LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

21 May 1989

Part X: INTRODUCING "TINY"

INTRODUCTION

In the last installment, I showed you the general idea for the top-down development of a compiler. I gave you the first few steps of the process for compilers for Pascal and C, but I stopped far short of pushing it through to completion. The reason was simple: if we're going to produce a real, functional compiler for any language, I'd rather do it for KISS, the language that I've been defining in this tutorial series.

In this installment, we're going to do just that, for a subset of KISS which I've chosen to call TINY.

The process will be essentially that outlined in Installment IX, except for one notable difference. In that installment, I suggested that you begin with a full BNF description of the language. That's fine for something like Pascal or C, for which the language definition is firm. In the case of TINY, however, we don't yet have a full description ... we seem to be defining the language as we go. That's OK. In fact, it's preferable, since we can tailor the language slightly as we go, to keep the parsing easy.

So in the development that follows, we'll actually be doing a top-down development of BOTH the language and its compiler. The BNF description will grow along with the compiler.

In this process, there will be a number of decisions to be made, each of which will influence the BNF and therefore the nature of the language. At each decision point I'll try to remember to explain the decision and the rationale behind my choice. That way, if you happen to hold a different opinion and would prefer a different option, you can choose it instead. You now have the background to do that. I guess the important thing to note is that nothing we do here is cast in concrete. When YOU'RE designing YOUR language, you should feel free to do it YOUR way.

Many of you may be asking at this point: Why bother starting over from scratch? We had a working subset of KISS as the outcome of Installment VII (lexical scanning). Why not just extend it as needed? The answer is threefold. First of all, I have been making a number of changes to further simplify the program ... changes like encapsulating the code generation procedures, so that we can convert to a different target machine more easily. Second, I want you to see how the development can indeed be done from the top down as outlined in the last installment. Finally, we both need the practice. Each time I go through this exercise, I get a little better at it, and you will, also.

GETTING STARTED

Many years ago there were languages called Tiny BASIC, Tiny Pascal, and Tiny C, each of which was a subset of its parent full language. Tiny BASIC, for example, had only single-character variable names and global variables. It supported only a single data type. Sound familiar? At this point we have almost all the tools we need to build a compiler like that.

Yet a language called Tiny-anything still carries some baggage inherited from its parent language. I've often wondered if this is a good idea. Granted, a language based upon some parent language will have the advantage of familiarity, but there may also be some peculiar syntax carried over from the parent that may tend to add unnecessary complexity to the compiler. (Nowhere is this more true than in Small C.)

I've wondered just how small and simple a compiler could be made and still be useful, if it were designed from the outset to be both easy to use and to parse. Let's find out. This language will just be called "TINY," period. It's a subset of KISS, which I also haven't fully defined, so that at least makes us consistent (!). I suppose you could call it TINY KISS. But that opens up a whole can of worms involving cuter and cuter (and perhaps more risque) names, so let's just stick with TINY.

The main limitations of TINY will be because of the things we haven't yet covered, such as data types. Like its cousins Tiny C and Tiny BASIC, TINY will have only one data type, the 16-bit integer. The first version we develop will also have no procedure calls and will use single-character variable names, although as you will see we can remove these restrictions without much effort.

The language I have in mind will share some of the good features of Pascal, C, and Ada. Taking a lesson from the comparison of the Pascal and C compilers in the previous installment, though, TINY will have a decided Pascal flavor. Wherever feasible, a language structure will be bracketed by keywords or symbols, so that the parser will know where it's going without having to guess.

One other ground rule: As we go, I'd like to keep the compiler producing real, executable code. Even though it may not DO much at the beginning, it will at least do it correctly.

Finally, I'll use a couple of Pascal restrictions that make sense: All data and procedures must be declared before they are used. That makes good sense, even though for now the only data type we'll use is a word. This rule in turn means that the only reasonable place to put the executable code for the main program is at the end of the listing.

The top-level definition will be similar to Pascal:

```
     <program> ::= PROGRAM <top-level decl> <main> '.'
```

Already, we've reached a decision point. My first thought was to make the main block optional. It doesn't seem to make sense to write a "program" with no main program, but it does make sense if we're allowing for multiple modules, linked together. As a matter of fact, I intend to allow for this in KISS. But then we begin to open up a can of worms that I'd rather leave closed for now. For example, the term "PROGRAM" really becomes a misnomer. The MODULE of Modula-2 or the Unit of Turbo Pascal would be more appropriate. Second, what about scope rules? We'd need a convention for dealing with name visibility across modules. Better for now to just keep it simple and ignore the idea altogether.

There's also a decision in choosing to require the main program to be last. I toyed with the idea of making its position optional, as in C. The nature of SK*DOS, the OS I'm compiling for, make this very easy to do. But this doesn't really make much sense in view of the Pascal-like requirement that all data and procedures be declared before they're referenced. Since the main program can only call procedures that have already been declared, the only position that makes sense is at the end, a la Pascal.

Given the BNF above, let's write a parser that just recognizes the brackets:

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

procedure Prog;
begin
   Match('p');
   Header;
   Prolog;
   Match('.');
   Epilog;
end;
{--------------------------------------------------------------}
```

The procedure Header just emits the startup code required by the assembler:

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

procedure Header;
begin
   WriteLn('WARMST', TAB, 'EQU $A01E');
end;
{--------------------------------------------------------------}
```

The procedures Prolog and Epilog emit the code for identifying the main program, and for returning to the OS:

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

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


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

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

The main program just calls Prog, and then looks for a clean ending:

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

begin
   Init;
   Prog;
   if Look <> CR then Abort('Unexpected data after ''.''');
end.
{--------------------------------------------------------------}
```

At this point, TINY will accept only one input "program," the null program:

```
     PROGRAM .   (or 'p.' in our shorthand.)
```

Note, though, that the compiler DOES generate correct code for this program. It will run, and do what you'd expect the null program to do, that is, nothing but return gracefully to the OS.

As a matter of interest, one of my favorite compiler benchmarks is to compile, link, and execute the null program in whatever language is involved. You can learn a lot about the implementation by measuring the overhead in time required to compile what should be a trivial case. It's also interesting to measure the amount of code produced. In many compilers, the code can be fairly large, because they always include the whole run-time library whether they need it or not. Early versions of Turbo Pascal produced a 12K object file for this case. VAX C generates 50K!

The smallest null programs I've seen are those produced by Modula-2 compilers, and they run about 200-800 bytes.

In the case of TINY, we HAVE no run-time library as yet, so the object code is indeed tiny: two bytes. That's got to be a record, and it's likely to remain one since it is the minimum size required by the OS.

The next step is to process the code for the main program. I'll use the Pascal BEGIN-block:

```
     <main> ::= BEGIN <block> END
```

Here, again, we have made a decision. We could have chosen to require a "PROCEDURE MAIN" sort of declaration, similar to C. I must admit that this is not a bad idea at all ... I don't particularly like the Pascal approach since I tend to have trouble locating the main program in a Pascal listing. But the alternative is a little awkward, too, since you have to deal with the error condition where the user omits the main program or misspells its name. Here I'm taking the easy way out.

Another solution to the "where is the main program" problem might be to require a name for the program, and then bracket the main by

```
     BEGIN <name>
     END <name>
```

similar to the convention of Modula 2. This adds a bit of "syntactic sugar" to the language. Things like this are easy to add or change to your liking, if the language is your own design.

To parse this definition of a main block, change procedure Prog to read:

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

procedure Prog;
begin
   Match('p');
   Header;
   Main;
   Match('.');
end;
{--------------------------------------------------------------}
```

and add the new procedure:

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

procedure Main;
begin
   Match('b');
   Prolog;
```

```
   Match('e');
   Epilog;
end;
{--------------------------------------------------------------}
```

Now, the only legal program is:

```
     PROGRAM BEGIN END . (or 'pbe.')
```

Aren't we making progress??? Well, as usual it gets better. You might try some deliberate errors here, like omitting the 'b' or the 'e', and see what happens. As always, the compiler should flag all illegal inputs.

DECLARATIONS

The obvious next step is to decide what we mean by a declaration. My intent here is to have two kinds of declarations: variables and procedures/functions. At the top level, only global declarations are allowed, just as in C.

For now, there can only be variable declarations, identified by the keyword VAR (abbreviated 'v'):

```
     <top-level decls> ::= ( <data declaration> )*

     <data declaration> ::= VAR <var-list>
```

Note that since there is only one variable type, there is no need to declare the type. Later on, for full KISS, we can easily add a type description.

The procedure Prog becomes:

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

procedure Prog;
begin
   Match('p');
   Header;
   TopDecls;
   Main;
   Match('.');
end;
{--------------------------------------------------------------}
```

Now, add the two new procedures:

```
{--------------------------------------------------------------}
{ Process a Data Declaration }

procedure Decl;
begin
   Match('v');
   GetChar;
end;
```

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

procedure TopDecls;
begin
   while Look <> 'b' do
      case Look of
        'v': Decl;
      else Abort('Unrecognized Keyword ''' + Look + '''');
      end;
end;
{--------------------------------------------------------------}
```

Note that at this point, Decl is just a stub. It generates no code, and it doesn't process a list ... every variable must occur in a separate VAR statement.

OK, now we can have any number of data declarations, each starting with a 'v' for VAR, before the BEGIN-block. Try a few cases and see what happens.

DECLARATIONS AND SYMBOLS

That looks pretty good, but we're still only generating the null program for output. A real compiler would issue assembler directives to allocate storage for the variables. It's about time we actually produced some code.

With a little extra code, that's an easy thing to do from procedure Decl. Modify it as follows:

```
{--------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
var Name: char;
begin
   Match('v');
   Alloc(GetName);
end;
{--------------------------------------------------------------}
```

The procedure Alloc just issues a command to the assembler to allocate storage:

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

procedure Alloc(N: char);
begin
   WriteLn(N, ':', TAB, 'DC 0');
end;
{--------------------------------------------------------------}
```

Give this one a whirl. Try an input that declares some variables, such as:

```
     pvxvyvzbe.
```

See how the storage is allocated? Simple, huh? Note also that the entry point, "MAIN," comes out in the right place.

For the record, a "real" compiler would also have a symbol table to record the variables being used. Normally, the symbol table is necessary to record the type of each variable. But since in this case all variables have the same type, we don't need a symbol table for that reason. As it turns out, we're going to find a symbol necessary even without different types, but let's postpone that need until it arises.

Of course, we haven't really parsed the correct syntax for a data declaration, since it involves a variable list. Our version only permits a single variable. That's easy to fix, too.

The BNF for <var-list> is

```
<var-list> ::= <ident> (, <ident>)*
```

Adding this syntax to Decl gives this new version:

```
{--------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
var Name: char;
begin
   Match('v');
   Alloc(GetName);
   while Look = ',' do begin
      GetChar;
      Alloc(GetName);
   end;
end;
{--------------------------------------------------------------}
```

OK, now compile this code and give it a try. Try a number of lines of VAR declarations, try a list of several variables on one line, and try combinations of the two. Does it work?

INITIALIZERS

As long as we're dealing with data declarations, one thing that's always bothered me about Pascal is that it doesn't allow initializing data items in the declaration. That feature is admittedly sort of a frill, and it may be out of place in a language that purports to be a minimal language. But it's also SO easy to add that it seems a shame not to do so. The BNF becomes:

```
<var-list> ::= <var> ( <var> )*

<var> ::= <ident> [ = <integer> ]
```

Change Alloc as follows:

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

procedure Alloc(N: char);
begin
```

```
   Write(N, ':', TAB, 'DC ');
   if Look = '=' then begin
      Match('=');
      WriteLn(GetNum);
      end
   else
      WriteLn('0');
end;
{--------------------------------------------------------------}
```

There you are: an initializer with six added lines of Pascal.

OK, try this version of TINY and verify that you can, indeed, give the variables initial values.

By golly, this thing is starting to look real! Of course, it still doesn't DO anything, but it looks good, doesn't it?

Before leaving this section, I should point out that we've used two versions of function GetNum. One, the earlier one, returns a character value, a single digit. The other accepts a multi-digit integer and returns an integer value. Either one will work here, since WriteLn will handle either type. But there's no reason to limit ourselves to single-digit values here, so the correct version to use is the one that returns an integer. Here it is:

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

function GetNum: integer;
var Val: integer;
begin
   Val := 0;
   if not IsDigit(Look) then Expected('Integer');
   while IsDigit(Look) do begin
      Val := 10 * Val + Ord(Look) - Ord('0');
      GetChar;
   end;
   GetNum := Val;
end;
{--------------------------------------------------------------}
```

As a matter of fact, strictly speaking we should allow for expressions in the data field of the initializer, or at the very least for negative values. For now, let's just allow for negative values by changing the code for Alloc as follows:

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

procedure Alloc(N: char);
begin
   if InTable(N) then Abort('Duplicate Variable Name ' + N);
   ST[N] := 'v';
   Write(N, ':', TAB, 'DC ');
   if Look = '=' then begin
      Match('=');
      If Look = '-' then begin
         Write(Look);
         Match('-');
      end;
      WriteLn(GetNum);
      end
   else
```

```
      WriteLn('0');
end;
{--------------------------------------------------------------}
```

Now you should be able to initialize variables with negative and/or multi-digit values.

THE SYMBOL TABLE

There's one problem with the compiler as it stands so far: it doesn't do anything to record a variable when we declare it. So the compiler is perfectly content to allocate storage for several variables with the same name. You can easily verify this with an input like

```
     pvavavabe.
```

Here we've declared the variable A three times. As you can see, the compiler will cheerfully accept that, and generate three identical labels. Not good.

Later on, when we start referencing variables, the compiler will also let us reference variables that don't exist. The assembler will catch both of these error conditions, but it doesn't seem friendly at all to pass such errors along to the assembler. The compiler should catch such things at the source language level.

So even though we don't need a symbol table to record data types, we ought to install one just to check for these two conditions. Since at this point we are still restricted to single-character variable names, the symbol table can be trivial. To provide for it, first add the following declaration at the beginning of your program:

```
     var ST: array['A'..'Z'] of char;
```

and insert the following function:

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

function InTable(n: char): Boolean;
begin
   InTable := ST[n] <> ' ';
end;
{--------------------------------------------------------------}
```

We also need to initialize the table to all blanks. The following lines in Init will do the job:

```
var i: char;
begin
   for i := 'A' to 'Z' do
      ST[i] := ' ';
   ...
```

Finally, insert the following two lines at the beginning of

Alloc:

```
   if InTable(N) then Abort('Duplicate Variable Name ' + N);
   ST[N] := 'v';
```

That should do it. The compiler will now catch duplicate declarations. Later, we can also use InTable when generating references to the variables.

EXECUTABLE STATEMENTS

At this point, we can generate a null program that has some data variables declared and possibly initialized. But so far we haven't arranged to generate the first line of executable code.

Believe it or not, though, we almost have a usable language! What's missing is the executable code that must go into the main program. But that code is just assignment statements and control statements ... all stuff we have done before. So it shouldn't take us long to provide for them, as well.

The BNF definition given earlier for the main program included a statement block, which we have so far ignored:

```
     <main> ::= BEGIN <block> END
```

For now, we can just consider a block to be a series of assignment statements:

```
     <block> ::= (Assignment)*
```

Let's start things off by adding a parser for the block. We'll begin with a stub for the assignment statement:

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

procedure Assignment;
begin
   GetChar;
end;


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

procedure Block;
begin
   while Look <> 'e' do
      Assignment;
end;
{--------------------------------------------------------------}
```

Modify procedure Main to call Block as shown below:

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

```
procedure Main;
begin
   Match('b');
   Prolog;
   Block;
   Match('e');
   Epilog;
end;
{--------------------------------------------------------------}
```

This version still won't generate any code for the "assignment statements" ... all it does is to eat characters until it sees the 'e' for 'END.' But it sets the stage for what is to follow.

The next step, of course, is to flesh out the code for an assignment statement. This is something we've done many times before, so I won't belabor it. This time, though, I'd like to deal with the code generation a little differently. Up till now, we've always just inserted the Emits that generate output code in line with the parsing routines. A little unstructured, perhaps, but it seemed the most straightforward approach, and made it easy to see what kind of code would be emitted for each construct.

However, I realize that most of you are using an 80x86 computer, so the 68000 code generated is of little use to you. Several of you have asked me if the CPU-dependent code couldn't be collected into one spot where it would be easier to retarget to another CPU. The answer, of course, is yes.

To accomplish this, insert the following "code generation" routines:

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

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


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

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


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

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


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

procedure LoadVar(Name: char);
```

```
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;


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

procedure Store(Name: char);
begin
   if not InTable(Name) then Undefined(Name);
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)')
end;
{---------------------------------------------------------------}
```

The nice part of this approach, of course, is that we can retarget the compiler to a new CPU simply by rewriting these "code generator" procedures. In addition, we will find later that we can improve the code quality by tweaking these routines a bit, without having to modify the compiler proper.

Note that both LoadVar and Store check the symbol table to make sure that the variable is defined. The error handler Undefined simply calls Abort:

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

procedure Undefined(n: string);
begin
   Abort('Undefined Identifier ' + n);
end;
{--------------------------------------------------------------}
```

OK, we are now finally ready to begin processing executable code. We'll do that by replacing the stub version of procedure Assignment.

We've been down this road many times before, so this should all be familiar to you. In fact, except for the changes associated with the code generation, we could just copy the procedures from Part VII. Since we are making some changes, I won't just copy them, but we will go a little faster than usual.

The BNF for the assignment statement is:

```
     <assignment> ::= <ident> = <expression>

     <expression> ::= <first term> ( <addop> <term> )*

     <first term> ::= <first factor> <rest>

     <term> ::= <factor> <rest>

     <rest> ::= ( <mulop> <factor> )*

     <first factor> ::= [ <addop> ] <factor>

     <factor> ::= <var> | <number> | ( <expression> )
```

This version of the BNF is also a bit different than we've used before ... yet another "variation on the theme of an expression." This particular version has what I consider to be the best treatment of the unary minus. As you'll see later, it lets us handle negative constant values efficiently. It's worth mentioning here that we have often seen the advantages of "tweaking" the BNF as we go, to help make the language easy to parse. What you're looking at here is a bit different: we've tweaked the BNF to make the CODE GENERATION more efficient! That's a first for this series.

Anyhow, the following code implements the BNF:

```
{---------------------------------------------------------------}
{ 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
      LoadVar(GetName)
   else
      LoadConst(GetNum);
end;


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

procedure NegFactor;
begin
   Match('-');
   if IsDigit(Look) then
      LoadConst(-GetNum)
   else begin
      Factor;
      Negate;
   end;
end;


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

procedure FirstFactor;
begin
   case Look of
     '+': begin
             Match('+');
             Factor;
          end;
     '-': NegFactor;
   else  Factor;
   end;
end;


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

procedure Multiply;
begin
   Match('*');
   Factor;
   PopMul;
end;


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

procedure Divide;
begin
   Match('/');
   Factor;
   PopDiv;
end;
```

```
{--------------------------------------------------------------}
{ Common Code Used by Term and FirstTerm }

procedure Term1;
begin
   while IsMulop(Look) do begin
      Push;
      case Look of
       '*': Multiply;
       '/': Divide;
      end;
   end;
end;


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

procedure Term;
begin
   Factor;
   Term1;
end;


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

procedure FirstTerm;
begin
   FirstFactor;
   Term1;
end;


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

procedure Add;
begin
   Match('+');
   Term;
   PopAdd;
end;


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

procedure Subtract;
begin
   Match('-');
   Term;
   PopSub;
end;


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

procedure Expression;
begin
   FirstTerm;
   while IsAddop(Look) do begin
      Push;
      case Look of
       '+': Add;
```

```
       '-': Subtract;
      end;
   end;
end;


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

procedure Assignment;
var Name: char;
begin
   Name := GetName;
   Match('=');
   Expression;
   Store(Name);
end;
{--------------------------------------------------------------}
```

OK, if you've got all this code inserted, then compile it and check it out. You should be seeing reasonable-looking code, representing a complete program that will assemble and execute. We have a compiler!

BOOLEANS

The next step should also be familiar to you. We must add Boolean expressions and relational operations. Again, since we've already dealt with them more than once, I won't elaborate much on them, except where they are different from what we've done before. Again, we won't just copy from other files because I've changed a few things just a bit. Most of the changes just involve encapsulating the machine-dependent parts as we did for the arithmetic operations. I've also modified procedure NotFactor somewhat, to parallel the structure of FirstFactor. Finally, I corrected an error in the object code for the relational operators: The Scc instruction I used only sets the low 8 bits of D0. We want all 16 bits set for a logical true, so I've added an instruction to sign-extend the low byte.

To begin, we're going to need some more recognizers:

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

Also, we're going to need some more code generation routines:

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

procedure NotIt;
begin
   EmitLn('NOT 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;
{--------------------------------------------------------------}
```

All of this gives us the tools we need. The BNF for the Boolean expressions is:

```
     <bool-expr> ::= <bool-term> ( <orop> <bool-term> )*

     <bool-term> ::= <not-factor> ( <andop> <not-factor> )*

     <not-factor> ::= [ '!' ] <relation>

     <relation> ::= <expression> [ <relop> <expression> ]
```

Sharp-eyed readers might note that this syntax does not include the non-terminal "bool-factor" used in earlier versions. It was needed then because I also allowed for the Boolean constants TRUE and FALSE. But remember that in TINY there is no distinction made between Boolean and arithmetic types ... they can be freely intermixed. So there is really no need for these predefined values ... we can just use -1 and 0, respectively.

In C terminology, we could always use the defines:

```
     #define TRUE -1
     #define FALSE 0
```

(That is, if TINY had a preprocessor.) Later on, when we allow for declarations of constants, these two values will be predefined by the language.

The reason that I'm harping on this is that I've already tried the alternative, which is to include TRUE and FALSE as keywords. The problem with that approach is that it then requires lexical scanning for EVERY variable name in every expression. If you'll recall, I pointed out in Installment VII that this slows the compiler down considerably. As long as keywords can't be in expressions, we need to do the scanning only at the beginning of every new statement ... quite an improvement. So using the syntax above not only simplifies the parsing, but speeds up the scanning as well.

OK, given that we're all satisfied with the syntax above, the corresponding code is shown below:

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

procedure Equals;
begin
   Match('=');
   Expression;
   PopCompare;
   SetEqual;
end;


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

procedure NotEquals;
begin
   Match('#');
   Expression;
   PopCompare;
   SetNEqual;
end;


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

procedure Less;
begin
   Match('<');
   Expression;
   PopCompare;
   SetLess;
end;


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

procedure Greater;
begin
   Match('>');
   Expression;
   PopCompare;
   SetGreater;
end;


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


procedure Relation;
begin
   Expression;
   if IsRelop(Look) then begin
      Push;
      case Look of
       '=': Equals;
       '#': NotEquals;
       '<': Less;
       '>': Greater;
      end;
   end;
end;
```

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

procedure NotFactor;
begin
   if Look = '!' then begin
      Match('!');
      Relation;
      NotIt;
      end
   else
      Relation;
end;


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

procedure BoolTerm;
begin
   NotFactor;
   while Look = '&' do begin
      Push;
      Match('&');
      NotFactor;
      PopAnd;
   end;
end;


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

procedure BoolOr;
begin
   Match('|');
   BoolTerm;
   PopOr;
end;


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

procedure BoolXor;
begin
   Match('~');
   BoolTerm;
   PopXor;
end;


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

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

To tie it all together, don't forget to change the references to Expression in procedures Factor and Assignment so that they call BoolExpression instead.

OK, if you've got all that typed in, compile it and give it a whirl. First, make sure you can still parse an ordinary arithmetic expression. Then, try a Boolean one. Finally, make sure that you can assign the results of relations. Try, for example:

```
     pvx,y,zbx=z>ye.
```

which stands for:

```
     PROGRAM
     VAR X,Y,Z
     BEGIN
     X = Z > Y
     END.
```

See how this assigns a Boolean value to X?

CONTROL STRUCTURES

We're almost home. With Boolean expressions in place, it's a simple matter to add control structures. For TINY, we'll only allow two kinds of them, the IF and the WHILE:

```
     <if> ::= IF <bool-expression> <block> [ ELSE <block>] ENDIF

     <while> ::= WHILE <bool-expression> <block> ENDWHILE
```

Once again, let me spell out the decisions implicit in this syntax, which departs strongly from that of C or Pascal. In both of those languages, the "body" of an IF or WHILE is regarded as a single statement. If you intend to use a block of more than one statement, you have to build a compound statement using BEGIN-END (in Pascal) or '{}' (in C). In TINY (and KISS) there is no such thing as a compound statement ... single or multiple they're all just blocks to these languages.

In KISS, all the control structures will have explicit and unique keywords bracketing the statement block, so there can be no confusion as to where things begin and end. This is the modern approach, used in such respected languages as Ada and Modula 2, and it completely eliminates the problem of the "dangling else."

Note that I could have chosen to use the same keyword END to end all the constructs, as is done in Pascal. (The closing '}' in C serves the same purpose.) But this has always led to confusion, which is why Pascal programmers tend to write things like

```
     end { loop }

or   end { if }
```

As I explained in Part V, using unique terminal keywords does increase the size of the keyword list and therefore slows down the scanning, but in this case it seems a small price to pay for the added insurance. Better to find the errors at compile time rather than run time.

One last thought: The two constructs above each have the non-terminals

<bool-expression> and <block>

juxtaposed with no separating keyword. In Pascal we would expect the keywords THEN and DO in these locations.

I have no problem with leaving out these keywords, and the parser has no trouble either, ON CONDITION that we make no errors in the bool-expression part. On the other hand, if we were to include these extra keywords we would get yet one more level of insurance at very little cost, and I have no problem with that, either. Use your best judgment as to which way to go.

OK, with that bit of explanation let's proceed. As usual, we're going to need some new code generation routines. These generate the code for conditional and unconditional branches:

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

Except for the encapsulation of the code generation, the code to parse the control constructs is the same as you've seen before:

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

procedure Block; Forward;


procedure DoIf;
var L1, L2: string;
begin
   Match('i');
   BoolExpression;
   L1 := NewLabel;
   L2 := L1;
   BranchFalse(L1);
   Block;
   if Look = 'l' then begin
      Match('l');
      L2 := NewLabel;
      Branch(L2);
      PostLabel(L1);
      Block;
```

```
   end;
   PostLabel(L2);
   Match('e');
end;


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

procedure DoWhile;
var L1, L2: string;
begin
   Match('w');
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   BoolExpression;
   BranchFalse(L2);
   Block;
   Match('e');
   Branch(L1);
   PostLabel(L2);
end;
{--------------------------------------------------------------}
```

To tie everything together, we need only modify procedure Block to recognize the "keywords" for the IF and WHILE. As usual, we expand the definition of a block like so:

```
     <block> ::= ( <statement> )*
```

where

```
     <statement> ::= <if> | <while> | <assignment>
```

The corresponding code is:

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

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

OK, add the routines I've given, compile and test them. You should be able to parse the single-character versions of any of the control constructs. It's looking pretty good!

As a matter of fact, except for the single-character limitation we've got a virtually complete version of TINY. I call it, with tongue planted firmly in cheek, TINY Version 0.1.

LEXICAL SCANNING

Of course, you know what's next: We have to convert the program so that it can deal with multi-character keywords, newlines, and whitespace. We have just gone through all that in Part VII. We'll use the distributed scanner technique that I showed you in that installment. The actual implementation is a little different because the way I'm handling newlines is different.

To begin with, let's simply allow for whitespace. This involves only adding calls to SkipWhite at the end of the three routines, GetName, GetNum, and Match. A call to SkipWhite in Init primes the pump in case there are leading spaces.

Next, we need to deal with newlines. This is really a two-step process, since the treatment of the newlines with single-character tokens is different from that for multi-character ones. We can eliminate some work by doing both steps at once, but I feel safer taking things one step at a time.

Insert the new procedure:

```
{--------------------------------------------------------------}
{ Skip Over an End-of-Line }

procedure NewLine;
begin
   while Look = CR do begin
      GetChar;
      if Look = LF then GetChar;
      SkipWhite;
   end;
end;
{--------------------------------------------------------------}
```

Note that we have seen this procedure before in the form of Procedure Fin. I've changed the name since this new one seems more descriptive of the actual function. I've also changed the code to allow for multiple newlines and lines with nothing but white space.

The next step is to insert calls to NewLine wherever we decide a newline is permissible. As I've pointed out before, this can be very different in different languages. In TINY, I've decided to allow them virtually anywhere. This means that we need calls to NewLine at the BEGINNING (not the end, as with SkipWhite) of the procedures GetName, GetNum, and Match.

For procedures that have while loops, such as TopDecl, we need a call to NewLine at the beginning of the procedure AND at the bottom of each loop. That way, we can be assured that NewLine has just been called at the beginning of each pass through the loop.

If you've got all this done, try the program out and verify that it will indeed handle white space and newlines.

If it does, then we're ready to deal with multi-character tokens and keywords. To begin, add the additional declarations (copied almost verbatim from Part VII):

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

    ST: Array['A'..'Z'] 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',
               'VAR', 'BEGIN', 'END', 'PROGRAM');

const KWcode: string[NKW1] = 'xilewevbep';
{--------------------------------------------------------------}
```

Next, add the three procedures, also from Part VII:

```
{--------------------------------------------------------------}
{ 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 and Scan it for Keywords }

procedure Scan;
begin
   GetName;
   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 + '''');
end;
{--------------------------------------------------------------}
```

Now, we have to make a fairly large number of subtle changes to the remaining procedures. First, we must change the function GetName to a procedure, again as we did in Part VII:

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

procedure GetName;
begin
   NewLine;
   if not IsAlpha(Look) then Expected('Name');
   Value := '';
   while IsAlNum(Look) do begin
      Value := Value + UpCase(Look);
      GetChar;
   end;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

Note that this procedure leaves its result in the global string Value.

Next, we have to change every reference to GetName to reflect its new form. These occur in Factor, Assignment, and Decl:

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

procedure BoolExpression; Forward;

procedure Factor;
begin
   if Look = '(' then begin
      Match('(');
      BoolExpression;
      Match(')');
      end
   else if IsAlpha(Look) then begin
      GetName;
      LoadVar(Value[1]);
      end
   else
      LoadConst(GetNum);
end;
{--------------------------------------------------------------}
.
.
{--------------------------------------------------------------}
{ Parse and Translate an Assignment Statement }

procedure Assignment;
var Name: char;
begin
   Name := Value[1];
   Match('=');
   BoolExpression;
```

```
   Store(Name);
end;
{--------------------------------------------------------------}
.
.
{--------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
begin
   GetName;
   Alloc(Value[1]);
   while Look = ',' do begin
      Match(',');
      GetName;
      Alloc(Value[1]);
   end;
end;
{--------------------------------------------------------------}
```

(Note that we're still only allowing single-character variable names, so we take the easy way out here and simply use the first character of the string.)

Finally, we must make the changes to use Token instead of Look as the test character and to call Scan at the appropriate places. Mostly, this involves deleting calls to Match, occasionally replacing calls to Match by calls to MatchString, and Replacing calls to NewLine by calls to Scan. Here are the affected routines:

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

procedure Block; Forward;


procedure DoIf;
var L1, L2: string;
begin
   BoolExpression;
   L1 := NewLabel;
   L2 := L1;
   BranchFalse(L1);
   Block;
   if Token = 'l' then begin
      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
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   BoolExpression;
```

```
   BranchFalse(L2);
   Block;
   MatchString('ENDWHILE');
   Branch(L1);
   PostLabel(L2);
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;
      else Assignment;
      end;
      Scan;
   end;
end;


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

procedure TopDecls;
begin
   Scan;
   while Token <> 'b' do begin
      case Token of
        'v': Decl;
      else Abort('Unrecognized Keyword ' + Value);
      end;
      Scan;
   end;
end;


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

procedure Main;
begin
   MatchString('BEGIN');
   Prolog;
   Block;
   MatchString('END');
   Epilog;
end;

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

procedure Prog;
begin
   MatchString('PROGRAM');
   Header;
   TopDecls;
   Main;
   Match('.');
end;


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

```
{ Initialize }

procedure Init;
var i: char;
begin
   for i := 'A' to 'Z' do
      ST[i] := ' ';
   GetChar;
   Scan;
end;
{--------------------------------------------------------------}
```

That should do it. If all the changes got in correctly, you should now be parsing programs that look like programs. (If you didn't make it through all the changes, don't despair. A complete listing of the final form is given later.)

Did it work? If so, then we're just about home. In fact, with a few minor exceptions we've already got a compiler that's usable. There are still a few areas that need improvement.

MULTI-CHARACTER VARIABLE NAMES

One of those is the restriction that we still have, requiring single-character variable names. Now that we can handle multi-character keywords, this one begins to look very much like an arbitrary and unnecessary limitation. And indeed it is. Basically, its only virtue is that it permits a trivially simple implementation of the symbol table. But that's just a convenience to the compiler writers, and needs to be eliminated.

We've done this step before. This time, as usual, I'm doing it a little differently. I think the approach used here keeps things just about as simple as possible.

The natural way to implement a symbol table in Pascal is by declaring a record type, and making the symbol table an array of such records. Here, though, we don't really need a type field yet (there is only one kind of entry allowed so far), so we only need an array of symbols. This has the advantage that we can use the existing procedure Lookup to search the symbol table as well as the keyword list. As it turns out, even when we need more fields we can still use the same approach, simply by storing the other fields in separate arrays.

OK, here are the changes that need to be made. First, add the new typed constant:

```
      NEntry: integer = 0;
```

Then change the definition of the symbol table as follows:

```
const MaxEntry = 100;

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

(Note that ST is _NOT_ declared as a SymTab. That declaration is a phony one to get Lookup to work. A SymTab would take up too much RAM space, and so one is never actually allocated.)

Next, we need to replace InTable:

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

function InTable(n: Symbol): Boolean;
begin
   InTable := Lookup(@ST, n, MaxEntry) <> 0;
end;
{--------------------------------------------------------------}
```

We also need a new procedure, AddEntry, that adds a new entry to the table:

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

procedure AddEntry(N: Symbol; T: char);
begin
   if InTable(N) then Abort('Duplicate Identifier ' + N);
   if NEntry = MaxEntry then Abort('Symbol Table Full');
   Inc(NEntry);
   ST[NEntry] := N;
   SType[NEntry] := T;
end;
{--------------------------------------------------------------}
```

This procedure is called by Alloc:

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

procedure Alloc(N: Symbol);
begin
   if InTable(N) then Abort('Duplicate Variable Name ' + N);
   AddEntry(N, 'v');
.
.
.
{--------------------------------------------------------------}
```

Finally, we must change all the routines that currently treat the variable name as a single character. These include LoadVar and Store (just change the type from char to string), and Factor, Assignment, and Decl (just change Value[1] to Value).

One last thing: change procedure Init to clear the array as shown:

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

procedure Init;
var i: integer;
begin
   for i := 1 to MaxEntry do begin
      ST[i] := '';
      SType[i] := ' ';
   end;
   GetChar;
   Scan;
```

```
end;
{--------------------------------------------------------------}
```

That should do it. Try it out and verify that you can, indeed, use multi-character variable names.

MORE RELOPS

We still have one remaining single-character restriction: the one on relops. Some of the relops are indeed single characters, but others require two. These are '<=' and '>='. I also prefer the Pascal '<>' for "not equals," instead of '#'.

If you'll recall, in Part VII I pointed out that the conventional way to deal with relops is to include them in the list of keywords, and let the lexical scanner find them. But, again, this requires scanning throughout the expression parsing process, whereas so far we've been able to limit the use of the scanner to the beginning of a statement.

I mentioned then that we can still get away with this, since the multi-character relops are so few and so limited in their usage. It's easy to just treat them as special cases and handle them in an ad hoc manner.

The changes required affect only the code generation routines and procedures Relation and friends. First, we're going to need two more code generation routines:

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

Then, modify the relation parsing routines as shown below:

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

procedure LessOrEqual;
begin
   Match('=');
   Expression;
   PopCompare;
   SetLessOrEqual;
end;
```

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

procedure NotEqual;
begin
   Match('>');
   Expression;
   PopCompare;
   SetNEqual;
end;


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

procedure Less;
begin
   Match('<');
   case Look of
     '=': LessOrEqual;
     '>': NotEqual;
   else begin
           Expression;
           PopCompare;
           SetLess;
        end;
   end;
end;


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

procedure Greater;
begin
   Match('>');
   if Look = '=' then begin
      Match('=');
      Expression;
      PopCompare;
      SetGreaterOrEqual;
      end
   else begin
      Expression;
      PopCompare;
      SetGreater;
   end;
end;
{--------------------------------------------------------------}
```

That's all it takes. Now you can process all the relops. Try it.

INPUT/OUTPUT

We now have a complete, working language, except for one minor embarassment: we have no way to get data in or out. We need some I/O.

Now, the convention these days, established in C and continued in Ada and Modula 2, is to leave I/O statements out of the language itself, and just include them in the subroutine library. That would be fine, except that so far we have no provision for subroutines. Anyhow, with this approach you run into the problem of variable-length argument lists. In Pascal, the I/O statements are built into the language because they are the only ones for which the argument list can have a variable number of entries. In C, we settle for kludges like scanf and printf, and must pass the argument count to the called procedure. In Ada and Modula 2 we must use the awkward (and SLOW!) approach of a separate call for each argument.

So I think I prefer the Pascal approach of building the I/O in, even though we don't need to.

As usual, for this we need some more code generation routines. These turn out to be the easiest of all, because all we do is to call library procedures to do the work:

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

procedure ReadVar;
begin
   EmitLn('BSR READ');
   Store(Value);
end;


{--------------------------------------------------------------}
{ Write Variable from Primary Register }

procedure WriteVar;
begin
   EmitLn('BSR WRITE');
end;
{--------------------------------------------------------------}
```

The idea is that READ loads the value from input to the D0, and WRITE outputs it from there.

These two procedures represent our first encounter with a need for library procedures ... the components of a Run Time Library (RTL). Of course, someone (namely us) has to write these routines, but they're not part of the compiler itself. I won't even bother showing the routines here, since these are obviously very much OS-dependent. I _WILL_ simply say that for SK*DOS, they are particularly simple ... almost trivial. One reason I won't show them here is that you can add all kinds of fanciness to the things, for example by prompting in READ for the inputs, and by giving the user a chance to reenter a bad input.

But that is really separate from compiler design, so for now I'll just assume that a library call TINYLIB.LIB exists. Since we now need it loaded, we need to add a statement to include it in procedure Header:

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

procedure Header;
begin

   WriteLn('WARMST', TAB, 'EQU $A01E');
   EmitLn('LIB TINYLIB');
end;
{--------------------------------------------------------------}
```

That takes care of that part. Now, we also need to recognize the read and write commands. We can do this by adding two more keywords to our list:

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

const NKW =   11;
      NKW1 = 12;

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

const KWcode: string[NKW1] = 'xileweRWvbep';
{--------------------------------------------------------------}
```

(Note how I'm using upper case codes here to avoid conflict with the 'w' of WHILE.)

Next, we need procedures for processing the read/write statement and its argument list:

```
{--------------------------------------------------------------}
{ Process a Read Statement }
procedure DoRead;
begin
   Match('(');
   GetName;
   ReadVar;
   while Look = ',' do begin
      Match(',');
      GetName;
      ReadVar;
   end;
   Match(')');
end;


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

procedure DoWrite;
begin
   Match('(');
   Expression;
   WriteVar;
   while Look = ',' do begin
      Match(',');
      Expression;
      WriteVar;
   end;
   Match(')');
end;
{--------------------------------------------------------------}
```

Finally, we must expand procedure Block to handle the new statement types:

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

That's all there is to it. _NOW_ we have a language!

CONCLUSION

At this point we have TINY completely defined. It's not much ... actually a toy compiler. TINY has only one data type and no subroutines ... but it's a complete, usable language. While you're not likely to be able to write another compiler in it, or do anything else very seriously, you could write programs to read some input, perform calculations, and output the results. Not too bad for a toy.

Most importantly, we have a firm base upon which to build further extensions. I know you'll be glad to hear this: this is the last time I'll start over in building a parser ... from now on I intend to just add features to TINY until it becomes KISS. Oh, there'll be other times we will need to try things out with new copies of the Cradle, but once we've found out how to do those things they'll be incorporated into TINY.

What will those features be? Well, for starters we need subroutines and functions. Then we need to be able to handle different types, including arrays, strings, and other structures. Then we need to deal with the idea of pointers. All this will be upcoming in future installments.

See you then.

For references purposes, the complete listing of TINY Version 1.0 is shown below:

```
{--------------------------------------------------------------}
program Tiny10;

{--------------------------------------------------------------}
{ 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 =   11;
      NKW1 = 12;

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

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


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


{--------------------------------------------------------------}
{ 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];
end;


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

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


{--------------------------------------------------------------}
{ Skip Over an End-of-Line }

procedure NewLine;
begin
   while Look = CR do begin
      GetChar;
      if Look = LF then GetChar;
      SkipWhite;
   end;
end;


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

procedure Match(x: char);
begin
   NewLine;
   if Look = x then GetChar
   else Expected('''' + x + '''');
   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;


{--------------------------------------------------------------}
{ 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, MaxEntry);
end;


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

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


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

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


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

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


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

function GetNum: integer;
var Val: integer;
begin
   NewLine;
   if not IsDigit(Look) then Expected('Integer');
   Val := 0;
   while IsDigit(Look) do begin
      Val := 10 * Val + Ord(Look) - Ord('0');
      GetChar;
   end;
   GetNum := Val;
   SkipWhite;
end;
```

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

procedure Scan;
begin
   GetName;
   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 + '''');
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: integer);
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
   if not InTable(Name) then Undefined(Name);
   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 ReadVar;
begin
   EmitLn('BSR READ');
   Store(Value[1]);
end;


{ Write Variable from Primary Register }

procedure WriteVar;
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;


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

procedure BoolExpression; Forward;

procedure Factor;
```

```
begin
   if Look = '(' then begin
      Match('(');
      BoolExpression;
      Match(')');
      end
   else if IsAlpha(Look) then begin
      GetName;
      LoadVar(Value);
      end
   else
      LoadConst(GetNum);
end;


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

procedure NegFactor;
begin
   Match('-');
   if IsDigit(Look) then
      LoadConst(-GetNum)
   else begin
      Factor;
      Negate;
   end;
end;


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

procedure FirstFactor;
begin
   case Look of
     '+': begin
            Match('+');
            Factor;
          end;
     '-': NegFactor;
   else  Factor;
   end;
end;


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

procedure Multiply;
begin
   Match('*');
   Factor;
   PopMul;
end;


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

procedure Divide;
begin
   Match('/');
   Factor;
   PopDiv;
end;
```

```
{---------------------------------------------------------------}
{ Common Code Used by Term and FirstTerm }

procedure Term1;
begin
   while IsMulop(Look) do begin
      Push;
      case Look of
       '*': Multiply;
       '/': Divide;
      end;
   end;
end;


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

procedure Term;
begin
   Factor;
   Term1;
end;


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

procedure FirstTerm;
begin
   FirstFactor;
   Term1;
end;


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

procedure Add;
begin
   Match('+');
   Term;
   PopAdd;
end;


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

procedure Subtract;
begin
   Match('-');
   Term;
   PopSub;
end;


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

procedure Expression;
begin
   FirstTerm;
   while IsAddop(Look) do begin
      Push;
      case Look of
```

```
       '+': Add;
       '-': Subtract;
      end;
   end;
end;


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

procedure Equal;
begin
   Match('=');
   Expression;
   PopCompare;
   SetEqual;
end;


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

procedure LessOrEqual;
begin
   Match('=');
   Expression;
   PopCompare;
   SetLessOrEqual;
end;


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

procedure NotEqual;
begin
   Match('>');
   Expression;
   PopCompare;
   SetNEqual;
end;


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

procedure Less;
begin
   Match('<');
   case Look of
     '=': LessOrEqual;
     '>': NotEqual;
   else begin
           Expression;
           PopCompare;
           SetLess;
        end;
   end;
end;


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

procedure Greater;
begin
   Match('>');
```

```
   if Look = '=' then begin
      Match('=');
      Expression;
      PopCompare;
      SetGreaterOrEqual;
      end
   else begin
      Expression;
      PopCompare;
      SetGreater;
   end;
end;


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


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


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

procedure NotFactor;
begin
   if Look = '!' then begin
      Match('!');
      Relation;
      NotIt;
      end
   else
      Relation;
end;


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

procedure BoolTerm;
begin
   NotFactor;
   while Look = '&' do begin
      Push;
      Match('&');
      NotFactor;
      PopAnd;
   end;
end;


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

procedure BoolOr;
begin
```

```
   Match('|');
   BoolTerm;
   PopOr;
end;


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

procedure BoolXor;
begin
   Match('~');
   BoolTerm;
   PopXor;
end;


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

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


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

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


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

procedure Block; Forward;


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


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

procedure DoRead;
begin
   Match('(');
   GetName;
   ReadVar;
   while Look = ',' do begin
      Match(',');
      GetName;
      ReadVar;
   end;
   Match(')');
end;


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

procedure DoWrite;
begin
   Match('(');
   Expression;
   WriteVar;
   while Look = ',' do begin
      Match(',');
      Expression;
      WriteVar;
   end;
   Match(')');
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(N: Symbol);
begin
   if InTable(N) then Abort('Duplicate Variable Name ' + N);
   AddEntry(N, 'v');
   Write(N, ':', TAB, 'DC ');
   if Look = '=' then begin
      Match('=');
      If Look = '-' then begin
         Write(Look);
         Match('-');
      end;
      WriteLn(GetNum);
      end
   else
      WriteLn('0');
end;


{--------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
begin
   GetName;
   Alloc(Value);
   while Look = ',' do begin
      Match(',');
      GetName;
      Alloc(Value);
   end;
end;


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

procedure TopDecls;
begin
   Scan;
   while Token <> 'b' do begin
      case Token of
        'v': Decl;
      else Abort('Unrecognized Keyword ' + Value);
      end;
      Scan;
   end;
end;


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

procedure Main;
begin
   MatchString('BEGIN');
   Prolog;
```

```
   Block;
   MatchString('END');
   Epilog;
end;


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

procedure Prog;
begin
   MatchString('PROGRAM');
   Header;
   TopDecls;
   Main;
   Match('.');
end;


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

procedure Init;
var i: integer;
begin
   for i := 1 to MaxEntry do begin
      ST[i] := '';
      SType[i] := ' ';
   end;
   GetChar;
   Scan;
end;


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

begin
   Init;
   Prog;
   if Look <> CR then Abort('Unexpected data after ''.''');
end.
{--------------------------------------------------------------}
```