LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

4 Aug 1988

Part III: MORE EXPRESSIONS

INTRODUCTION

In the last installment, we examined the techniques used to parse and translate a general math expression. We ended up with a simple parser that could handle arbitrarily complex expressions, with two restrictions:

- o No variables were allowed, only numeric factors

- o The numeric factors were limited to single digits

In this installment, we'll get rid of those restrictions. We'll also extend what we've done to include assignment statements function calls and. Remember, though, that the second restriction was mainly self-imposed ... a choice of convenience on our part, to make life easier and to let us concentrate on the fundamental concepts. As you'll see in a bit, it's an easy restriction to get rid of, so don't get too hung up about it.

We'll use the trick when it serves us to do so, confident that we can discard it when we're ready to.

VARIABLES

Most expressions that we see in practice involve variables, such as

```
               b * b + 4 * a * c
```

No parser is much good without being able to deal with them. Fortunately, it's also quite easy to do.

Remember that in our parser as it currently stands, there are two kinds of factors allowed: integer constants and expressions within parentheses. In BNF notation,

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

The '|' stands for "or", meaning of course that either form is a legal form for a factor. Remember, too, that we had no trouble knowing which was which ... the lookahead character is a left paren '(' in one case, and a digit in the other.

It probably won't come as too much of a surprise that a variable is just another kind of factor. So we extend the BNF above to read:

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

Again, there is no ambiguity: if the lookahead character is a letter, we have a variable; if a digit, we have a number. Back when we translated the number, we just issued code to load the number, as immediate data, into D0. Now we do the same, only we load a variable.

A minor complication in the code generation arises from the fact that most 68000 operating systems, including the SK*DOS that I'm using, require the code to be written in "position-independent" form, which basically means that everything is PC-relative. The format for a load in this language is

```
               MOVE X(PC),D0
```

where X is, of course, the variable name. Armed with that, let's modify the current version of Factor to read:

```
{---------------------------------------------------------------}
{ 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
      EmitLn('MOVE ' + GetName + '(PC),D0')
   else
      EmitLn('MOVE #' + GetNum + ',D0');
```

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

I've remarked before how easy it is to add extensions to the parser, because of the way it's structured. You can see that this still holds true here. This time it cost us all of two extra lines of code. Notice, too, how the if-else-else structure exactly parallels the BNF syntax equation.

OK, compile and test this new version of the parser. That didn't hurt too badly, did it?

FUNCTIONS

There is only one other common kind of factor supported by most languages: the function call. It's really too early for us to deal with functions well, because we haven't yet addressed the issue of parameter passing. What's more, a "real" language would include a mechanism to support more than one type, one of which should be a function type. We haven't gotten there yet, either. But I'd still like to deal with functions now for a couple of reasons.   First, it lets us finally wrap up the parser in something very close to its final form, and second, it brings up a new issue which is very much worth talking about.

Up till now, we've been able to write what is called a "predictive parser." That means that at any point, we can know by looking at the current lookahead character exactly what to do next. That isn't the case when we add functions. Every language has some naming rules for what constitutes a legal identifier. For the present, ours is simply that it is one of the letters 'a'..'z'. The problem is that a variable name and a function name obey the same rules.   So how can we tell which is which? One way is to require that they each be declared before they are used.   Pascal takes that approach. The other is that we might require a function to be followed by a (possibly empty) parameter list. That's the rule used in C.

Since we don't yet have a mechanism for declaring types, let's use the C rule for now. Since we also don't have a mechanism to deal with parameters, we can only handle empty lists, so our function calls will have the form

```
                    x()  .
```

Since we're not dealing with parameter lists yet, there is nothing to do but to call the function, so we need only to issue a BSR (call) instead of a MOVE.

Now that there are two possibilities for the "If IsAlpha" branch of the test in Factor, let's treat them in a separate procedure. Modify Factor to read:

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

and insert before it the new procedure

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

OK, compile and test this version. Does it parse all legal expressions? Does it correctly flag badly formed ones?

The important thing to notice is that even though we no longer have a predictive parser, there is little or no complication added with the recursive descent approach that we're using. At the point where Factor finds an identifier (letter), it doesn't know whether it's a variable name or a function name, nor does it really care. It simply passes it on to Ident and leaves it up to that procedure to figure it out. Ident, in turn, simply tucks away the identifier and then reads one more character to decide which kind of identifier it's dealing with.

Keep this approach in mind. It's a very powerful concept, and it should be used whenever you encounter an ambiguous situation requiring further lookahead. Even if you had to look several tokens ahead, the principle would still work.

MORE ON ERROR HANDLING

As long as we're talking philosophy, there's another important issue to point out: error handling. Notice that although the parser correctly rejects (almost) every malformed expression we can throw at it, with a meaningful error message, we haven't really had to do much work to make that happen. In fact, in the whole parser per se (from Ident through Expression) there are only two calls to the error routine, Expected. Even those aren't necessary ... if you'll look again in Term and Expression, you'll see that those statements can't be reached. I put them in early on as a bit of insurance, but they're no longer needed. Why don't you delete them now?

So how did we get this nice error handling virtually for free? It's simply that I've carefully avoided reading a character directly using GetChar. Instead, I've relied on the error handling in GetName, GetNum, and Match to do all the error checking for me. Astute readers will notice that some of the calls to Match (for example, the ones in Add and Subtract) are also unnecessary ... we already know what the character is by the time we get there ... but it maintains a certain symmetry to leave them in, and the general rule to always use Match instead of GetChar is a good one.

I mentioned an "almost" above. There is a case where our error handling leaves a bit to be desired. So far we haven't told our parser what and end-of-line looks like, or what to do with embedded white space. So a space character (or any other character not part of the recognized character set) simply causes the parser to terminate, ignoring the unrecognized characters.

It could be argued that this is reasonable behavior at this point. In a "real" compiler, there is usually another statement following the one we're working on, so any characters not treated as part of our expression will either be used for or rejected as part of the next one.

But it's also a very easy thing to fix up, even if it's only temporary. All we have to do is assert that the expression should end with an end-of-line , i.e., a carriage return.

To see what I'm talking about, try the input line

```
               1+2 <space> 3+4
```

See how the space was treated as a terminator? Now, to make the compiler properly flag this, add the line

```
               if Look <> CR then Expected('Newline');
```

in the main program, just after the call to Expression. That catches anything left over in the input stream. Don't forget to define CR in the const statement:

```
               CR = ^M;
```

As usual, recompile the program and verify that it does what it's supposed to.

ASSIGNMENT STATEMENTS

OK, at this point we have a parser that works very nicely. I'd like to point out that we got it using only 88 lines of executable code, not counting what was in the cradle. The compiled object file is a whopping 4752 bytes. Not bad, considering we weren't trying very hard to save either source code or object size. We just stuck to the KISS principle.

Of course, parsing an expression is not much good without having something to do with it afterwards. Expressions USUALLY (but not always) appear in assignment statements, in the form

```
          <Ident> = <Expression>
```

We're only a breath away from being able to parse an assignment statement, so let's take that last step. Just after procedure Expression, add the following new procedure:

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

procedure Assignment;
```

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

Note again that the code exactly parallels the BNF. And notice further that the error checking was painless, handled by GetName and Match.

The reason for the two lines of assembler has to do with a peculiarity in the 68000, which requires this kind of construct for PC-relative code.

Now change the call to Expression, in the main program, to one to Assignment. That's all there is to it.

Son of a gun! We are actually compiling assignment statements. If those were the only kind of statements in a language, all we'd have to do is put this in a loop and we'd have a full-fledged compiler!

Well, of course they're not the only kind. There are also little items like control statements (IFs and loops), procedures, declarations, etc. But cheer up. The arithmetic expressions that we've been dealing with are among the most challenging in a language. Compared to what we've already done, control statements will be easy. I'll be covering them in the fifth installment. And the other statements will all fall in line, as long as we remember to KISS.

MULTI-CHARACTER TOKENS

Throughout this series, I've been carefully restricting everything we do to single-character tokens, all the while assuring you that it wouldn't be difficult to extend to multi-character ones. I don't know if you believed me or not ... I wouldn't really blame you if you were a bit skeptical. I'll continue to use that approach in the sessions which follow, because it helps keep complexity away. But I'd like to back up those assurances, and wrap up this portion of the parser, by showing you just how easy that extension really is. In the process, we'll also provide for embedded white space. Before you make the next few changes, though, save the current version of the parser away under another name. I have some more uses for it in the next installment, and we'll be working with the single-character version.

Most compilers separate out the handling of the input stream into a separate module called the lexical scanner. The idea is that the scanner deals with all the character-by-character input, and returns the separate units (tokens) of the stream. There may come a time when we'll want to do something like that, too, but for now there is no need. We can handle the multi-character tokens that we need by very slight and very local modifications to GetName and GetNum.

The usual definition of an identifier is that the first character must be a letter, but the rest can be alphanumeric (letters or numbers). To deal with this, we need one other recognizer function

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

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

Add this function to your parser. I put mine just after IsDigit. While you're at it, might as well include it as a permanent member of Cradle, too.

Now, we need to modify function GetName to return a string instead of a character:

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

function GetName: string;
var Token: string;
begin
   Token := '';
   if not IsAlpha(Look) then Expected('Name');
   while IsAlNum(Look) do begin
      Token := Token + UpCase(Look);
      GetChar;
   end;
   GetName := Token;
end;
{--------------------------------------------------------------}
```

Similarly, modify GetNum to read:

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

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

Amazingly enough, that is virtually all the changes required to the parser! The local variable Name in procedures Ident and Assignment was originally declared as "char", and must now be declared string[8]. (Clearly, we could make the string length longer if we chose, but most assemblers limit the length anyhow.) Make this change, and then recompile and test. _NOW_ do you believe that it's a simple change?

WHITE SPACE

Before we leave this parser for awhile, let's address the issue of white space. As it stands now, the parser will barf (or simply terminate) on a single space character embedded anywhere in the input stream. That's pretty unfriendly behavior. So let's "productionize" the thing a bit by eliminating this last restriction.

The key to easy handling of white space is to come up with a simple rule for how the parser should treat the input stream, and to enforce that rule everywhere. Up till now, because white space wasn't permitted, we've been able to assume that after each parsing action, the lookahead character Look contains the next meaningful character, so we could test it immediately. Our design was based upon this principle.

It still sounds like a good rule to me, so that's the one we'll use. This means that every routine that advances the input stream must skip over white space, and leave the next non-white character in Look. Fortunately, because we've been careful to use GetName, GetNum, and Match for most of our input processing, it is only those three routines (plus Init) that we need to modify.

Not surprisingly, we start with yet another new recognizer routine:

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

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

We also need a routine that will eat white-space characters, until it finds a non-white one:

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

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

Now, add calls to SkipWhite to Match, GetName, and GetNum as shown below:

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

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

```
end;


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

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


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

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

(Note that I rearranged Match a bit, without changing the functionality.)

Finally, we need to skip over leading blanks where we "prime the pump" in Init:

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

procedure Init;
begin
   GetChar;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

Make these changes and recompile the program. You will find that you will have to move Match below SkipWhite, to avoid an error message from the Pascal compiler. Test the program as always to make sure it works properly.

Since we've made quite a few changes during this session, I'm reproducing the entire parser below:

```
{--------------------------------------------------------------}
program parse;

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

```
{ Constant Declarations }

const TAB = ^I;
       CR = ^M;

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

var Look: char;              { Lookahead Character }

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

procedure GetChar;
begin
   Read(Look);
end;

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

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


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

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


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

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


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

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


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

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


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

```
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 White Space }

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


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

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


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

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


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

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


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

```
function GetNum: string;
var Value: string;
begin
   Value := '';
   if not IsDigit(Look) then Expected('Integer');
   while IsDigit(Look) do begin
      Value := Value + Look;
      GetChar;
   end;
   GetNum := Value;
   SkipWhite;
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 Identifier }

procedure Ident;
var Name: string[8];
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;


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


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

procedure Assignment;
var Name: string[8];
begin
   Name := GetName;
   Match('=');
   Expression;
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)')
end;


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

procedure Init;
begin
   GetChar;
   SkipWhite;
end;


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

begin
   Init;
   Assignment;
   If Look <> CR then Expected('NewLine');
end.
{--------------------------------------------------------------}
```

Now the parser is complete. It's got every feature we can put in a one-line "compiler." Tuck it away in a safe place. Next time we'll move on to a new subject, but we'll still be talking about expressions for quite awhile. Next installment, I plan to talk a bit about interpreters as opposed to compilers, and show you how the structure of the parser changes a bit as we change what sort of action has to be taken. The information we pick up there will serve us in good stead later on, even if you have no interest in interpreters. See you next time.