LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

24 July 1988

Part IV: INTERPRETERS

INTRODUCTION

In the first three installments of this series, we've looked at parsing and compiling math expressions, and worked our way gradually and methodically from dealing with very simple one-term, one-character "expressions" up through more general ones, finally arriving at a very complete parser that could parse and translate complete assignment statements, with multi-character tokens, embedded white space, and function calls. This time, I'm going to walk you through the process one more time, only with the goal of interpreting rather than compiling object code.

Since this is a series on compilers, why should we bother with interpreters? Simply because I want you to see how the nature of the parser changes as we change the goals. I also want to unify the concepts of the two types of translators, so that you can see not only the differences, but also the similarities.

Consider the assignment statement

```
x = 2 * y + 3
```

In a compiler, we want the target CPU to execute this assignment at EXECUTION time. The translator itself doesn't do any arithmetic ... it only issues the object code that will cause the CPU to do it when the code is executed. For the example above, the compiler would issue code to compute the expression and store the results in variable x.

For an interpreter, on the other hand, no object code is generated. Instead, the arithmetic is computed immediately, as the parsing is going on. For the example, by the time parsing of the statement is complete, x will have a new value.

The approach we've been taking in this whole series is called "syntax-driven translation." As you are aware by now, the structure of the parser is very closely tied to the syntax of the productions we parse. We have built Pascal procedures that recognize every language construct. Associated with each of these constructs (and procedures) is a corresponding "action," which does whatever makes sense to do once a construct has been recognized. In our compiler so far, every action involves emitting object code, to be executed later at execution time. In an interpreter, every action involves something to be done immediately.

What I'd like you to see here is that the layout ... the structure ... of the parser doesn't change. It's only the actions that change. So if you can write an interpreter for a given language, you can also write a compiler, and vice versa. Yet, as you will see, there ARE differences, and significant ones. Because the actions are different, the procedures that do the recognizing end up being written differently. Specifically, in the interpreter the recognizing procedures end up being coded as FUNCTIONS that return numeric values to their callers. None of the parsing routines for our compiler did that.

Our compiler, in fact, is what we might call a "pure" compiler. Each time a construct is recognized, the object code is emitted IMMEDIATELY. (That's one reason the code is not very efficient.) The interpreter we'll be building here is a pure interpreter, in the sense that there is no translation, such as "tokenizing," performed on the source code. These represent the two extremes of translation. In the real world, translators are rarely so pure, but tend to have bits of each technique.

I can think of several examples. I've already mentioned one: most interpreters, such as Microsoft BASIC, for example, translate the source code (tokenize it) into an intermediate form so that it'll be easier to parse real time.

Another example is an assembler. The purpose of an assembler, of course, is to produce object code, and it normally does that on a one-to-one basis: one object instruction per line of source code. But almost every assembler also permits expressions as arguments. In this case, the expressions are always constant expressions, and so the assembler isn't supposed to issue object code for them. Rather, it "interprets" the expressions and computes the corresponding constant result, which is what it actually emits as object code.

As a matter of fact, we could use a bit of that ourselves. The translator we built in the previous installment will dutifully spit out object code for complicated expressions, even though every term in the expression is a constant. In that case it

would be far better if the translator behaved a bit more like an interpreter, and just computed the equivalent constant result.

There is a concept in compiler theory called "lazy" translation. The idea is that you typically don't just emit code at every action. In fact, at the extreme you don't emit anything at all, until you absolutely have to. To accomplish this, the actions associated with the parsing routines typically don't just emit code. Sometimes they do, but often they simply return information back to the caller. Armed with such information, the caller can then make a better choice of what to do.

For example, given the statement

```
x = x + 3 - 2 - (5 - 4)  ,
```

our compiler will dutifully spit out a stream of 18 instructions to load each parameter into registers, perform the arithmetic, and store the result. A lazier evaluation would recognize that the arithmetic involving constants can be evaluated at compile time, and would reduce the expression to

```
x = x + 0  .
```

An even lazier evaluation would then be smart enough to figure out that this is equivalent to

```
x = x  ,
```

which calls for no action at all. We could reduce 18 instructions to zero!

Note that there is no chance of optimizing this way in our translator as it stands, because every action takes place immediately.

Lazy expression evaluation can produce significantly better object code than we have been able to so far. I warn you, though: it complicates the parser code considerably, because each routine now has to make decisions as to whether to emit object code or not. Lazy evaluation is certainly not named that because it's easier on the compiler writer!

Since we're operating mainly on the KISS principle here, I won't go into much more depth on this subject. I just want you to be aware that you can get some code optimization by combining the techniques of compiling and interpreting. In particular, you should know that the parsing routines in a smarter translator will generally return things to their caller, and sometimes expect things as well. That's the main reason for going over interpretation in this installment.

THE INTERPRETER

OK, now that you know WHY we're going into all this, let's do it. Just to give you practice, we're going to start over with a bare cradle and build up the translator all over again. This time, of course, we can go a bit faster.

Since we're now going to do arithmetic, the first thing we need to do is to change function GetNum, which up till now has always returned a character (or string). Now, it's better for it to return an integer. MAKE A COPY of the cradle (for goodness's sake, don't change the version in Cradle itself!!) and modify GetNum as follows:

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

function GetNum: integer;
begin
   if not IsDigit(Look) then Expected('Integer');
   GetNum := Ord(Look) - Ord('0');
   GetChar;
end;
{--------------------------------------------------------------}
```

Now, write the following version of Expression:

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

function Expression: integer;
begin
   Expression := GetNum;
end;
{--------------------------------------------------------------}
```

Finally, insert the statement

```
   Writeln(Expression);
```

at the end of the main program. Now compile and test.

All this program does is to "parse" and translate a single integer "expression." As always, you should make sure that it does that with the digits 0..9, and gives an error message for anything else. Shouldn't take you very long!

OK, now let's extend this to include addops. Change Expression to read:

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

function Expression: integer;
var Value: integer;
begin
   if IsAddop(Look) then
      Value := 0
   else
      Value := GetNum;
   while IsAddop(Look) do begin
      case Look of
       '+': begin
              Match('+');
              Value := Value + GetNum;
            end;
       '-': begin
              Match('-');
              Value := Value - GetNum;
            end;
      end;
   end;
   Expression := Value;
end;
{--------------------------------------------------------------}
```

The structure of Expression, of course, parallels what we did before, so we shouldn't have too much trouble debugging it. There's been a SIGNIFICANT development, though, hasn't there? Procedures Add and Subtract went away! The reason is that the action to be taken requires BOTH arguments of the operation. I could have chosen to retain the procedures and pass into them the value of the expression to date, which is Value. But it seemed cleaner to me to keep Value as strictly a local variable, which meant that the code for Add and Subtract had to be moved in line. This result suggests that, while the structure we had developed was nice and clean for our simple-minded translation scheme, it probably wouldn't do for use with lazy evaluation. That's a little tidbit we'll probably want to keep in mind for later.

OK, did the translator work? Then let's take the next step. It's not hard to figure out what procedure Term should now look like. Change every call to GetNum in function Expression to a call to Term, and then enter the following form for Term:

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

function Term: integer;
var Value: integer;
begin
   Value := GetNum;
   while Look in ['*', '/'] do begin
      case Look of
       '*': begin
               Match('*');
               Value := Value * GetNum;
            end;
       '/': begin
               Match('/');
               Value := Value div GetNum;
            end;
      end;
   end;
   Term := Value;
end;
{--------------------------------------------------------------}
```

Now, try it out. Don't forget two things: first, we're dealing with integer division, so, for example, 1/3 should come out zero. Second, even though we can output multi-digit results, our input is still restricted to single digits.

That seems like a silly restriction at this point, since we have already seen how easily function GetNum can be extended. So let's go ahead and fix it right now. The new version is

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

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

```
      GetChar;
   end;
   GetNum := Value;
end;
{--------------------------------------------------------------}
```

If you've compiled and tested this version of the interpreter, the next step is to install function Factor, complete with parenthesized expressions. We'll hold off a bit longer on the variable names. First, change the references to GetNum, in function Term, so that they call Factor instead. Now code the following version of Factor:

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

function Expression: integer; Forward;

function Factor: integer;
begin
   if Look = '(' then begin
      Match('(');
      Factor := Expression;
      Match(')');
      end
   else
       Factor := GetNum;
end;
{--------------------------------------------------------------}
```

That was pretty easy, huh? We're rapidly closing in on a useful interpreter.

A LITTLE PHILOSOPHY

Before going any further, there's something I'd like to call to your attention. It's a concept that we've been making use of in all these sessions, but I haven't explicitly mentioned it up till now. I think it's time, because it's a concept so useful, and so powerful, that it makes all the difference between a parser that's trivially easy, and one that's too complex to deal with.

In the early days of compiler technology, people had a terrible time figuring out how to deal with things like operator precedence ... the way that multiply and divide operators take precedence over add and subtract, etc. I remember a colleague of some thirty years ago, and how excited he was to find out how to do it. The technique used involved building two stacks, upon which you pushed each operator or operand. Associated with each operator was a precedence level, and the rules required that you only actually performed an operation ("reducing" the stack) if the precedence level showing on top of the stack was correct. To make life more interesting, an operator like ')' had different precedence levels, depending upon whether or not it was already on the stack. You had to give it one value before you put it on the stack, and another to decide when to take it off. Just for the experience, I worked all of this out for myself a few years ago, and I can tell you that it's very tricky.

We haven't had to do anything like that. In fact, by now the parsing of an arithmetic statement should seem like child's play. How did we get so lucky? And where did the precedence stacks go?

A similar thing is going on in our interpreter above. You just KNOW that in order for it to do the computation of arithmetic statements (as opposed to the parsing of them), there have to be numbers pushed onto a stack somewhere. But where is the stack?

Finally, in compiler textbooks, there are a number of places where stacks and other structures are discussed. In the other leading parsing method (LR), an explicit stack is used. In fact, the technique is very much like the old way of doing arithmetic expressions. Another concept is that of a parse tree. Authors like to draw diagrams of the tokens in a statement, connected into a tree with operators at the internal nodes. Again, where are the trees and stacks in our technique? We haven't seen any. The answer in all cases is that the structures are implicit, not explicit. In any computer language, there is a stack involved every time you call a subroutine. Whenever a subroutine is called, the return address is pushed onto the CPU stack. At the end of the subroutine, the address is popped back off and control is transferred there. In a recursive language such as Pascal, there can also be local data pushed onto the stack, and it, too, returns when it's needed.

For example, function Expression contains a local parameter called Value, which it fills by a call to Term. Suppose, in its next call to Term for the second argument, that Term calls Factor, which recursively calls Expression again. That "in-stance" of Expression gets another value for its copy of Value. What happens to the first Value? Answer: it's still on the stack, and will be there again when we return from our call sequence.

In other words, the reason things look so simple is that we've been making maximum use of the resources of the language. The hierarchy levels and the parse trees are there, all right, but they're hidden within the structure of the parser, and they're taken care of by the order with which the various procedures are called. Now that you've seen how we do it, it's probably hard to imagine doing it any other way. But I can tell you that it took a lot of years for compiler writers to get that smart. The early compilers were too complex too imagine. Funny how things get easier with a little practice.

The reason I've brought all this up is as both a lesson and a warning. The lesson: things can be easy when you do them right. The warning: take a look at what you're doing. If, as you branch out on your own, you begin to find a real need for a separate stack or tree structure, it may be time to ask yourself if you're looking at things the right way. Maybe you just aren't using the facilities of the language as well as you could be.

The next step is to add variable names. Now, though, we have a slight problem. For the compiler, we had no problem in dealing with variable names ... we just issued the names to the assembler and let the rest of the program take care of allocating storage for them. Here, on the other hand, we need to be able to fetch the values of the variables and return them as the return values of Factor. We need a storage mechanism for these variables.

Back in the early days of personal computing, Tiny BASIC lived. It had a grand total of 26 possible variables: one for each letter of the alphabet. This fits nicely with our concept of single-character tokens, so we'll try the same trick. In the beginning of your interpreter, just after the declaration of variable Look, insert the line:

```
               Table: Array['A'..'Z'] of integer;
```

We also need to initialize the array, so add this procedure:

```
{--------------------------------------------------------------}
{ Initialize the Variable Area }

procedure InitTable;
var i: char;
begin
   for i := 'A' to 'Z' do
      Table[i] := 0;
end;
{--------------------------------------------------------------}
```

You must also insert a call to InitTable, in procedure Init. DON'T FORGET to do that, or the results may surprise you!

Now that we have an array of variables, we can modify Factor to use it. Since we don't have a way (so far) to set the variables, Factor will always return zero values for them, but let's go ahead and extend it anyway. Here's the new version:

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

function Expression: integer; Forward;

function Factor: integer;
begin
   if Look = '(' then begin
      Match('(');
      Factor := Expression;
      Match(')');
      end
   else if IsAlpha(Look) then
      Factor := Table[GetName]
   else
       Factor := GetNum;
end;
{--------------------------------------------------------------}
```

As always, compile and test this version of the program. Even though all the variables are now zeros, at least we can correctly parse the complete expressions, as well as catch any badly formed expressions.

I suppose you realize the next step: we need to do an assignment statement so we can put something INTO the variables. For now, let's stick to one-liners, though we will soon be handling multiple statements.

The assignment statement parallels what we did before:

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



procedure Assignment;
```

```
var Name: char;
begin
   Name := GetName;
   Match('=');
   Table[Name] := Expression;
end;
{--------------------------------------------------------------}
```

To test this, I added a temporary write statement in the main program, to print out the value of A. Then I tested it with various assignments to it.

Of course, an interpretive language that can only accept a single line of program is not of much value. So we're going to want to handle multiple statements. This merely means putting a loop around the call to Assignment. So let's do that now. But what should be the loop exit criterion? Glad you asked, because it brings up a point we've been able to ignore up till now.

One of the most tricky things to handle in any translator is to determine when to bail out of a given construct and go look for something else. This hasn't been a problem for us so far because we've only allowed for a single kind of construct ... either an expression or an assignment statement. When we start adding loops and different kinds of statements, you'll find that we have to be very careful that things terminate properly. If we put our interpreter in a loop, we need a way to quit. Terminating on a newline is no good, because that's what sends us back for another line. We could always let an unrecognized character take us out, but that would cause every run to end in an error message, which certainly seems uncool.

What we need is a termination character. I vote for Pascal's ending period ('.'). A minor complication is that Turbo ends every normal line with TWO characters, the carriage return (CR) and line feed (LF). At the end of each line, we need to eat these characters before processing the next one. A natural way to do this would be with procedure Match, except that Match's error message prints the character, which of course for the CR and/or LF won't look so great. What we need is a special procedure for this, which we'll no doubt be using over and over. Here it is:

```
{--------------------------------------------------------------}
{ Recognize and Skip Over a Newline }

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

Insert this procedure at any convenient spot ... I put mine just after Match. Now, rewrite the main program to look like this:

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

begin
```

```
   Init;
   repeat
      Assignment;
      NewLine;
   until Look = '.';
end.
{--------------------------------------------------------------}
```

Note that the test for a CR is now gone, and that there are also no error tests within NewLine itself. That's OK, though ... whatever is left over in terms of bogus characters will be caught at the beginning of the next assignment statement.

Well, we now have a functioning interpreter. It doesn't do us a lot of good, however, since we have no way to read data in or write it out. Sure would help to have some I/O!

Let's wrap this session up, then, by adding the I/O routines. Since we're sticking to single-character tokens, I'll use '?' to stand for a read statement, and '!' for a write, with the character immediately following them to be used as a one-token "parameter list." Here are the routines:

```
{--------------------------------------------------------------}
{ Input Routine }

procedure Input;
begin
   Match('?');
   Read(Table[GetName]);
end;


{--------------------------------------------------------------}
{ Output Routine }

procedure Output;
begin
   Match('!');
   WriteLn(Table[GetName]);
end;
{--------------------------------------------------------------}
```

They aren't very fancy, I admit ... no prompt character on input, for example ... but they get the job done.

The corresponding changes in the main program are shown below. Note that we use the usual trick of a case statement based upon the current lookahead character, to decide what to do.

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

begin
   Init;
   repeat
      case Look of
       '?': Input;
       '!': Output;
       else Assignment;
      end;
      NewLine;
   until Look = '.';
end.
{--------------------------------------------------------------}
```

You have now completed a real, working interpreter. It's pretty sparse, but it works just like the "big boys." It includes three kinds of program statements (and can tell the difference!), 26 variables, and I/O statements. The only things that it lacks, really, are control statements, subroutines, and some kind of program editing function. The program editing part, I'm going to pass on. After all, we're not here to build a product, but to learn things. The control statements, we'll cover in the next installment, and the subroutines soon after. I'm anxious to get on with that, so we'll leave the interpreter as it stands.

I hope that by now you're convinced that the limitation of single-character names and the processing of white space are easily taken care of, as we did in the last session. This time, if you'd like to play around with these extensions, be my guest ... they're "left as an exercise for the student." See you next time.