LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

31 August 1988

Part VI: BOOLEAN EXPRESSIONS

INTRODUCTION

In Part V of this series, we took a look at control constructs, and developed parsing routines to translate them into object code. We ended up with a nice, relatively rich set of constructs.

As we left the parser, though, there was one big hole in our capabilities: we did not address the issue of the branch condition. To fill the void, I introduced to you a dummy parse routine called Condition, which only served as a place-keeper for the real thing.

One of the things we'll do in this session is to plug that hole by expanding Condition into a true parser/translator.

THE PLAN

We're going to approach this installment a bit differently than any of the others. In those other installments, we started out immediately with experiments using the Pascal compiler, building up the parsers from very rudimentary beginnings to their final forms, without spending much time in planning beforehand. That's called coding without specs, and it's usually frowned upon. We could get away with it before because the rules of arithmetic are pretty well established ... we know what a '+' sign is supposed to mean without having to discuss it at length. The same is true for branches and loops. But the ways in which programming languages implement logic vary quite a bit from language to language. So before we begin serious coding, we'd better first make up our minds what it is we want. And the way to do that is at the level of the BNF syntax rules (the GRAMMAR).

THE GRAMMAR

For some time now, we've been implementing BNF syntax equations for arithmetic expressions, without ever actually writing them down all in one place. It's time that we did so. They are:

```
<expression> ::= <unary op> <term> [<addop> <term>]*
<term>       ::= <factor> [<mulop> factor]*
<factor>     ::= <integer> | <variable> | ( <expression> )
```

(Remember, the nice thing about this grammar is that it enforces the operator precedence hierarchy that we normally expect for algebra.)

Actually, while we're on the subject, I'd like to amend this grammar a bit right now. The way we've handled the unary minus is a bit awkward. I've found that it's better to write the grammar this way:

```
<expression>    ::= <term> [<addop> <term>]*
<term>          ::= <signed factor> [<mulop> factor]*
<signed factor> ::= [<addop>] <factor>
<factor>        ::= <integer> | <variable> | (<expression>)
```

This puts the job of handling the unary minus onto Factor, which is where it really belongs.

This doesn't mean that you have to go back and recode the programs you've already written, although you're free to do so if you like. But I will be using the new syntax from now on.

Now, it probably won't come as a shock to you to learn that we can define an analogous grammar for Boolean algebra. A typical set or rules is:

```
<b-expression>::= <b-term> [<orop> <b-term>]*
<b-term>      ::= <not-factor> [AND <not-factor>]*
<not-factor>  ::= [NOT] <b-factor>
<b-factor>    ::= <b-literal> | <b-variable> | (<b-expression>)
```

Notice that in this grammar, the operator AND is analogous to '*', and OR (and exclusive OR) to '+'. The NOT operator is analogous to a unary minus. This hierarchy is not absolutely standard ... some languages, notably Ada, treat all logical operators as having the same precedence level ... but it seems

natural.

Notice also the slight difference between the way the NOT and the unary minus are handled. In algebra, the unary minus is considered to go with the whole term, and so never appears but once in a given term. So an expression like

```
a * -b
```

or worse yet,

```
a - -b
```

is not allowed. In Boolean algebra, though, the expression

```
a AND NOT b
```

makes perfect sense, and the syntax shown allows for that.

RELOPS

OK, assuming that you're willing to accept the grammar I've shown here, we now have syntax rules for both arithmetic and Boolean algebra. The sticky part comes in when we have to combine the two. Why do we have to do that? Well, the whole subject came up because of the need to process the "predicates" (conditions) associated with control statements such as the IF. The predicate is required to have a Boolean value; that is, it must evaluate to either TRUE or FALSE. The branch is then taken or not taken, depending on that value. What we expect to see going on in procedure Condition, then, is the evaluation of a Boolean expression.

But there's more to it than that. A pure Boolean expression can indeed be the predicate of a control statement ... things like

```
IF a AND NOT b THEN ....
```

But more often, we see Boolean algebra show up in such things as

```
IF (x >= 0) and (x <= 100) THEN ...
```

Here, the two terms in parens are Boolean expressions, but the individual terms being compared: x, 0, and 100, are NUMERIC in nature. The RELATIONAL OPERATORS >= and <= are the catalysts by which the Boolean and the arithmetic ingredients get merged together.

Now, in the example above, the terms being compared are just that: terms. However, in general each side can be a math expression. So we can define a RELATION to be:

```
<relation> ::= <expression> <relop> <expression>  ,
```

where the expressions we're talking about here are the old numeric type, and the relops are any of the usual symbols

```
=, <> (or !=), <, >, <=, and >=
```

If you think about it a bit, you'll agree that, since this kind of predicate has a single Boolean value, TRUE or FALSE, as its result, it is really just another kind of factor. So we can expand the definition of a Boolean factor above to read:

```
<b-factor> ::=    <b-literal>
                | <b-variable>
                | (<b-expression>)
                | <relation>
```

THAT's the connection! The relops and the relation they define serve to wed the two kinds of algebra. It is worth noting that this implies a hierarchy where the arithmetic expression has a HIGHER precedence that a Boolean factor, and therefore than all the Boolean operators. If you write out the precedence levels for all the operators, you arrive at the following list:

Level	Syntax Element	Operator
0	factor	literal, variable
1	signed factor	unary minus
2	term	*, /
3	expression	+, -
4	b-factor	literal, variable, relop
5	not-factor	NOT
6	b-term	AND
7	b-expression	OR, XOR

If we're willing to accept that many precedence levels, this

grammar seems reasonable. Unfortunately, it won't work! The grammar may be great in theory, but it's no good at all in the practice of a top-down parser. To see the problem, consider the code fragment:

```
IF ((((((A + B + C) < 0 ) AND ....
```

When the parser is parsing this code, it knows after it sees the IF token that a Boolean expression is supposed to be next. So it can set up to begin evaluating such an expression. But the first expression in the example is an ARITHMETIC expression, A + B + C. What's worse, at the point that the parser has read this much of the input line:

```
IF ((((((A   ,
```

it still has no way of knowing which kind of expression it's dealing with. That won't do, because we must have different recognizers for the two cases. The situation can be handled without changing any of our definitions, but only if we're willing to accept an arbitrary amount of backtracking to work our way out of bad guesses. No compiler writer in his right mind would agree to that.

What's going on here is that the beauty and elegance of BNF grammar has met face to face with the realities of compiler technology.

To deal with this situation, compiler writers have had to make compromises so that a single parser can handle the grammar without backtracking.

FIXING THE GRAMMAR

The problem that we've encountered comes up because our definitions of both arithmetic and Boolean factors permit the use of parenthesized expressions. Since the definitions are recursive, we can end up with any number of levels of parentheses, and the parser can't know which kind of expression it's dealing with.

The solution is simple, although it ends up causing profound changes to our grammar. We can only allow parentheses in one kind of factor. The way to do that varies considerably from language to language. This is one place where there is NO agreement or convention to help us.

When Niklaus Wirth designed Pascal, the desire was to limit the number of levels of precedence (fewer parse routines, after all). So the OR and exclusive OR operators are treated just like an Addop and processed at the level of a math expression. Similarly, the AND is treated like a Mulop and processed with Term. The precedence levels are

Level	Syntax Element	Operator
0	factor	literal, variable
1	signed factor	unary minus, NOT
2	term	*, /, AND
3	expression	+, -, OR

Notice that there is only ONE set of syntax rules, applying to both kinds of operators. According to this grammar, then, expressions like

```
x + (y AND NOT z) DIV 3
```

are perfectly legal. And, in fact, they ARE ... as far as the parser is concerned. Pascal doesn't allow the mixing of arithmetic and Boolean variables, and things like this are caught at the SEMANTIC level, when it comes time to generate code for them, rather than at the syntax level.

The authors of C took a diametrically opposite approach: they treat the operators as different, and have something much more akin to our seven levels of precedence. In fact, in C there are no fewer than 17 levels! That's because C also has the operators '=', '+=' and its kin, '<<', '>>', '++', '--', etc. Ironically, although in C the arithmetic and Boolean operators are treated separately, the variables are NOT ... there are no Boolean or logical variables in C, so a Boolean test can be made on any integer value.

We'll do something that's sort of in-between. I'm tempted to stick mostly with the Pascal approach, since that seems the simplest from an implementation point of view, but it results in some funnies that I never liked very much, such as the fact that, in the expression

```
IF (c >= 'A') and (c <= 'Z') then ...
```

the parens above are REQUIRED. I never understood why before,

and neither my compiler nor any human ever explained it very well, either. But now, we can all see that the 'and' operator, having the precedence of a multiply, has a higher one than the relational operators, so without the parens the expression is equivalent to

```
     IF c >= ('A' and c) <= 'Z' then
```

which doesn't make sense.

In any case, I've elected to separate the operators into different levels, although not as many as in C.

```
 <b-expression> ::= <b-term> [<orop> <b-term>]*
 <b-term>       ::= <not-factor> [AND <not-factor>]*
 <not-factor>   ::= [NOT] <b-factor>
 <b-factor>     ::= <b-literal> | <b-variable> | <relation>
 <relation>     ::= | <expression> [<relop> <expression]
 <expression>   ::= <term> [<addop> <term>]*
 <term>         ::= <signed factor> [<mulop> factor]*
 <signed factor>::= [<addop>] <factor>
 <factor>       ::= <integer> | <variable> | (<b-expression>)
```

This grammar results in the same set of seven levels that I showed earlier. Really, it's almost the same grammar ... I just removed the option of parenthesized b-expressions as a possible b-factor, and added the relation as a legal form of b-factor.

There is one subtle but crucial difference, which is what makes the whole thing work. Notice the square brackets in the definition of a relation. This means that the relop and the second expression are OPTIONAL.

A strange consequence of this grammar (and one shared by C) is that EVERY expression is potentially a Boolean expression. The parser will always be looking for a Boolean expression, but will "settle" for an arithmetic one. To be honest, that's going to slow down the parser, because it has to wade through more layers of procedure calls. That's one reason why Pascal compilers tend to compile faster than C compilers. If it's raw speed you want, stick with the Pascal syntax.

THE PARSER

Now that we've gotten through the decision-making process, we can press on with development of a parser. You've done this with me several times now, so you know the drill: we begin with a fresh copy of the cradle, and begin adding procedures one by one. So let's do it.

We begin, as we did in the arithmetic case, by dealing only with Boolean literals rather than variables. This gives us a new kind of input token, so we're also going to need a new recognizer, and a new procedure to read instances of that token type. Let's start by defining the two new procedures:

```
{--------------------------------------------------------------}
{ Recognize a Boolean Literal }

function IsBoolean(c: char): Boolean;
begin
   IsBoolean := UpCase(c) in ['T', 'F'];
end;
```

```
{-------------------------------------------------------------}
{ Get a Boolean Literal }

function GetBoolean: Boolean;
var c: char;
begin
   if not IsBoolean(Look) then Expected('Boolean Literal');
   GetBoolean := UpCase(Look) = 'T';
   GetChar;
end;
{-------------------------------------------------------------}
```

Type these routines into your program. You can test them by adding into the main program the print statement

```
   WriteLn(GetBoolean);
```

OK, compile the program and test it. As usual, it's not very impressive so far, but it soon will be.

Now, when we were dealing with numeric data we had to arrange to generate code to load the values into D0. We need to do the same for Boolean data. The usual way to encode Boolean variables is to let 0 stand for FALSE, and some other value for TRUE. Many languages, such as C, use an integer 1 to represent it. But I prefer FFFF hex (or -1), because a bitwise NOT also becomes a Boolean NOT. So now we need to emit the right assembler code to load those values. The first cut at the Boolean expression parser (BoolExpression, of course) is:

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

procedure BoolExpression;
begin
   if not IsBoolean(Look) then Expected('Boolean Literal');
   if GetBoolean then
      EmitLn('MOVE #-1,D0')
   else
      EmitLn('CLR D0');
end;
{-------------------------------------------------------------}
```

Add this procedure to your parser, and call it from the main program (replacing the print statement you had just put there). As you can see, we still don't have much of a parser, but the output code is starting to look more realistic.

Next, of course, we have to expand the definition of a Boolean expression. We already have the BNF rule:

```
 <b-expression> ::= <b-term> [<orop> <b-term>]*
```

I prefer the Pascal versions of the "orops", OR and XOR. But since we are keeping to single-character tokens here, I'll encode those with '|' and '~'. The next version of BoolExpression is

almost a direct copy of the arithmetic procedure Expression:

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

procedure BoolOr;
begin
   Match('|');
   BoolTerm;
   EmitLn('OR (SP)+,D0');
end;




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

procedure BoolXor;
begin
   Match('~');
   BoolTerm;
   EmitLn('EOR (SP)+,D0');
end;


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

procedure BoolExpression;
begin
   BoolTerm;
   while IsOrOp(Look) do begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '|': BoolOr;
       '~': BoolXor;
      end;
   end;
end;
{---------------------------------------------------------------}
```

Note the new recognizer IsOrOp, which is also a copy, this time of IsAddOp:

```
{---------------------------------------------------------------}
{ Recognize a Boolean Orop }

function IsOrop(c: char): Boolean;
begin
   IsOrop := c in ['|', '~'];
end;
{---------------------------------------------------------------}
```

OK, rename the old version of BoolExpression to BoolTerm, then enter the code above. Compile and test this version. At this point, the output code is starting to look pretty good. Of course, it doesn't make much sense to do a lot of Boolean algebra on constant values, but we'll soon be expanding the types of Booleans we deal with.

You've probably already guessed what the next step is: The Boolean version of Term.

Rename the current procedure BoolTerm to NotFactor, and enter the following new version of BoolTerm. Note that is is much simpler than the numeric version, since there is no equivalent of division.

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

procedure BoolTerm;
begin
   NotFactor;
   while Look = '&' do begin
      EmitLn('MOVE D0,-(SP)');
      Match('&');
      NotFactor;
      EmitLn('AND (SP)+,D0');
   end;
end;
{--------------------------------------------------------------}
```

Now, we're almost home. We are translating complex Boolean expressions, although only for constant values. The next step is to allow for the NOT. Write the following procedure:

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

procedure NotFactor;
begin
   if Look = '!' then begin
      Match('!');
      BoolFactor;
      EmitLn('EOR #-1,D0');
      end
   else
      BoolFactor;
end;
{--------------------------------------------------------------}
```

And rename the earlier procedure to BoolFactor. Now try that. At this point the parser should be able to handle any Boolean expression you care to throw at it. Does it? Does it trap badly formed expressions?

If you've been following what we did in the parser for math expressions, you know that what we did next was to expand the definition of a factor to include variables and parens. We don't have to do that for the Boolean factor, because those little items get taken care of by the next step. It takes just a one line addition to BoolFactor to take care of relations:

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

procedure BoolFactor;
begin
   if IsBoolean(Look) then
      if GetBoolean then
         EmitLn('MOVE #-1,D0')
      else
         EmitLn('CLR D0')
```

```
      else Relation;
end;
{-------------------------------------------------------------}
```

You might be wondering when I'm going to provide for Boolean variables and parenthesized Boolean expressions. The answer is, I'm NOT! Remember, we took those out of the grammar earlier. Right now all I'm doing is encoding the grammar we've already agreed upon. The compiler itself can't tell the difference between a Boolean variable or expression and an arithmetic one ... all of those will be handled by Relation, either way.

Of course, it would help to have some code for Relation. I don't feel comfortable, though, adding any more code without first checking out what we already have. So for now let's just write a dummy version of Relation that does nothing except eat the current character, and write a little message:

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

procedure Relation;
begin
   WriteLn('<Relation>');
   GetChar;
end;
{-------------------------------------------------------------}
```

OK, key in this code and give it a try. All the old things should still work ... you should be able to generate the code for ANDs, ORs, and NOTs. In addition, if you type any alphabetic character you should get a little <Relation> place-holder, where a Boolean factor should be. Did you get that? Fine, then let's move on to the full-blown version of Relation.

To get that, though, there is a bit of groundwork that we must lay first. Recall that a relation has the form

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

Since we have a new kind of operator, we're also going to need a new Boolean function to recognize it. That function is shown below. Because of the single-character limitation, I'm sticking to the four operators that can be encoded with such a character (the "not equals" is encoded by '#').

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

function IsRelop(c: char): Boolean;
begin
   IsRelop := c in ['=', '#', '<', '>'];
end;
{-------------------------------------------------------------}
```

Now, recall that we're using a zero or a -1 in register D0 to represent a Boolean value, and also that the loop constructs expect the flags to be set to correspond. In implementing all this on the 68000, things get a a little bit tricky.

Since the loop constructs operate only on the flags, it would be nice (and also quite efficient) just to set up those flags, and not load anything into D0 at all. This would be fine for the loops and branches, but remember that the relation can be used ANYWHERE a Boolean factor could be used. We may be storing its result to a Boolean variable. Since we can't know at this point how the result is going to be used, we must allow for BOTH cases.

Comparing numeric data is easy enough ... the 68000 has an operation for that ... but it sets the flags, not a value. What's more, the flags will always be set the same (zero if equal, etc.), while we need the zero flag set differently for the each of the different relops.

The solution is found in the 68000 instruction Scc, which sets a byte value to 0000 or FFFF (funny how that works!) depending upon the result of the specified condition. If we make the destination byte to be D0, we get the Boolean value needed.

Unfortunately, there's one final complication: unlike almost every other instruction in the 68000 set, Scc does NOT reset the condition flags to match the data being stored. So we have to do one last step, which is to test D0 and set the flags to match it. It must seem to be a trip around the moon to get what we want: we first perform the test, then test the flags to set data into D0, then test D0 to set the flags again. It is sort of roundabout, but it's the most straightforward way to get the flags right, and after all it's only a couple of instructions.

I might mention here that this area is, in my opinion, the one that represents the biggest difference between the efficiency of hand-coded assembler language and compiler-generated code. We have seen already that we lose efficiency in arithmetic operations, although later I plan to show you how to improve that a bit. We've also seen that the control constructs themselves can be done quite efficiently ... it's usually very difficult to improve on the code generated for an IF or a WHILE. But virtually every compiler I've ever seen generates terrible code, compared to assembler, for the computation of a Boolean function, and particularly for relations. The reason is just what I've hinted at above. When I'm writing code in assembler, I go ahead and perform the test the most convenient way I can, and then set up the branch so that it goes the way it should. In effect, I "tailor" every branch to the situation. The compiler can't do that (practically), and it also can't know that we don't want to store the result of the test as a Boolean variable. So it must generate the code in a very strict order, and it often ends up loading the result as a Boolean that never gets used for anything.

In any case, we're now ready to look at the code for Relation. It's shown below with its companion procedures:

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

procedure Equals;
begin
   Match('=');
   Expression;
   EmitLn('CMP (SP)+,D0');
   EmitLn('SEQ D0');
end;
```

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

procedure NotEquals;
begin
   Match('#');
   Expression;
   EmitLn('CMP (SP)+,D0');
   EmitLn('SNE D0');
end;


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

procedure Less;
begin
   Match('<');
   Expression;
   EmitLn('CMP (SP)+,D0');
   EmitLn('SGE D0');
end;


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

procedure Greater;
begin
   Match('>');
   Expression;
   EmitLn('CMP (SP)+,D0');
   EmitLn('SLE D0');
end;


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

procedure Relation;
begin
   Expression;
   if IsRelop(Look) then begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '=': Equals;
       '#': NotEquals;
       '<': Less;
       '>': Greater;
      end;
   EmitLn('TST D0');
   end;
end;
{--------------------------------------------------------------}
```

Now, that call to Expression looks familiar! Here is where the editor of your system comes in handy. We have already generated code for Expression and its buddies in previous sessions. You can copy them into your file now. Remember to use the single-character versions. Just to be certain, I've duplicated the arithmetic procedures below. If you're observant, you'll also see that I've changed them a little to make them correspond to the latest version of the syntax. This change is NOT necessary, so you may prefer to hold off on that until you're sure

everything is working.

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

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


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

procedure Expression; Forward;

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


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


procedure SignedFactor;
begin
   if Look = '+' then
      GetChar;
   if Look = '-' then begin
      GetChar;
      if IsDigit(Look) then
         EmitLn('MOVE #-' + GetNum + ',D0')
      else begin
         Factor;
         EmitLn('NEG D0');
      end;
   end
   else Factor;
end;




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

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


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

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


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

procedure Term;
begin
   SignedFactor;
   while Look in ['*', '/'] do begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '*': Multiply;
       '/': Divide;
      end;
   end;
end;


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

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


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

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




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

procedure Expression;
begin
```

```
   Term;
   while IsAddop(Look) do begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '+': Add;
       '-': Subtract;
      end;
   end;
end;
{--------------------------------------------------------------}
```

There you have it ... a parser that can handle both arithmetic AND Boolean algebra, and things that combine the two through the use of relops. I suggest you file away a copy of this parser in a safe place for future reference, because in our next step we're going to be chopping it up.

MERGING WITH CONTROL CONSTRUCTS

At this point, let's go back to the file we had previously built that parses control constructs. Remember those little dummy procedures called Condition and Expression? Now you know what goes in their places!

I warn you, you're going to have to do some creative editing here, so take your time and get it right. What you need to do is to copy all of the procedures from the logic parser, from Ident through BoolExpression, into the parser for control constructs. Insert them at the current location of Condition. Then delete that procedure, as well as the dummy Expression. Next, change every call to Condition to refer to BoolExpression instead. Finally, copy the procedures IsMulop, IsOrOp, IsRelop, IsBoolean, and GetBoolean into place. That should do it.

Compile the resulting program and give it a try. Since we haven't used this program in awhile, don't forget that we used single-character tokens for IF, WHILE, etc. Also don't forget that any letter not a keyword just gets echoed as a block.

Try

```
     ia=bxlye
```

which stands for "IF a=b X ELSE Y ENDIF".

What do you think? Did it work? Try some others.

ADDING ASSIGNMENTS

As long as we're this far, and we already have the routines for expressions in place, we might as well replace the "blocks" with real assignment statements. We've already done that before, so it won't be too hard. Before taking that step, though, we need to fix something else.

We're soon going to find that the one-line "programs" that we're having to write here will really cramp our style. At the moment we have no cure for that, because our parser doesn't recognize the end-of-line characters, the carriage return (CR) and the line feed (LF). So before going any further let's plug that hole.

There are a couple of ways to deal with the CR/LFs. One (the

C/Unix approach) is just to treat them as additional white space characters and ignore them. That's actually not such a bad approach, but it does sort of produce funny results for our parser as it stands now. If it were reading its input from a source file as any self-respecting REAL compiler does, there would be no problem. But we're reading input from the keyboard, and we're sort of conditioned to expect something to happen when we hit the return key. It won't, if we just skip over the CR and LF (try it). So I'm going to use a different method here, which is NOT necessarily the best approach in the long run. Consider it a temporary kludge until we're further along.

Instead of skipping the CR/LF, We'll let the parser go ahead and catch them, then introduce a special procedure, analogous to SkipWhite, that skips them only in specified "legal" spots.

Here's the procedure:

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

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

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

Now, add two calls to Fin in procedure Block, like this:

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

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

Now, you'll find that you can use multiple-line "programs." The only restriction is that you can't separate an IF or WHILE token from its predicate.

Now we're ready to include the assignment statements. Simply change that call to Other in procedure Block to a call to Assignment, and add the following procedure, copied from one of our earlier programs. Note that Assignment now calls BoolExpression, so that we can assign Boolean variables.

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

procedure Assignment;
var Name: char;
begin
   Name := GetName;
   Match('=');
   BoolExpression;
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)');
end;
{-------------------------------------------------------------}
```

With that change, you should now be able to write reasonably realistic-looking programs, subject only to our limitation on single-character tokens. My original intention was to get rid of that limitation for you, too. However, that's going to require a fairly major change to what we've done so far. We need a true lexical scanner, and that requires some structural changes. They are not BIG changes that require us to throw away all of what we've done so far ... with care, it can be done with very minimal changes, in fact. But it does require that care.

This installment has already gotten pretty long, and it contains some pretty heavy stuff, so I've decided to leave that step until next time, when you've had a little more time to digest what we've done and are ready to start fresh.

In the next installment, then, we'll build a lexical scanner and eliminate the single-character barrier once and for all. We'll also write our first complete compiler, based on what we've done in this session. See you then.