# LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

19 August 1988

## Part V: CONTROL CONSTRUCTS

## INTRODUCTION

In the first four installments of this series, we've been concentrating on the parsing of math expressions and assignment statements. In this installment, we'll take off on a new and exciting tangent: that of parsing and translating control constructs such as IF statements.

This subject is dear to my heart, because it represents a turning point for me. I had been playing with the parsing of expressions, just as we have done in this series, but I still felt that I was a LONG way from being able to handle a complete language. After all, REAL languages have branches and loops and subroutines and all that. Perhaps you've shared some of the same thoughts. Awhile back, though, I had to produce control constructs for a structured assembler preprocessor I was writing. Imagine my surprise to discover that it was far easier than the expression parsing I had already been through. I remember thinking, "Hey! This is EASY!" After we've finished this session, I'll bet you'll be thinking so, too.

## THE PLAN

In what follows, we'll be starting over again with a bare cradle, and as we've done twice before now, we'll build things up one at a time. We'll also be retaining the concept of single-character tokens that has served us so well to date. This means that the "code" will look a little funny, with 'i' for IF, 'w' for WHILE, etc. But it helps us get the concepts down pat without fussing over lexical scanning. Fear not ... eventually we'll see something looking like "real" code.

I also don't want to have us get bogged down in dealing with statements other than branches, such as the assignment statements we've been working on. We've already demonstrated that we can handle them, so there's no point carrying them around as excess baggage during this exercise. So what I'll do instead is to use an anonymous statement, "other", to take the place of the non-control statements and serve as a place-holder for them. We have to generate some kind of object code for them (we're back into compiling, not interpretation), so for want of anything else I'll just echo the character input.

OK, then, starting with yet another copy of the cradle, let's define the procedure:

```
{--------------------------------------------------------------}
{ Recognize and Translate an "Other" }

procedure Other;
begin
   EmitLn(GetName);
end;
{--------------------------------------------------------------}
```

Now include a call to it in the main program, thus:

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

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

Run the program and see what you get. Not very exciting, is it? But hang in there, it's a start, and things will get better.

The first thing we need is the ability to deal with more than one statement, since a single-line branch is pretty limited. We did that in the last session on interpreting, but this time let's get a little more formal. Consider the following BNF:

```
          <program> ::= <block> END

          <block> ::= [ <statement> ]*
```

This says that, for our purposes here, a program is defined as a block, followed by an END statement. A block, in turn, consists of zero or more statements.  We only have one kind  of statement, so far.

What signals the end of a block?  It's  simply any construct that isn't an "other"  statement.    For  now, that means only the END statement.

Armed with these ideas, we can proceed to build  up  our  parser. The code for a program (we  have  to call it DoProgram, or Pascal will complain, is:

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

procedure DoProgram;
begin
   Block;
   if Look <> 'e' then Expected('End');
   EmitLn('END')
end;
{--------------------------------------------------------------}
```

Notice  that  I've  arranged to emit  an  "END"  command  to  the assembler, which sort of  punctuates  the  output code, and makes sense considering that we're parsing a complete program here.

The code for Block is:

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

procedure Block;
begin
   while not(Look in ['e']) do begin
      Other;
   end;
end;
{--------------------------------------------------------------}
```

(From the form of the procedure, you just KNOW we're going  to be adding to it in a bit!)

OK, enter these routines into your program.  Replace the  call to Block in the main program, by  a  call  to DoProgram.  Now try it and  see  how  it works.  Well, it's still not  much,  but  we're getting closer.

## SOME GROUNDWORK

Before we begin to define the various control constructs, we need to  lay a bit more groundwork.  First, a word of warning: I won't be using the same syntax  for these constructs as you're familiar with  from Pascal or C.  For example, the Pascal syntax for an IF is:

```
     IF <condition> THEN <statement>
```

(where the statement, of course, may be compound).

The C version is similar:

```
     IF ( <condition> ) <statement>
```

Instead, I'll be using something that looks more like Ada:

```
     IF <condition> <block> ENDIF
```

In other words, the IF construct has a specific termination symbol. This avoids the dangling-else of Pascal and C and also precludes the need for the brackets {} or begin-end. The syntax I'm showing you here, in fact, is that of the language KISS that I'll be detailing in later installments. The other constructs will also be slightly different. That shouldn't be a real problem for you. Once you see how it's done, you'll realize that it really doesn't matter so much which specific syntax is involved. Once the syntax is defined, turning it into code is straightforward.

Now, all of the constructs we'll be dealing with here involve transfer of control, which at the assembler-language level means conditional and/or unconditional branches. For example, the simple IF statement

```
          IF <condition> A ENDIF B ....
```

must get translated into

```
          Branch if NOT condition to L
          A
     L:   B
          ...
```

It's clear, then, that we're going to need some more procedures to help us deal with these branches. I've defined two of them below. Procedure NewLabel generates unique labels. This is done via the simple expedient of calling every label 'Lnn', where nn is a label number starting from zero. Procedure PostLabel just outputs the labels at the proper place.

Here are the two routines:

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

Notice that we've added a new global variable, LCount, so you need to change the VAR declarations at the top of the program to look like this:

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

Also, add the following extra initialization to Init:

```
   LCount := 0;
```

(DON'T forget that, or your labels can look really strange!)

At this point I'd also like to show you a new kind of notation. If you compare the form of the IF statement above with the assembler code that must be produced, you can see that there are certain actions associated with each of the keywords in the statement:

```
     IF:  First, get the condition and issue the code for it.
          Then, create a unique label and emit a branch if false.

     ENDIF: Emit the label.
```

These actions can be shown very concisely if we write the syntax this way:

```
     IF
     <condition>    { Condition;
                      L = NewLabel;
                      Emit(Branch False to L); }
     <block>
     ENDIF          { PostLabel(L) }
```

This is an example of syntax-directed translation. We've been doing it all along ... we've just never written it down this way before. The stuff in curly brackets represents the ACTIONS to be taken. The nice part about this representation is that it not only shows what we have to recognize, but also the actions we have to perform, and in which order. Once we have this syntax, the code almost writes itself.

About the only thing left to do is to be a bit more specific about what we mean by "Branch if false."

I'm assuming that there will be code executed for <condition> that will perform Boolean algebra and compute some result. It should also set the condition flags corresponding to that result. Now, the usual convention for a Boolean variable is to let 0000 represent "false," and anything else (some use FFFF, some 0001) represent "true."

On the 68000 the condition flags are set whenever any data is moved or calculated. If the data is a 0000 (corresponding to a false condition, remember), the zero flag will be set. The code for "Branch on zero" is BEQ. So for our purposes here,

```
               BEQ  <=> Branch if false
               BNE  <=> Branch if true
```

It's the nature of the beast that most of the branches we see will be BEQ's ... we'll be branching AROUND the code that's supposed to be executed when the condition is true.

## THE IF STATEMENT

With that bit of explanation out of the way, we're finally ready to begin coding the IF-statement parser. In fact, we've almost already done it! As usual, I'll be using our single-character approach, with the character 'i' for IF, and 'e' for ENDIF (as well as END ... that dual nature causes no confusion). I'll also, for now, skip completely the character for the branch condition, which we still have to define.

The code for DoIf is:

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

procedure Block; Forward;


procedure DoIf;
var L: string;
begin
   Match('i');
   L := NewLabel;
   Condition;
   EmitLn('BEQ ' + L);
   Block;
   Match('e');
   PostLabel(L);
end;
{---------------------------------------------------------------}
```

Add this routine to your program, and change Block to reference it as follows:

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

procedure Block;
begin
   while not(Look in ['e']) do begin
      case Look of
       'i': DoIf;
       'o': Other;
      end;
   end;
end;
{---------------------------------------------------------------}
```

Notice the reference to procedure Condition. Eventually, we'll write a routine that can parse and translate any Boolean condition we care to give it. But that's a whole installment by itself (the next one, in fact). For now, let's just make it a dummy that emits some text. Write the following routine:

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

Procedure Condition;
begin
   EmitLn('<condition>');
end;
{--------------------------------------------------------------}
```

Insert this procedure in your program just before DoIf. Now run the program. Try a string like

```
     aibece
```

As you can see, the parser seems to recognize the construct and inserts the object code at the right places. Now try a set of nested IF's, like

```
     aibicedefe
```

It's starting to look real, eh?

Now that we have the general idea (and the tools such as the notation and the procedures NewLabel and PostLabel), it's a piece of cake to extend the parser to include other constructs. The first (and also one of the trickiest) is to add the ELSE clause to IF. The BNF is

```
     IF <condition> <block> [ ELSE <block>] ENDIF
```

The tricky part arises simply because there is an optional part, which doesn't occur in the other constructs.

The corresponding output code should be

```
          <condition>
          BEQ L1
          <block>
          BRA L2
     L1:  <block>
     L2:  ...
```

This leads us to the following syntax-directed translation:

```
     IF
     <condition>    { L1 = NewLabel;
                      L2 = NewLabel;
                      Emit(BEQ L1) }
     <block>
     ELSE           { Emit(BRA L2);
                      PostLabel(L1) }
     <block>
     ENDIF          { PostLabel(L2) }
```

Comparing this with the case for an ELSE-less IF gives us a clue as to how to handle both situations. The code below does it. (Note that I use an 'l' for the ELSE, since 'e' is otherwise occupied):

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

procedure DoIf;
var L1, L2: string;
begin
   Match('i');
   Condition;
   L1 := NewLabel;
   L2 := L1;
   EmitLn('BEQ ' + L1);
   Block;
   if Look = 'l' then begin
      Match('l');
      L2 := NewLabel;
      EmitLn('BRA ' + L2);
      PostLabel(L1);
      Block;
   end;
   Match('e');
   PostLabel(L2);
end;
{--------------------------------------------------------------}
```

There you have it.  A complete IF parser/translator, in  19 lines of code.

Give it a try now.  Try something like

```
   aiblcede
```

Did it work?  Now, just  to  be  sure we haven't broken the ELSE-less case, try

```
   aibece
```

Now try some nested IF's.  Try anything you like,  including some badly formed statements.   Just  remember that 'e' is not a legal "other" statement.

## THE WHILE STATEMENT

The next type of statement should be easy, since we  already have the process  down  pat.    The  syntax  I've chosen for the WHILE statement is

```
          WHILE <condition> <block> ENDWHILE
```

I know,  I  know,  we  don't  REALLY  need separate kinds of terminators for each construct ... you can see that by the fact that in our one-character version, 'e' is used for all of them.  But I also remember  MANY debugging sessions in Pascal, trying to track down a wayward END that the compiler obviously thought I meant to put  somewhere  else.   It's been my experience that specific and unique  keywords,  although  they add to the  vocabulary  of  the language,  give  a  bit of error-checking that is worth the extra work for the compiler writer.

Now,  consider  what  the  WHILE  should be translated into.   It should be:

```
     L1:  <condition>
          BEQ L2
          <block>
          BRA L1
     L2:
```

As before, comparing the two representations gives us the actions
needed at each point.

```
     WHILE          { L1 = NewLabel;
                      PostLabel(L1) }
     <condition>    { Emit(BEQ L2) }
     <block>
     ENDWHILE       { Emit(BRA L1);
                      PostLabel(L2) }
```

The code follows immediately from the syntax:

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

procedure DoWhile;
var L1, L2: string;
begin
   Match('w');
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   Condition;
   EmitLn('BEQ ' + L2);
   Block;
   Match('e');
   EmitLn('BRA ' + L1);
   PostLabel(L2);
end;
{--------------------------------------------------------------}
```

Since we've got a new statement, we have to add a call to it
within procedure Block:

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

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

No other changes are necessary.

OK, try the new program.  Note that this  time,  the  <condition> code is INSIDE the upper label, which is just where we wanted it. Try some nested loops.  Try some loops within IF's, and some IF's within loops.  If you get  a  bit  confused as to what you should type, don't be discouraged:  you  write  bugs in other languages, too, don't you?  It'll look a lot  more  meaningful  when  we get full keywords.

I hope by now that you're beginning to  get  the  idea  that this really  IS easy.  All we have to do to accomodate a new construct is to work out  the  syntax-directed translation of it.  The code almost falls out  from  there,  and  it doesn't affect any of the other routines.  Once you've gotten the feel of the thing, you'll see that you  can  add  new  constructs  about as fast as you can dream them up.

## THE LOOP STATEMENT

We could stop right here, and  have  a language that works.  It's been  shown  many  times that a high-order language with only two constructs, the IF and the WHILE, is sufficient  to  write structured  code.   But we're on a roll now, so let's  richen  up  the repertoire a bit.

This construct is even easier, since it has no condition  test at all  ... it's an infinite loop.  What's the point of such a loop? Not much, by  itself,  but  later  on  we're going to add a BREAK command,  that  will  give us a way out.  This makes the language considerably richer than Pascal, which  has  no  break,  and also avoids the funny  WHILE(1) or WHILE TRUE of C and Pascal.

The syntax is simply

```
     LOOP <block> ENDLOOP
```

and the syntax-directed translation is:

```
     LOOP           { L = NewLabel;
                      PostLabel(L) }
     <block>
     ENDLOOP        { Emit(BRA L }
```

The corresponding code is shown below.  Since  I've  already used 'l'  for  the  ELSE, I've used  the  last  letter,  'p',  as  the "keyword" this time.

```
{--------------------------------------------------------------}
{ Parse and Translate a LOOP Statement }

procedure DoLoop;
var L: string;
begin
   Match('p');
   L := NewLabel;
   PostLabel(L);
   Block;
   Match('e');
   EmitLn('BRA ' + L);
end;
{--------------------------------------------------------------}
```

When you insert this routine, don't forget to add a line in Block to call it.

## REPEAT-UNTIL

Here's one construct that I lifted right from Pascal.  The syntax is

```
     REPEAT <block> UNTIL <condition>  ,
```

and the syntax-directed translation is:

```
     REPEAT         { L = NewLabel;
                      PostLabel(L) }
     <block>
     UNTIL
     <condition>    { Emit(BEQ L) }
```

As usual, the code falls out pretty easily:

```
{--------------------------------------------------------------}
{ Parse and Translate a REPEAT Statement }

procedure DoRepeat;
var L: string;
begin
   Match('r');
   L := NewLabel;
   PostLabel(L);
   Block;
   Match('u');
   Condition;
   EmitLn('BEQ ' + L);
end;
{--------------------------------------------------------------}
```

As  before, we have to add the call  to  DoRepeat  within  Block. This time, there's a difference, though.  I decided  to  use  'r' for REPEAT (naturally), but I also decided to use 'u'  for UNTIL. This means that the 'u' must be added to the set of characters in the while-test.  These  are  the  characters  that signal an exit from the current  block  ... the "follow" characters, in compiler jargon.

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

procedure Block;
begin
   while not(Look in ['e', 'l', 'u']) do begin
      case Look of
       'i': DoIf;
       'w': DoWhile;
       'p': DoLoop;
       'r': DoRepeat;
       else Other;
      end;
   end;
```

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

## THE FOR LOOP

The FOR loop is a very handy one to have around, but it's a bear to translate. That's not so much because the construct itself is hard ... it's only a loop after all ... but simply because it's hard to implement in assembler language. Once the code is figured out, the translation is straightforward enough.

C fans love the FOR-loop of that language (and, in fact, it's easier to code), but I've chosen instead a syntax very much like the one from good ol' BASIC:

```
     FOR <ident> = <expr1> TO <expr2> <block> ENDFOR
```

The translation of a FOR loop can be just about as difficult as you choose to make it, depending upon the way you decide to define the rules as to how to handle the limits. Does expr2 get evaluated every time through the loop, for example, or is it treated as a constant limit? Do you always go through the loop at least once, as in FORTRAN, or not? It gets simpler if you adopt the point of view that the construct is equivalent to:

```
     <ident> = <expr1>
     TEMP = <expr2>
     WHILE <ident> <= TEMP
     <block>
     ENDWHILE
```

Notice that with this definition of the loop, <block> will not be executed at all if <expr1> is initially larger than <expr2>.

The 68000 code needed to do this is trickier than anything we've done so far. I had a couple of tries at it, putting both the counter and the upper limit on the stack, both in registers, etc. I finally arrived at a hybrid arrangement, in which the loop counter is in memory (so that it can be accessed within the loop), and the upper limit is on the stack. The translated code came out like this:

```
          <ident>             get name of loop counter
          <expr1>             get initial value
          LEA <ident>(PC),A0  address the loop counter
          SUBQ #1,D0          predecrement it
          MOVE D0,(A0)        save it
          <expr1>             get upper limit
          MOVE D0,-(SP)       save it on stack

     L1:  LEA <ident>(PC),A0  address loop counter
          MOVE (A0),D0        fetch it to D0
          ADDQ #1,D0          bump the counter
          MOVE D0,(A0)        save new value
          CMP (SP),D0         check for range
          BLE L2              skip out if D0 > (SP)
          <block>
          BRA L1              loop for next pass
     L2:  ADDQ #2,SP          clean up the stack
```

Wow!    That  seems like a lot of code ...  the  line  containing
<block> seems to almost get lost.  But that's the best I could do
with it.   I guess it helps to keep in mind that it's really only
sixteen  words,  after  all.  If  anyone else can  optimize  this
better, please let me know.

Still, the parser  routine  is  pretty  easy now that we have the
code:

```
{--------------------------------------------------------------}
{ Parse and Translate a FOR Statement }

procedure DoFor;
var L1, L2: string;
    Name: char;
begin
   Match('f');
   L1 := NewLabel;
   L2 := NewLabel;
   Name := GetName;
   Match('=');
   Expression;
   EmitLn('SUBQ #1,D0');
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)');
   Expression;
   EmitLn('MOVE D0,-(SP)');
   PostLabel(L1);
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE (A0),D0');
   EmitLn('ADDQ #1,D0');
   EmitLn('MOVE D0,(A0)');
   EmitLn('CMP (SP),D0');
   EmitLn('BGT ' + L2);
   Block;
   Match('e');
   EmitLn('BRA ' + L1);
   PostLabel(L2);
   EmitLn('ADDQ #2,SP');
end;
{--------------------------------------------------------------}
```

Since we don't have  expressions  in this parser, I used the same
trick as for Condition, and wrote the routine

```
{--------------------------------------------------------------}
{ Parse and Translate an Expression }
{ This version is a dummy }

Procedure Expression;
begin
   EmitLn('<expr>');
end;
{--------------------------------------------------------------}
```

Give it a try.  Once again,  don't  forget  to  add  the  call in
Block.    Since  we don't have any input for the dummy version of
Expression, a typical input line would look something like

```
     afi=bece
```

Well, it DOES generate a lot of code, doesn't it?    But at least
it's the RIGHT code.

## THE DO STATEMENT

All this made me wish for a simpler version of the FOR loop.  The reason for all the code above is the need to have the loop counter accessible as a variable within the loop.  If all we need is a counting loop to make us go through something a specified number of times, but don't need access to the counter itself, there is a much easier solution.  The 68000 has a "decrement and branch nonzero" instruction built in which is ideal for counting. For good measure, let's add this construct, too.   This will be the last of our loop structures.

The syntax and its translation is:

```
     DO
     <expr>         { Emit(SUBQ #1,D0);
                      L = NewLabel;
                      PostLabel(L);
                      Emit(MOVE D0,-(SP) }
     <block>
     ENDDO          { Emit(MOVE (SP)+,D0;
                      Emit(DBRA D0,L) }
```

That's quite a bit simpler!  The loop will execute  <expr> times. Here's the code:

```
{---------------------------------------------------------------}
{ Parse and Translate a DO Statement }

procedure Dodo;
var L: string;
begin
   Match('d');
   L := NewLabel;
   Expression;
   EmitLn('SUBQ #1,D0');
   PostLabel(L);
   EmitLn('MOVE D0,-(SP)');
   Block;
   EmitLn('MOVE (SP)+,D0');
   EmitLn('DBRA D0,' + L);
end;
{---------------------------------------------------------------}
```

I think you'll have to agree, that's a whole lot simpler than the classical FOR.  Still, each construct has its place.

## THE BREAK STATEMENT

Earlier I promised you a BREAK statement to accompany LOOP.  This is one I'm sort of proud of.  On the face of it a BREAK seems really tricky.  My first approach was to just use it as an extra terminator to Block, and split all the loops into two parts, just as I did with the ELSE half of an IF.  That turns out not to work, though, because the BREAK statement is almost certainly not going to show up at the same level as the loop itself.  The most likely place for a BREAK is right after an IF, which would cause it to exit to the IF construct, not the enclosing loop.  WRONG. The BREAK has to exit the inner LOOP, even if it's nested down into several levels of IFs.

My next thought was that I would just store away, in some global variable, the ending label of the innermost loop. That doesn't work either, because there may be a break from an inner loop followed by a break from an outer one. Storing the label for the inner loop would clobber the label for the outer one. So the global variable turned into a stack. Things were starting to get messy.

Then I decided to take my own advice. Remember in the last session when I pointed out how well the implicit stack of a recursive descent parser was serving our needs? I said that if you begin to see the need for an external stack you might be doing something wrong. Well, I was. It is indeed possible to let the recursion built into our parser take care of everything, and the solution is so simple that it's surprising.

The secret is to note that every BREAK statement has to occur within a block ... there's no place else for it to be. So all we have to do is to pass into Block the exit address of the innermost loop. Then it can pass the address to the routine that translates the break instruction. Since an IF statement doesn't change the loop level, procedure DoIf doesn't need to do anything except pass the label into ITS blocks (both of them). Since loops DO change the level, each loop construct simply ignores whatever label is above it and passes its own exit label along.

All this is easier to show you than it is to describe. I'll demonstrate with the easiest loop, which is LOOP:

```
{--------------------------------------------------------------}
{ Parse and Translate a LOOP Statement }

procedure DoLoop;
var L1, L2: string;
begin
   Match('p');
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   Block(L2);
   Match('e');
   EmitLn('BRA ' + L1);
   PostLabel(L2);
end;
{--------------------------------------------------------------}
```

Notice that DoLoop now has TWO labels, not just one. The second is to give the BREAK instruction a target to jump to. If there is no BREAK within the loop, we've wasted a label and cluttered up things a bit, but there's no harm done.

Note also that Block now has a parameter, which for loops will always be the exit address. The new version of Block is:

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

procedure Block(L: string);
begin
   while not(Look in ['e', 'l', 'u']) do begin
      case Look of
       'i': DoIf(L);
       'w': DoWhile;
       'p': DoLoop;
       'r': DoRepeat;
       'f': DoFor;
       'd': DoDo;
       'b': DoBreak(L);
       else Other;
      end;
   end;
end;
{--------------------------------------------------------------}
```

Again,  notice  that  all Block does with the label is to pass it into DoIf and  DoBreak.    The  loop  constructs  don't  need it, because they are going to pass their own label anyway.

The new version of DoIf is:

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

procedure Block(L: string); Forward;


procedure DoIf(L: string);
var L1, L2: string;
begin
   Match('i');
   Condition;
   L1 := NewLabel;
   L2 := L1;
   EmitLn('BEQ ' + L1);
   Block(L);
   if Look = 'l' then begin
      Match('l');
      L2 := NewLabel;
      EmitLn('BRA ' + L2);
      PostLabel(L1);
      Block(L);
   end;
   Match('e');
   PostLabel(L2);
end;
{--------------------------------------------------------------}
```

Here,  the  only  thing  that  changes  is  the addition  of  the parameter to procedure Block.  An IF statement doesn't change the loop  nesting level, so DoIf just passes the  label  along.    No matter how many levels of IF nesting we have, the same label will be used.

Now, remember that DoProgram also calls Block, so it now needs to pass it a label.  An  attempt  to  exit the outermost block is an error, so DoProgram  passes  a  null  label  which  is  caught by DoBreak:

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

procedure DoBreak(L: string);
begin
   Match('b');
   if L <> '' then
      EmitLn('BRA ' + L)
   else Abort('No loop to break from');
```

```
end;


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

{ Parse and Translate a Program }

procedure DoProgram;
begin
   Block('');
   if Look <> 'e' then Expected('End');
   EmitLn('END')
end;
{--------------------------------------------------------------}
```

That  ALMOST takes care of everything.  Give it a try, see if you can "break" it <pun>.  Careful, though.  By this time  we've used so many letters, it's hard to think of characters that aren't now representing  reserved  words.   Remember:  before  you  try the program, you're going to have to edit every occurence of Block in the other loop constructs to include the new parameter.   Do  it just like I did for LOOP.

I  said ALMOST above.  There is one slight problem: if you take a hard  look  at  the code generated for DO, you'll see that if you break  out  of  this loop, the value of the loop counter is still left on the stack.  We're going to have to fix that!  A shame ... that was one  of  our  smaller  routines, but it can't be helped. Here's a version that doesn't have the problem:

```
{--------------------------------------------------------------}
{ Parse and Translate a DO Statement }

procedure Dodo;
var L1, L2: string;
begin
   Match('d');
   L1 := NewLabel;
   L2 := NewLabel;
   Expression;
   EmitLn('SUBQ #1,D0');
   PostLabel(L1);
   EmitLn('MOVE D0,-(SP)');
   Block(L2);
   EmitLn('MOVE (SP)+,D0');
   EmitLn('DBRA D0,' + L1);
   EmitLn('SUBQ #2,SP');
   PostLabel(L2);
   EmitLn('ADDQ #2,SP');
end;
{--------------------------------------------------------------}
```

The  two  extra  instructions,  the  SUBQ and ADDQ, take care  of leaving the stack in the right shape.

## CONCLUSION

At this point we have created a number of control  constructs ... a richer set, really, than that provided by almost any other programming language.  And,  except  for the FOR loop, it was pretty easy to do.  Even that one was tricky only because it's tricky in assembler language.

I'll conclude this session here.  To wrap the thing up with a red ribbon, we really should have a go at having real keywords instead of these mickey-mouse single-character things.  You've already seen that the extension to multi-character words is not difficult, but in this case it will make a big difference in the appearance of our input code.  I'll save that little bit for the next installment.  In that installment we'll also address Boolean expressions, so we can get rid of the dummy version of Condition that we've used here.  See you then.

For reference purposes, here is the completed parser for this session:

```
{--------------------------------------------------------------}
program Branch;

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

const TAB = ^I;
      CR  = ^M;


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

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


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

procedure GetChar;
begin
   Read(Look);
end;


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

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


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

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


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

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

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

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

function IsAddop(c: char): boolean;
begin
   IsAddop := 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;


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

function GetName: char;
begin
   if not IsAlpha(Look) then Expected('Name');
   GetName := UpCase(Look);
   GetChar;
end;




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

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


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

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


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

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


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

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


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

{ Output a String with Tab and CRLF }

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


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

procedure Condition;
begin
   EmitLn('<condition>');
end;
```

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

procedure Expression;
begin
   EmitLn('<expr>');
end;


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

procedure Block(L: string); Forward;


procedure DoIf(L: string);
var L1, L2: string;
begin
   Match('i');
   Condition;
   L1 := NewLabel;
   L2 := L1;
   EmitLn('BEQ ' + L1);
   Block(L);
   if Look = 'l' then begin
      Match('l');
      L2 := NewLabel;
      EmitLn('BRA ' + L2);
      PostLabel(L1);
      Block(L);
   end;
   Match('e');
   PostLabel(L2);
end;


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

procedure DoWhile;
var L1, L2: string;
begin
   Match('w');
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   Condition;
   EmitLn('BEQ ' + L2);
   Block(L2);
   Match('e');
   EmitLn('BRA ' + L1);
   PostLabel(L2);
end;


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

procedure DoLoop;
var L1, L2: string;
begin
   Match('p');
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   Block(L2);
   Match('e');
   EmitLn('BRA ' + L1);
   PostLabel(L2);
end;


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

procedure DoRepeat;
var L1, L2: string;
begin
   Match('r');
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   Block(L2);
   Match('u');
   Condition;
   EmitLn('BEQ ' + L1);
   PostLabel(L2);
end;


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

procedure DoFor;
var L1, L2: string;
    Name: char;
begin
   Match('f');
   L1 := NewLabel;
   L2 := NewLabel;
   Name := GetName;
   Match('=');
   Expression;
   EmitLn('SUBQ #1,D0');
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)');
   Expression;
   EmitLn('MOVE D0,-(SP)');
   PostLabel(L1);
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE (A0),D0');
   EmitLn('ADDQ #1,D0');
   EmitLn('MOVE D0,(A0)');
   EmitLn('CMP (SP),D0');
   EmitLn('BGT ' + L2);
   Block(L2);
   Match('e');
   EmitLn('BRA ' + L1);
   PostLabel(L2);
   EmitLn('ADDQ #2,SP');
end;




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

procedure Dodo;
var L1, L2: string;
begin
```

```
   Match('d');
   L1 := NewLabel;
   L2 := NewLabel;
   Expression;
   EmitLn('SUBQ #1,D0');
   PostLabel(L1);
   EmitLn('MOVE D0,-(SP)');
   Block(L2);
   EmitLn('MOVE (SP)+,D0');
   EmitLn('DBRA D0,' + L1);
   EmitLn('SUBQ #2,SP');
   PostLabel(L2);
   EmitLn('ADDQ #2,SP');
end;


{--------------------------------------------------------------}
{ Recognize and Translate a BREAK }

procedure DoBreak(L: string);
begin
   Match('b');
   EmitLn('BRA ' + L);
end;


{--------------------------------------------------------------}
{ Recognize and Translate an "Other" }

procedure Other;
begin
   EmitLn(GetName);
end;


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

procedure Block(L: string);
begin
   while not(Look in ['e', 'l', 'u']) do begin
      case Look of
       'i': DoIf(L);
       'w': DoWhile;
       'p': DoLoop;
       'r': DoRepeat;
       'f': DoFor;
       'd': DoDo;
       'b': DoBreak(L);
       else Other;
      end;
   end;
end;




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

{ Parse and Translate a Program }

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

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

{ Initialize }

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


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

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