) do begin
      case Token of
       'i': DoIf;
       'w': DoWhile;
       'R': DoRead;
       'W': DoWrite;
      else Assignment;
      end;
      Scan;
   end;
end;
{--------------------------------------------------------------}
```

As you can see, Block is oriented to individual program statements. At each pass through the loop, we know that we are at the beginning of a statement. We exit the block when we have scanned an END or an ELSE.

But suppose that we see a semicolon instead. The procedure as it's shown above can't handle that, because procedure Scan only expects and can only accept tokens that begin with a letter.

I tinkered around for quite awhile to come up with a fix. I found many possible approaches, but none were very satisfying. I finally figured out the reason.

Recall that when we started with our single-character parsers, we adopted a convention that the lookahead character would always be prefetched. That is, we would have the character that corresponds to our current position in the input stream fetched into the global character Look, so that we could examine it as many times as needed. The rule we adopted was that EVERY recognizer, if it found its target token, would advance Look to the next character in the input stream.

That simple and fixed convention served us very well when we had single-character tokens, and it still does. It would make a lot of sense to apply the same rule to multi-character tokens.

But when we got into lexical scanning, I began to violate that simple rule. The scanner of Part X did indeed advance to the next token if it found an identifier or keyword, but it DIDN'T do that if it found a carriage return, a whitespace character, or an operator.

Now, that sort of mixed-mode operation gets us into deep trouble in procedure Block, because whether or not the input stream has been advanced depends upon the kind of token we encounter. If it's a keyword or the target of an assignment statement, the "cursor," as defined by the contents of Look, has been advanced to the next token OR to the beginning of whitespace. If, on the other hand, the token is a semicolon, or if we have hit a carriage return, the cursor has NOT advanced.

Needless to say, we can add enough logic to keep us on track. But it's tricky, and makes the whole parser very fragile.

There's a much better way, and that's just to adopt that same rule that's worked so well before, to apply to TOKENS as well as single characters. In other words, we'll prefetch tokens just as we've always done for characters. It seems so obvious once you think about it that way.

Interestingly enough, if we do things this way the problem that
we've had with newline characters goes away. We can just lump
them in as whitespace characters, which means that the handling
of newlines becomes very trivial, and MUCH less prone to error
than we've had to deal with in the past.

THE SOLUTION

Let's begin to fix the problem by re-introducing the two
procedures:

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

procedure GetName;
begin
   SkipWhite;
   if Not IsAlpha(Look) then Expected('Identifier');
   Token := 'x';
   Value := '';
   repeat
      Value := Value + UpCase(Look);
      GetChar;
   until not IsAlNum(Look);
end;


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

procedure GetNum;
begin
   SkipWhite;
   if not IsDigit(Look) then Expected('Number');
   Token := '#';
   Value := '';
   repeat
      Value := Value + Look;
      GetChar;
   until not IsDigit(Look);
end;
{--------------------------------------------------------------}
```

These two procedures are functionally almost identical to the
ones I showed you in Part VII. They each fetch the current
token, either an identifier or a number, into the global string
Value.   They also set the encoded version, Token, to the
appropriate code. The input stream is left with Look containing
the first character NOT part of the token.

We can do the same thing for operators, even multi-character
operators, with a procedure such as:

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

procedure GetOp;
begin
   Token := Look;
   Value := '';
   repeat
      Value := Value + Look;
      GetChar;
   until IsAlpha(Look) or IsDigit(Look) or IsWhite(Look);
end;
{--------------------------------------------------------------}
```

Note that GetOp returns, as its encoded token, the FIRST character of the operator. This is important, because it means that we can now use that single character to drive the parser, instead of the lookahead character.

We need to tie these procedures together into a single procedure that can handle all three cases. The following procedure will read any one of the token types and always leave the input stream advanced beyond it:

```
{--------------------------------------------------------------}
{ Get the Next Input Token }

procedure Next;
begin
   SkipWhite;
   if IsAlpha(Look) then GetName
   else if IsDigit(Look) then GetNum
   else GetOp;
end;
{--------------------------------------------------------------}
```

***NOTE that here I have put SkipWhite BEFORE the calls rather than after. This means that, in general, the variable Look will NOT have a meaningful value in it, and therefore we should NOT use it as a test value for parsing, as we have been doing so far. That's the big departure from our normal approach.

Now, remember that before I was careful not to treat the carriage return (CR) and line feed (LF) characters as white space. This was because, with SkipWhite called as the last thing in the scanner, the encounter with LF would trigger a read statement. If we were on the last line of the program, we couldn't get out until we input another line with a non-white character. That's why I needed the second procedure, NewLine, to handle the CRLF's.

But now, with the call to SkipWhite coming first, that's exactly the behavior we want. The compiler must know there's another token coming or it wouldn't be calling Next. In other words, it hasn't found the terminating END yet. So we're going to insist on more data until we find something.

All this means that we can greatly simplify both the program and the concepts, by treating CR and LF as whitespace characters, and eliminating NewLine. You can do that simply by modifying the function IsWhite:

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

function IsWhite(c: char): boolean;
begin
   IsWhite := c in [' ', TAB, CR, LF];
end;
{--------------------------------------------------------------}
```

We've already tried similar routines in Part VII, but you might as well try these new ones out. Add them to a copy of the Cradle and call Next with the following main program:

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

begin
   Init;
   repeat
      Next;
      WriteLn(Token, ' ', Value);
   until Token = '.';
end.
{--------------------------------------------------------------}
```

Compile it and verify that you can separate a program into a series of tokens, and that you get the right encoding for each token.

This ALMOST works, but not quite. There are two potential problems: First, in KISS/TINY almost all of our operators are single-character operators. The only exceptions are the relops >=, <=, and <>. It seems a shame to treat all operators as strings and do a string compare, when only a single character compare will almost always suffice. Second, and much more important, the thing doesn't WORK when two operators appear together, as in (a+b)*(c+d). Here the string following 'b' would be interpreted as a single operator ")*(."

It's possible to fix that problem. For example, we could just give GetOp a list of legal characters, and we could treat the parentheses as different operator types than the others. But this begins to get messy.

Fortunately, there's a better way that solves all the problems. Since almost all the operators are single characters, let's just treat them that way, and let GetOp get only one character at a time. This not only simplifies GetOp, but also speeds things up quite a bit. We still have the problem of the relops, but we were treating them as special cases anyway.

So here's the final version of GetOp:

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

procedure GetOp;
begin
   SkipWhite;
   Token := Look;
   Value := Look;
   GetChar;
end;
{--------------------------------------------------------------}
```

Note that I still give the string Value a value. If you're truly concerned about efficiency, you could leave this out. When we're expecting an operator, we will only be testing Token anyhow, so the value of the string won't matter. But to me it seems to be good practice to give the thing a value just in case.

Try this new version with some realistic-looking code. You should be able to separate any program into its individual tokens, with the caveat that the two-character relops will scan into two separate tokens. That's OK ... we'll parse them that way.

Now, in Part VII the function of Next was combined with procedure Scan, which also checked every identifier against a list of keywords and encoded each one that was found. As I mentioned at the time, the last thing we would want to do is to use such a procedure in places where keywords should not appear, such as in expressions. If we did that, the keyword list would be scanned for every identifier appearing in the code. Not good.

The right way to deal with that is to simply separate the functions of fetching tokens and looking for keywords. The version of Scan shown below does NOTHING but check for keywords. Notice that it operates on the current token and does NOT advance the input stream.

```
{--------------------------------------------------------------}
{ Scan the Current Identifier for Keywords }

procedure Scan;
begin
   if Token = 'x' then
      Token := KWcode[Lookup(Addr(KWlist), Value, NKW) + 1];
end;
{--------------------------------------------------------------}
```

There is one last detail. In the compiler there are a few places that we must actually check the string value of the token. Mainly, this is done to distinguish between the different END's, but there are a couple of other places. (I should note in passing that we could always eliminate the need for matching END characters by encoding each one to a different character. Right now we are definitely taking the lazy man's route.)

The following version of MatchString takes the place of the character-oriented Match. Note that, like Match, it DOES advance the input stream.

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

procedure MatchString(x: string);
begin
   if Value <> x then Expected('''' + x + '''');
   Next;
end;
{--------------------------------------------------------------}
```

FIXING UP THE COMPILER

Armed with these new scanner procedures, we can now begin to fix the compiler to use them properly. The changes are all quite minor, but there are quite a few places where changes are necessary. Rather than showing you each place, I will give you the general idea and then just give the finished product.

First of all, the code for procedure Block doesn't change, though its function does:

```
{--------------------------------------------------------------}
{ Parse and Translate a Block of Statements }
```

```
procedure Block;
begin
   Scan;
   while not(Token in ['e', 'l']) do begin
      case Token of
       'i': DoIf;
       'w': DoWhile;
       'R': DoRead;
       'W': DoWrite;
      else Assignment;
      end;
      Scan;
   end;
end;
{--------------------------------------------------------------}
```

Remember that the new version of Scan doesn't advance the input stream, it only scans for keywords. The input stream must be advanced by each procedure that Block calls.

In general, we have to replace every test on Look with a similar test on Token. For example:

```
{---------------------------------------------------------------}
{ Parse and Translate a Boolean Expression }

procedure BoolExpression;
begin
   BoolTerm;
   while IsOrOp(Token) do begin
      Push;
      case Token of
       '|': BoolOr;
       '~': BoolXor;
      end;
   end;
end;
{--------------------------------------------------------------}
```

In procedures like Add, we don't have to use Match anymore. We need only call Next to advance the input stream:

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

procedure Add;
begin
   Next;
   Term;
   PopAdd;
end;
{-------------------------------------------------------------}
```

Control structures are actually simpler. We just call Next to advance over the control keywords:

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

procedure Block; Forward;
procedure DoIf;
var L1, L2: string;
begin
   Next;
   BoolExpression;
   L1 := NewLabel;
   L2 := L1;
   BranchFalse(L1);
   Block;
   if Token = 'l' then begin
      Next;
      L2 := NewLabel;
      Branch(L2);
      PostLabel(L1);
      Block;
   end;
   PostLabel(L2);
   MatchString('ENDIF');
end;
{--------------------------------------------------------------}
```

That's about the extent of the REQUIRED changes. In the listing of TINY Version 1.1 below, I've also made a number of other "improvements" that aren't really required. Let me explain them briefly:

(1) I've deleted the two procedures Prog and Main, and combined their functions into the main program. They didn't seem to add to program clarity ... in fact they seemed to just muddy things up a little.

(2) I've deleted the keywords PROGRAM and BEGIN from the keyword list. Each one only occurs in one place, so it's not necessary to search for it.

(3) Having been bitten by an overdose of cleverness, I've reminded myself that TINY is supposed to be a minimalist program. Therefore I've replaced the fancy handling of unary minus with the dumbest one I could think of. A giant step backwards in code quality, but a great simplification of the compiler. KISS is the right place to use the other version.

(4) I've added some error-checking routines such as CheckTable and CheckDup, and replaced in-line code by calls to them. This cleans up a number of routines.

(5) I've taken the error checking out of code generation routines like Store, and put it in the parser where it belongs. See Assignment, for example.

(6) There was an error in InTable and Locate that caused them to search all locations instead of only those with valid data in them. They now search only valid cells. This allows us to eliminate the initialization of the symbol table, which was done in Init.

(7) Procedure AddEntry now has two arguments, which helps to make things a bit more modular.

(8) I've cleaned up the code for the relational operators by the addition of the new procedures CompareExpression and NextExpression.

(9) I fixed an error in the Read routine ... the earlier value

      did not check for a valid variable name.

CONCLUSION

The resulting compiler for TINY is given below. Other than the removal of the keyword PROGRAM, it parses the same language as before. It's just a bit cleaner, and more importantly it's considerably more robust. I feel good about it.

The next installment will be another digression: the discussion of semicolons and such that got me into this mess in the first place. THEN we'll press on into procedures and types. Hang in there with me. The addition of those features will go a long way towards removing KISS from the "toy language" category. We're getting very close to being able to write a serious compiler.

TINY VERSION 1.1

```
{--------------------------------------------------------------}
program Tiny11;

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

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

      LCount: integer = 0;
      NEntry: integer = 0;


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


const MaxEntry = 100;

var ST   : array[1..MaxEntry] of Symbol;
    SType: array[1..MaxEntry] of char;


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

const NKW =   9;
      NKW1 = 10;

const KWlist: array[1..NKW] of Symbol =
              ('IF', 'ELSE', 'ENDIF', 'WHILE', 'ENDWHILE',
               'READ', 'WRITE', 'VAR', 'END');

const KWcode: string[NKW1] = 'xileweRWve';


{--------------------------------------------------------------}
{ 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;

{--------------------------------------------------------------}
{ Report an Undefined Identifier }

procedure Undefined(n: string);
begin
   Abort('Undefined Identifier ' + n);
end;


{--------------------------------------------------------------}
{ Report a Duplicate Identifier }

procedure Duplicate(n: string);
begin
   Abort('Duplicate Identifier ' + n);
end;


{--------------------------------------------------------------}
{ Check to Make Sure the Current Token is an Identifier }

procedure CheckIdent;
begin
   if Token <> 'x' then Expected('Identifier');
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 a Boolean Orop }

function IsOrop(c: char): boolean;
begin
   IsOrop := c in ['|', '~'];
end;


{--------------------------------------------------------------}
{ Recognize a Relop }

function IsRelop(c: char): boolean;
begin
   IsRelop := c in ['=', '#', '<', '>'];
end;


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

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


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

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


{--------------------------------------------------------------}
{ Locate a Symbol in Table }
{ Returns the index of the entry.  Zero if not present. }

function Locate(N: Symbol): integer;
begin
   Locate := Lookup(@ST, n, NEntry);
end;


{--------------------------------------------------------------}
{ Look for Symbol in Table }

function InTable(n: Symbol): Boolean;
begin
   InTable := Lookup(@ST, n, NEntry) <> 0;
end;


{--------------------------------------------------------------}
{ Check to See if an Identifier is in the Symbol Table         }
{ Report an error if it's not. }


procedure CheckTable(N: Symbol);
begin
   if not InTable(N) then Undefined(N);
end;


{--------------------------------------------------------------}
{ Check the Symbol Table for a Duplicate Identifier }
{ Report an error if identifier is already in table. }
```

```
procedure CheckDup(N: Symbol);
begin
   if InTable(N) then Duplicate(N);
end;


{--------------------------------------------------------------}
{ Add a New Entry to Symbol Table }

procedure AddEntry(N: Symbol; T: char);
begin
   CheckDup(N);
   if NEntry = MaxEntry then Abort('Symbol Table Full');
   Inc(NEntry);
   ST[NEntry] := N;
   SType[NEntry] := T;
end;


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

procedure GetName;
begin
   SkipWhite;
   if Not IsAlpha(Look) then Expected('Identifier');
   Token := 'x';
   Value := '';
   repeat
      Value := Value + UpCase(Look);
      GetChar;
   until not IsAlNum(Look);
end;


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

procedure GetNum;
begin
   SkipWhite;
   if not IsDigit(Look) then Expected('Number');
   Token := '#';
   Value := '';
   repeat
      Value := Value + Look;
      GetChar;
   until not IsDigit(Look);
end;


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

procedure GetOp;
begin
   SkipWhite;
   Token := Look;
   Value := Look;
   GetChar;
end;


{--------------------------------------------------------------}
{ Get the Next Input Token }
```

```
procedure Next;
begin
   SkipWhite;
   if IsAlpha(Look) then GetName
   else if IsDigit(Look) then GetNum
   else GetOp;
end;


{--------------------------------------------------------------}
{ Scan the Current Identifier for Keywords }

procedure Scan;
begin
   if Token = 'x' then
      Token := KWcode[Lookup(Addr(KWlist), Value, NKW) + 1];
end;


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

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


{--------------------------------------------------------------}
{ 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;


{--------------------------------------------------------------}
{ Clear the Primary Register }

procedure Clear;
begin
   EmitLn('CLR D0');
end;


{--------------------------------------------------------------}
{ Negate the Primary Register }

procedure Negate;
begin
   EmitLn('NEG D0');
end;


{--------------------------------------------------------------}
{ Complement the Primary Register }

procedure NotIt;
begin
   EmitLn('NOT D0');
end;


{--------------------------------------------------------------}
{ Load a Constant Value to Primary Register }

procedure LoadConst(n: string);
begin
   Emit('MOVE #');
   WriteLn(n, ',D0');
end;


{--------------------------------------------------------------}
{ Load a Variable to Primary Register }

procedure LoadVar(Name: string);
begin
   if not InTable(Name) then Undefined(Name);
   EmitLn('MOVE ' + Name + '(PC),D0');
end;


{--------------------------------------------------------------}
{ Push Primary onto Stack }

procedure Push;
begin
   EmitLn('MOVE D0,-(SP)');
end;


{--------------------------------------------------------------}
{ Add Top of Stack to Primary }

procedure PopAdd;
begin
   EmitLn('ADD (SP)+,D0');
end;
```

```
{--------------------------------------------------------------}
{ Subtract Primary from Top of Stack }

procedure PopSub;
begin
   EmitLn('SUB (SP)+,D0');
   EmitLn('NEG D0');
end;


{--------------------------------------------------------------}
{ Multiply Top of Stack by Primary }

procedure PopMul;
begin
   EmitLn('MULS (SP)+,D0');
end;


{--------------------------------------------------------------}
{ Divide Top of Stack by Primary }

procedure PopDiv;
begin
   EmitLn('MOVE (SP)+,D7');
   EmitLn('EXT.L D7');
   EmitLn('DIVS D0,D7');
   EmitLn('MOVE D7,D0');
end;


{--------------------------------------------------------------}
{ AND Top of Stack with Primary }

procedure PopAnd;
begin
   EmitLn('AND (SP)+,D0');
end;


{--------------------------------------------------------------}
{ OR Top of Stack with Primary }

procedure PopOr;
begin
   EmitLn('OR (SP)+,D0');
end;


{--------------------------------------------------------------}
{ XOR Top of Stack with Primary }

procedure PopXor;
begin
   EmitLn('EOR (SP)+,D0');
end;


{--------------------------------------------------------------}
{ Compare Top of Stack with Primary }

procedure PopCompare;
begin
   EmitLn('CMP (SP)+,D0');
end;
```

```
{---------------------------------------------------------------}
{ Set D0 If Compare was = }

procedure SetEqual;
begin
   EmitLn('SEQ D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was != }

procedure SetNEqual;
begin
   EmitLn('SNE D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was > }

procedure SetGreater;
begin
   EmitLn('SLT D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was < }

procedure SetLess;
begin
   EmitLn('SGT D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was <= }

procedure SetLessOrEqual;
begin
   EmitLn('SGE D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was >= }

procedure SetGreaterOrEqual;
begin
   EmitLn('SLE D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Store Primary to Variable }

procedure Store(Name: string);
begin
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)')
end;


{---------------------------------------------------------------}
{ Branch Unconditional  }

procedure Branch(L: string);
begin
   EmitLn('BRA ' + L);
end;


{---------------------------------------------------------------}
{ Branch False }

procedure BranchFalse(L: string);
begin
   EmitLn('TST D0');
   EmitLn('BEQ ' + L);
end;


{---------------------------------------------------------------}
{ Read Variable to Primary Register }

procedure ReadIt(Name: string);
begin
   EmitLn('BSR READ');
   Store(Name);
end;


{ Write from Primary Register }

procedure WriteIt;
begin
   EmitLn('BSR WRITE');
end;


{--------------------------------------------------------------}
{ Write Header Info }

procedure Header;
begin
   WriteLn('WARMST', TAB, 'EQU $A01E');
end;


{--------------------------------------------------------------}
{ Write the Prolog }

procedure Prolog;
begin
   PostLabel('MAIN');
end;


{--------------------------------------------------------------}
{ Write the Epilog }

procedure Epilog;
begin
   EmitLn('DC WARMST');
   EmitLn('END MAIN');
end;
```

```
{---------------------------------------------------------------}
{ Allocate Storage for a Static Variable }

procedure Allocate(Name, Val: string);
begin
   WriteLn(Name, ':', TAB, 'DC ', Val);
end;


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

procedure BoolExpression; Forward;

procedure Factor;
begin
   if Token = '(' then begin
      Next;
      BoolExpression;
      MatchString(')');
      end
   else begin
      if Token = 'x' then
         LoadVar(Value)
      else if Token = '#' then
         LoadConst(Value)
      else Expected('Math Factor');
      Next;
   end;
end;


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

procedure Multiply;
begin
   Next;
   Factor;
   PopMul;
end;


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

procedure Divide;
begin
   Next;
   Factor;
   PopDiv;
end;


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

procedure Term;
begin
   Factor;
   while IsMulop(Token) do begin
      Push;
      case Token of
       '*': Multiply;
       '/': Divide;
      end;
   end;
end;


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

procedure Add;
begin
   Next;
   Term;
   PopAdd;
end;


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

procedure Subtract;
begin
   Next;
   Term;
   PopSub;
end;


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

procedure Expression;
begin
   if IsAddop(Token) then
      Clear
   else
      Term;
   while IsAddop(Token) do begin
      Push;
      case Token of
       '+': Add;
       '-': Subtract;
      end;
   end;
end;


{---------------------------------------------------------------}
{ Get Another Expression and Compare }

procedure CompareExpression;
begin
   Expression;
   PopCompare;
end;


{---------------------------------------------------------------}
{ Get The Next Expression and Compare }

procedure NextExpression;
begin
   Next;
   CompareExpression;
end;


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

```
{ Recognize and Translate a Relational "Equals" }

procedure Equal;
begin
   NextExpression;
   SetEqual;
end;


{---------------------------------------------------------------}
{ Recognize and Translate a Relational "Less Than or Equal" }

procedure LessOrEqual;
begin
   NextExpression;
   SetLessOrEqual;
end;


{---------------------------------------------------------------}
{ Recognize and Translate a Relational "Not Equals" }

procedure NotEqual;
begin
   NextExpression;
   SetNEqual;
end;


{---------------------------------------------------------------}
{ Recognize and Translate a Relational "Less Than" }

procedure Less;
begin
   Next;
   case Token of
     '=': LessOrEqual;
     '>': NotEqual;
   else begin
           CompareExpression;
           SetLess;
        end;
   end;
end;


{---------------------------------------------------------------}
{ Recognize and Translate a Relational "Greater Than" }

procedure Greater;
begin
   Next;
   if Token = '=' then begin
      NextExpression;
      SetGreaterOrEqual;
      end
   else begin
      CompareExpression;
      SetGreater;
   end;
end;


{---------------------------------------------------------------}
{ Parse and Translate a Relation }
```

```
procedure Relation;
begin
   Expression;
   if IsRelop(Token) then begin
      Push;
      case Token of
       '=': Equal;
       '<': Less;
       '>': Greater;
      end;
   end;
end;


{--------------------------------------------------------------}
{ Parse and Translate a Boolean Factor with Leading NOT }

procedure NotFactor;
begin
   if Token = '!' then begin
      Next;
      Relation;
      NotIt;
      end
   else
      Relation;
end;


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

procedure BoolTerm;
begin
   NotFactor;
   while Token = '&' do begin
      Push;
      Next;
      NotFactor;
      PopAnd;
   end;
end;


{-------------------------------------------------------------}
{ Recognize and Translate a Boolean OR }

procedure BoolOr;
begin
   Next;
   BoolTerm;
   PopOr;
end;


{-------------------------------------------------------------}
{ Recognize and Translate an Exclusive Or }

procedure BoolXor;
begin
   Next;
   BoolTerm;
   PopXor;
end;


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

```
{ Parse and Translate a Boolean Expression }

procedure BoolExpression;
begin
   BoolTerm;
   while IsOrOp(Token) do begin
      Push;
      case Token of
       '|': BoolOr;
       '~': BoolXor;
      end;
   end;
end;


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

procedure Assignment;
var Name: string;
begin
   CheckTable(Value);
   Name := Value;
   Next;
   MatchString('=');
   BoolExpression;
   Store(Name);
end;


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

procedure Block; Forward;

procedure DoIf;
var L1, L2: string;
begin
   Next;
   BoolExpression;
   L1 := NewLabel;
   L2 := L1;
   BranchFalse(L1);
   Block;
   if Token = 'l' then begin
      Next;
      L2 := NewLabel;
      Branch(L2);
      PostLabel(L1);
      Block;
   end;
   PostLabel(L2);
   MatchString('ENDIF');
end;


{--------------------------------------------------------------}
{ Parse and Translate a WHILE Statement }

procedure DoWhile;
var L1, L2: string;
begin
   Next;
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   BoolExpression;
   BranchFalse(L2);
   Block;
   MatchString('ENDWHILE');
   Branch(L1);
   PostLabel(L2);
end;


{--------------------------------------------------------------}
{ Read a Single Variable }

procedure ReadVar;
begin
   CheckIdent;
   CheckTable(Value);
   ReadIt(Value);
   Next;
end;


{--------------------------------------------------------------}
{ Process a Read Statement }

procedure DoRead;
begin
   Next;
   MatchString('(');
   ReadVar;
   while Token = ',' do begin
      Next;
      ReadVar;
   end;
   MatchString(')');
end;


{--------------------------------------------------------------}
{ Process a Write Statement }

procedure DoWrite;
begin
   Next;
   MatchString('(');
   Expression;
   WriteIt;
   while Token = ',' do begin
      Next;
      Expression;
      WriteIt;
   end;
   MatchString(')');
end;


{--------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure Block;
begin
   Scan;
   while not(Token in ['e', 'l']) do begin
      case Token of
       'i': DoIf;
       'w': DoWhile;
       'R': DoRead;
       'W': DoWrite;
      else Assignment;
      end;
      Scan;
   end;
end;


{--------------------------------------------------------------}
{ Allocate Storage for a Variable }

procedure Alloc;
begin
   Next;
   if Token <> 'x' then Expected('Variable Name');
   CheckDup(Value);
   AddEntry(Value, 'v');
   Allocate(Value, '0');
   Next;
end;


{--------------------------------------------------------------}
{ Parse and Translate Global Declarations }

procedure TopDecls;
begin
   Scan;
   while Token = 'v' do
      Alloc;
      while Token = ',' do
         Alloc;
end;


{--------------------------------------------------------------}
{ Initialize }

procedure Init;
begin
   GetChar;
   Next;
end;


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

begin
   Init;
   MatchString('PROGRAM');
   Header;
   TopDecls;
   MatchString('BEGIN');
   Prolog;
   Block;
   MatchString('END');
   Epilog;
end.
{--------------------------------------------------------------}
```