LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

16 April 1989

Part IX: A TOP VIEW

INTRODUCTION

In the previous installments, we have learned many of the techniques required to build a full-blown compiler. We've done both assignment statements (with Boolean and arithmetic expressions), relational operators, and control constructs. We still haven't addressed procedure or function calls, but even so we could conceivably construct a mini-language without them. I've always thought it would be fun to see just how small a language one could build that would still be useful. We're ALMOST in a position to do that now. The problem is: though we know how to parse and translate the constructs, we still don't know quite how to put them all together into a language.

In those earlier installments, the development of our programs had a decidedly bottom-up flavor. In the case of expression parsing, for example, we began with the very lowest level constructs, the individual constants and variables, and worked

our way up to more complex expressions.

Most people regard the top-down design approach as being better than the bottom-up one. I do too, but the way we did it certainly seemed natural enough for the kinds of things we were parsing.

You mustn't get the idea, though, that the incremental approach that we've been using in all these tutorials is inherently bottom-up. In this installment I'd like to show you that the approach can work just as well when applied from the top down ... maybe better. We'll consider languages such as C and Pascal, and see how complete compilers can be built starting from the top.

In the next installment, we'll apply the same technique to build a complete translator for a subset of the KISS language, which I'll be calling TINY. But one of my goals for this series is that you will not only be able to see how a compiler for TINY or KISS works, but that you will also be able to design and build compilers for your own languages. The C and Pascal examples will help. One thing I'd like you to see is that the natural structure of the compiler depends very much on the language being translated, so the simplicity and ease of construction of the compiler depends very much on letting the language set the program structure.

It's a bit much to produce a full C or Pascal compiler here, and we won't try. But we can flesh out the top levels far enough so that you can see how it goes.

Let's get started.

THE TOP LEVEL

One of the biggest mistakes people make in a top-down design is failing to start at the true top. They think they know what the overall structure of the design should be, so they go ahead and write it down.

Whenever I start a new design, I always like to do it at the absolute beginning. In program design language (PDL), this top level looks something like:

```
begin
   solve the problem
end
```

OK, I grant you that this doesn't give much of a hint as to what the next level is, but I like to write it down anyway, just to give me that warm feeling that I am indeed starting at the top.

For our problem, the overall function of a compiler is to compile a complete program. Any definition of the language, written in BNF, begins here. What does the top level BNF look like? Well, that depends quite a bit on the language to be translated. Let's take a look at Pascal.

THE STRUCTURE OF PASCAL

Most texts for Pascal include a BNF or "railroad-track" definition of the language. Here are the first few lines of one:

```
     <program> ::= <program-header> <block> '.'

     <program-header> ::= PROGRAM <ident>

     <block> ::= <declarations> <statements>
```

We can write recognizers to deal with each of these elements, just as we've done before. For each one, we'll use our familiar single-character tokens to represent the input, then flesh things out a little at a time. Let's begin with the first recognizer: the program itself.

To translate this, we'll start with a fresh copy of the Cradle. Since we're back to single-character names, we'll just use a 'p' to stand for 'PROGRAM.'

To a fresh copy of the cradle, add the following code, and insert a call to it from the main program:

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

procedure Prog;
var  Name: char;
begin
   Match('p');            { Handles program header part }
   Name := GetName;
   Prolog(Name);
   Match('.');
   Epilog(Name);
end;
{--------------------------------------------------------------}
```

The procedures Prolog and Epilog perform whatever is required to let the program interface with the operating system, so that it can execute as a program. Needless to say, this part will be VERY OS-dependent. Remember, I've been emitting code for a 68000 running under the OS I use, which is SK*DOS. I realize most of you are using PC's and would rather see something else, but I'm in this thing too deep to change now!

Anyhow, SK*DOS is a particularly easy OS to interface to. Here is the code for Prolog and Epilog:

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

procedure Prolog;
begin
   EmitLn('WARMST EQU $A01E');
end;


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

procedure Epilog(Name: char);
begin
   EmitLn('DC WARMST');
   EmitLn('END ' + Name);
end;
{--------------------------------------------------------------}
```

As usual, add this code and try out the "compiler." At this point, there is only one legal input:

```
     px.   (where x is any single letter, the program name)
```

Well, as usual our first effort is rather unimpressive, but by now I'm sure you know that things will get more interesting. There is one important thing to note: THE OUTPUT IS A WORKING, COMPLETE, AND EXECUTABLE PROGRAM (at least after it's assembled).

This is very important. The nice feature of the top-down approach is that at any stage you can compile a subset of the complete language and get a program that will run on the target machine. From here on, then, we need only add features by fleshing out the language constructs. It's all very similar to what we've been doing all along, except that we're approaching it from the other end.

FLESHING IT OUT

To flesh out the compiler, we only have to deal with language features one by one. I like to start with a stub procedure that does nothing, then add detail in incremental fashion. Let's begin by processing a block, in accordance with its PDL above. We can do this in two stages. First, add the null procedure:

```
{--------------------------------------------------------------}
{ Parse and Translate a Pascal Block }

procedure DoBlock(Name: char);
begin
end;
{--------------------------------------------------------------}
```

and modify Prog to read:

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

procedure Prog;
var  Name: char;
begin
   Match('p');
   Name := GetName;
   Prolog;
   DoBlock(Name);
   Match('.');
   Epilog(Name);
end;
{--------------------------------------------------------------}
```

That certainly shouldn't change the behavior of the program, and it doesn't. But now the definition of Prog is complete, and we can proceed to flesh out DoBlock. That's done right from its BNF definition:

```
{--------------------------------------------------------------}
{ Parse and Translate a Pascal Block }
```

```
procedure DoBlock(Name: char);
begin
   Declarations;
   PostLabel(Name);
   Statements;
end;
{--------------------------------------------------------------}
```

The procedure PostLabel was defined in the installment on branches. Copy it into your cradle.

I probably need to explain the reason for inserting the label where I have. It has to do with the operation of SK*DOS. Unlike some OS's, SK*DOS allows the entry point to the main program to be anywhere in the program. All you have to do is to give that point a name. The call to PostLabel puts that name just before the first executable statement in the main program. How does SK*DOS know which of the many labels is the entry point, you ask? It's the one that matches the END statement at the end of the program.

OK, now we need stubs for the procedures Declarations and Statements. Make them null procedures as we did before.

Does the program still run the same? Then we can move on to the next stage.

DECLARATIONS

The BNF for Pascal declarations is:

```
     <declarations> ::= ( <label list>    |
                          <constant list> |
                          <type list>     |
                          <variable list> |
                          <procedure>     |
                          <function>         )*
```

(Note that I'm using the more liberal definition used by Turbo Pascal. In the standard Pascal definition, each of these parts must be in a specific order relative to the rest.)

As usual, let's let a single character represent each of these declaration types. The new form of Declarations is:

```
{--------------------------------------------------------------}
{ Parse and Translate the Declaration Part }

procedure Declarations;
begin
   while Look in ['l', 'c', 't', 'v', 'p', 'f'] do
      case Look of
       'l': Labels;
       'c': Constants;
       't': Types;
       'v': Variables;
       'p': DoProcedure;
       'f': DoFunction;
      end;
end;
{--------------------------------------------------------------}
```

Of course, we need stub procedures for each of these declaration types. This time, they can't quite be null procedures, since otherwise we'll end up with an infinite While loop. At the very least, each recognizer must eat the character that invokes it. Insert the following procedures:

```
{--------------------------------------------------------------}
{ Process Label Statement }

procedure Labels;
begin
   Match('l');
end;


{--------------------------------------------------------------}
{ Process Const Statement }

procedure Constants;
begin
   Match('c');
end;


{--------------------------------------------------------------}
{ Process Type Statement }
procedure Types;
begin
   Match('t');
end;


{--------------------------------------------------------------}
{ Process Var Statement }

procedure Variables;
begin
   Match('v');
end;


{--------------------------------------------------------------}
{ Process Procedure Definition }

procedure DoProcedure;
begin
   Match('p');
end;


{--------------------------------------------------------------}
{ Process Function Definition }

procedure DoFunction;
begin
   Match('f');
end;
{--------------------------------------------------------------}
```

Now try out the compiler with a few representative inputs. You can mix the declarations any way you like, as long as the last character in the program is'.' to indicate the end of the program. Of course, none of the declarations actually declare anything, so you don't need (and can't use) any characters other

than those standing for the keywords.

We can flesh out the statement part in a similar way. The BNF for it is:

```
     <statements> ::= <compound statement>

     <compound statement> ::= BEGIN <statement>
                                 (';' <statement>) END
```

Note that statements can begin with any identifier except END. So the first stub form of procedure Statements is:

```
{---------------------------------------------------------------}
{ Parse and Translate the Statement Part }

procedure Statements;
begin
   Match('b');
   while Look <> 'e' do
      GetChar;
   Match('e');
end;
{---------------------------------------------------------------}
```

At this point the compiler will accept any number of declarations, followed by the BEGIN block of the main program. This block itself can contain any characters at all (except an END), but it must be present.

The simplest form of input is now

```
     'pxbe.'
```

Try it. Also try some combinations of this. Make some deliberate errors and see what happens.

At this point you should be beginning to see the drill. We begin with a stub translator to process a program, then we flesh out each procedure in turn, based upon its BNF definition. Just as the lower-level BNF definitions add detail and elaborate upon the higher-level ones, the lower-level recognizers will parse more detail of the input program. When the last stub has been expanded, the compiler will be complete. That's top-down design/implementation in its purest form.

You might note that even though we've been adding procedures, the output of the program hasn't changed. That's as it should be. At these top levels there is no emitted code required. The recognizers are functioning as just that: recognizers. They are accepting input sentences, catching bad ones, and channeling good input to the right places, so they are doing their job. If we were to pursue this a bit longer, code would start to appear.

The next step in our expansion should probably be procedure Statements. The Pascal definition is:

```
    <statement> ::= <simple statement> | <structured statement>

    <simple statement> ::= <assignment> | <procedure call> | null

    <structured statement> ::= <compound statement> |
```

```
                        <if statement>       |
                        <case statement>     |
                        <while statement>    |
                        <repeat statement>   |
                        <for statement>      |
                        <with statement>
```

These are starting to look familiar. As a matter of fact, you have already gone through the process of parsing and generating code for both assignment statements and control structures. This is where the top level meets our bottom-up approach of previous sessions. The constructs will be a little different from those we've been using for KISS, but the differences are nothing you can't handle.

I think you can get the picture now as to the procedure. We begin with a complete BNF description of the language. Starting at the top level, we code up the recognizer for that BNF statement, using stubs for the next-level recognizers. Then we flesh those lower-level statements out one by one.

As it happens, the definition of Pascal is very compatible with the use of BNF, and BNF descriptions of the language abound. Armed with such a description, you will find it fairly straightforward to continue the process we've begun.

You might have a go at fleshing a few of these constructs out, just to get a feel for it. I don't expect you to be able to complete a Pascal compiler here ... there are too many things such as procedures and types that we haven't addressed yet ... but it might be helpful to try some of the more familiar ones. It will do you good to see executable programs coming out the other end.

If I'm going to address those issues that we haven't covered yet, I'd rather do it in the context of KISS. We're not trying to build a complete Pascal compiler just yet, so I'm going to stop the expansion of Pascal here. Let's take a look at a very different language.

THE STRUCTURE OF C

The C language is quite another matter, as you'll see. Texts on C rarely include a BNF definition of the language. Probably that's because the language is quite hard to write BNF for.

One reason I'm showing you these structures now is so that I can impress upon you these two facts:

(1) The definition of the language drives the structure of the compiler. What works for one language may be a disaster for another. It's a very bad idea to try to force a given structure upon the compiler. Rather, you should let the BNF drive the structure, as we have done here.

(2) A language that is hard to write BNF for will probably be hard to write a compiler for, as well. C is a popular language, and it has a reputation for letting you do virtually anything that is possible to do. Despite the success of Small C, C is _NOT_ an easy language to parse.

A C program has less structure than its Pascal counterpart. At the top level, everything in C is a static declaration, either of data or of a function. We can capture this thought like this:

```
     <program> ::= ( <global declaration> )*

     <global declaration> ::= <data declaration>  |
                              <function>
```

In Small C, functions can only have the default type int, which is not declared. This makes the input easy to parse: the first token is either "int," "char," or the name of a function. In Small C, the preprocessor commands are also processed by the compiler proper, so the syntax becomes:

```
     <global declaration> ::= '#' <preprocessor command>  |
                              'int' <data list>           |
                              'char' <data list>          |
                              <ident> <function body>     |
```

Although we're really more interested in full C here, I'll show you the code corresponding to this top-level structure for Small C.

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

procedure Prog;
begin
   while Look <> ^Z do begin
      case Look of
       '#': PreProc;
       'i': IntDecl;
       'c': CharDecl;
      else DoFunction(Int);
      end;
   end;
end;
{-------------------------------------------------------------}
```

Note that I've had to use a ^Z to indicate the end of the source. C has no keyword such as END or the '.' to otherwise indicate the end.

With full C, things aren't even this easy. The problem comes about because in full C, functions can also have types. So when the compiler sees a keyword like "int," it still doesn't know whether to expect a data declaration or a function definition. Things get more complicated since the next token may not be a name ... it may start with an '*' or '(', or combinations of the two.

More specifically, the BNF for full C begins with:

```
     <program> ::= ( <top-level decl> )*

     <top-level decl> ::= <function def> | <data decl>

     <data decl> ::= [<class>] <type> <decl-list>

     <function def> ::= [<class>] [<type>] <function decl>
```

You can now see the problem: The first two parts of the declarations for data and functions can be the same. Because of

the ambiguity in the grammar as written above, it's not a suitable grammar for a recursive-descent parser. Can we transform it into one that is suitable? Yes, with a little work. Suppose we write it this way:

```
<top-level decl> ::= [<class>] <decl>

<decl> ::= <type> <typed decl> | <function decl>

<typed decl> ::= <data list> | <function decl>
```

We can build a parsing routine for the class and type definitions, and have them store away their findings and go on, without their ever having to "know" whether a function or a data declaration is being processed.

To begin, key in the following version of the main program:

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

begin
   Init;
   while Look <> ^Z do begin
      GetClass;
      GetType;
      TopDecl;
   end;
end.

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

For the first round, just make the three procedures stubs that do nothing _BUT_ call GetChar.

Does this program work? Well, it would be hard put NOT to, since we're not really asking it to do anything. It's been said that a C compiler will accept virtually any input without choking. It's certainly true of THIS compiler, since in effect all it does is to eat input characters until it finds a ^Z.

Next, let's make GetClass do something worthwhile. Declare the global variable

```
var Class: char;
```

and change GetClass to do the following:

```
{-------------------------------------------------------------}
{  Get a Storage Class Specifier }

Procedure GetClass;
begin
   if Look in ['a', 'x', 's'] then begin
      Class := Look;
      GetChar;
      end
   else Class := 'a';
end;
{-------------------------------------------------------------}
```

Here, I've used three single characters to represent the three storage classes "auto," "extern," and "static." These are not the only three possible classes ... there are also "register" and "typedef," but this should give you the picture. Note that the default class is "auto."

We can do a similar thing for types. Enter the following procedure next:

```
{---------------------------------------------------------------}
{  Get a Type Specifier }

procedure GetType;
begin
   Typ := ' ';
   if Look = 'u' then begin
      Sign := 'u';
      Typ := 'i';
      GetChar;
      end
   else Sign := 's';
   if Look in ['i', 'l', 'c'] then begin
      Typ := Look;
      GetChar;
   end;
end;
{---------------------------------------------------------------}
```

Note that you must add two more global variables, Sign and Typ.

With these two procedures in place, the compiler will process the class and type definitions and store away their findings. We can now process the rest of the declaration.

We are by no means out of the woods yet, because there are still many complexities just in the definition of the type, before we even get to the actual data or function names. Let's pretend for the moment that we have passed all those gates, and that the next thing in the input stream is a name. If the name is followed by a left paren, we have a function declaration. If not, we have at least one data item, and possibly a list, each element of which can have an initializer.

Insert the following version of TopDecl:

```
{---------------------------------------------------------------}
{ Process a Top-Level Declaration }

procedure TopDecl;
var Name: char;
begin
   Name := Getname;
   if Look = '(' then
      DoFunc(Name)
   else
      DoData(Name);
end;
{---------------------------------------------------------------}
```

(Note that, since we have already read the name, we must pass it along to the appropriate routine.)

Finally, add the two procedures DoFunc and DoData:

```
{-------------------------------------------------------------}
{ Process a Function Definition }

procedure DoFunc(n: char);
begin
   Match('(');
   Match(')');
   Match('{');
   Match('}');
   if Typ = ' ' then Typ := 'i';
   Writeln(Class, Sign, Typ, ' function ', n);
end;

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

procedure DoData(n: char);
begin
   if Typ = ' ' then Expected('Type declaration');
   Writeln(Class, Sign, Typ, ' data ', n);
   while Look = ',' do begin
      Match(',');
      n := GetName;
      WriteLn(Class, Sign, Typ, ' data ', n);
   end;
   Match(';');
end;
{-------------------------------------------------------------}
```

Since we're still a long way from producing executable code, I decided to just have these two routines tell us what they found.

OK, give this program a try. For data declarations, it's OK to give a list separated by commas. We can't process initializers as yet. We also can't process argument lists for the functions, but the "(){}" characters should be there.

We're still a _VERY_ long way from having a C compiler, but what we have is starting to process the right kinds of inputs, and is recognizing both good and bad inputs. In the process, the natural structure of the compiler is starting to take form.

Can we continue this until we have something that acts more like a compiler. Of course we can. Should we? That's another matter. I don't know about you, but I'm beginning to get dizzy, and we've still got a long way to go to even get past the data declarations.

At this point, I think you can see how the structure of the compiler evolves from the language definition. The structures we've seen for our two examples, Pascal and C, are as different as night and day. Pascal was designed at least partly to be easy to parse, and that's reflected in the compiler. In general, in Pascal there is more structure and we have a better idea of what kinds of constructs to expect at any point. In C, on the other hand, the program is essentially a list of declarations, terminated only by the end of file.

We could pursue both of these structures much farther, but remember that our purpose here is not to build a Pascal or a C compiler, but rather to study compilers in general. For those of you who DO want to deal with Pascal or C, I hope I've given you enough of a start so that you can take it from here (although

you'll soon need some of the stuff we still haven't covered yet, such as typing and procedure calls).   For the rest of you, stay with me through the next installment. There, I'll be leading you through the development of a complete compiler for TINY, a subset of KISS.

See you then.