Let's Build a Compiler

Let's Build a Compiler

Jack Crenshaw

Chapter 1

Part I: Introduction - 24 July 1988

Introduction

This series of articles is a tutorial on the theory and practice of developing language parsers and compilers. Before we are finished, we will have covered every aspect of compiler construction, designed a new programming language, and built a working compiler.

Though I am not a computer scientist by education (my Ph.D. is in a different field, Physics), I have been interested in compilers for many years. I have bought and tried to digest the contents of virtually every book on the subject ever written. I don't mind telling you that it was slow going. Compiler texts are written for Computer Science majors, and are tough sledding for the rest of us. But over the years a bit of it began to seep in. What really caused it to jell was when I began to branch off on my own and begin to try things on my own computer. Now I plan to share with you what I have learned. At the end of this series you will by no means be a computer scientist, nor will you know all the esoterics of compiler theory. I intend to completely ignore the more theoretical aspects of the subject. What you *will* know is all the practical aspects that one needs to know to build a working system.

This is a "learn-by-doing" series. In the course of the series I will be performing experiments on a computer. You will be expected to follow along, repeating the experiments that I do, and performing some on your own. I will be using Turbo Pascal 4.0 on a PC clone. I will periodically insert examples written in TP. These will be executable code, which you will be expected to copy into your own computer and run. If you don't have a copy of Turbo, you will be severely limited in how well you will be able to follow what's going on. If you don't have a copy, I urge you to get one. After all, it's an excellent product, good for many other uses!

Some articles on compilers show you examples, or show you (as in the case of Small-C) a finished product, which you can then copy and use without a whole lot of understanding of how it works. I hope to do much more than that. I hope to teach you *how* the things get done, so that you can go off on your own and not only reproduce what I have done, but improve on it.

This is admittedly an ambitious undertaking, and it won't be done in one page. I expect to do it in the course of a number of articles. Each article will cover a single aspect of compiler theory, and will pretty much stand alone. If all you're interested in at a given time is one aspect, then you need to look only at that one article. Each article will be uploaded as it is complete, so you will have to wait for the last one before you can consider yourself finished. Please be patient.

The average text on compiler theory covers a lot of ground that we won't be covering here. The typical sequence is:

- An introductory chapter describing what a compiler is.
- A chapter or two on syntax equations, using Backus-Naur Form (BNF).
- A chapter or two on lexical scanning, with emphasis on deterministic and non-deterministic finite automata.
- Several chapters on parsing theory, beginning with top-down recursive descent, and ending with LALR parsers.
- A chapter on intermediate languages, with emphasis on P-code and similar reverse polish representations.
- Many chapters on alternative ways to handle subroutines and parameter passing, type declarations, and such.
- A chapter toward the end on code generation, usually for some imaginary CPU with a simple instruction set. Most readers (and in fact, most college classes) never make it this far.

- A final chapter or two on optimization. This chapter often goes unread, too.

I'll be taking a much different approach in this series. To begin with, I won't dwell long on options. I'll be giving you *a* way that works. If you want to explore options, well and good ... I encourage you to do so ... but I'll be sticking to what I know. I also will skip over most of the theory that puts people to sleep. Don't get me wrong: I don't belittle the theory, and it's vitally important when it comes to dealing with the more tricky parts of a given language. But I believe in putting first things first. Here we'll be dealing with the 95% of compiler techniques that don't need a lot of theory to handle.

I also will discuss only one approach to parsing: top-down, recursive descent parsing, which is the *only* technique that's at all amenable to hand-crafting a compiler. The other approaches are only useful if you have a tool like YACC, and also don't care how much memory space the final product uses.

I also take a page from the work of Ron Cain, the author of the original Small C. Whereas almost all other compiler authors have historically used an intermediate language like P-code and divided the compiler into two parts (a front end that produces P-code, and a back end that processes P-code to produce executable object code), Ron showed us that it is a straightforward matter to make a compiler directly produce executable object code, in the form of assembler language statements. The code will *not* be the world's tightest code ... producing optimized code is a much more difficult job. But it will work, and work reasonably well. Just so that I don't leave you with the impression that our end product will be worthless, I *do* intend to show you how to "soup up" the compiler with some optimization.

Finally, I'll be using some tricks that I've found to be most helpful in letting me understand what's going on without wading through a lot of boiler plate. Chief among these is the use of single-character tokens, with no embedded spaces, for the early design work. I figure that if I can get a parser to recognize and deal with `I-T-L`, I can get it to do the same with `IF-THEN-ELSE`. And I can. In the second "lesson," I'll show you just how easy it is to extend a simple parser to handle tokens of arbitrary length. As another trick, I completely ignore file I/O, figuring that if I can read source from the keyboard and output object to the screen, I can also do it from/to disk files. Experience has proven that once a translator is working correctly, it's a straightforward matter to redirect the I/O to files. The last trick is that I make no attempt to do error correction/recovery. The programs we'll be building will *recognize* errors, and will not *crash*, but they will simply stop on the first error ... just like good ol' Turbo does. There will be other tricks that you'll see as you go. Most of them can't be found in any compiler textbook, but they work.

A word about style and efficiency. As you will see, I tend to write programs in *very* small, easily understood pieces. None of the procedures we'll be working with will be more than about 15-20 lines long. I'm a fervent devotee of the KISS (Keep It Simple, Sidney) school of software development. I try to never do something tricky or complex, when something simple will do. Inefficient? Perhaps, but you'll like the results. As Brian Kernighan has said, *first* make it run, *then* make it run fast. If, later on, you want to go back and tighten up the code in one of our products, you'll be able to do so, since the code will be quite understandable. If you do so, however, I urge you to wait until the program is doing everything you want it to.

I also have a tendency to delay building a module until I discover that I need it. Trying to anticipate every possible future contingency can drive you crazy, and you'll generally guess wrong anyway. In this modern day of screen editors and fast compilers, I don't hesitate to change a module when I feel I need a more powerful one. Until then, I'll write only what I need.

One final caveat: One of the principles we'll be sticking to here is that we don't fool around with P-code or imaginary CPUs, but that we will start out on day one producing working, executable object code, at least in the form of assembler language source. However, you may not like my choice of assembler language ... it's 68000 code, which is what works on my system (under SK*DOS). I think you'll find, though, that the translation to any other CPU such as the 80x86 will be quite obvious, though, so I don't see a problem here. In fact, I hope someone out there who knows the '86 language better than I do will offer us the equivalent object code fragments as we need them.

The Cradle

Every program needs some boiler plate ... I/O routines, error message routines, etc. The programs we develop here will be no exceptions. I've tried to hold this stuff to an absolute minimum, however, so that we can concentrate on the important stuff without losing it among the trees. The code given below represents about the minimum that we need to get anything done. It consists of some I/O routines, an error-handling routine and a skeleton, null main program. I call it our cradle. As we develop other routines, we'll add them to the cradle, and add the calls to them as we need to. Make a copy of the cradle and save it, because we'll be using it more than once.

There are many different ways to organize the scanning activities of a parser. In Unix systems, authors tend to use getc and ungetc. I've had very good luck with the approach shown here, which is to use a single, global, lookahead character. Part of the initialization procedure (the only part, so far!) serves to "prime the pump"

by reading the first character from the input stream. No other special techniques are required with Turbo 4.0 ... each successive call to `GetChar` will read the next character in the stream.

```
{--------------------------------------------------------------}
program Cradle;

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

const TAB = ^I;

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

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

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

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

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


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

{ Recognize a Decimal Digit }

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


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

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


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

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


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

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

procedure Init;
begin
   GetChar;
end;


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

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

That's it for this introduction. Copy the code above into TP and compile it. Make sure that it compiles and runs correctly. Then proceed to the first lesson, which is on expression parsing.

Chapter 2

Part II: Expression Parsing - 24 July 1988

Introduction

If you've read the introduction document to this series, you will already know what we're about. You will also have copied the cradle software into your Turbo Pascal system, and have compiled it. So you should be ready to go.

The purpose of this article is for us to learn how to parse and translate mathematical expressions. What we would like to see as output is a series of assembler-language statements that perform the desired actions. For purposes of definition, an expression is the right-hand side of an equation, as in `x = 2*y + 3/(4*z)`.

In the early going, I'll be taking things in *very* small steps. That's so that the beginners among you won't get totally lost. There are also some very good lessons to be learned early on, that will serve us well later. For the more experienced readers: bear with me. We'll get rolling soon enough.

Single Digits

In keeping with the whole theme of this series (KISS, remember?), let's start with the absolutely most simple case we can think of. That, to me, is an expression consisting of a single digit.

Before starting to code, make sure you have a baseline copy of the "cradle" that I gave last time. We'll be using it again for other experiments. Then add this code:

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

procedure Expression;
begin
   EmitLn('MOVE #' + GetNum + ',D0')
end;
{--------------------------------------------------------------}
```

And add the line `Expression;` to the main program so that it reads:

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

Now run the program. Try any single-digit number as input. You should get a single line of assembler-language output. Now try any other character as input, and you'll see that the parser properly reports an error.

CONGRATULATIONS! You have just written a working translator!

OK, I grant you that it's pretty limited. But don't brush it off too lightly. This little "compiler" does, on a very limited scale, exactly what any larger compiler does: it correctly recognizes legal statements in the input "language" that we have defined for it, and it produces correct, executable assembler code, suitable for

assembling into object format. Just as importantly, it correctly recognizes statements that are *not* legal, and gives a meaningful error message. Who could ask for more? As we expand our parser, we'd better make sure those two characteristics always hold true.

There are some other features of this tiny program worth mentioning. First, you can see that we don't separate code generation from parsing ... as soon as the parser knows what we want done, it generates the object code directly. In a real compiler, of course, the reads in `GetChar` would be from a disk file, and the writes to another disk file, but this way is much easier to deal with while we're experimenting.

Also note that an expression must leave a result somewhere. I've chosen the 68000 register `D0`. I could have made some other choices, but this one makes sense.

Binary Expressions

Now that we have that under our belt, let's branch out a bit. Admittedly, an "expression" consisting of only one character is not going to meet our needs for long, so let's see what we can do to extend it. Suppose we want to handle expressions of the form: `1+2` or `4-3` or, in general, `<term> +/- <term>`. (That's a bit of Backus-Naur Form, or BNF.)

To do this we need a procedure that recognizes a term and leaves its result somewhere, and another that recognizes and distinguishes between a `+` and a `-` and generates the appropriate code. But if `Expression` is going to leave its result in `D0`, where should `Term` leave its result? Answer: the same place. We're going to have to save the first result of `Term` somewhere before we get the next one.

OK, basically what we want to do is have procedure `Term` do what `Expression` was doing before. So just *rename* procedure `Expression` as `Term`, and enter the following new version of `Expression`:

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

procedure Expression;
begin
   Term;
   EmitLn('MOVE D0,D1');
   case Look of
    '+': Add;
    '-': Subtract;
   else Expected('Addop');
   end;
end;
{---------------------------------------------------------------}
```

Next, just above `Expression` enter these two procedures:

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

procedure Add;
begin
   Match('+');
   Term;
   EmitLn('ADD D1,D0');
end;


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

procedure Subtract;
begin
   Match('-');
   Term;
   EmitLn('SUB D1,D0');
end;
{---------------------------------------------------------------}
```

When you're finished with that, the order of the routines should be:

- `Term` (The *old* Expression)
- `Add`
- `Subtract`
- `Expression`

Now run the program. Try any combination you can think of of two single digits, separated by a `+` or a `-`. You should get a series of four assembler-language instructions out of each run. Now try some expressions with deliberate errors in them. Does the parser catch the errors?

Take a look at the object code generated. There are two observations we can make. First, the code generated is *not* what we would write ourselves. The sequence

```
MOVE #n,D0
MOVE D0,D1
```

is inefficient. If we were writing this code by hand, we would probably just load the data directly to `D1`.

There is a message here: code generated by our parser is less efficient than the code we would write by hand. Get used to it. That's going to be true throughout this series. It's true of all compilers to some extent. Computer scientists have devoted whole lifetimes to the issue of code optimization, and there are indeed things that can be done to improve the quality of code output. Some compilers do quite well, but there is a heavy price to pay in complexity, and it's a losing battle anyway ... there will probably never come a time when a good assembler-language programmer can't out-program a compiler. Before this session is over, I'll briefly mention some ways that we can do a little optimization, just to show you that we can indeed improve things without too much trouble. But remember, we're here to learn, not to see how tight we can make the object code. For now, and really throughout this series of articles, we'll studiously ignore optimization and concentrate on getting out code that works.

Speaking of which: ours *doesn't*! The code is *wrong*! As things are working now, the subtraction process subtracts `D1` (which has the *first* argument in it) from `D0` (which has the second). That's the wrong way, so we end up with the wrong sign for the result. So let's fix up procedure `Subtract` with a sign-changer, so that it reads

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

procedure Subtract;
begin
   Match('-');
   Term;
   EmitLn('SUB D1,D0');
   EmitLn('NEG D0');
end;
{--------------------------------------------------------------}
```

Now our code is even less efficient, but at least it gives the right answer! Unfortunately, the rules that give the meaning of math expressions require that the terms in an expression come out in an inconvenient order for us. Again, this is just one of those facts of life you learn to live with. This one will come back to haunt us when we get to division.

OK, at this point we have a parser that can recognize the sum or difference of two digits. Earlier, we could only recognize a single digit. But real expressions can have either form (or an infinity of others). For kicks, go back and run the program with the single input line `1`.

Didn't work, did it? And why should it? We just finished telling our parser that the only kinds of expressions that are legal are those with two terms. We must rewrite procedure `Expression` to be a lot more broadminded, and this is where things start to take the shape of a real parser.

General Expressions

In the *real* world, an expression can consist of one or more terms, separated by "addops" (`+` or `-`). In BNF, this is written

```
<expression> ::= <term> [<addop> <term>]*
```

We can accommodate this definition of an expression with the addition of a simple loop to procedure `Expression`:

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

procedure Expression;
begin
   Term;
   while Look in ['+', '-'] do begin
      EmitLn('MOVE D0,D1');
      case Look of
       '+': Add;
       '-': Subtract;
      else Expected('Addop');
      end;
   end;
end;
{---------------------------------------------------------------}
```

Now we're getting somewhere! This version handles any number of terms, and it only cost us two extra lines of code. As we go on, you'll discover that this is characteristic of top-down parsers ... it only takes a few lines of code to accommodate extensions to the language. That's what makes our incremental approach possible. Notice, too, how well the code of procedure `Expression` matches the BNF definition. That, too, is characteristic of the method. As you get proficient in the approach, you'll find that you can turn BNF into parser code just about as fast as you can type!

OK, compile the new version of our parser, and give it a try. As usual, verify that the "compiler" can handle any legal expression, and will give a meaningful error message for an illegal one. Neat, eh? You might note that in our test version, any error message comes out sort of buried in whatever code had already been generated. But remember, that's just because we are using the CRT as our "output file" for this series of experiments. In a production version, the two outputs would be separated ... one to the output file, and one to the screen.

Using the Stack

At this point I'm going to violate my rule that we don't introduce any complexity until it's absolutely necessary, long enough to point out a problem with the code we're generating. As things stand now, the parser uses `D0` for the "primary" register, and `D1` as a place to store the partial sum. That works fine for now, because as long as we deal with only the "addops" `+` and `-`, any new term can be added in as soon as it is found. But in general that isn't true. Consider, for example, the expression `1+(2-(3+(4-5)))`.

If we put the `1` in `D1`, where do we put the `2`? Since a general expression can have any degree of complexity, we're going to run out of registers fast!

Fortunately, there's a simple solution. Like every modern microprocessor, the 68000 has a stack, which is the perfect place to save a variable number of items. So instead of moving the term in `D0` to `D1`, let's just push it onto the stack. For the benefit of those unfamiliar with 68000 assembler language, a push is written `-(SP)` and a pop, `(SP)+`.

So let's change the `EmitLn` in `Expression` to read `EmitLn('MOVE D0,-(SP)');` and the two lines in `Add` and `Subtract` to `EmitLn('ADD (SP)+,D0');` and `EmitLn('SUB (SP)+,D0');`, respectively. Now try the parser again and make sure we haven't broken it.

Once again, the generated code is less efficient than before, but it's a necessary step, as you'll see.

Multiplication and Division

Now let's get down to some *really* serious business. As you all know, there are other math operators than "addops" ... expressions can also have multiply and divide operations. You also know that there is an implied operator *precedence*, or hierarchy, associated with expressions, so that in an expression like `2 + 3 * 4`, we know that we're supposed to multiply *first*, then add. (See why we needed the stack?)

In the early days of compiler technology, people used some rather complex techniques to insure that the operator precedence rules were obeyed. It turns out, though, that none of this is necessary ... the rules can be

accommodated quite nicely by our top-down parsing technique. Up till now, the only form that we've considered for a term is that of a single decimal digit.

More generally, we can define a term as a *product* of *factors*; i.e.

```
<term> ::= <factor>  [ <mulop> <factor ]*
```

What is a `factor`? For now, it's what a term used to be ... a single digit.

Notice the symmetry: a term has the same form as an expression. As a matter of fact, we can add to our parser with a little judicious copying and renaming. But to avoid confusion, the listing below is the complete set of parsing routines. (Note the way we handle the reversal of operands in `Divide`.)

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

procedure Factor;
begin
   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('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;
      else Expected('Mulop');
      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 Look in ['+', '-'] do begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '+': Add;
       '-': Subtract;
      else Expected('Addop');
      end;
   end;
end;
{--------------------------------------------------------------}
```

Hot dog! A *nearly* functional parser/translator, in only 55 lines of Pascal! The output is starting to look really useful, if you continue to overlook the inefficiency, which I hope you will. Remember, we're not trying to produce tight code here.

Parentheses

We can wrap up this part of the parser with the addition of parentheses with math expressions. As you know, parentheses are a mechanism to force a desired operator precedence. So, for example, in the expression `2*(3+4)`, the parentheses force the addition before the multiply. Much more importantly, though, parentheses give us a mechanism for defining expressions of any degree of complexity, as in `(1+2)/((3+4)+(5-6))`.

The key to incorporating parentheses into our parser is to realize that no matter how complicated an expression enclosed by parentheses may be, to the rest of the world it looks like a simple factor. That is, one of the forms for a factor is:

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

This is where the recursion comes in. An expression can contain a factor which contains another expression which contains a factor, etc., ad infinitum.

Complicated or not, we can take care of this by adding just a few lines of Pascal to procedure `Factor`:

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

procedure Expression; Forward;

procedure Factor;
begin
   if Look = '(' then begin
      Match('(');
      Expression;
      Match(')');
      end
   else
      EmitLn('MOVE #' + GetNum + ',D0');
end;
{--------------------------------------------------------------}
```

Note again how easily we can extend the parser, and how well the Pascal code matches the BNF syntax.

As usual, compile the new version and make sure that it correctly parses legal sentences, and flags illegal ones with an error message.

Unary Minus

At this point, we have a parser that can handle just about any expression, right? OK, try this input sentence: `-1`.

WOOPS! It doesn't work, does it? Procedure `Expression` expects everything to start with an integer, so it coughs up the leading minus sign. You'll find that `+3` won't work either, nor will something like `-(3-2)`.

There are a couple of ways to fix the problem. The easiest (although not necessarily the best) way is to stick an imaginary leading zero in front of expressions of this type, so that `-3` becomes `0-3`. We can easily patch this into our existing version of `Expression`:

```
{--------------------------------------------------------------}
{ 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;
      else Expected('Addop');
      end;
   end;
end;
{--------------------------------------------------------------}
```

I *told* you that making changes was easy! This time it cost us only three new lines of Pascal. Note the new reference to function `IsAddop`. Since the test for an addop appeared twice, I chose to embed it in the new function. The form of `IsAddop` should be apparent from that for `IsAlpha`. Here it is:

```
{--------------------------------------------------------------}
{ Recognize an Addop }

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

```
   IsAddop := c in ['+', '-'];
end;
{---------------------------------------------------------------}
```

OK, make these changes to the program and recompile. You should also include `IsAddop` in your baseline copy of the cradle. We'll be needing it again later. Now try the input `-1` again. Wow! The efficiency of the code is pretty poor ... six lines of code just for loading a simple constant ... but at least it's correct. Remember, we're not trying to replace Turbo Pascal here.

At this point we're just about finished with the structure of our expression parser. This version of the program should correctly parse and compile just about any expression you care to throw at it. It's still limited in that we can only handle factors involving single decimal digits. But I hope that by now you're starting to get the message that we can accommodate further extensions with just some minor changes to the parser. You probably won't be surprised to hear that a variable or even a function call is just another kind of a factor.

In the next session, I'll show you just how easy it is to extend our parser to take care of these things too, and I'll also show you just how easily we can accommodate multi-character numbers and variable names. So you see, we're not far at all from a truly useful parser.

A Word about Optimization

Earlier in this session, I promised to give you some hints as to how we can improve the quality of the generated code. As I said, the production of tight code is not the main purpose of this series of articles. But you need to at least know that we aren't just wasting our time here ... that we can indeed modify the parser further to make it produce better code, without throwing away everything we've done to date. As usual, it turns out that *some* optimization is not that difficult to do ... it simply takes some extra code in the parser.

There are two basic approaches we can take:

- Try to fix up the code after it's generated

  This is the concept of "peephole" optimization. The general idea it that we know what combinations of instructions the compiler is going to generate, and we also know which ones are pretty bad (such as the code for `-1`, above). So all we do is to scan the produced code, looking for those combinations, and replacing them by better ones. It's sort of a macro expansion, in reverse, and a fairly straightforward exercise in pattern-matching. The only complication, really, is that there may be a *lot* of such combinations to look for. It's called peephole optimization simply because it only looks at a small group of instructions at a time. Peephole optimization can have a dramatic effect on the quality of the code, with little change to the structure of the compiler itself. There is a price to pay, though, in both the speed, size, and complexity of the compiler. Looking for all those combinations calls for a lot of `IF` tests, each one of which is a source of error. And, of course, it takes time.

  In the classical implementation of a peephole optimizer, it's done as a second pass to the compiler. The output code is written to disk, and then the optimizer reads and processes the disk file again. As a matter of fact, you can see that the optimizer could even be a separate *program* from the compiler proper. Since the optimizer only looks at the code through a small "window" of instructions (hence the name), a better implementation would be to simply buffer up a few lines of output, and scan the buffer after each `EmitLn`.

- Try to generate better code in the first place

  This approach calls for us to look for special cases *before* we `Emit` them. As a trivial example, we should be able to identify a constant zero, and `Emit` a `CLR` instead of a load, or even do nothing at all, as in an add of zero, for example. Closer to home, if we had chosen to recognize the unary minus in `Factor` instead of in `Expression`, we could treat constants like `-1` as ordinary constants, rather then generating them from positive ones. None of these things are difficult to deal with ... they only add extra tests in the code, which is why I haven't included them in our program. The way I see it, once we get to the point that we have a working compiler, generating useful code that executes, we can always go back and tweak the thing to tighten up the code produced. That's why there are Release 2.0's in the world.

There *is* one more type of optimization worth mentioning, that seems to promise pretty tight code without too much hassle. It's my "invention" in the sense that I haven't seen it suggested in print anywhere, though I have no illusions that it's original with me.

This is to avoid such a heavy use of the stack, by making better use of the CPU registers. Remember back when we were doing only addition and subtraction, that we used registers `D0` and `D1`, rather than the stack? It worked, because with only those two operations, the "stack" never needs more than two entries.

Well, the 68000 has eight data registers. Why not use them as a privately managed stack? The key is to recognize that, at any point in its processing, the parser *knows* how many items are on the stack, so it can

indeed manage it properly. We can define a private "stack pointer" that keeps track of which stack level we're at, and addresses the corresponding register. Procedure `Factor`, for example, would not cause data to be loaded into register `D0`, but into whatever the current "top-of-stack" register happened to be.

What we're doing in effect is to replace the CPU's RAM stack with a locally managed stack made up of registers. For most expressions, the stack level will never exceed eight, so we'll get pretty good code out. Of course, we also have to deal with those odd cases where the stack level *does* exceed eight, but that's no problem either. We simply let the stack spill over into the CPU stack. For levels beyond eight, the code is no worse than what we're generating now, and for levels less than eight, it's considerably better.

For the record, I have implemented this concept, just to make sure it works before I mentioned it to you. It does. In practice, it turns out that you can't really use all eight levels ... you need at least one register free to reverse the operand order for division (sure wish the 68000 had an `XTHL`, like the 8080!). For expressions that include function calls, we would also need a register reserved for them. Still, there is a nice improvement in code size for most expressions.

So, you see, getting better code isn't that difficult, but it does add complexity to the our translator ... complexity we can do without at this point. For that reason, I *strongly* suggest that we continue to ignore efficiency issues for the rest of this series, secure in the knowledge that we can indeed improve the code quality without throwing away what we've done.

Next lesson, I'll show you how to deal with variables factors and function calls. I'll also show you just how easy it is to handle multi-character tokens and embedded white space.

Chapter 3

Part III: More Expressions - 4 Aug 1988

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:

- No variables were allowed, only numeric factors
- 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 (`IF`s 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.

Chapter 4

Part IV: Interpreters - 24 July 1988

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 "instance" 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`es 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.

Chapter 5

Part V: Control Constructs - 19 August 1988

Introduction

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

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

The Plan

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

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

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

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

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

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

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

begin
```

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

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

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

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

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

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

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

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

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

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

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

The code for `Block` is:

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

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

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

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

Some Groundwork

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

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

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

The C version is similar:

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

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

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

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

Now, all of the constructs we'll be dealing with here involve transfer of control, which at the assembler-language level means conditional and/or unconditional branches. For example, the simple `IF` statement `IF <condition> A ENDIF B ...` must get translated into

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

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

Here are the two routines:

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

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


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

procedure PostLabel(L: string);
begin
   WriteLn(L, ':');
end;
{---------------------------------------------------------------}
```

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

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

Also, add the following extra initialization to Init:

```
LCount := 0;
```

(*Don't* forget that, or your labels can look really strange!)

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

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

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

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

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

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

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

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

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

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

The IF Statement

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

The code for `DoIf` is:

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

procedure Block; Forward;


procedure DoIf;
var L: string;
begin
   Match('i');
   L := NewLabel;
   Condition;
   EmitLn('BEQ ' + L);
   Block;
```

```
   Match('e');
   PostLabel(L);
end;
{--------------------------------------------------------------}
```

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

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

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

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

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

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

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

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

It's starting to look real, eh?

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

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

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

The corresponding output code should be

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

This leads us to the following syntax-directed translation:

```
IF
<condition>     { L1 = NewLabel;
                  L2 = NewLabel;
                  Emit(BEQ L1) }
```

```
<block>
ELSE           { Emit(BRA L2);
                 PostLabel(L1) }
<block>
ENDIF          { PostLabel(L2) }
```

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

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

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

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

Give it a try now. Try something like aiblcede.

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

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

The WHILE Statement

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

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

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

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

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

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

The code follows immediately from the syntax:

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

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

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

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

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

No other changes are necessary.

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

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

The LOOP Statement

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

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

The syntax is simply `LOOP <block> ENDLOOP` and the syntax-directed translation is:

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

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

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

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

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

REPEAT-UNTIL

Here's one construct that I lifted right from Pascal. The syntax is `REPEAT <block> UNTIL <condition>`, and the syntax-directed translation is:

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

As usual, the code falls out pretty easily:

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

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

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

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

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

The FOR Loop

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

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

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

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

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

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

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

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

L1:   LEA <ident>(PC),A0  address loop counter
      MOVE (A0),D0        fetch it to D0
      ADDQ #1,D0          bump the counter
```

```
      MOVE D0,(A0)        save new value
      CMP (SP),D0         check for range
      BLE L2              skip out if D0 > (SP)
      <block>
      BRA L1              loop for next pass
L2:   ADDQ #2,SP          clean up the stack
```

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

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

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

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

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

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

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

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

Well, it *does* generate a lot of code, doesn't it? But at least it's the *right* code.

The DO Statement

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

The syntax and its translation is:

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

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

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

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

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

The BREAK Statement

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

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

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

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

loops *do* change the level, each loop construct simply ignores whatever label is above it and passes its own exit label along.

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

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

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

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

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

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

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

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

The new version of `DoIf` is:

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

procedure Block(L: string); Forward;


procedure DoIf(L: string);
var L1, L2: string;
begin
   Match('i');
   Condition;
```

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

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

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

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

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


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

{ Parse and Translate a Program }

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

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

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

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

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

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

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

Conclusion

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

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

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

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

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

const TAB = ^I;
      CR  = ^M;


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

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


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

procedure GetChar;
begin
   Read(Look);
end;


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

procedure Error(s: string);
```

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


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

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


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

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

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

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


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

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


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

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


{---------------------------------------------------------------}
{ Recognize an Addop }

function IsAddop(c: char): boolean;
begin
   IsAddop := c in ['+', '-'];
end;
```

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

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


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

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


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

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




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

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


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

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


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

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

```
end;


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

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


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

{ Output a String with Tab and CRLF }

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


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

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




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

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


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

procedure Block(L: string); Forward;


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

```
      EmitLn('BRA ' + L2);
      PostLabel(L1);
      Block(L);
   end;
   Match('e');
   PostLabel(L2);
end;


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

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


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

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


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

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

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

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




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

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


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

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


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

procedure Other;
begin
   EmitLn(GetName);
end;


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

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




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

{ Parse and Translate a Program }

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


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

{ Initialize }

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

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

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

Chapter 6

Part VI: Boolean Expressions - 31 August 1988

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 their 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.

Chapter 7

Part VII: Lexical Scanning - 7 November 1988

Introduction

In the last installment, I left you with a compiler that would *almost* work, except that we were still limited to single-character tokens. The purpose of this session is to get rid of that restriction, once and for all. This means that we must deal with the concept of the lexical scanner.

Maybe I should mention why we need a lexical scanner at all ... after all, we've been able to manage all right without one, up till now, even when we provided for multi-character tokens.

The *only* reason, really, has to do with keywords. It's a fact of computer life that the syntax for a keyword has the same form as that for any other identifier. We can't tell until we get the complete word whether or not it *is* a keyword. For example, the variable `IFILE` and the keyword `IF` look just alike, until you get to the third character. In the examples to date, we were always able to make a decision based upon the first character of the token, but that's no longer possible when keywords are present. We need to know that a given string is a keyword *before* we begin to process it. And that's why we need a scanner.

In the last session, I also promised that we would be able to provide for normal tokens without making wholesale changes to what we have already done. I didn't lie ... we can, as you will see later. But every time I set out to install these elements of the software into the parser we have already built, I had bad feelings about it. The whole thing felt entirely too much like a band-aid. I finally figured out what was causing the problem: I was installing lexical scanning software without first explaining to you what scanning is all about, and what the alternatives are. Up till now, I have studiously avoided giving you a lot of theory, and certainly not alternatives. I generally don't respond well to the textbooks that give you twenty-five different ways to do something, but no clue as to which way best fits your needs. I've tried to avoid that pitfall by just showing you *one* method, that *works*.

But this is an important area. While the lexical scanner is hardly the most exciting part of a compiler, it often has the most profound effect on the general "look & feel" of the language, since after all it's the part closest to the user. I have a particular structure in mind for the scanner to be used with KISS. It fits the look & feel that I want for that language. But it may not work at all for the language *you're* cooking up, so in this one case I feel that it's important for you to know your options.

So I'm going to depart, again, from my usual format. In this session we'll be getting much deeper than usual into the basic theory of languages and grammars. I'll also be talking about areas *other* than compilers in which lexical scanning plays an important role. Finally, I will show you some alternatives for the structure of the lexical scanner. Then, and only then, will we get back to our parser from the last installment. Bear with me ... I think you'll find it's worth the wait. In fact, since scanners have many applications outside of compilers, you may well find this to be the most useful session for you.

Lexical Scanning

Lexical scanning is the process of scanning the stream of input characters and separating it into strings called tokens. Most compiler texts start here, and devote several chapters to discussing various ways to build scanners. This approach has its place, but as you have already seen, there is a lot you can do without ever even addressing the issue, and in fact the scanner we'll end up with here won't look much like what the texts describe. The reason? Compiler theory and, consequently, the programs resulting from it, must deal with the most general kind of parsing rules. We don't. In the real world, it is possible to specify the language syntax in such a way that a pretty simple scanner will suffice. And as always, KISS is our motto.

Typically, lexical scanning is done in a separate part of the compiler, so that the parser per se sees only a stream of input tokens. Now, theoretically it is not necessary to separate this function from the rest of the parser. There is only one set of syntax equations that define the whole language, so in theory we could write the whole parser in one module.

Why the separation? The answer has both practical and theoretical bases.

In 1956, Noam Chomsky defined the "Chomsky Hierarchy" of grammars. They are:

- Type 0: Unrestricted (e.g., English)
- Type 1: Context-Sensitive
- Type 2: Context-Free
- Type 3: Regular

A few features of the typical programming language (particularly the older ones, such as FORTRAN) are Type 1, but for the most part all modern languages can be described using only the last two types, and those are all we'll be dealing with here.

The neat part about these two types is that there are very specific ways to parse them. It has been shown that any regular grammar can be parsed using a particular form of abstract machine called the state machine (finite automaton). We have already implemented state machines in some of our recognizers.

Similarly, Type 2 (context-free) grammars can always be parsed using a push-down automaton (a state machine augmented by a stack). We have also implemented these machines. Instead of implementing a literal stack, we have relied on the built-in stack associated with recursive coding to do the job, and that in fact is the preferred approach for top-down parsing.

Now, it happens that in real, practical grammars, the parts that qualify as regular expressions tend to be the lower-level parts, such as the definition of an identifier:

```
<ident> ::= <letter> [ <letter> | <digit> ]*
```

Since it takes a different kind of abstract machine to parse the two types of grammars, it makes sense to separate these lower-level functions into a separate module, the lexical scanner, which is built around the idea of a state machine. The idea is to use the simplest parsing technique needed for the job.

There is another, more practical reason for separating scanner from parser. We like to think of the input source file as a stream of characters, which we process right to left without backtracking. In practice that isn't possible. Almost every language has certain keywords such as `IF`, `WHILE`, and `END`. As I mentioned earlier, we can't really know whether a given character string is a keyword, until we've reached the end of it, as defined by a space or other delimiter. So in that sense, we MUST save the string long enough to find out whether we have a keyword or not. That's a limited form of backtracking.

So the structure of a conventional compiler involves splitting up the functions of the lower-level and higher-level parsing. The lexical scanner deals with things at the character level, collecting characters into strings, etc., and passing them along to the parser proper as indivisible tokens. It's also considered normal to let the scanner have the job of identifying keywords.

State Machines and Alternatives

I mentioned that the regular expressions can be parsed using a state machine. In most compiler texts, and indeed in most compilers as well, you will find this taken literally. There is typically a real implementation of the state machine, with integers used to define the current state, and a table of actions to take for each combination of current state and input character. If you write a compiler front end using the popular Unix tools LEX and YACC, that's what you'll get. The output of LEX is a state machine implemented in C, plus a table of actions corresponding to the input grammar given to LEX. The YACC output is similar ... a canned table-driven parser, plus the table corresponding to the language syntax.

That is not the only choice, though. In our previous installments, you have seen over and over that it is possible to implement parsers without dealing specifically with tables, stacks, or state variables. In fact, in Installment V I warned you that if you find yourself needing these things you might be doing something wrong, and not taking advantage of the power of Pascal. There are basically two ways to define a state machine's state: explicitly, with a state number or code, and implicitly, simply by virtue of the fact that I'm at a certain place in the code (if it's Tuesday, this must be Belgium). We've relied heavily on the implicit approaches before, and I think you'll find that they work well here, too.

In practice, it may not even be necessary to *have* a well-defined lexical scanner. This isn't our first experience at dealing with multi-character tokens. In Installment III, we extended our parser to provide for them, and

we didn't even *need* a lexical scanner. That was because in that narrow context, we could always tell, just by looking at the single lookahead character, whether we were dealing with a number, a variable, or an operator. In effect, we built a distributed lexical scanner, using procedures `GetName` and `GetNum`.

With keywords present, we can't know anymore what we're dealing with, until the entire token is read. This leads us to a more localized scanner; although, as you will see, the idea of a distributed scanner still has its merits.

Some Experiments in Scanning

Before getting back to our compiler, it will be useful to experiment a bit with the general concepts.

Let's begin with the two definitions most often seen in real programming languages:

```
<ident> ::= <letter> [ <letter> | <digit> ]*
<number ::= [<digit>]+
```

(Remember, the `*` indicates zero or more occurrences of the terms in brackets, and the `+`, one or more.)

We have already dealt with similar items in Installment III. Let's begin (as usual) with a bare cradle. Not surprisingly, we are going to need a new recognizer:

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

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

Using this let's write the following two routines, which are very similar to those we've used before:

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

function GetName: string;
var x: string[8];
begin
   x := '';
   if not IsAlpha(Look) then Expected('Name');
   while IsAlNum(Look) do begin
     x := x + UpCase(Look);
     GetChar;
   end;
   GetName := x;
end;


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

function GetNum: string;
var x: string[16];
begin
   x := '';
   if not IsDigit(Look) then Expected('Integer');
   while IsDigit(Look) do begin
     x := x + Look;
     GetChar;
   end;
   GetNum := x;
end;
{--------------------------------------------------------------}
```

(Notice that this version of `GetNum` returns a string, not an integer as before.)

You can easily verify that these routines work by calling them from the main program, as in `WriteLn( GetName);`.

This program will print any legal name typed in (maximum eight characters, since that's what we told `GetName`). It will reject anything else.

Test the other routine similarly.

White Space

We also have dealt with embedded white space before, using the two routines `IsWhite` and `SkipWhite`. Make sure that these routines are in your current version of the cradle, and add the the line `SkipWhite;` at the end of both `GetName` and `GetNum`.

Now, let's define the new procedure:

```
{--------------------------------------------------------------}
{ Lexical Scanner }

Function Scan: string;
begin
   if IsAlpha(Look) then
      Scan := GetName
   else if IsDigit(Look) then
      Scan := GetNum
   else begin
      Scan := Look;
      GetChar;
   end;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

We can call this from the new main program:

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


begin
   Init;
   repeat
      Token := Scan;
      writeln(Token);
   until Token = CR;
end.
{--------------------------------------------------------------}
```

(You will have to add the declaration of the string `Token` at the beginning of the program. Make it any convenient length, say 16 characters.)

Now, run the program. Note how the input string is, indeed, separated into distinct tokens.

State Machines

For the record, a parse routine like `GetName` does indeed implement a state machine. The state is implicit in the current position in the code. A very useful trick for visualizing what's going on is the syntax diagram, or "railroad-track" diagram. It's a little difficult to draw one in this medium, so I'll use them very sparingly, but the figure below should give you the idea:

```
        |-----> Other--------------------------> Error
        |
Start -------> Letter ---------------> Other -----> Finish
        ^                         V
        |                         |
        |<----- Letter <----------|
        |                         |
        |<----- Digit  <----------
```

As you can see, this diagram shows how the logic flows as characters are read. Things begin, of course, in the start state, and end when a character other than an alphanumeric is found. If the first character is not alpha, an error occurs. Otherwise the machine will continue looping until the terminating delimiter is found.

Note that at any point in the flow, our position is entirely dependent on the past history of the input characters. At that point, the action to be taken depends only on the current state, plus the current input character. That's what make this a state machine.

Because of the difficulty of drawing railroad-track diagrams in this medium, I'll continue to stick to syntax equations from now on. But I highly recommend the diagrams to you for anything you do that involves parsing. After a little practice you can begin to see how to write a parser directly from the diagrams. Parallel paths get coded into guarded actions (guarded by `IF`s or `CASE` statements), serial paths into sequential calls. It's almost like working from a schematic.

We didn't even discuss `SkipWhite`, which was introduced earlier, but it also is a simple state machine, as is `GetNum`. So is their parent procedure, `Scan`. Little machines make big machines.

The neat thing that I'd like you to note is how painlessly this implicit approach creates these state machines. I personally prefer it a lot over the table-driven approach. It also results is a small, tight, and fast scanner.

Newlines

Moving right along, let's modify our scanner to handle more than one line. As I mentioned last time, the most straightforward way to do this is to simply treat the newline characters, carriage return and line feed, as white space. This is, in fact, the way the C standard library routine, `iswhite`, works. We didn't actually try this before. I'd like to do it now, so you can get a feel for the results.

To do this, simply modify the single executable line of `IsWhite` to read:

```
IsWhite := c in [' ', TAB, CR, LF];
```

We need to give the main program a new stop condition, since it will never see a CR. Let's just use:

```
until Token = '.';
```

OK, compile this program and run it. Try a couple of lines, terminated by the period. I used:

```
now is the time
for all good men.
```

Hey, what happened? When I tried it, I didn't get the last token, the period. The program didn't halt. What's more, when I pressed the `enter` key a few times, I still didn't get the period.

If you're still stuck in your program, you'll find that typing a period on a new line will terminate it.

What's going on here? The answer is that we're hanging up in `SkipWhite`. A quick look at that routine will show that as long as we're typing null lines, we're going to just continue to loop. After `SkipWhite` encounters an LF, it tries to execute a `GetChar`. But since the input buffer is now empty, `GetChar`'s read statement insists on having another line. Procedure `Scan` gets the terminating period, all right, but it calls `SkipWhite` to clean up, and `SkipWhite` won't return until it gets a non-null line.

This kind of behavior is not quite as bad as it seems. In a real compiler, we'd be reading from an input file instead of the console, and as long as we have some procedure for dealing with end-of-files, everything will come out OK. But for reading data from the console, the behavior is just too bizarre. The fact of the matter is that the C/Unix convention is just not compatible with the structure of our parser, which calls for a lookahead character. The code that the Bell wizards have implemented doesn't use that convention, which is why they need `ungetc`.

OK, let's fix the problem. To do that, we need to go back to the old definition of `IsWhite` (delete the CR and LF characters) and make use of the procedure `Fin` that I introduced last time. If it's not in your current version of the cradle, put it there now.

Also, modify the main program to read:

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


begin
   Init;
   repeat
      Token := Scan;
      writeln(Token);
      if Token = CR then Fin;
   until Token = '.';
end.
{--------------------------------------------------------------}
```

Note the "guard" test preceding the call to `Fin`. That's what makes the whole thing work, and ensures that we don't try to read a line ahead.

Try the code now. I think you'll like it better.

If you refer to the code we did in the last installment, you'll find that I quietly sprinkled calls to `Fin` throughout the code, wherever a line break was appropriate. This is one of those areas that really affects the look & feel that I mentioned. At this point I would urge you to experiment with different arrangements and see how you like them. If you want your language to be truly free-field, then newlines should be transparent. In this case, the best approach is to put the following lines at the *beginning* of `Scan`:

```
while Look = CR do
   Fin;
```

If, on the other hand, you want a line-oriented language like Assembler, BASIC, or FORTRAN (or even Ada... note that it has comments terminated by newlines), then you'll need for `Scan` to return CRs as tokens. It must also eat the trailing LF. The best way to do that is to use this line, again at the beginning of `Scan`:

```
if Look = LF then Fin;
```

For other conventions, you'll have to use other arrangements. In my example of the last session, I allowed newlines only at specific places, so I was somewhere in the middle ground. In the rest of these sessions, I'll be picking ways to handle newlines that I happen to like, but I want you to know how to choose other ways for yourselves.

Operators

We could stop now and have a pretty useful scanner for our purposes. In the fragments of KISS that we've built so far, the only tokens that have multiple characters are the identifiers and numbers. All operators were single characters. The only exception I can think of is the relops `<=`, `>=`, and `<>`, but they could be dealt with as special cases.

Still, other languages have multi-character operators, such as the `:=` of Pascal or the `++` and `>>` of C. So while we may not need multi-character operators, it's nice to know how to get them if necessary.

Needless to say, we can handle operators very much the same way as the other tokens. Let's start with a recognizer:

```
{--------------------------------------------------------------}
{ Recognize Any Operator }

function IsOp(c: char): boolean;
begin
   IsOp := c in ['+', '-', '*', '/', '<', '>', ':', '='];
end;
{--------------------------------------------------------------}
```

It's important to note that we *don't* have to include every possible operator in this list. For example, the parentheses aren't included, nor is the terminating period. The current version of `Scan` handles single-character operators just fine as it is. The list above includes only those characters that can appear in multi-character operators. (For specific languages, of course, the list can always be edited.)

Now, let's modify `Scan` to read:

```
{--------------------------------------------------------------}
{ Lexical Scanner }

Function Scan: string;
begin
   while Look = CR do
      Fin;
   if IsAlpha(Look) then
      Scan := GetName
   else if IsDigit(Look) then
      Scan := GetNum
   else if IsOp(Look) then
      Scan := GetOp
   else begin
      Scan := Look;
      GetChar;
   end;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

Try the program now. You will find that any code fragments you care to throw at it will be neatly broken up into individual tokens.

Lists, Commas and Command Lines

Before getting back to the main thrust of our study, I'd like to get on my soapbox for a moment.

How many times have you worked with a program or operating system that had rigid rules about how you must separate items in a list? (Try, the last time you used MSDOS!) Some programs require spaces as delimiters, and some require commas. Worst of all, some require both, in different places. Most are pretty unforgiving about violations of their rules.

I think this is inexcusable. It's too easy to write a parser that will handle both spaces and commas in a flexible way. Consider the following procedure:

```
{--------------------------------------------------------------}
{ Skip Over a Comma }

procedure SkipComma;
begin
   SkipWhite;
   if Look = ',' then begin
      GetChar;
      SkipWhite;
   end;
end;
{--------------------------------------------------------------}
```

This eight-line procedure will skip over a delimiter consisting of any number (including zero) of spaces, with zero or one comma embedded in the string.

Temporarily, change the call to `SkipWhite` in `Scan` to a call to `SkipComma`, and try inputting some lists. Works nicely, eh? Don't you wish more software authors knew about `SkipComma`?

For the record, I found that adding the equivalent of `SkipComma` to my Z80 assembler-language programs took all of 6 (six) extra bytes of code. Even in a 64K machine, that's not a very high price to pay for user-friendliness!

I think you can see where I'm going here. Even if you never write a line of a compiler code in your life, there are places in every program where you can use the concepts of parsing. Any program that processes a command line needs them. In fact, if you think about it for a bit, you'll have to conclude that any time you write a program that processes user inputs, you're defining a language. People communicate with languages, and the syntax implicit in your program defines that language. The real question is: are you going to define it deliberately and explicitly, or just let it turn out to be whatever the program ends up parsing?

I claim that you'll have a better, more user-friendly program if you'll take the time to define the syntax explicitly. Write down the syntax equations or draw the railroad-track diagrams, and code the parser using the techniques I've shown you here. You'll end up with a better program, and it will be easier to write, to boot.

Getting Fancy

OK, at this point we have a pretty nice lexical scanner that will break an input stream up into tokens. We could use it as it stands and have a serviceable compiler. But there are some other aspects of lexical scanning that we need to cover.

The main consideration is `<shudder>` efficiency. Remember when we were dealing with single-character tokens, every test was a comparison of a single character, Look, with a byte constant. We also used the Case statement heavily.

With the multi-character tokens being returned by `Scan`, all those tests now become string comparisons. Much slower. And not only slower, but more awkward, since there is no string equivalent of the Case statement in Pascal. It seems especially wasteful to test for what used to be single characters ... the `=`, `+`, and other operators ... using string comparisons.

Using string comparison is not impossible ... Ron Cain used just that approach in writing Small C. Since we're sticking to the KISS principle here, we would be truly justified in settling for this approach. But then I would have failed to tell you about one of the key approaches used in "real" compilers.

You have to remember: the lexical scanner is going to be called a *LOT*! Once for every token in the whole source program, in fact. Experiments have indicated that the average compiler spends anywhere from 20% to 40% of its time in the scanner routines. If there were ever a place where efficiency deserves real consideration, this is it.

For this reason, most compiler writers ask the lexical scanner to do a little more work, by "tokenizing" the input stream. The idea is to match every token against a list of acceptable keywords and operators, and return unique codes for each one recognized. In the case of ordinary variable names or numbers, we just return a code that says what kind of token they are, and save the actual string somewhere else.

One of the first things we're going to need is a way to identify keywords. We can always do it with successive `IF` tests, but it surely would be nice if we had a general-purpose routine that could compare a given string with a table of keywords. (By the way, we're also going to need such a routine later, for dealing with symbol tables.) This usually presents a problem in Pascal, because standard Pascal doesn't allow for arrays of variable lengths. It's a real bother to have to declare a different search routine for every table. Standard Pascal also doesn't allow for initializing arrays, so you tend to see code like

```
Table[1] := 'IF';
Table[2] := 'ELSE';
.
.
Table[n] := 'END';
```

which can get pretty old if there are many keywords.

Fortunately, Turbo Pascal 4.0 has extensions that eliminate both of these problems. Constant arrays can be declared using TP's "typed constant" facility, and the variable dimensions can be handled with its C-like extensions for pointers.

First, modify your declarations like this:

```
{--------------------------------------------------------------}
{ Type Declarations  }

type Symbol = string[8];

     SymTab = array[1..1000] of Symbol;

     TabPtr = ^SymTab;
{--------------------------------------------------------------}
```

(The dimension used in SymTab is not real ... no storage is allocated by the declaration itself, and the number need only be "big enough.")

Now, just beneath those declarations, add the following:

```
{-------------------------------------------------------------}
{ Definition of Keywords and Token Types }

const KWlist: array [1..4] of Symbol =
              ('IF', 'ELSE', 'ENDIF', 'END');

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

Next, insert the following new function:

```
{-------------------------------------------------------------}
{ Table Lookup }

{ If the input string matches a table entry, return the entry
  index.  If not, return a zero.  }

function Lookup(T: TabPtr; s: string; n: integer): integer;
var i: integer;
    found: boolean;
begin
   found := false;
   i := n;
   while (i > 0) and not found do
      if s = T^[i] then
         found := true
      else
         dec(i);
   Lookup := i;
end;
{-------------------------------------------------------------}
```

To test it, you can temporarily change the main program as follows:

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


begin
   ReadLn(Token);
   WriteLn(Lookup(Addr(KWList), Token, 4));
end.
{-------------------------------------------------------------}
```

Notice how `Lookup` is called: The `Addr` function sets up a pointer to `KWList`, which gets passed to `Lookup`.

OK, give this a try. Since we're bypassing `Scan` here, you'll have to type the keywords in upper case to get any matches.

Now that we can recognize keywords, the next thing is to arrange to return codes for them.

So what kind of code should we return? There are really only two reasonable choices. This seems like an ideal application for the Pascal enumerated type. For example, you can define something like

```
SymType = (IfSym, ElseSym, EndifSym, EndSym, Ident, Number,
               Operator);
```

and arrange to return a variable of this type. Let's give it a try. Insert the line above into your type definitions.

Now, add the two variable declarations:

```
    Token: Symtype;          { Current Token  }
    Value: String[16];       { String Token of Look }
```

Modify the scanner to read:

```
{--------------------------------------------------------------}
{ Lexical Scanner }

procedure Scan;
var k: integer;
begin
   while Look = CR do
      Fin;
   if IsAlpha(Look) then begin
      Value := GetName;
      k := Lookup(Addr(KWlist), Value, 4);
      if k = 0 then
         Token := Ident
      else
         Token := SymType(k - 1);
      end
   else if IsDigit(Look) then begin
      Value := GetNum;
      Token := Number;
      end
   else if IsOp(Look) then begin
      Value := GetOp;
      Token := Operator;
      end
   else begin
      Value := Look;
      Token := Operator;
      GetChar;
   end;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

(Notice that Scan is now a procedure, not a function.)

Finally, modify the main program to read:

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

begin
   Init;
   repeat
      Scan;
      case Token of
        Ident: write('Ident ');
        Number: Write('Number ');
        Operator: Write('Operator ');
        IfSym, ElseSym, EndifSym, EndSym: Write('Keyword ');
      end;
      Writeln(Value);
   until Token = EndSym;
end.
{--------------------------------------------------------------}
```

What we've done here is to replace the string `Token` used earlier with an enumerated type. `Scan` returns the type in variable `Token`, and returns the string itself in the new variable `Value`.

OK, compile this and give it a whirl. If everything goes right, you should see that we are now recognizing keywords.

What we have now is working right, and it was easy to generate from what we had earlier. However, it still seems a little "busy" to me. We can simplify things a bit by letting `GetName`, `GetNum`, `GetOp`, and `Scan` be procedures working with the global variables `Token` and `Value`, thereby eliminating the local copies. It also seems a little cleaner to move the table lookup into `GetName`. The new form for the four procedures is, then:

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

procedure GetName;
var k: integer;
begin
   Value := '';
   if not IsAlpha(Look) then Expected('Name');
   while IsAlNum(Look) do begin
     Value := Value + UpCase(Look);
     GetChar;
   end;
   k := Lookup(Addr(KWlist), Value, 4);
   if k = 0 then
      Token := Ident
   else
      Token := SymType(k-1);
end;

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

procedure GetNum;
begin
   Value := '';
   if not IsDigit(Look) then Expected('Integer');
   while IsDigit(Look) do begin
     Value := Value + Look;
     GetChar;
   end;
   Token := Number;
end;


{--------------------------------------------------------------}
{ Get an Operator }

procedure GetOp;
begin
   Value := '';
   if not IsOp(Look) then Expected('Operator');
   while IsOp(Look) do begin
     Value := Value + Look;
     GetChar;
   end;
   Token := Operator;
end;


{--------------------------------------------------------------}
{ Lexical Scanner }

procedure Scan;
var k: integer;
```

```
begin
   while Look = CR do
      Fin;
   if IsAlpha(Look) then
      GetName
   else if IsDigit(Look) then
      GetNum
   else if IsOp(Look) then
      GetOp
   else begin
      Value := Look;
      Token := Operator;
      GetChar;
   end;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

Returning a Character

Essentially every scanner I've ever seen that was written in Pascal used the mechanism of an enumerated type that I've just described. It is certainly a workable mechanism, but it doesn't seem the simplest approach to me.

For one thing, the list of possible symbol types can get pretty long. Here, I've used just one symbol, `Operator`, to stand for all of the operators, but I've seen other designs that actually return different codes for each one.

There is, of course, another simple type that can be returned as a code: the character. Instead of returning the enumeration value `Operator` for a `+` sign, what's wrong with just returning the character itself? A character is just as good a variable for encoding the different token types, it can be used in case statements easily, and it's sure a lot easier to type. What could be simpler?

Besides, we've already had experience with the idea of encoding keywords as single characters. Our previous programs are already written that way, so using this approach will minimize the changes to what we've already done.

Some of you may feel that this idea of returning character codes is too mickey-mouse. I must admit it gets a little awkward for multi-character operators like `<=`. If you choose to stay with the enumerated type, fine. For the rest, I'd like to show you how to change what we've done above to support that approach.

First, you can delete the `SymType` declaration now ... we won't be needing that. And you can change the type of `Token` to `char`.

Next, to replace `SymType`, add the following constant string:

```
const KWcode: string[5] = 'xilee';
```

(I'll be encoding all idents with the single character `x`.)

Lastly, modify `Scan` and its relatives as follows:

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

procedure GetName;
begin
   Value := '';
   if not IsAlpha(Look) then Expected('Name');
   while IsAlNum(Look) do begin
     Value := Value + UpCase(Look);
     GetChar;
   end;
   Token := KWcode[Lookup(Addr(KWlist), Value, 4) + 1];
end;


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

```
{ Get a Number }

procedure GetNum;
begin
   Value := '';
   if not IsDigit(Look) then Expected('Integer');
   while IsDigit(Look) do begin
     Value := Value + Look;
     GetChar;
   end;
   Token := '#';
end;


{---------------------------------------------------------------}
{ Get an Operator }

procedure GetOp;
begin
   Value := '';
   if not IsOp(Look) then Expected('Operator');
   while IsOp(Look) do begin
     Value := Value + Look;
     GetChar;
   end;
   if Length(Value) = 1 then
      Token := Value[1]
   else
      Token := '?';
end;


{---------------------------------------------------------------}
{ Lexical Scanner }

procedure Scan;
var k: integer;
begin
   while Look = CR do
      Fin;
   if IsAlpha(Look) then
      GetName
   else if IsDigit(Look) then
      GetNum
   else if IsOp(Look) then begin
      GetOp
   else begin
      Value := Look;
      Token := '?';
      GetChar;
   end;
   SkipWhite;
end;


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


begin
   Init;
```

```
   repeat
      Scan;
      case Token of
        'x': write('Ident ');
        '#': Write('Number ');
        'i', 'l', 'e': Write('Keyword ');
        else Write('Operator ');
      end;
      Writeln(Value);
   until Value = 'END';
end.
{--------------------------------------------------------------}
```

This program should work the same as the previous version. A minor difference in structure, maybe, but it seems more straightforward to me.

Distributed vs Centralized Scanners

The structure for the lexical scanner that I've just shown you is very conventional, and about 99% of all compilers use something very close to it. This is not, however, the only possible structure, or even always the best one.

The problem with the conventional approach is that the scanner has no knowledge of context. For example, it can't distinguish between the assignment operator `=` and the relational operator `=` (perhaps that's why both C and Pascal use different strings for the two). All the scanner can do is to pass the operator along to the parser, which can hopefully tell from the context which operator is meant. Similarly, a keyword like `IF` has no place in the middle of a math expression, but if one happens to appear there, the scanner will see no problem with it, and will return it to the parser, properly encoded as an `IF`.

With this kind of approach, we are not really using all the information at our disposal. In the middle of an expression, for example, the parser "knows" that there is no need to look for keywords, but it has no way of telling the scanner that. So the scanner continues to do so. This, of course, slows down the compilation.

In real-world compilers, the designers often arrange for more information to be passed between parser and scanner, just to avoid this kind of problem. But that can get awkward, and certainly destroys a lot of the modularity of the structure.

The alternative is to seek some way to use the contextual information that comes from knowing where we are in the parser. This leads us back to the notion of a distributed scanner, in which various portions of the scanner are called depending upon the context.

In KISS, as in most languages, keywords *only* appear at the beginning of a statement. In places like expressions, they are not allowed. Also, with one minor exception (the multi-character relops) that is easily handled, all operators are single characters, which means that we don't need `GetOp` at all.

So it turns out that even with multi-character tokens, we can still always tell from the current lookahead character exactly what kind of token is coming, except at the very beginning of a statement.

Even at that point, the *only* kind of token we can accept is an identifier. We need only to determine if that identifier is a keyword or the target of an assignment statement.

We end up, then, still needing only `GetName` and `GetNum`, which are used very much as we've used them in earlier installments.

It may seem at first to you that this is a step backwards, and a rather primitive approach. In fact, it is an improvement over the classical scanner, since we're using the scanning routines only where they're really needed. In places where keywords are not allowed, we don't slow things down by looking for them.

Merging Scanner and Parser

Now that we've covered all of the theory and general aspects of lexical scanning that we'll be needing, I'm *finally* ready to back up my claim that we can accommodate multi-character tokens with minimal change to our previous work. To keep things short and simple I will restrict myself here to a subset of what we've done before; I'm allowing only one control construct (the `IF`) and no Boolean expressions. That's enough to demonstrate the parsing of both keywords and expressions. The extension to the full set of constructs should be pretty apparent from what we've already done.

All the elements of the program to parse this subset, using single-character tokens, exist already in our previous programs. I built it by judicious copying of these files, but I wouldn't dare try to lead you through that process. Instead, to avoid any confusion, the whole program is shown below:

```
{--------------------------------------------------------------}
program KISS;

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

const TAB = ^I;
      CR  = ^M;
      LF  = ^J;

{--------------------------------------------------------------}
{ Type Declarations  }

type Symbol = string[8];

     SymTab = array[1..1000] of Symbol;

     TabPtr = ^SymTab;


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

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


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

procedure GetChar;
begin
   Read(Look);
end;


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

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


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

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


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

procedure Expected(s: string);
begin
```

```
   Abort(s + ' Expected');
end;

{---------------------------------------------------------------}
{ 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 Character }

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 a Mulop }

function IsMulop(c: char): boolean;
begin
   IsMulop := 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 + '''');
   GetChar;
   SkipWhite;
end;


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

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


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

function GetName: char;
begin
   while Look = CR do
      Fin;
   if not IsAlpha(Look) then Expected('Name');
   Getname := UpCase(Look);
   GetChar;
   SkipWhite;
end;


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

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


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

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

```
   Inc(LCount);
end;


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

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


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

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


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

{ Output a String with Tab and CRLF }

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


{---------------------------------------------------------------}
{ Parse and Translate 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;
var s: boolean;
begin
   s := Look = '-';
   if IsAddop(Look) then begin
      GetChar;
      SkipWhite;
   end;
   Factor;
   if s then
      EmitLn('NEG 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;


{---------------------------------------------------------------}
{ Completion of Term Processing  (called by Term and FirstTerm }

procedure Term1;
begin
   while IsMulop(Look) do begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '*': Multiply;
       '/': Divide;
      end;
```

```
   end;
end;


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

procedure Term;
begin
   Factor;
   Term1;
end;


{---------------------------------------------------------------}
{ Parse and Translate a Math Term with Possible Leading Sign }

procedure FirstTerm;
begin
   SignedFactor;
   Term1;
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
   FirstTerm;
   while IsAddop(Look) do begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '+': Add;
       '-': Subtract;
      end;
   end;
end;
```

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

Procedure Condition;
begin
   EmitLn('Condition');
end;


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

procedure Block;
 Forward;

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


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


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

procedure Block;
begin
   while not(Look in ['e', 'l']) do begin
      case Look of
       'i': DoIf;
```

```
       CR: while Look = CR do
              Fin;
       else Assignment;
      end;
   end;
end;


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

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


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

{ Initialize }

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


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

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

A couple of comments:

1. The form for the expression parser, using `FirstTerm`, etc., is a little different from what you've seen before. It's yet another variation on the same theme. Don't let it throw you ... the change is not required for what follows.

2. Note that, as usual, I had to add calls to `Fin` at strategic spots to allow for multiple lines.

Before we proceed to adding the scanner, first copy this file and verify that it does indeed parse things correctly. Don't forget the "codes": `i` for `IF`, `l` for `ELSE`, and `e` for `END` or `ENDIF`.

If the program works, then let's press on. In adding the scanner modules to the program, it helps to have a systematic plan. In all the parsers we've written to date, we've stuck to a convention that the current lookahead character should always be a non-blank character. We preload the lookahead character in `Init`, and keep the "pump primed" after that. To keep the thing working right at newlines, we had to modify this a bit and treat the newline as a legal token.

In the multi-character version, the rule is similar: The current lookahead character should always be left at the *beginning* of the next token, or at a newline.

The multi-character version is shown next. To get it, I've made the following changes:

- Added the variables `Token` and `Value`, and the type definitions needed by `Lookup`.
- Added the definitions of `KWList` and `KWcode`.
- Added `Lookup`.

- Replaced `GetName` and `GetNum` by their multi-character versions. (Note that the call to `Lookup` has been moved out of `GetName`, so that it will not be executed for calls within an expression.)
- Created a new, vestigial `Scan` that calls `GetName`, then scans for keywords.
- Created a new procedure, `MatchString`, that looks for a specific keyword. Note that, unlike `Match`, `MatchString` does *not* read the next keyword.
- Modified `Block` to call `Scan`.
- Changed the calls to `Fin` a bit. `Fin` is now called within `GetName`.

Here is the program in its entirety:

```
{--------------------------------------------------------------}
program KISS;

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

const TAB = ^I;
      CR  = ^M;
      LF  = ^J;

{--------------------------------------------------------------}
{ Type Declarations  }

type Symbol = string[8];

     SymTab = array[1..1000] of Symbol;

     TabPtr = ^SymTab;


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

var Look  : char;              { Lookahead Character }
    Token : char;              { Encoded Token       }
    Value : string[16];        { Unencoded Token     }
    Lcount: integer;           { Label Counter       }


{--------------------------------------------------------------}
{ Definition of Keywords and Token Types }

const KWlist: array [1..4] of Symbol =
              ('IF', 'ELSE', 'ENDIF', 'END');

const KWcode: string[5] = 'xilee';


{--------------------------------------------------------------}
{ 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 Character }

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 a Mulop }

function IsMulop(c: char): boolean;
begin
   IsMulop := 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 + '''');
   GetChar;
   SkipWhite;
end;


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

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


{---------------------------------------------------------------}
{ Table Lookup }

function Lookup(T: TabPtr; s: string; n: integer): integer;
var i: integer;
    found: boolean;
begin
   found := false;
   i := n;
   while (i > 0) and not found do
      if s = T^[i] then
         found := true
```

```
      else
         dec(i);
   Lookup := i;
end;


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

procedure GetName;
begin
   while Look = CR do
      Fin;
   if not IsAlpha(Look) then Expected('Name');
   Value := '';
   while IsAlNum(Look) do begin
     Value := Value + UpCase(Look);
     GetChar;
   end;
   SkipWhite;
end;


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

procedure GetNum;
begin
   if not IsDigit(Look) then Expected('Integer');
   Value := '';
   while IsDigit(Look) do begin
     Value := Value + Look;
     GetChar;
   end;
   Token := '#';
   SkipWhite;
end;


{--------------------------------------------------------------}
{ Get an Identifier and Scan it for Keywords }

procedure Scan;
begin
   GetName;
   Token := KWcode[Lookup(Addr(KWlist), Value, 4) + 1];
end;


{--------------------------------------------------------------}
{ Match a Specific Input String }

procedure MatchString(x: string);
begin
   if Value <> x then Expected('''' + x + '''');
end;


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

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


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

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


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

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


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

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


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

procedure Ident;
begin
   GetName;
   if Look = '(' then begin
      Match('(');
      Match(')');
      EmitLn('BSR ' + Value);
      end
   else
      EmitLn('MOVE ' + Value + '(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 begin
      GetNum;
      EmitLn('MOVE #' + Value + ',D0');
   end;
end;


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

procedure SignedFactor;
var s: boolean;
begin
   s := Look = '-';
   if IsAddop(Look) then begin
      GetChar;
      SkipWhite;
   end;
   Factor;
   if s then
      EmitLn('NEG 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;


{---------------------------------------------------------------}
{ Completion of Term Processing  (called by Term and FirstTerm }

procedure Term1;
begin
   while IsMulop(Look) do begin
      EmitLn('MOVE D0,-(SP)');
```

```
      case Look of
       '*': Multiply;
       '/': Divide;
      end;
   end;
end;
{--------------------------------------------------------------}
{ Parse and Translate a Math Term }

procedure Term;
begin
   Factor;
   Term1;
end;


{--------------------------------------------------------------}
{ Parse and Translate a Math Term with Possible Leading Sign }

procedure FirstTerm;
begin
   SignedFactor;
   Term1;
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
   FirstTerm;
   while IsAddop(Look) do begin
      EmitLn('MOVE D0,-(SP)');
      case Look of
       '+': Add;
       '-': Subtract;
      end;
```

```
   end;
end;


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

Procedure Condition;
begin
   EmitLn('Condition');
end;


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

procedure Block; Forward;


procedure DoIf;
var L1, L2: string;
begin
   Condition;
   L1 := NewLabel;
   L2 := L1;
   EmitLn('BEQ ' + L1);
   Block;
   if Token = 'l' then begin
      L2 := NewLabel;
      EmitLn('BRA ' + L2);
      PostLabel(L1);
      Block;
   end;
   PostLabel(L2);
   MatchString('ENDIF');
end;


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

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


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

procedure Block;
begin
   Scan;
   while not (Token in ['e', 'l']) do begin
      case Token of
```

```
        'i': DoIf;
        else Assignment;
       end;
       Scan;
   end;
end;


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

{ Parse and Translate a Program }

procedure DoProgram;
begin
   Block;
   MatchString('END');
   EmitLn('END')
end;


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

{ Initialize }

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


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

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

Compare this program with its single-character counterpart. I think you will agree that the differences are minor.

Conclusion

At this point, you have learned how to parse and generate code for expressions, Boolean expressions, and control structures. You have now learned how to develop lexical scanners, and how to incorporate their elements into a translator. You have still not seen *all* the elements combined into one program, but on the basis of what we've done before you should find it a straightforward matter to extend our earlier programs to include scanners.

We are very close to having all the elements that we need to build a real, functional compiler. There are still a few things missing, notably procedure calls and type definitions. We will deal with those in the next few sessions. Before doing so, however, I thought it would be fun to turn the translator above into a true compiler. That's what we'll be doing in the next installment.

Up till now, we've taken a rather bottom-up approach to parsing, beginning with low-level constructs and working our way up. In the next installment, I'll also be taking a look from the top down, and we'll discuss how the structure of the translator is altered by changes in the language definition.

See you then.

Chapter 8

Part VIII: A Little Philosophy - 2 April 1989

Introduction

This is going to be a different kind of session than the others in our series on parsing and compiler construction. For this session, there won't be any experiments to do or code to write. This once, I'd like to just talk with you for a while. Mercifully, it will be a short session, and then we can take up where we left off, hopefully with renewed vigor.

When I was in college, I found that I could always follow a prof's lecture a lot better if I knew where they were going with it. I'll bet you were the same.

So I thought maybe it's about time I told you where we're going with this series: what's coming up in future installments, and in general what all this is about. I'll also share some general thoughts concerning the usefulness of what we've been doing.

The Road Home

So far, we've covered the parsing and translation of arithmetic expressions, Boolean expressions, and combinations connected by relational operators. We've also done the same for control constructs. In all of this we've leaned heavily on the use of top-down, recursive descent parsing, BNF definitions of the syntax, and direct generation of assembly-language code. We also learned the value of such tricks as single-character tokens to help us see the forest through the trees. In the last installment we dealt with lexical scanning, and I showed you simple but powerful ways to remove the single-character barriers.

Throughout the whole study, I've emphasized the KISS philosophy ... Keep It Simple, Sidney ... and I hope by now you've realized just how simple this stuff can really be. While there are for sure areas of compiler theory that are truly intimidating, the ultimate message of this series is that in practice you can just politely sidestep many of these areas. If the language definition cooperates or, as in this series, if you can define the language as you go, it's possible to write down the language definition in BNF with reasonable ease. And, as we've seen, you can crank out parse procedures from the BNF just about as fast as you can type.

As our compiler has taken form, it's gotten more parts, but each part is quite small and simple, and very much like all the others.

At this point, we have many of the makings of a real, practical compiler. As a matter of fact, we already have all we need to build a toy compiler for a language as powerful as, say, Tiny BASIC. In the next couple of installments, we'll go ahead and define that language.

To round out the series, we still have a few items to cover. These include:

- Procedure calls, with and without parameters
- Local and global variables
- Basic types, such as character and integer types
- Arrays
- Strings
- User-defined types and structures
- Tree-structured parsers and intermediate languages

- Optimization

These will all be covered in future installments. When we're finished, you'll have all the tools you need to design and build your own languages, and the compilers to translate them.

I can't design those languages for you, but I can make some comments and recommendations. I've already sprinkled some throughout past installments. You've seen, for example, the control constructs I prefer.

These constructs are going to be part of the languages I build. I have three languages in mind at this point, two of which you will see in installments to come:

- TINY: A minimal, but usable language on the order of Tiny BASIC or Tiny C. It won't be very practical, but it will have enough power to let you write and run real programs that do something worthwhile.
- KISS: The language I'm building for my own use. KISS is intended to be a systems programming language. It won't have strong typing or fancy data structures, but it will support most of the things I want to do with a higher-order language (HOL), except perhaps writing compilers.

I've also been toying for years with the idea of a HOL-like assembler, with structured control constructs and HOL-like assignment statements. That, in fact, was the impetus behind my original foray into the jungles of compiler theory. This one may never be built, simply because I've learned that it's actually easier to implement a language like KISS, that only uses a subset of the CPU instructions. As you know, assembly language can be bizarre and irregular in the extreme, and a language that maps one-for-one onto it can be a real challenge. Still, I've always felt that the syntax used in conventional assemblers is dumb ... why is `MOVE.L A,B` better, or easier to translate, than `B=A`?

I think it would be an interesting exercise to develop a "compiler" that would give the programmer complete access to and control over the full complement of the CPU instruction set, and would allow you to generate programs as efficient as assembly language, without the pain of learning a set of mnemonics. Can it be done? I don't know. The real question may be, "Will the resulting language be any easier to write than assembly"? If not, there's no point in it. I think that it can be done, but I'm not completely sure yet how the syntax should look.

Perhaps you have some comments or suggestions on this one. I'd love to hear them.

You probably won't be surprised to learn that I've already worked ahead in most of the areas that we will cover. I have some good news: Things never get much harder than they've been so far. It's possible to build a complete, working compiler for a real language, using nothing but the same kinds of techniques you've learned so far. And *that* brings up some interesting questions.

Why Is It So Simple?

Before embarking on this series, I always thought that compilers were just naturally complex computer programs ... the ultimate challenge. Yet the things we have done here have usually turned out to be quite simple, sometimes even trivial.

For awhile, I thought is was simply because I hadn't yet gotten into the meat of the subject. I had only covered the simple parts. I will freely admit to you that, even when I began the series, I wasn't sure how far we would be able to go before things got too complex to deal with in the ways we have so far. But at this point I've already been down the road far enough to see the end of it. Guess what?

THERE ARE NO HARD PARTS!

Then, I thought maybe it was because we were not generating very good object code. Those of you who have been following the series and trying sample compiles know that, while the code works and is rather foolproof, its efficiency is pretty awful. I figured that if we were concentrating on turning out tight code, we would soon find all that missing complexity.

To some extent, that one is true. In particular, my first few efforts at trying to improve efficiency introduced complexity at an alarming rate. But since then I've been tinkering around with some simple optimizations and I've found some that result in very respectable code quality, *without* adding a lot of complexity.

Finally, I thought that perhaps the saving grace was the "toy compiler" nature of the study. I have made no pretense that we were ever going to be able to build a compiler to compete with Borland and Microsoft. And yet, again, as I get deeper into this thing the differences are starting to fade away.

Just to make sure you get the message here, let me state it flat out:

USING THE TECHNIQUES WE'VE USED HERE, IT IS POSSIBLE TO BUILD A PRODUCTION-QUALITY, WORKING COMPILER WITHOUT ADDING A LOT OF COMPLEXITY TO WHAT WE'VE ALREADY DONE.

Since the series began I've received some comments from you. Most of them echo my own thoughts: "This is easy! Why do the textbooks make it seem so hard?" Good question.

Recently, I've gone back and looked at some of those texts again, and even bought and read some new ones. Each time, I come away with the same feeling: These guys have made it seem too hard.

What's going on here? Why does the whole thing seem difficult in the texts, but easy to us? Are we that much smarter than Aho, Ullman, Brinch Hansen, and all the rest?

Hardly. But we are doing some things differently, and more and more I'm starting to appreciate the value of our approach, and the way that it simplifies things. Aside from the obvious shortcuts that I outlined in Part I, like single-character tokens and console I/O, we have made some implicit assumptions and done some things differently from those who have designed compilers in the past. As it turns out, our approach makes life a lot easier.

So why didn't all those other guys use it?

You have to remember the context of some of the earlier compiler development. These people were working with very small computers of limited capacity. Memory was very limited, the CPU instruction set was minimal, and programs ran in batch mode rather than interactively. As it turns out, these caused some key design decisions that have really complicated the designs. Until recently, I hadn't realized how much of classical compiler design was driven by the available hardware.

Even in cases where these limitations no longer apply, people have tended to structure their programs in the same way, since that is the way they were taught to do it.

In our case, we have started with a blank sheet of paper. There is a danger there, of course, that you will end up falling into traps that other people have long since learned to avoid. But it also has allowed us to take different approaches that, partly by design and partly by pure dumb luck, have allowed us to gain simplicity.

Here are the areas that I think have led to complexity in the past:

- **Limited RAM Forcing Multiple Passes**

  I just read "Brinch Hansen on Pascal Compilers" (an excellent book, BTW). He developed a Pascal compiler for a PC, but he started the effort in 1981 with a 64K system, and so almost every design decision he made was aimed at making the compiler fit into RAM. To do this, their compiler has three passes, one of which is the lexical scanner. There is no way he could, for example, use the distributed scanner I introduced in the last installment, because the program structure wouldn't allow it. He also required not one but two intermediate languages, to provide the communication between phases.

  All the early compiler writers had to deal with this issue: Break the compiler up into enough parts so that it will fit in memory. When you have multiple passes, you need to add data structures to support the information that each pass leaves behind for the next. That adds complexity, and ends up driving the design. Lee's book, "The Anatomy of a Compiler," mentions a FORTRAN compiler developed for an IBM 1401. It had no fewer than 63 separate passes! Needless to say, in a compiler like this the separation into phases would dominate the design.

  Even in situations where RAM is plentiful, people have tended to use the same techniques because that is what they're familiar with. It wasn't until Turbo Pascal came along that we found how simple a compiler could be if you started with different assumptions.

- **Batch Processing**

  In the early days, batch processing was the only choice ... there was no interactive computing. Even today, compilers run in essentially batch mode.

  In a mainframe compiler as well as many micro compilers, considerable effort is expended on error recovery ... it can consume as much as 30-40% of the compiler and completely drive the design. The idea is to avoid halting on the first error, but rather to keep going at all costs, so that you can tell the programmer about as many errors in the whole program as possible.

  All of that harks back to the days of the early mainframes, where turnaround time was measured in hours or days, and it was important to squeeze every last ounce of information out of each run.

  In this series, I've been very careful to avoid the issue of error recovery, and instead our compiler simply halts with an error message on the first error. I will frankly admit that it was mostly because I wanted to take the easy way out and keep things simple. But this approach, pioneered by Borland in Turbo Pascal, also has a lot going for it anyway. Aside from keeping the compiler simple, it also fits very well with the idea of an interactive system. When compilation is fast, and especially when you have an editor such as Borland's that will take you right to the point of the error, then it makes a lot of sense to stop there, and just restart the compilation after the error is fixed.

- **Large Programs**

  Early compilers were designed to handle large programs ... essentially infinite ones. In those days there was little choice; the idea of subroutine libraries and separate compilation were still in the future. Again, this assumption led to multi-pass designs and intermediate files to hold the results of partial processing.

  Brinch Hansen's stated goal was that the compiler should be able to compile itself. Again, because of his limited RAM, this drove him to a multi-pass design. He needed as little resident compiler code as possible, so that the necessary tables and other data structures would fit into RAM.

I haven't stated this one yet, because there hasn't been a need ... we've always just read and written the data as streams, anyway. But for the record, my plan has always been that, in a production compiler, the source and object data should all coexist in RAM with the compiler, a la the early Turbo Pascals. That's why I've been careful to keep routines like `GetChar` and `Emit` as separate routines, in spite of their small size. It will be easy to change them to read to and write from memory.

- **Emphasis on Efficiency**

  John Backus has stated that, when he and his colleagues developed the original FORTRAN compiler, they *knew* that they had to make it produce tight code. In those days, there was a strong sentiment against HOLs and in favor of assembly language, and efficiency was the reason. If FORTRAN didn't produce very good code by assembly standards, the users would simply refuse to use it. For the record, that FORTRAN compiler turned out to be one of the most efficient ever built, in terms of code quality. But it *was* complex!

  Today, we have CPU power and RAM size to spare, so code efficiency is not so much of an issue. By studiously ignoring this issue, we have indeed been able to Keep It Simple. Ironically, though, as I have said, I have found some optimizations that we can add to the basic compiler structure, without having to add a lot of complexity. So in this case we get to have our cake and eat it too: we will end up with reasonable code quality, anyway.

- **Limited Instruction Sets**

  The early computers had primitive instruction sets. Things that we take for granted, such as stack operations and indirect addressing, came only with great difficulty.

  Example: In most compiler designs, there is a data structure called the literal pool. The compiler typically identifies all literals used in the program, and collects them into a single data structure. All references to the literals are done indirectly to this pool. At the end of the compilation, the compiler issues commands to set aside storage and initialize the literal pool.

  We haven't had to address that issue at all. When we want to load a literal, we just do it, in line, as in `MOVE #3,D0`.

  There is something to be said for the use of a literal pool, particularly on a machine like the 8086 where data and code can be separated. Still, the whole thing adds a fairly large amount of complexity with little in return.

  Of course, without the stack we would be lost. In a micro, both subroutine calls and temporary storage depend heavily on the stack, and we have used it even more than necessary to ease expression parsing.

- **Desire for Generality**

  Much of the content of the typical compiler text is taken up with issues we haven't addressed here at all ... things like automated translation of grammars, or generation of LALR parse tables. This is not simply because the authors want to impress you. There are good, practical reasons why the subjects are there.

  We have been concentrating on the use of a recursive-descent parser to parse a deterministic grammar, i.e., a grammar that is not ambiguous and, therefore, can be parsed with one level of lookahead. I haven't made much of this limitation, but the fact is that this represents a small subset of possible grammars. In fact, there is an infinite number of grammars that we can't parse using our techniques. The LR technique is a more powerful one, and can deal with grammars that we can't.

  In compiler theory, it's important to know how to deal with these other grammars, and how to transform them into grammars that are easier to deal with. For example, many (but not all) ambiguous grammars can be transformed into unambiguous ones. The way to do this is not always obvious, though, and so many people have devoted years to develop ways to transform them automatically.

  In practice, these issues turn out to be considerably less important. Modern languages tend to be designed to be easy to parse, anyway. That was a key motivation in the design of Pascal. Sure, there are pathological grammars that you would be hard pressed to write unambiguous BNF for, but in the real world the best answer is probably to avoid those grammars!

  In our case, of course, we have sneakily let the language evolve as we go, so we haven't painted ourselves into any corners here. You may not always have that luxury. Still, with a little care you should be able to keep the parser simple without having to resort to automatic translation of the grammar.

We have taken a vastly different approach in this series. We started with a clean sheet of paper, and developed techniques that work in the context that we are in; that is, a single-user PC with rather ample CPU power and RAM space. We have limited ourselves to reasonable grammars that are easy to parse, we have used the instruction set of the CPU to advantage, and we have not concerned ourselves with efficiency. *That*'s why it's been easy.

Does this mean that we are forever doomed to be able to build only toy compilers? No, I don't think so. As I've said, we can add certain optimizations without changing the compiler structure. If we want to process large files, we can always add file buffering to do that. These things do not affect the overall program design.

And I think that's a key factor. By starting with small and limited cases, we have been able to concentrate on a structure for the compiler that is natural for the job. Since the structure naturally fits the job, it is almost bound to be simple and transparent. Adding capability doesn't have to change that basic structure. We can simply expand things like the file structure or add an optimization layer. I guess my feeling is that, back when resources were tight, the structures people ended up with were artificially warped to make them work under those conditions, and weren't optimum structures for the problem at hand.

Conclusion

Anyway, that's my arm-waving guess as to how we've been able to keep things simple. We started with something simple and let it evolve naturally, without trying to force it into some traditional mold.

We're going to press on with this. I've given you a list of the areas we'll be covering in future installments. With those installments, you should be able to build complete, working compilers for just about any occasion, and build them simply. If you *really* want to build production-quality compilers, you'll be able to do that, too.

For those of you who are chafing at the bit for more parser code, I apologize for this digression. I just thought you'd like to have things put into perspective a bit. Next time, we'll get back to the mainstream of the tutorial.

So far, we've only looked at pieces of compilers, and while we have many of the makings of a complete language, we haven't talked about how to put it all together. That will be the subject of our next two installments. Then we'll press on into the new subjects I listed at the beginning of this installment.

See you then.

Chapter 9

Part IX: A Top View - 16 April 1989

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.

Chapter 10

Part X: Introducing "TINY" - 21 May 1989

Introduction

In the last installment, I showed you the general idea for the top-down development of a compiler. I gave you the first few steps of the process for compilers for Pascal and C, but I stopped far short of pushing it through to completion. The reason was simple: if we're going to produce a real, functional compiler for any language, I'd rather do it for KISS, the language that I've been defining in this tutorial series.

In this installment, we're going to do just that, for a subset of KISS which I've chosen to call TINY.

The process will be essentially that outlined in Installment IX, except for one notable difference. In that installment, I suggested that you begin with a full BNF description of the language. That's fine for something like Pascal or C, for which the language definition is firm. In the case of TINY, however, we don't yet have a full description ... we seem to be defining the language as we go. That's OK. In fact, it's preferable, since we can tailor the language slightly as we go, to keep the parsing easy.

So in the development that follows, we'll actually be doing a top-down development of *both* the language and its compiler. The BNF description will grow along with the compiler.

In this process, there will be a number of decisions to be made, each of which will influence the BNF and therefore the nature of the language. At each decision point I'll try to remember to explain the decision and the rationale behind my choice. That way, if you happen to hold a different opinion and would prefer a different option, you can choose it instead. You now have the background to do that. I guess the important thing to note is that nothing we do here is cast in concrete. When *you're* designing *your* language, you should feel free to do it *your* way.

Many of you may be asking at this point: Why bother starting over from scratch? We had a working subset of KISS as the outcome of Installment VII (lexical scanning). Why not just extend it as needed? The answer is threefold. First of all, I have been making a number of changes to further simplify the program ... changes like encapsulating the code generation procedures, so that we can convert to a different target machine more easily. Second, I want you to see how the development can indeed be done from the top down as outlined in the last installment. Finally, we both need the practice. Each time I go through this exercise, I get a little better at it, and you will, also.

Getting Started

Many years ago there were languages called Tiny BASIC, Tiny Pascal, and Tiny C, each of which was a subset of its parent full language. Tiny BASIC, for example, had only single-character variable names and global variables. It supported only a single data type. Sound familiar? At this point we have almost all the tools we need to build a compiler like that.

Yet a language called Tiny-anything still carries some baggage inherited from its parent language. I've often wondered if this is a good idea. Granted, a language based upon some parent language will have the advantage of familiarity, but there may also be some peculiar syntax carried over from the parent that may tend to add unnecessary complexity to the compiler. (Nowhere is this more true than in Small C.)

I've wondered just how small and simple a compiler could be made and still be useful, if it were designed from the outset to be both easy to use and to parse. Let's find out. This language will just be called "TINY," period. It's a subset of KISS, which I also haven't fully defined, so that at least makes us consistent (!). I suppose you could call it TINY KISS. But that opens up a whole can of worms involving cuter and cuter (and perhaps more risque) names, so let's just stick with TINY.

The main limitations of TINY will be because of the things we haven't yet covered, such as data types. Like its cousins Tiny C and Tiny BASIC, TINY will have only one data type, the 16-bit integer. The first version we develop will also have no procedure calls and will use single-character variable names, although as you will see we can remove these restrictions without much effort.

The language I have in mind will share some of the good features of Pascal, C, and Ada. Taking a lesson from the comparison of the Pascal and C compilers in the previous installment, though, TINY will have a decided Pascal flavor. Wherever feasible, a language structure will be bracketed by keywords or symbols, so that the parser will know where it's going without having to guess.

One other ground rule: As we go, I'd like to keep the compiler producing real, executable code. Even though it may not *do* much at the beginning, it will at least do it correctly.

Finally, I'll use a couple of Pascal restrictions that make sense: All data and procedures must be declared before they are used. That makes good sense, even though for now the only data type we'll use is a word. This rule in turn means that the only reasonable place to put the executable code for the main program is at the end of the listing.

The top-level definition will be similar to Pascal:

```
<program> ::= PROGRAM <top-level decl> <main> '.'
```

Already, we've reached a decision point. My first thought was to make the main block optional. It doesn't seem to make sense to write a "program" with no main program, but it does make sense if we're allowing for multiple modules, linked together. As a matter of fact, I intend to allow for this in KISS. But then we begin to open up a can of worms that I'd rather leave closed for now. For example, the term `PROGRAM` really becomes a misnomer. The `MODULE` of Modula-2 or the `Unit` of Turbo Pascal would be more appropriate. Second, what about scope rules? We'd need a convention for dealing with name visibility across modules. Better for now to just keep it simple and ignore the idea altogether.

There's also a decision in choosing to require the main program to be last. I toyed with the idea of making its position optional, as in C. The nature of SK*DOS, the OS I'm compiling for, make this very easy to do. But this doesn't really make much sense in view of the Pascal-like requirement that all data and procedures be declared before they're referenced. Since the main program can only call procedures that have already been declared, the only position that makes sense is at the end, a la Pascal.

Given the BNF above, let's write a parser that just recognizes the brackets:

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

procedure Prog;
begin
   Match('p');
   Header;
   Prolog;
   Match('.');
   Epilog;
end;
{--------------------------------------------------------------}
```

The procedure `Header` just emits the startup code required by the assembler:

```
{--------------------------------------------------------------}
{ Write Header Info }

procedure Header;
begin
   WriteLn('WARMST', TAB, 'EQU $A01E');
end;
{--------------------------------------------------------------}
```

The procedures `Prolog` and `Epilog` emit the code for identifying the main program, and for returning to the OS:

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

procedure Prolog;
begin
   PostLabel('MAIN');
end;


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

procedure Epilog;
begin
   EmitLn('DC WARMST');
   EmitLn('END MAIN');
end;
{--------------------------------------------------------------}
```

The main program just calls `Prog`, and then looks for a clean ending:

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

begin
   Init;
   Prog;
   if Look <> CR then Abort('Unexpected data after ''.''');
end.
{--------------------------------------------------------------}
```

At this point, TINY will accept only one input "program," the null program: `PROGRAM .` (or `p.` in our shorthand.) Note, though, that the compiler DOES generate correct code for this program. It will run, and do what you'd expect the null program to do, that is, nothing but return gracefully to the OS.

As a matter of interest, one of my favorite compiler benchmarks is to compile, link, and execute the null program in whatever language is involved. You can learn a lot about the implementation by measuring the overhead in time required to compile what should be a trivial case. It's also interesting to measure the amount of code produced. In many compilers, the code can be fairly large, because they always include the whole run-time library whether they need it or not. Early versions of Turbo Pascal produced a 12K object file for this case. VAX C generates 50K!

The smallest null programs I've seen are those produced by Modula-2 compilers, and they run about 200-800 bytes.

In the case of TINY, we *have* no run-time library as yet, so the object code is indeed tiny: two bytes. That's got to be a record, and it's likely to remain one since it is the minimum size required by the OS.

The next step is to process the code for the main program. I'll use the Pascal `BEGIN`-block:

```
<main> ::= BEGIN <block> END
```

Here, again, we have made a decision. We could have chosen to require a `PROCEDURE MAIN` sort of declaration, similar to C. I must admit that this is not a bad idea at all ... I don't particularly like the Pascal approach since I tend to have trouble locating the main program in a Pascal listing. But the alternative is a little awkward, too, since you have to deal with the error condition where the user omits the main program or misspells its name. Here I'm taking the easy way out.

Another solution to the "where is the main program" problem might be to require a name for the program, and then bracket the main by

```
BEGIN <name>
END <name>
```

similar to the convention of Modula 2. This adds a bit of "syntactic sugar" to the language. Things like this are easy to add or change to your liking, if the language is your own design.

To parse this definition of a main block, change procedure `Prog` to read:

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

procedure Prog;
begin
   Match('p');
   Header;
   Main;
   Match('.');
end;
{--------------------------------------------------------------}
```

and add the new procedure:

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

procedure Main;
begin
   Match('b');
   Prolog;
   Match('e');
   Epilog;
end;
{--------------------------------------------------------------}
```

Now, the only legal program is: `PROGRAM BEGIN END .` (or `pbe.`)

Aren't we making progress??? Well, as usual it gets better. You might try some deliberate errors here, like omitting the `b` or the `e`, and see what happens. As always, the compiler should flag all illegal inputs.

Declarations

The obvious next step is to decide what we mean by a declaration. My intent here is to have two kinds of declarations: variables and procedures/functions. At the top level, only global declarations are allowed, just as in C.

For now, there can only be variable declarations, identified by the keyword `VAR` (abbreviated `v`):

```
<top-level decls> ::= ( <data declaration> )*

<data declaration> ::= VAR <var-list>
```

Note that since there is only one variable type, there is no need to declare the type. Later on, for full KISS, we can easily add a type description.

The procedure `Prog` becomes:

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

procedure Prog;
begin
   Match('p');
   Header;
   TopDecls;
   Main;
   Match('.');
end;
{--------------------------------------------------------------}
```

Now, add the two new procedures:

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

procedure Decl;
begin
   Match('v');
   GetChar;
end;


{--------------------------------------------------------------}
{ Parse and Translate Global Declarations }

procedure TopDecls;
begin
   while Look <> 'b' do
      case Look of
        'v': Decl;
      else Abort('Unrecognized Keyword ''' + Look + '''');
      end;
end;
{--------------------------------------------------------------}
```

Note that at this point, `Decl` is just a stub. It generates no code, and it doesn't process a list ... every variable must occur in a separate `VAR` statement.

OK, now we can have any number of data declarations, each starting with a `v` for `VAR`, before the `BEGIN`-block. Try a few cases and see what happens.

Declarations and Symbols

That looks pretty good, but we're still only generating the null program for output. A real compiler would issue assembler directives to allocate storage for the variables. It's about time we actually produced some code.

With a little extra code, that's an easy thing to do from procedure `Decl`. Modify it as follows:

```
{--------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
var Name: char;
begin
   Match('v');
   Alloc(GetName);
end;
{--------------------------------------------------------------}
```

The procedure `Alloc` just issues a command to the assembler to allocate storage:

```
{--------------------------------------------------------------}
{ Allocate Storage for a Variable }

procedure Alloc(N: char);
begin
   WriteLn(N, ':', TAB, 'DC 0');
end;
{--------------------------------------------------------------}
```

Give this one a whirl. Try an input that declares some variables, such as: `pvxvyvzbe.`.

See how the storage is allocated? Simple, huh? Note also that the entry point, `MAIN`, comes out in the right place.

For the record, a "real" compiler would also have a symbol table to record the variables being used. Normally, the symbol table is necessary to record the type of each variable. But since in this case all variables have the same type, we don't need a symbol table for that reason. As it turns out, we're going to find a symbol necessary even without different types, but let's postpone that need until it arises.

Of course, we haven't really parsed the correct syntax for a data declaration, since it involves a variable list. Our version only permits a single variable. That's easy to fix, too.

The BNF for `<var-list>` is

```
<var-list> ::= <ident> (, <ident>)*
```

Adding this syntax to `Decl` gives this new version:

```
{--------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
var Name: char;
begin
   Match('v');
   Alloc(GetName);
   while Look = ',' do begin
      GetChar;
      Alloc(GetName);
   end;
end;
{--------------------------------------------------------------}
```

OK, now compile this code and give it a try. Try a number of lines of `VAR` declarations, try a list of several variables on one line, and try combinations of the two. Does it work?

Initializers

As long as we're dealing with data declarations, one thing that's always bothered me about Pascal is that it doesn't allow initializing data items in the declaration. That feature is admittedly sort of a frill, and it may be out of place in a language that purports to be a minimal language. But it's also SO easy to add that it seems a shame not to do so. The BNF becomes:

```
<var-list> ::= <var> ( <var> )*

<var> ::= <ident> [ = <integer> ]
```

Change `Alloc` as follows:

```
{--------------------------------------------------------------}
{ Allocate Storage for a Variable }

procedure Alloc(N: char);
begin
   Write(N, ':', TAB, 'DC ');
   if Look = '=' then begin
      Match('=');
      WriteLn(GetNum);
      end
   else
      WriteLn('0');
end;
{--------------------------------------------------------------}
```

There you are: an initializer with six added lines of Pascal.

OK, try this version of TINY and verify that you can, indeed, give the variables initial values.

By golly, this thing is starting to look real! Of course, it still doesn't *do* anything, but it looks good, doesn't it?

Before leaving this section, I should point out that we've used two versions of function `GetNum`. One, the earlier one, returns a character value, a single digit. The other accepts a multi-digit integer and returns an integer value. Either one will work here, since WriteLn will handle either type. But there's no reason to limit ourselves to single-digit values here, so the correct version to use is the one that returns an integer. Here it is:

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

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

As a matter of fact, strictly speaking we should allow for expressions in the data field of the initializer, or at the very least for negative values. For now, let's just allow for negative values by changing the code for `Alloc` as follows:

```
{--------------------------------------------------------------}
{ Allocate Storage for a Variable }

procedure Alloc(N: char);
begin
   if InTable(N) then Abort('Duplicate Variable Name ' + N);
   ST[N] := 'v';
   Write(N, ':', TAB, 'DC ');
   if Look = '=' then begin
      Match('=');
      If Look = '-' then begin
         Write(Look);
         Match('-');
      end;
      WriteLn(GetNum);
      end
   else
      WriteLn('0');
end;
{--------------------------------------------------------------}
```

Now you should be able to initialize variables with negative and/or multi-digit values.

The Symbol Table

There's one problem with the compiler as it stands so far: it doesn't do anything to record a variable when we declare it. So the compiler is perfectly content to allocate storage for several variables with the same name. You can easily verify this with an input like `pvavavabe.`.

Here we've declared the variable `A` three times. As you can see, the compiler will cheerfully accept that, and generate three identical labels. Not good.

Later on, when we start referencing variables, the compiler will also let us reference variables that don't exist. The assembler will catch both of these error conditions, but it doesn't seem friendly at all to pass such errors along to the assembler. The compiler should catch such things at the source language level.

So even though we don't need a symbol table to record data types, we ought to install one just to check for these two conditions. Since at this point we are still restricted to single-character variable names, the symbol table can be trivial. To provide for it, first add the following declaration at the beginning of your program:

```
var ST: array['A'..'Z'] of char;
```

and insert the following function:

```
{--------------------------------------------------------------}
{ Look for Symbol in Table }

function InTable(n: char): Boolean;
begin
   InTable := ST[n] <> ' ';
end;
{--------------------------------------------------------------}
```

We also need to initialize the table to all blanks. The following lines in `Init` will do the job:

```
var i: char;
begin
   for i := 'A' to 'Z' do
      ST[i] := ' ';
   ...
```

Finally, insert the following two lines at the beginning of `Alloc`:

```
if InTable(N) then Abort('Duplicate Variable Name ' + N);
ST[N] := 'v';
```

That should do it. The compiler will now catch duplicate declarations. Later, we can also use `InTable` when generating references to the variables.

Executable Statements

At this point, we can generate a null program that has some data variables declared and possibly initialized. But so far we haven't arranged to generate the first line of executable code.

Believe it or not, though, we almost have a usable language! What's missing is the executable code that must go into the main program. But that code is just assignment statements and control statements ... all stuff we have done before. So it shouldn't take us long to provide for them, as well.

The BNF definition given earlier for the main program included a statement block, which we have so far ignored:

```
<main> ::= BEGIN <block> END
```

For now, we can just consider a block to be a series of assignment statements:

```
<block> ::= (Assignment)*
```

Let's start things off by adding a parser for the block. We'll begin with a stub for the assignment statement:

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

procedure Assignment;
begin
   GetChar;
end;


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

```
{ Parse and Translate a Block of Statements }

procedure Block;
begin
   while Look <> 'e' do
      Assignment;
end;
{--------------------------------------------------------------}
```

Modify procedure `Main` to call `Block` as shown below:

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

procedure Main;
begin
   Match('b');
   Prolog;
   Block;
   Match('e');
   Epilog;
end;
{--------------------------------------------------------------}
```

This version still won't generate any code for the "assignment statements" ... all it does is to eat characters until it sees the `e` for `END`. But it sets the stage for what is to follow.

The next step, of course, is to flesh out the code for an assignment statement. This is something we've done many times before, so I won't belabor it. This time, though, I'd like to deal with the code generation a little differently. Up till now, we've always just inserted the Emits that generate output code in line with the parsing routines. A little unstructured, perhaps, but it seemed the most straightforward approach, and made it easy to see what kind of code would be emitted for each construct.

However, I realize that most of you are using an 80x86 computer, so the 68000 code generated is of little use to you. Several of you have asked me if the CPU-dependent code couldn't be collected into one spot where it would be easier to retarget to another CPU. The answer, of course, is yes.

To accomplish this, insert the following "code generation" routines:

```
{--------------------------------------------------------------}
{ Clear the Primary Register }

procedure Clear;
begin
   EmitLn('CLR D0');
end;


{--------------------------------------------------------------}
{ Negate the Primary Register }

procedure Negate;
begin
   EmitLn('NEG D0');
end;


{--------------------------------------------------------------}
{ Load a Constant Value to Primary Register }

procedure LoadConst(n: integer);
begin
   Emit('MOVE #');
   WriteLn(n, ',D0');
```

```
end;


{---------------------------------------------------------------}
{ Load a Variable to Primary Register }

procedure LoadVar(Name: char);
begin
   if not InTable(Name) then Undefined(Name);
   EmitLn('MOVE ' + Name + '(PC),D0');
end;


{---------------------------------------------------------------}
{ Push Primary onto Stack }

procedure Push;
begin
   EmitLn('MOVE D0,-(SP)');
end;


{---------------------------------------------------------------}
{ Add Top of Stack to Primary }

procedure PopAdd;
begin
   EmitLn('ADD (SP)+,D0');
end;


{---------------------------------------------------------------}
{ Subtract Primary from Top of Stack }

procedure PopSub;
begin
   EmitLn('SUB (SP)+,D0');
   EmitLn('NEG D0');
end;


{---------------------------------------------------------------}
{ Multiply Top of Stack by Primary }

procedure PopMul;
begin
   EmitLn('MULS (SP)+,D0');
end;


{---------------------------------------------------------------}
{ Divide Top of Stack by Primary }

procedure PopDiv;
begin
   EmitLn('MOVE (SP)+,D7');
   EmitLn('EXT.L D7');
   EmitLn('DIVS D0,D7');
   EmitLn('MOVE D7,D0');
end;
```

```
{--------------------------------------------------------------}
{ Store Primary to Variable }

procedure Store(Name: char);
begin
   if not InTable(Name) then Undefined(Name);
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)')
end;
{--------------------------------------------------------------}
```

The nice part of this approach, of course, is that we can retarget the compiler to a new CPU simply by rewriting these "code generator" procedures. In addition, we will find later that we can improve the code quality by tweaking these routines a bit, without having to modify the compiler proper.

Note that both `LoadVar` and `Store` check the symbol table to make sure that the variable is defined. The error handler `Undefined` simply calls Abort:

```
{--------------------------------------------------------------}
{ Report an Undefined Identifier }

procedure Undefined(n: string);
begin
   Abort('Undefined Identifier ' + n);
end;
{--------------------------------------------------------------}
```

OK, we are now finally ready to begin processing executable code. We'll do that by replacing the stub version of procedure `Assignment`.

We've been down this road many times before, so this should all be familiar to you. In fact, except for the changes associated with the code generation, we could just copy the procedures from Part VII. Since we are making some changes, I won't just copy them, but we will go a little faster than usual.

The BNF for the assignment statement is:

```
<assignment> ::= <ident> = <expression>

<expression> ::= <first term> ( <addop> <term> )*

<first term> ::= <first factor> <rest>

<term> ::= <factor> <rest>

<rest> ::= ( <mulop> <factor> )*

<first factor> ::= [ <addop> ] <factor>

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

This version of the BNF is also a bit different than we've used before ... yet another "variation on the theme of an expression." This particular version has what I consider to be the best treatment of the unary minus. As you'll see later, it lets us handle negative constant values efficiently. It's worth mentioning here that we have often seen the advantages of "tweaking" the BNF as we go, to help make the language easy to parse. What you're looking at here is a bit different: we've tweaked the BNF to make the *code generation* more efficient! That's a first for this series.

Anyhow, the following code implements the BNF:

```
{--------------------------------------------------------------}
{ 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
      LoadVar(GetName)
   else
      LoadConst(GetNum);
end;


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

procedure NegFactor;
begin
   Match('-');
   if IsDigit(Look) then
      LoadConst(-GetNum)
   else begin
      Factor;
      Negate;
   end;
end;


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

procedure FirstFactor;
begin
   case Look of
     '+': begin
             Match('+');
             Factor;
          end;
     '-': NegFactor;
   else  Factor;
   end;
end;


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

procedure Multiply;
begin
   Match('*');
   Factor;
   PopMul;
end;


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

procedure Divide;
begin
```

```
   Match('/');
   Factor;
   PopDiv;
end;


{---------------------------------------------------------------}
{ Common Code Used by Term and FirstTerm }

procedure Term1;
begin
   while IsMulop(Look) do begin
      Push;
      case Look of
       '*': Multiply;
       '/': Divide;
      end;
   end;
end;


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

procedure Term;
begin
   Factor;
   Term1;
end;


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

procedure FirstTerm;
begin
   FirstFactor;
   Term1;
end;


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

procedure Add;
begin
   Match('+');
   Term;
   PopAdd;
end;


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

procedure Subtract;
begin
   Match('-');
   Term;
   PopSub;
end;
```

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

procedure Expression;
begin
   FirstTerm;
   while IsAddop(Look) do begin
      Push;
      case Look of
       '+': Add;
       '-': Subtract;
      end;
   end;
end;


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

procedure Assignment;
var Name: char;
begin
   Name := GetName;
   Match('=');
   Expression;
   Store(Name);
end;
{--------------------------------------------------------------}
```

OK, if you've got all this code inserted, then compile it and check it out. You should be seeing reasonable-looking code, representing a complete program that will assemble and execute. We have a compiler!

Booleans

The next step should also be familiar to you. We must add Boolean expressions and relational operations. Again, since we've already dealt with them more than once, I won't elaborate much on them, except where they are different from what we've done before. Again, we won't just copy from other files because I've changed a few things just a bit. Most of the changes just involve encapsulating the machine-dependent parts as we did for the arithmetic operations. I've also modified procedure `NotFactor` somewhat, to parallel the structure of `FirstFactor`. Finally, I corrected an error in the object code for the relational operators: The `Scc` instruction I used only sets the low 8 bits of `D0`. We want all 16 bits set for a logical true, so I've added an instruction to sign-extend the low byte.

To begin, we're going to need some more recognizers:

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

function IsOrop(c: char): boolean;
begin
   IsOrop := c in ['|', '~'];
end;


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

function IsRelop(c: char): boolean;
begin
```

```
   IsRelop := c in ['=', '#', '<', '>'];
end;
{---------------------------------------------------------------}
```

Also, we're going to need some more code generation routines:

```
{---------------------------------------------------------------}
{ Complement the Primary Register }

procedure NotIt;
begin
   EmitLn('NOT D0');
end;
{---------------------------------------------------------------}
.
.
.
{---------------------------------------------------------------}
{ AND Top of Stack with Primary }

procedure PopAnd;
begin
   EmitLn('AND (SP)+,D0');
end;


{---------------------------------------------------------------}
{ OR Top of Stack with Primary }

procedure PopOr;
begin
   EmitLn('OR (SP)+,D0');
end;


{---------------------------------------------------------------}
{ XOR Top of Stack with Primary }

procedure PopXor;
begin
   EmitLn('EOR (SP)+,D0');
end;


{---------------------------------------------------------------}
{ Compare Top of Stack with Primary }

procedure PopCompare;
begin
   EmitLn('CMP (SP)+,D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was = }

procedure SetEqual;
begin
   EmitLn('SEQ D0');
   EmitLn('EXT D0');
end;
```

```
{--------------------------------------------------------------}
{ Set D0 If Compare was != }

procedure SetNEqual;
begin
   EmitLn('SNE D0');
   EmitLn('EXT D0');
end;


{--------------------------------------------------------------}
{ Set D0 If Compare was > }

procedure SetGreater;
begin
   EmitLn('SLT D0');
   EmitLn('EXT D0');
end;


{--------------------------------------------------------------}
{ Set D0 If Compare was < }

procedure SetLess;
begin
   EmitLn('SGT D0');
   EmitLn('EXT D0');
end;
{--------------------------------------------------------------}
```

All of this gives us the tools we need. The BNF for the Boolean expressions is:

```
<bool-expr> ::= <bool-term> ( <orop> <bool-term> )*

<bool-term> ::= <not-factor> ( <andop> <not-factor> )*

<not-factor> ::= [ '!' ] <relation>

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

Sharp-eyed readers might note that this syntax does not include the non-terminal "bool-factor" used in earlier versions. It was needed then because I also allowed for the Boolean constants `TRUE` and `FALSE`. But remember that in TINY there is no distinction made between Boolean and arithmetic types ... they can be freely intermixed. So there is really no need for these predefined values ... we can just use -1 and 0, respectively.

In C terminology, we could always use the defines:

```
#define TRUE -1
#define FALSE 0
```

(That is, if TINY had a preprocessor.) Later on, when we allow for declarations of constants, these two values will be predefined by the language.

The reason that I'm harping on this is that I've already tried the alternative, which is to include `TRUE` and `FALSE` as keywords. The problem with that approach is that it then requires lexical scanning for *every* variable name in every expression. If you'll recall, I pointed out in Installment VII that this slows the compiler down considerably. As long as keywords can't be in expressions, we need to do the scanning only at the beginning of every new statement ... quite an improvement. So using the syntax above not only simplifies the parsing, but speeds up the scanning as well.

OK, given that we're all satisfied with the syntax above, the corresponding code is shown below:

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

procedure Equals;
begin
   Match('=');
   Expression;
   PopCompare;
   SetEqual;
end;


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

procedure NotEquals;
begin
   Match('#');
   Expression;
   PopCompare;
   SetNEqual;
end;


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

procedure Less;
begin
   Match('<');
   Expression;
   PopCompare;
   SetLess;
end;


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

procedure Greater;
begin
   Match('>');
   Expression;
   PopCompare;
   SetGreater;
end;


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


procedure Relation;
begin
   Expression;
   if IsRelop(Look) then begin
      Push;
      case Look of
       '=': Equals;
       '#': NotEquals;
       '<': Less;
```

```
       '>': Greater;
      end;
   end;
end;


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

procedure NotFactor;
begin
   if Look = '!' then begin
      Match('!');
      Relation;
      NotIt;
      end
   else
      Relation;
end;


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

procedure BoolTerm;
begin
   NotFactor;
   while Look = '&' do begin
      Push;
      Match('&');
      NotFactor;
      PopAnd;
   end;
end;


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

procedure BoolOr;
begin
   Match('|');
   BoolTerm;
   PopOr;
end;


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

procedure BoolXor;
begin
   Match('~');
   BoolTerm;
   PopXor;
end;


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

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

To tie it all together, don't forget to change the references to `Expression` in procedures `Factor` and `Assignment` so that they call `BoolExpression` instead.

OK, if you've got all that typed in, compile it and give it a whirl. First, make sure you can still parse an ordinary arithmetic expression. Then, try a Boolean one. Finally, make sure that you can assign the results of relations. Try, for example, `pvx,y,zbx=z>ye.`, which stands for:

```
PROGRAM
VAR X,Y,Z
BEGIN
X = Z > Y
END.
```

See how this assigns a Boolean value to `X`?

Control Structures

We're almost home. With Boolean expressions in place, it's a simple matter to add control structures. For TINY, we'll only allow two kinds of them, the `IF` and the `WHILE`:

```
<if> ::= IF <bool-expression> <block> [ ELSE <block>] ENDIF

<while> ::= WHILE <bool-expression> <block> ENDWHILE
```

Once again, let me spell out the decisions implicit in this syntax, which departs strongly from that of C or Pascal. In both of those languages, the "body" of an `IF` or `WHILE` is regarded as a single statement. If you intend to use a block of more than one statement, you have to build a compound statement using `BEGIN-END` (in Pascal) or `{}` (in C). In TINY (and KISS) there is no such thing as a compound statement ... single or multiple they're all just blocks to these languages.

In KISS, all the control structures will have explicit and unique keywords bracketing the statement block, so there can be no confusion as to where things begin and end. This is the modern approach, used in such respected languages as Ada and Modula 2, and it completely eliminates the problem of the "dangling else."

Note that I could have chosen to use the same keyword `END` to end all the constructs, as is done in Pascal. (The closing `}` in C serves the same purpose.) But this has always led to confusion, which is why Pascal programmers tend to write things like `end { loop }` or `end { if }`.

As I explained in Part V, using unique terminal keywords does increase the size of the keyword list and therefore slows down the scanning, but in this case it seems a small price to pay for the added insurance. Better to find the errors at compile time rather than run time.

One last thought: The two constructs above each have the non-terminals `<bool-expression>` and `<block>` juxtaposed with no separating keyword. In Pascal we would expect the keywords THEN and DO in these locations.

I have no problem with leaving out these keywords, and the parser has no trouble either, *on condition* that we make no errors in the bool-expression part. On the other hand, if we were to include these extra keywords we would get yet one more level of insurance at very little cost, and I have no problem with that, either. Use your best judgment as to which way to go.

OK, with that bit of explanation let's proceed. As usual, we're going to need some new code generation routines. These generate the code for conditional and unconditional branches:

```
{---------------------------------------------------------------}
{ Branch Unconditional  }

procedure Branch(L: string);
begin
   EmitLn('BRA ' + L);
end;


{---------------------------------------------------------------}
{ Branch False }

procedure BranchFalse(L: string);
begin
   EmitLn('TST D0');
   EmitLn('BEQ ' + L);
end;
{---------------------------------------------------------------}
```

Except for the encapsulation of the code generation, the code to parse the control constructs is the same as you've seen before:

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

procedure Block; Forward;


procedure DoIf;
var L1, L2: string;
begin
   Match('i');
   BoolExpression;
   L1 := NewLabel;
   L2 := L1;
   BranchFalse(L1);
   Block;
   if Look = 'l' then begin
      Match('l');
      L2 := NewLabel;
      Branch(L2);
      PostLabel(L1);
      Block;
   end;
   PostLabel(L2);
   Match('e');
end;


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

procedure DoWhile;
var L1, L2: string;
begin
   Match('w');
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   BoolExpression;
   BranchFalse(L2);
```

```
   Block;
   Match('e');
   Branch(L1);
   PostLabel(L2);
end;
{--------------------------------------------------------------}
```

To tie everything together, we need only modify procedure `Block` to recognize the "keywords" for the `IF` and `WHILE`. As usual, we expand the definition of a block like so:

```
<block> ::= ( <statement> )*

<statement> ::= <if> | <while> | <assignment>
```

The corresponding code is:

```
{--------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

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

OK, add the routines I've given, compile and test them. You should be able to parse the single-character versions of any of the control constructs. It's looking pretty good!

As a matter of fact, except for the single-character limitation we've got a virtually complete version of TINY. I call it, with tongue planted firmly in cheek, TINY Version 0.1.

Lexical Scanning

Of course, you know what's next: We have to convert the program so that it can deal with multi-character keywords, newlines, and whitespace. We have just gone through all that in Part VII. We'll use the distributed scanner technique that I showed you in that installment. The actual implementation is a little different because the way I'm handling newlines is different.

To begin with, let's simply allow for whitespace. This involves only adding calls to `SkipWhite` at the end of the three routines, `GetName`, `GetNum`, and `Match`. A call to `SkipWhite` in `Init` primes the pump in case there are leading spaces.

Next, we need to deal with newlines. This is really a two-step process, since the treatment of the newlines with single-character tokens is different from that for multi-character ones. We can eliminate some work by doing both steps at once, but I feel safer taking things one step at a time.

Insert the new procedure:

```
{--------------------------------------------------------------}
{ Skip Over an End-of-Line }

procedure NewLine;
begin
   while Look = CR do begin
      GetChar;
      if Look = LF then GetChar;
      SkipWhite;
   end;
```

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

Note that we have seen this procedure before in the form of Procedure `Fin`. I've changed the name since this new one seems more descriptive of the actual function. I've also changed the code to allow for multiple newlines and lines with nothing but white space.

The next step is to insert calls to `NewLine` wherever we decide a newline is permissible. As I've pointed out before, this can be very different in different languages. In TINY, I've decided to allow them virtually anywhere. This means that we need calls to `NewLine` at the *beginning* (not the end, as with `SkipWhite`) of the procedures `GetName`, `GetNum`, and `Match`.

For procedures that have while loops, such as `TopDecl`, we need a call to `NewLine` at the beginning of the procedure *and* at the bottom of each loop. That way, we can be assured that `NewLine` has just been called at the beginning of each pass through the loop.

If you've got all this done, try the program out and verify that it will indeed handle white space and newlines.

If it does, then we're ready to deal with multi-character tokens and keywords. To begin, add the additional declarations (copied almost verbatim from Part VII):

```
{--------------------------------------------------------------}
{ Type Declarations }

type Symbol = string[8];

     SymTab = array[1..1000] of Symbol;

     TabPtr = ^SymTab;


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

var Look : char;             { Lookahead Character }
    Token: char;             { Encoded Token       }
    Value: string[16];       { Unencoded Token     }

    ST: Array['A'..'Z'] of char;

{--------------------------------------------------------------}
{ Definition of Keywords and Token Types }

const NKW =   9;
      NKW1 = 10;

const KWlist: array[1..NKW] of Symbol =
              ('IF', 'ELSE', 'ENDIF', 'WHILE', 'ENDWHILE',
               'VAR', 'BEGIN', 'END', 'PROGRAM');

const KWcode: string[NKW1] = 'xilewevbep';
{--------------------------------------------------------------}
```

Next, add the three procedures, also from Part VII:

```
{--------------------------------------------------------------}
{ Table Lookup }

function Lookup(T: TabPtr; s: string; n: integer): integer;
var i: integer;
    found: Boolean;
begin
   found := false;
   i := n;
   while (i > 0) and not found do
```

```
      if s = T^[i] then
         found := true
      else
         dec(i);
   Lookup := i;
end;
{--------------------------------------------------------------}
.
.
{--------------------------------------------------------------}
{ Get an Identifier and Scan it for Keywords }

procedure Scan;
begin
   GetName;
   Token := KWcode[Lookup(Addr(KWlist), Value, NKW) + 1];
end;
{--------------------------------------------------------------}
.
.
{--------------------------------------------------------------}
{ Match a Specific Input String }

procedure MatchString(x: string);
begin
   if Value <> x then Expected('''' + x + '''');
end;
{--------------------------------------------------------------}
```

Now, we have to make a fairly large number of subtle changes to the remaining procedures. First, we must change the function `GetName` to a procedure, again as we did in Part VII:

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

procedure GetName;
begin
   NewLine;
   if not IsAlpha(Look) then Expected('Name');
   Value := '';
   while IsAlNum(Look) do begin
      Value := Value + UpCase(Look);
      GetChar;
   end;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

Note that this procedure leaves its result in the global string `Value`.

Next, we have to change every reference to `GetName` to reflect its new form. These occur in `Factor`, `Assignment`, and `Decl`:

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

procedure BoolExpression; Forward;

procedure Factor;
begin
   if Look = '(' then begin
      Match('(');
```

```
      BoolExpression;
      Match(')');
      end
   else if IsAlpha(Look) then begin
      GetName;
      LoadVar(Value[1]);
      end
   else
      LoadConst(GetNum);
end;
{---------------------------------------------------------------}
.
.
{---------------------------------------------------------------}
{ Parse and Translate an Assignment Statement }

procedure Assignment;
var Name: char;
begin
   Name := Value[1];
   Match('=');
   BoolExpression;
   Store(Name);
end;
{---------------------------------------------------------------}
.
.
{---------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
begin
   GetName;
   Alloc(Value[1]);
   while Look = ',' do begin
      Match(',');
      GetName;
      Alloc(Value[1]);
   end;
end;
{---------------------------------------------------------------}
```

(Note that we're still only allowing single-character variable names, so we take the easy way out here and simply use the first character of the string.)

Finally, we must make the changes to use `Token` instead of `Look` as the test character and to call `Scan` at the appropriate places. Mostly, this involves deleting calls to `Match`, occasionally replacing calls to `Match` by calls to `MatchString`, and replacing calls to `NewLine` by calls to `Scan`. Here are the affected routines:

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

procedure Block; Forward;


procedure DoIf;
var L1, L2: string;
begin
   BoolExpression;
   L1 := NewLabel;
   L2 := L1;
   BranchFalse(L1);
```

```
   Block;
   if Token = 'l' then begin
      L2 := NewLabel;
      Branch(L2);
      PostLabel(L1);
      Block;
   end;
   PostLabel(L2);
   MatchString('ENDIF');
end;


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

procedure DoWhile;
var L1, L2: string;
begin
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   BoolExpression;
   BranchFalse(L2);
   Block;
   MatchString('ENDWHILE');
   Branch(L1);
   PostLabel(L2);
end;


{---------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure Block;
begin
   Scan;
   while not(Token in ['e', 'l']) do begin
      case Token of
       'i': DoIf;
       'w': DoWhile;
      else Assignment;
      end;
      Scan;
   end;
end;


{---------------------------------------------------------------}
{ Parse and Translate Global Declarations }

procedure TopDecls;
begin
   Scan;
   while Token <> 'b' do begin
      case Token of
        'v': Decl;
      else Abort('Unrecognized Keyword ' + Value);
      end;
      Scan;
   end;
end;
```

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

procedure Main;
begin
   MatchString('BEGIN');
   Prolog;
   Block;
   MatchString('END');
   Epilog;
end;

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

procedure Prog;
begin
   MatchString('PROGRAM');
   Header;
   TopDecls;
   Main;
   Match('.');
end;


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

procedure Init;
var i: char;
begin
   for i := 'A' to 'Z' do
      ST[i] := ' ';
   GetChar;
   Scan;
end;
{--------------------------------------------------------------}
```

That should do it. If all the changes got in correctly, you should now be parsing programs that look like programs. (If you didn't make it through all the changes, don't despair. A complete listing of the final form is given later.)

Did it work? If so, then we're just about home. In fact, with a few minor exceptions we've already got a compiler that's usable. There are still a few areas that need improvement.

Multi-Character Variable Names

One of those is the restriction that we still have, requiring single-character variable names. Now that we can handle multi-character keywords, this one begins to look very much like an arbitrary and unnecessary limitation. And indeed it is. Basically, its only virtue is that it permits a trivially simple implementation of the symbol table. But that's just a convenience to the compiler writers, and needs to be eliminated.

We've done this step before. This time, as usual, I'm doing it a little differently. I think the approach used here keeps things just about as simple as possible.

The natural way to implement a symbol table in Pascal is by declaring a record type, and making the symbol table an array of such records. Here, though, we don't really need a type field yet (there is only one kind of entry allowed so far), so we only need an array of symbols. This has the advantage that we can use the existing procedure `Lookup` to search the symbol table as well as the keyword list. As it turns out, even when we need more fields we can still use the same approach, simply by storing the other fields in separate arrays.

OK, here are the changes that need to be made. First, add the new typed constant:

```
NEntry: integer = 0;
```

Then change the definition of the symbol table as follows:

```
const MaxEntry = 100;

var ST   : array[1..MaxEntry] of Symbol;
```

(Note that `ST` is *not* declared as a `SymTab`. That declaration is a phony one to get `Lookup` to work. A `SymTab` would take up too much RAM space, and so one is never actually allocated.)

Next, we need to replace `InTable`:

```
{-------------------------------------------------------------}
{ Look for Symbol in Table }

function InTable(n: Symbol): Boolean;
begin
   InTable := Lookup(@ST, n, MaxEntry) <> 0;
end;
{-------------------------------------------------------------}
```

We also need a new procedure, `AddEntry`, that adds a new entry to the table:

```
{-------------------------------------------------------------}
{ Add a New Entry to Symbol Table }

procedure AddEntry(N: Symbol; T: char);
begin
   if InTable(N) then Abort('Duplicate Identifier ' + N);
   if NEntry = MaxEntry then Abort('Symbol Table Full');
   Inc(NEntry);
   ST[NEntry] := N;
   SType[NEntry] := T;
end;
{-------------------------------------------------------------}
```

This procedure is called by `Alloc`:

```
{-------------------------------------------------------------}
{ Allocate Storage for a Variable }

procedure Alloc(N: Symbol);
begin
   if InTable(N) then Abort('Duplicate Variable Name ' + N);
   AddEntry(N, 'v');
.
.
.
{-------------------------------------------------------------}
```

Finally, we must change all the routines that currently treat the variable name as a single character. These include `LoadVar` and `Store` (just change the type from `char` to `string`), and `Factor`, `Assignment`, and `Decl` (just change `Value[1]` to `Value`).

One last thing: change procedure `Init` to clear the array as shown:

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

procedure Init;
var i: integer;
begin
```

```
   for i := 1 to MaxEntry do begin
      ST[i] := '';
      SType[i] := ' ';
   end;
   GetChar;
   Scan;
end;
{---------------------------------------------------------------}
```

That should do it. Try it out and verify that you can, indeed, use multi-character variable names.

More Relops

We still have one remaining single-character restriction: the one on relops. Some of the relops are indeed single characters, but others require two. These are `<=` and `>=`. I also prefer the Pascal `<>` for "not equals," instead of `#`.

If you'll recall, in Part VII I pointed out that the conventional way to deal with relops is to include them in the list of keywords, and let the lexical scanner find them. But, again, this requires scanning throughout the expression parsing process, whereas so far we've been able to limit the use of the scanner to the beginning of a statement.

I mentioned then that we can still get away with this, since the multi-character relops are so few and so limited in their usage. It's easy to just treat them as special cases and handle them in an ad hoc manner.

The changes required affect only the code generation routines and procedures `Relation` and friends. First, we're going to need two more code generation routines:

```
{---------------------------------------------------------------}
{ Set D0 If Compare was <= }

procedure SetLessOrEqual;
begin
   EmitLn('SGE D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was >= }

procedure SetGreaterOrEqual;
begin
   EmitLn('SLE D0');
   EmitLn('EXT D0');
end;
{---------------------------------------------------------------}
```

Then, modify the relation parsing routines as shown below:

```
{---------------------------------------------------------------}
{ Recognize and Translate a Relational "Less Than or Equal" }

procedure LessOrEqual;
begin
   Match('=');
   Expression;
   PopCompare;
   SetLessOrEqual;
end;


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

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

procedure NotEqual;
begin
   Match('>');
   Expression;
   PopCompare;
   SetNEqual;
end;


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

procedure Less;
begin
   Match('<');
   case Look of
     '=': LessOrEqual;
     '>': NotEqual;
   else begin
           Expression;
           PopCompare;
           SetLess;
        end;
   end;
end;


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

procedure Greater;
begin
   Match('>');
   if Look = '=' then begin
      Match('=');
      Expression;
      PopCompare;
      SetGreaterOrEqual;
      end
   else begin
      Expression;
      PopCompare;
      SetGreater;
   end;
end;
{---------------------------------------------------------------}
```

That's all it takes. Now you can process all the relops. Try it.

Input/Output

We now have a complete, working language, except for one minor embarrassment: we have no way to get data in or out. We need some I/O.

Now, the convention these days, established in C and continued in Ada and Modula 2, is to leave I/O statements out of the language itself, and just include them in the subroutine library. That would be fine, except that so far we have no provision for subroutines. Anyhow, with this approach you run into the problem of variable-length argument lists. In Pascal, the I/O statements are built into the language because they are the only ones for which the argument list can have a variable number of entries. In C, we settle for kludges like

scanf and printf, and must pass the argument count to the called procedure. In Ada and Modula 2 we must use the awkward (and *slow*!) approach of a separate call for each argument.

So I think I prefer the Pascal approach of building the I/O in, even though we don't need to.

As usual, for this we need some more code generation routines. These turn out to be the easiest of all, because all we do is to call library procedures to do the work:

```
{-------------------------------------------------------------}
{ Read Variable to Primary Register }

procedure ReadVar;
begin
   EmitLn('BSR READ');
   Store(Value);
end;


{-------------------------------------------------------------}
{ Write Variable from Primary Register }

procedure WriteVar;
begin
   EmitLn('BSR WRITE');
end;
{-------------------------------------------------------------}
```

The idea is that `READ` loads the value from input to the `D0`, and `WRITE` outputs it from there.

These two procedures represent our first encounter with a need for library procedures ... the components of a Run Time Library (RTL). Of course, someone (namely us) has to write these routines, but they're not part of the compiler itself. I won't even bother showing the routines here, since these are obviously very much OS-dependent. I *will* simply say that for SK*DOS, they are particularly simple ... almost trivial. One reason I won't show them here is that you can add all kinds of fanciness to the things, for example by prompting in `READ` for the inputs, and by giving the user a chance to reenter a bad input.

But that is really separate from compiler design, so for now I'll just assume that a library call `TINYLIB.LIB` exists. Since we now need it loaded, we need to add a statement to include it in procedure `Header`:

```
{-------------------------------------------------------------}
{ Write Header Info }

procedure Header;
begin

   WriteLn('WARMST', TAB, 'EQU $A01E');
   EmitLn('LIB TINYLIB');
end;
{-------------------------------------------------------------}
```

That takes care of that part. Now, we also need to recognize the read and write commands. We can do this by adding two more keywords to our list:

```
{-------------------------------------------------------------}
{ Definition of Keywords and Token Types }

const NKW =   11;
      NKW1 = 12;

const KWlist: array[1..NKW] of Symbol =
              ('IF', 'ELSE', 'ENDIF', 'WHILE', 'ENDWHILE',
               'READ',    'WRITE',    'VAR',    'BEGIN',   'END',
'PROGRAM');

const KWcode: string[NKW1] = 'xileweRWvbep';
```

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

(Note how I'm using upper case codes here to avoid conflict with the `w` of WHILE.)

Next, we need procedures for processing the read/write statement and its argument list:

```
{---------------------------------------------------------------}
{ Process a Read Statement }
procedure DoRead;
begin
   Match('(');
   GetName;
   ReadVar;
   while Look = ',' do begin
      Match(',');
      GetName;
      ReadVar;
   end;
   Match(')');
end;


{---------------------------------------------------------------}
{ Process a Write Statement }

procedure DoWrite;
begin
   Match('(');
   Expression;
   WriteVar;
   while Look = ',' do begin
      Match(',');
      Expression;
      WriteVar;
   end;
   Match(')');
end;
{---------------------------------------------------------------}
```

Finally, we must expand procedure `Block` to handle the new statement types:

```
{---------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure Block;
begin
   Scan;
   while not(Token in ['e', 'l']) do begin
      case Token of
       'i': DoIf;
       'w': DoWhile;
       'R': DoRead;
       'W': DoWrite;
      else Assignment;
      end;
      Scan;
   end;
end;
{---------------------------------------------------------------}
```

That's all there is to it. *Now* we have a language!

Conclusion

At this point we have TINY completely defined. It's not much ... actually a toy compiler. TINY has only one data type and no subroutines ... but it's a complete, usable language. While you're not likely to be able to write another compiler in it, or do anything else very seriously, you could write programs to read some input, perform calculations, and output the results. Not too bad for a toy.

Most importantly, we have a firm base upon which to build further extensions. I know you'll be glad to hear this: this is the last time I'll start over in building a parser ... from now on I intend to just add features to TINY until it becomes KISS. Oh, there'll be other times we will need to try things out with new copies of the Cradle, but once we've found out how to do those things they'll be incorporated into TINY.

What will those features be? Well, for starters we need subroutines and functions. Then we need to be able to handle different types, including arrays, strings, and other structures. Then we need to deal with the idea of pointers. All this will be upcoming in future installments.

See you then.

For references purposes, the complete listing of TINY Version 1.0 is shown below:

```
{--------------------------------------------------------------}
program Tiny10;

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

const TAB = ^I;
      CR  = ^M;
      LF  = ^J;

      LCount: integer = 0;
      NEntry: integer = 0;


{--------------------------------------------------------------}
{ Type Declarations }

type Symbol = string[8];

     SymTab = array[1..1000] of Symbol;
     TabPtr = ^SymTab;


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

var Look : char;             { Lookahead Character }
    Token: char;             { Encoded Token       }
    Value: string[16];       { Unencoded Token     }


const MaxEntry = 100;

var ST   : array[1..MaxEntry] of Symbol;
    SType: array[1..MaxEntry] of char;


{--------------------------------------------------------------}
{ Definition of Keywords and Token Types }

const NKW =   11;
      NKW1 = 12;

const KWlist: array[1..NKW] of Symbol =
              ('IF', 'ELSE', 'ENDIF', 'WHILE', 'ENDWHILE',
```

```
               'READ',    'WRITE',    'VAR',    'BEGIN',   'END',
'PROGRAM');

const KWcode: string[NKW1] = 'xileweRWvbep';


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

{--------------------------------------------------------------}
{ Report an Undefined Identifier }

procedure Undefined(n: string);
begin
   Abort('Undefined Identifier ' + n);
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 Character }

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 a Mulop }

function IsMulop(c: char): boolean;
begin
   IsMulop := c in ['*', '/'];
end;


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

function IsOrop(c: char): boolean;
begin
   IsOrop := c in ['|', '~'];
end;


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

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


{---------------------------------------------------------------}
{ Skip Over an End-of-Line }

procedure NewLine;
begin
   while Look = CR do begin
      GetChar;
      if Look = LF then GetChar;
      SkipWhite;
   end;
end;


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

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


{---------------------------------------------------------------}
{ Table Lookup }

function Lookup(T: TabPtr; s: string; n: integer): integer;
var i: integer;
    found: Boolean;
begin
   found := false;
   i := n;
   while (i > 0) and not found do
      if s = T^[i] then
         found := true
      else
         dec(i);
   Lookup := i;
end;


{---------------------------------------------------------------}
{ Locate a Symbol in Table }
{ Returns the index of the entry.  Zero if not present. }

function Locate(N: Symbol): integer;
begin
   Locate := Lookup(@ST, n, MaxEntry);
```

```
end;


{--------------------------------------------------------------}
{ Look for Symbol in Table }

function InTable(n: Symbol): Boolean;
begin
   InTable := Lookup(@ST, n, MaxEntry) <> 0;
end;


{--------------------------------------------------------------}
{ Add a New Entry to Symbol Table }

procedure AddEntry(N: Symbol; T: char);
begin
   if InTable(N) then Abort('Duplicate Identifier ' + N);
   if NEntry = MaxEntry then Abort('Symbol Table Full');
   Inc(NEntry);
   ST[NEntry] := N;
   SType[NEntry] := T;
end;


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

procedure GetName;
begin
   NewLine;
   if not IsAlpha(Look) then Expected('Name');
   Value := '';
   while IsAlNum(Look) do begin
      Value := Value + UpCase(Look);
      GetChar;
   end;
   SkipWhite;
end;


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

function GetNum: integer;
var Val: integer;
begin
   NewLine;
   if not IsDigit(Look) then Expected('Integer');
   Val := 0;
   while IsDigit(Look) do begin
      Val := 10 * Val + Ord(Look) - Ord('0');
      GetChar;
   end;
   GetNum := Val;
   SkipWhite;
end;


{--------------------------------------------------------------}
{ Get an Identifier and Scan it for Keywords }
```

```
procedure Scan;
begin
   GetName;
   Token := KWcode[Lookup(Addr(KWlist), Value, NKW) + 1];
end;


{---------------------------------------------------------------}
{ Match a Specific Input String }

procedure MatchString(x: string);
begin
   if Value <> x then Expected('''' + x + '''');
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;


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

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


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

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


{---------------------------------------------------------------}
{ Clear the Primary Register }

procedure Clear;
```

```
begin
   EmitLn('CLR D0');
end;


{---------------------------------------------------------------}
{ Negate the Primary Register }

procedure Negate;
begin
   EmitLn('NEG D0');
end;


{---------------------------------------------------------------}
{ Complement the Primary Register }

procedure NotIt;
begin
   EmitLn('NOT D0');
end;


{---------------------------------------------------------------}
{ Load a Constant Value to Primary Register }

procedure LoadConst(n: integer);
begin
   Emit('MOVE #');
   WriteLn(n, ',D0');
end;


{---------------------------------------------------------------}
{ Load a Variable to Primary Register }

procedure LoadVar(Name: string);
begin
   if not InTable(Name) then Undefined(Name);
   EmitLn('MOVE ' + Name + '(PC),D0');
end;


{---------------------------------------------------------------}
{ Push Primary onto Stack }

procedure Push;
begin
   EmitLn('MOVE D0,-(SP)');
end;


{---------------------------------------------------------------}
{ Add Top of Stack to Primary }

procedure PopAdd;
begin
   EmitLn('ADD (SP)+,D0');
end;
```

```
{-----------------------------------------------------------------}
{ Subtract Primary from Top of Stack }

procedure PopSub;
begin
   EmitLn('SUB (SP)+,D0');
   EmitLn('NEG D0');
end;


{-----------------------------------------------------------------}
{ Multiply Top of Stack by Primary }

procedure PopMul;
begin
   EmitLn('MULS (SP)+,D0');
end;


{-----------------------------------------------------------------}
{ Divide Top of Stack by Primary }

procedure PopDiv;
begin
   EmitLn('MOVE (SP)+,D7');
   EmitLn('EXT.L D7');
   EmitLn('DIVS D0,D7');
   EmitLn('MOVE D7,D0');
end;


{-----------------------------------------------------------------}
{ AND Top of Stack with Primary }

procedure PopAnd;
begin
   EmitLn('AND (SP)+,D0');
end;


{-----------------------------------------------------------------}
{ OR Top of Stack with Primary }

procedure PopOr;
begin
   EmitLn('OR (SP)+,D0');
end;


{-----------------------------------------------------------------}
{ XOR Top of Stack with Primary }

procedure PopXor;
begin
   EmitLn('EOR (SP)+,D0');
end;


{-----------------------------------------------------------------}
{ Compare Top of Stack with Primary }
```

```
procedure PopCompare;
begin
   EmitLn('CMP (SP)+,D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was = }

procedure SetEqual;
begin
   EmitLn('SEQ D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was != }

procedure SetNEqual;
begin
   EmitLn('SNE D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was > }

procedure SetGreater;
begin
   EmitLn('SLT D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was < }

procedure SetLess;
begin
   EmitLn('SGT D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was <= }

procedure SetLessOrEqual;
begin
   EmitLn('SGE D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was >= }

procedure SetGreaterOrEqual;
begin
```

```
   EmitLn('SLE D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Store Primary to Variable }

procedure Store(Name: string);
begin
   if not InTable(Name) then Undefined(Name);
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)')
end;


{---------------------------------------------------------------}
{ Branch Unconditional  }

procedure Branch(L: string);
begin
   EmitLn('BRA ' + L);
end;


{---------------------------------------------------------------}
{ Branch False }

procedure BranchFalse(L: string);
begin
   EmitLn('TST D0');
   EmitLn('BEQ ' + L);
end;


{---------------------------------------------------------------}
{ Read Variable to Primary Register }

procedure ReadVar;
begin
   EmitLn('BSR READ');
   Store(Value[1]);
end;


{ Write Variable from Primary Register }

procedure WriteVar;
begin
   EmitLn('BSR WRITE');
end;


{---------------------------------------------------------------}
{ Write Header Info }

procedure Header;
begin
   WriteLn('WARMST', TAB, 'EQU $A01E');
end;
```

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

procedure Prolog;
begin
   PostLabel('MAIN');
end;


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

procedure Epilog;
begin
   EmitLn('DC WARMST');
   EmitLn('END MAIN');
end;


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

procedure BoolExpression; Forward;

procedure Factor;
begin
   if Look = '(' then begin
      Match('(');
      BoolExpression;
      Match(')');
      end
   else if IsAlpha(Look) then begin
      GetName;
      LoadVar(Value);
      end
   else
      LoadConst(GetNum);
end;


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

procedure NegFactor;
begin
   Match('-');
   if IsDigit(Look) then
      LoadConst(-GetNum)
   else begin
      Factor;
      Negate;
   end;
end;


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

procedure FirstFactor;
begin
```

```
   case Look of
     '+': begin
             Match('+');
             Factor;
          end;
     '-': NegFactor;
   else  Factor;
   end;
end;


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

procedure Multiply;
begin
   Match('*');
   Factor;
   PopMul;
end;


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

procedure Divide;
begin
   Match('/');
   Factor;
   PopDiv;
end;


{---------------------------------------------------------------}
{ Common Code Used by Term and FirstTerm }

procedure Term1;
begin
   while IsMulop(Look) do begin
      Push;
      case Look of
       '*': Multiply;
       '/': Divide;
      end;
   end;
end;


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

procedure Term;
begin
   Factor;
   Term1;
end;


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

```
procedure FirstTerm;
begin
   FirstFactor;
   Term1;
end;


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

procedure Add;
begin
   Match('+');
   Term;
   PopAdd;
end;


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

procedure Subtract;
begin
   Match('-');
   Term;
   PopSub;
end;


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

procedure Expression;
begin
   FirstTerm;
   while IsAddop(Look) do begin
      Push;
      case Look of
       '+': Add;
       '-': Subtract;
      end;
   end;
end;


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

procedure Equal;
begin
   Match('=');
   Expression;
   PopCompare;
   SetEqual;
end;


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

procedure LessOrEqual;
```

```
begin
   Match('=');
   Expression;
   PopCompare;
   SetLessOrEqual;
end;


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

procedure NotEqual;
begin
   Match('>');
   Expression;
   PopCompare;
   SetNEqual;
end;


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

procedure Less;
begin
   Match('<');
   case Look of
     '=': LessOrEqual;
     '>': NotEqual;
   else begin
           Expression;
           PopCompare;
           SetLess;
        end;
   end;
end;


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

procedure Greater;
begin
   Match('>');
   if Look = '=' then begin
      Match('=');
      Expression;
      PopCompare;
      SetGreaterOrEqual;
      end
   else begin
      Expression;
      PopCompare;
      SetGreater;
   end;
end;


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

```
procedure Relation;
begin
   Expression;
   if IsRelop(Look) then begin
      Push;
      case Look of
       '=': Equal;
       '<': Less;
       '>': Greater;
      end;
   end;
end;


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

procedure NotFactor;
begin
   if Look = '!' then begin
      Match('!');
      Relation;
      NotIt;
      end
   else
      Relation;
end;


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

procedure BoolTerm;
begin
   NotFactor;
   while Look = '&' do begin
      Push;
      Match('&');
      NotFactor;
      PopAnd;
   end;
end;


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

procedure BoolOr;
begin
   Match('|');
   BoolTerm;
   PopOr;
end;


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

procedure BoolXor;
begin
```

```
   Match('~');
   BoolTerm;
   PopXor;
end;


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

procedure BoolExpression;
begin
   BoolTerm;
   while IsOrOp(Look) do begin
      Push;
      case Look of
       '|': BoolOr;
       '~': BoolXor;
      end;
   end;
end;


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

procedure Assignment;
var Name: string;
begin
   Name := Value;
   Match('=');
   BoolExpression;
   Store(Name);
end;


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

procedure Block; Forward;


procedure DoIf;
var L1, L2: string;
begin
   BoolExpression;
   L1 := NewLabel;
   L2 := L1;
   BranchFalse(L1);
   Block;
   if Token = 'l' then begin
      L2 := NewLabel;
      Branch(L2);
      PostLabel(L1);
      Block;
   end;
   PostLabel(L2);
   MatchString('ENDIF');
end;


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

```
{ Parse and Translate a WHILE Statement }

procedure DoWhile;
var L1, L2: string;
begin
   L1 := NewLabel;
   L2 := NewLabel;
   PostLabel(L1);
   BoolExpression;
   BranchFalse(L2);
   Block;
   MatchString('ENDWHILE');
   Branch(L1);
   PostLabel(L2);
end;


{---------------------------------------------------------------}
{ Process a Read Statement }

procedure DoRead;
begin
   Match('(');
   GetName;
   ReadVar;
   while Look = ',' do begin
      Match(',');
      GetName;
      ReadVar;
   end;
   Match(')');
end;


{---------------------------------------------------------------}
{ Process a Write Statement }

procedure DoWrite;
begin
   Match('(');
   Expression;
   WriteVar;
   while Look = ',' do begin
      Match(',');
      Expression;
      WriteVar;
   end;
   Match(')');
end;


{---------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure Block;
begin
   Scan;
   while not(Token in ['e', 'l']) do begin
      case Token of
       'i': DoIf;
       'w': DoWhile;
```

```
        'R': DoRead;
        'W': DoWrite;
       else Assignment;
       end;
       Scan;
   end;
end;


{--------------------------------------------------------------}
{ Allocate Storage for a Variable }

procedure Alloc(N: Symbol);
begin
   if InTable(N) then Abort('Duplicate Variable Name ' + N);
   AddEntry(N, 'v');
   Write(N, ':', TAB, 'DC ');
   if Look = '=' then begin
      Match('=');
      If Look = '-' then begin
         Write(Look);
         Match('-');
      end;
      WriteLn(GetNum);
      end
   else
      WriteLn('0');
end;


{--------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
begin
   GetName;
   Alloc(Value);
   while Look = ',' do begin
      Match(',');
      GetName;
      Alloc(Value);
   end;
end;


{--------------------------------------------------------------}
{ Parse and Translate Global Declarations }

procedure TopDecls;
begin
   Scan;
   while Token <> 'b' do begin
      case Token of
        'v': Decl;
      else Abort('Unrecognized Keyword ' + Value);
      end;
      Scan;
   end;
end;
```

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

procedure Main;
begin
   MatchString('BEGIN');
   Prolog;
   Block;
   MatchString('END');
   Epilog;
end;


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

procedure Prog;
begin
   MatchString('PROGRAM');
   Header;
   TopDecls;
   Main;
   Match('.');
end;


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

procedure Init;
var i: integer;
begin
   for i := 1 to MaxEntry do begin
      ST[i] := '';
      SType[i] := ' ';
   end;
   GetChar;
   Scan;
end;


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

begin
   Init;
   Prog;
   if Look <> CR then Abort('Unexpected data after ''.''');
end.
{--------------------------------------------------------------}
```

Chapter 11

Part XI: Lexical Scan Revisited - 3 June 1989

Introduction

I've got some good news and some bad news. The bad news is that this installment is not the one I promised last time. What's more, the one after this one won't be, either.

The good news is the reason for this installment: I've found a way to simplify and improve the lexical scanning part of the compiler. Let me explain.

Background

If you'll remember, we talked at length about the subject of lexical scanners in Part VII, and I left you with a design for a distributed scanner that I felt was about as simple as I could make it ... more than most that I've seen elsewhere. We used that idea in Part X. The compiler structure that resulted was simple, and it got the job done.

Recently, though, I've begun to have problems, and they're the kind that send a message that you might be doing something wrong.

The whole thing came to a head when I tried to address the issue of semicolons. Several people have asked me about them, and whether or not KISS will have them separating the statements. My intention has been *not* to use semicolons, simply because I don't like them and, as you can see, they have not proved necessary.

But I know that many of you, like me, have gotten used to them, and so I set out to write a short installment to show you how they could easily be added, if you were so inclined.

Well, it turned out that they weren't easy to add at all. In fact it was darned difficult.

I guess I should have realized that something was wrong, because of the issue of newlines. In the last couple of installments we've addressed that issue, and I've shown you how to deal with newlines with a procedure called, appropriately enough, `NewLine`. In TINY Version 1.0, I sprinkled calls to this procedure in strategic spots in the code.

It seems that every time I've addressed the issue of newlines, though, I've found it to be tricky, and the resulting parser turned out to be quite fragile ... one addition or deletion here or there and things tended to go to pot. Looking back on it, I realize that there was a message in this that I just wasn't paying attention to.

When I tried to add semicolons on top of the newlines, that was the last straw. I ended up with much too complex a solution. I began to realize that something fundamental had to change.

So, in a way this installment will cause us to backtrack a bit and revisit the issue of scanning all over again. Sorry about that. That's the price you pay for watching me do this in real time. But the new version is definitely an improvement, and will serve us well for what is to come.

As I said, the scanner we used in Part X was about as simple as one can get. But anything can be improved. The new scanner is more like the classical scanner, and not as simple as before. But the overall compiler structure is even simpler than before. It's also more robust, and easier to add to and/or modify. I think that's worth the time spent in this digression. So in this installment, I'll be showing you the new structure. No doubt you'll be happy to know that, while the changes affect many procedures, they aren't very profound and so we lose very little of what's been done so far.

Ironically, the new scanner is much more conventional than the old one, and is very much like the more generic scanner I showed you earlier in Part VII. Then I started trying to get clever, and I almost clevered myself clean out of business. You'd think one day I'd learn: K-I-S-S!

The Problem

The problem begins to show itself in procedure `Block`, which I've reproduced below:

```
{--------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure Block;
begin
   Scan;
   while not(Token in ['e', 'l']) do begin
      case Token of
       'i': DoIf;
       'w': DoWhile;
       'R': DoRead;
       'W': DoWrite;
      else Assignment;
      end;
      Scan;
   end;
end;
{--------------------------------------------------------------}
```

As you can see, `Block` is oriented to individual program statements. At each pass through the loop, we know that we are at the beginning of a statement. We exit the block when we have scanned an `END` or an `ELSE`.

But suppose that we see a semicolon instead. The procedure as it's shown above can't handle that, because procedure `Scan` only expects and can only accept tokens that begin with a letter.

I tinkered around for quite awhile to come up with a fix. I found many possible approaches, but none were very satisfying. I finally figured out the reason.

Recall that when we started with our single-character parsers, we adopted a convention that the lookahead character would always be prefetched. That is, we would have the character that corresponds to our current position in the input stream fetched into the global character Look, so that we could examine it as many times as needed. The rule we adopted was that *every* recognizer, if it found its target token, would advance Look to the next character in the input stream.

That simple and fixed convention served us very well when we had single-character tokens, and it still does. It would make a lot of sense to apply the same rule to multi-character tokens.

But when we got into lexical scanning, I began to violate that simple rule. The scanner of Part X did indeed advance to the next token if it found an identifier or keyword, but it *didn't* do that if it found a carriage return, a whitespace character, or an operator.

Now, that sort of mixed-mode operation gets us into deep trouble in procedure `Block`, because whether or not the input stream has been advanced depends upon the kind of token we encounter. If it's a keyword or the target of an assignment statement, the "cursor," as defined by the contents of Look, has been advanced to the next token `OR` to the beginning of whitespace. If, on the other hand, the token is a semicolon, or if we have hit a carriage return, the cursor has *not* advanced.

Needless to say, we can add enough logic to keep us on track. But it's tricky, and makes the whole parser very fragile.

There's a much better way, and that's just to adopt that same rule that's worked so well before, to apply to *tokens* as well as single characters. In other words, we'll prefetch tokens just as we've always done for characters. It seems so obvious once you think about it that way.

Interestingly enough, if we do things this way the problem that we've had with newline characters goes away. We can just lump them in as whitespace characters, which means that the handling of newlines becomes very trivial, and *much* less prone to error than we've had to deal with in the past.

The Solution

Let's begin to fix the problem by re-introducing the two procedures:

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

procedure GetName;
```

```
begin
   SkipWhite;
   if Not IsAlpha(Look) then Expected('Identifier');
   Token := 'x';
   Value := '';
   repeat
      Value := Value + UpCase(Look);
      GetChar;
   until not IsAlNum(Look);
end;


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

procedure GetNum;
begin
   SkipWhite;
   if not IsDigit(Look) then Expected('Number');
   Token := '#';
   Value := '';
   repeat
      Value := Value + Look;
      GetChar;
   until not IsDigit(Look);
end;
{--------------------------------------------------------------}
```

These two procedures are functionally almost identical to the ones I showed you in Part VII. They each fetch the current token, either an identifier or a number, into the global string `Value`. They also set the encoded version, `Token`, to the appropriate code. The input stream is left with Look containing the first character *not* part of the token.

We can do the same thing for operators, even multi-character operators, with a procedure such as:

```
{--------------------------------------------------------------}
{ Get an Operator }

procedure GetOp;
begin
   Token := Look;
   Value := '';
   repeat
      Value := Value + Look;
      GetChar;
   until IsAlpha(Look) or IsDigit(Look) or IsWhite(Look);
end;
{--------------------------------------------------------------}
```

Note that `GetOp` returns, as its encoded token, the *first* character of the operator. This is important, because it means that we can now use that single character to drive the parser, instead of the lookahead character.

We need to tie these procedures together into a single procedure that can handle all three cases. The following procedure will read any one of the token types and always leave the input stream advanced beyond it:

```
{--------------------------------------------------------------}
{ Get the Next Input Token }

procedure Next;
begin
   SkipWhite;
   if IsAlpha(Look) then GetName
   else if IsDigit(Look) then GetNum
```

```
   else GetOp;
end;
{--------------------------------------------------------------}
```

NOTE that here I have put `SkipWhite` *before* the calls rather than after. This means that, in general, the variable Look will *not* have a meaningful value in it, and therefore we should *not* use it as a test value for parsing, as we have been doing so far. That's the big departure from our normal approach.

Now, remember that before I was careful not to treat the carriage return (CR) and line feed (LF) characters as white space. This was because, with `SkipWhite` called as the last thing in the scanner, the encounter with LF would trigger a read statement. If we were on the last line of the program, we couldn't get out until we input another line with a non-white character. That's why I needed the second procedure, `NewLine`, to handle the CRLFs.

But now, with the call to `SkipWhite` coming first, that's exactly the behavior we want. The compiler must know there's another token coming or it wouldn't be calling `Next`. In other words, it hasn't found the terminating `END` yet. So we're going to insist on more data until we find something.

All this means that we can greatly simplify both the program and the concepts, by treating CR and LF as whitespace characters, and eliminating `NewLine`. You can do that simply by modifying the function `IsWhite`:

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

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

We've already tried similar routines in Part VII, but you might as well try these new ones out. Add them to a copy of the Cradle and call `Next` with the following main program:

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

begin
   Init;
   repeat
      Next;
      WriteLn(Token, ' ', Value);
   until Token = '.';
end.
{--------------------------------------------------------------}
```

Compile it and verify that you can separate a program into a series of tokens, and that you get the right encoding for each token.

This *almost* works, but not quite. There are two potential problems: First, in KISS/TINY almost all of our operators are single-character operators. The only exceptions are the relops `>=`, `<=`, and `<>`. It seems a shame to treat all operators as strings and do a string compare, when only a single character compare will almost always suffice. Second, and much more important, the thing doesn't *work* when two operators appear together, as in `(a+b)*(c+d)`. Here the string following `b` would be interpreted as a single operator `)*(`.

It's possible to fix that problem. For example, we could just give `GetOp` a list of legal characters, and we could treat the parentheses as different operator types than the others. But this begins to get messy.

Fortunately, there's a better way that solves all the problems. Since almost all the operators are single characters, let's just treat them that way, and let `GetOp` get only one character at a time. This not only simplifies `GetOp`, but also speeds things up quite a bit. We still have the problem of the relops, but we were treating them as special cases anyway.

So here's the final version of `GetOp`:

```
{--------------------------------------------------------------}
{ Get an Operator }

procedure GetOp;
```

```
begin
   SkipWhite;
   Token := Look;
   Value := Look;
   GetChar;
end;
{--------------------------------------------------------------}
```

Note that I still give the string `Value` a value. If you're truly concerned about efficiency, you could leave this out. When we're expecting an operator, we will only be testing `Token` anyhow, so the value of the string won't matter. But to me it seems to be good practice to give the thing a value just in case.

Try this new version with some realistic-looking code. You should be able to separate any program into its individual tokens, with the caveat that the two-character relops will scan into two separate tokens. That's OK ... we'll parse them that way.

Now, in Part VII the function of `Next` was combined with procedure `Scan`, which also checked every identifier against a list of keywords and encoded each one that was found. As I mentioned at the time, the last thing we would want to do is to use such a procedure in places where keywords should not appear, such as in expressions. If we did that, the keyword list would be scanned for every identifier appearing in the code. Not good.

The right way to deal with that is to simply separate the functions of fetching tokens and looking for keywords. The version of `Scan` shown below does *nothing* but check for keywords. Notice that it operates on the current token and does *not* advance the input stream.

```
{--------------------------------------------------------------}
{ Scan the Current Identifier for Keywords }

procedure Scan;
begin
   if Token = 'x' then
      Token := KWcode[Lookup(Addr(KWlist), Value, NKW) + 1];
end;
{--------------------------------------------------------------}
```

There is one last detail. In the compiler there are a few places that we must actually check the string value of the token. Mainly, this is done to distinguish between the different `END`s, but there are a couple of other places. (I should note in passing that we could always eliminate the need for matching `END` characters by encoding each one to a different character. Right now we are definitely taking the lazy man's route.)

The following version of `MatchString` takes the place of the character-oriented `Match`. Note that, like `Match`, it *does* advance the input stream.

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

procedure MatchString(x: string);
begin
   if Value <> x then Expected('''' + x + '''');
   Next;
end;
{--------------------------------------------------------------}
```

Fixing up the Compiler

Armed with these new scanner procedures, we can now begin to fix the compiler to use them properly. The changes are all quite minor, but there are quite a few places where changes are necessary. Rather than showing you each place, I will give you the general idea and then just give the finished product.

First of all, the code for procedure `Block` doesn't change, though its function does:

```
{--------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure Block;
```

```
begin
   Scan;
   while not(Token in ['e', 'l']) do begin
      case Token of
       'i': DoIf;
       'w': DoWhile;
       'R': DoRead;
       'W': DoWrite;
      else Assignment;
      end;
      Scan;
   end;
end;
{-------------------------------------------------------------}
```

Remember that the new version of `Scan` doesn't advance the input stream, it only scans for keywords. The input stream must be advanced by each procedure that `Block` calls.

In general, we have to replace every test on `Look` with a similar test on `Token`. For example:

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

procedure BoolExpression;
begin
   BoolTerm;
   while IsOrOp(Token) do begin
      Push;
      case Token of
       '|': BoolOr;
       '~': BoolXor;
      end;
   end;
end;
{-------------------------------------------------------------}
```

In procedures like `Add`, we don't have to use `Match` anymore. We need only call `Next` to advance the input stream:

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

procedure Add;
begin
   Next;
   Term;
   PopAdd;
end;
{-------------------------------------------------------------}
```

Control structures are actually simpler. We just call `Next` to advance over the control keywords:

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

procedure Block; Forward;

procedure DoIf;
var L1, L2: string;
begin
   Next;
   BoolExpression;
```

```
   L1 := NewLabel;
   L2 := L1;
   BranchFalse(L1);
   Block;
   if Token = 'l' then begin
      Next;
      L2 := NewLabel;
      Branch(L2);
      PostLabel(L1);
      Block;
   end;
   PostLabel(L2);
   MatchString('ENDIF');
end;
{--------------------------------------------------------------}
```

That's about the extent of the *required* changes. In the listing of TINY Version 1.1 below, I've also made a number of other "improvements" that aren't really required. Let me explain them briefly:

1. I've deleted the two procedures `Prog` and `Main`, and combined their functions into the main program. They didn't seem to add to program clarity ... in fact they seemed to just muddy things up a little.
2. I've deleted the keywords `PROGRAM` and `BEGIN` from the keyword list. Each one only occurs in one place, so it's not necessary to search for it.
3. Having been bitten by an overdose of cleverness, I've reminded myself that TINY is supposed to be a minimalist program. Therefore I've replaced the fancy handling of unary minus with the dumbest one I could think of. A giant step backwards in code quality, but a great simplification of the compiler. KISS is the right place to use the other version.
4. I've added some error-checking routines such as `CheckTable` and `CheckDup`, and replaced in-line code by calls to them. This cleans up a number of routines.
5. I've taken the error checking out of code generation routines like `Store`, and put it in the parser where it belongs. See `Assignment`, for example.
6. There was an error in `InTable` and `Locate` that caused them to search all locations instead of only those with valid data in them. They now search only valid cells. This allows us to eliminate the initialization of the symbol table, which was done in `Init`.
7. Procedure `AddEntry` now has two arguments, which helps to make things a bit more modular.
8. I've cleaned up the code for the relational operators by the addition of the new procedures `CompareExpression` and `NextExpression`.
9. I fixed an error in the `Read` routine ... the earlier value did not check for a valid variable name.

Conclusion

The resulting compiler for TINY is given below. Other than the removal of the keyword `PROGRAM`, it parses the same language as before. It's just a bit cleaner, and more importantly it's considerably more robust. I feel good about it.

The next installment will be another digression: the discussion of semicolons and such that got me into this mess in the first place. *Then* we'll press on into procedures and types. Hang in there with me. The addition of those features will go a long way towards removing KISS from the "toy language" category. We're getting very close to being able to write a serious compiler.

TINY Version 1.1

```
{--------------------------------------------------------------}
program Tiny11;

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

```
const TAB = ^I;
      CR  = ^M;
      LF  = ^J;

      LCount: integer = 0;
      NEntry: integer = 0;


{---------------------------------------------------------------}
{ Type Declarations }

type Symbol = string[8];

     SymTab = array[1..1000] of Symbol;

     TabPtr = ^SymTab;


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

var Look : char;             { Lookahead Character }
    Token: char;             { Encoded Token       }
    Value: string[16];       { Unencoded Token     }


const MaxEntry = 100;

var ST   : array[1..MaxEntry] of Symbol;
    SType: array[1..MaxEntry] of char;


{---------------------------------------------------------------}
{ Definition of Keywords and Token Types }

const NKW =   9;
      NKW1 = 10;

const KWlist: array[1..NKW] of Symbol =
              ('IF', 'ELSE', 'ENDIF', 'WHILE', 'ENDWHILE',
               'READ', 'WRITE', 'VAR', 'END');

const KWcode: string[NKW1] = 'xileweRWve';


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

{---------------------------------------------------------------}
{ Report an Undefined Identifier }

procedure Undefined(n: string);
begin
   Abort('Undefined Identifier ' + n);
end;


{---------------------------------------------------------------}
{ Report a Duplicate Identifier }

procedure Duplicate(n: string);
begin
   Abort('Duplicate Identifier ' + n);
end;


{---------------------------------------------------------------}
{ Check to Make Sure the Current Token is an Identifier }

procedure CheckIdent;
begin
   if Token <> 'x' then Expected('Identifier');
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 Character }

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 a Mulop }

function IsMulop(c: char): boolean;
begin
   IsMulop := c in ['*', '/'];
end;


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

function IsOrop(c: char): boolean;
begin
   IsOrop := c in ['|', '~'];
end;


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

function IsRelop(c: char): boolean;
begin
   IsRelop := c in ['=', '#', '<', '>'];
end;


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

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


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

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


{---------------------------------------------------------------}
{ Table Lookup }

function Lookup(T: TabPtr; s: string; n: integer): integer;
var i: integer;
    found: Boolean;
begin
   found := false;
   i := n;
   while (i > 0) and not found do
      if s = T^[i] then
         found := true
      else
         dec(i);
   Lookup := i;
end;


{---------------------------------------------------------------}
{ Locate a Symbol in Table }
{ Returns the index of the entry.  Zero if not present. }

function Locate(N: Symbol): integer;
begin
   Locate := Lookup(@ST, n, NEntry);
end;


{---------------------------------------------------------------}
{ Look for Symbol in Table }

function InTable(n: Symbol): Boolean;
begin
   InTable := Lookup(@ST, n, NEntry) <> 0;
end;


{---------------------------------------------------------------}
{ Check to See if an Identifier is in the Symbol Table         }
{ Report an error if it's not. }


procedure CheckTable(N: Symbol);
begin
   if not InTable(N) then Undefined(N);
end;


{---------------------------------------------------------------}
{ Check the Symbol Table for a Duplicate Identifier }
{ Report an error if identifier is already in table. }
```

```
procedure CheckDup(N: Symbol);
begin
   if InTable(N) then Duplicate(N);
end;


{--------------------------------------------------------------}
{ Add a New Entry to Symbol Table }

procedure AddEntry(N: Symbol; T: char);
begin
   CheckDup(N);
   if NEntry = MaxEntry then Abort('Symbol Table Full');
   Inc(NEntry);
   ST[NEntry] := N;
   SType[NEntry] := T;
end;


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

procedure GetName;
begin
   SkipWhite;
   if Not IsAlpha(Look) then Expected('Identifier');
   Token := 'x';
   Value := '';
   repeat
      Value := Value + UpCase(Look);
      GetChar;
   until not IsAlNum(Look);
end;


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

procedure GetNum;
begin
   SkipWhite;
   if not IsDigit(Look) then Expected('Number');
   Token := '#';
   Value := '';
   repeat
      Value := Value + Look;
      GetChar;
   until not IsDigit(Look);
end;


{--------------------------------------------------------------}
{ Get an Operator }

procedure GetOp;
begin
   SkipWhite;
   Token := Look;
   Value := Look;
   GetChar;
end;
```

```
{--------------------------------------------------------------}
{ Get the Next Input Token }

procedure Next;
begin
   SkipWhite;
   if IsAlpha(Look) then GetName
   else if IsDigit(Look) then GetNum
   else GetOp;
end;


{--------------------------------------------------------------}
{ Scan the Current Identifier for Keywords }

procedure Scan;
begin
   if Token = 'x' then
      Token := KWcode[Lookup(Addr(KWlist), Value, NKW) + 1];
end;


{--------------------------------------------------------------}
{ Match a Specific Input String }

procedure MatchString(x: string);
begin
   if Value <> x then Expected('''' + x + '''');
   Next;
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;


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

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

```
   Inc(LCount);
end;


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

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


{---------------------------------------------------------------}
{ Clear the Primary Register }

procedure Clear;
begin
   EmitLn('CLR D0');
end;


{---------------------------------------------------------------}
{ Negate the Primary Register }

procedure Negate;
begin
   EmitLn('NEG D0');
end;


{---------------------------------------------------------------}
{ Complement the Primary Register }

procedure NotIt;
begin
   EmitLn('NOT D0');
end;


{---------------------------------------------------------------}
{ Load a Constant Value to Primary Register }

procedure LoadConst(n: string);
begin
   Emit('MOVE #');
   WriteLn(n, ',D0');
end;


{---------------------------------------------------------------}
{ Load a Variable to Primary Register }

procedure LoadVar(Name: string);
begin
   if not InTable(Name) then Undefined(Name);
   EmitLn('MOVE ' + Name + '(PC),D0');
end;


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

```
{ Push Primary onto Stack }

procedure Push;
begin
   EmitLn('MOVE D0,-(SP)');
end;


{---------------------------------------------------------------}
{ Add Top of Stack to Primary }

procedure PopAdd;
begin
   EmitLn('ADD (SP)+,D0');
end;


{---------------------------------------------------------------}
{ Subtract Primary from Top of Stack }

procedure PopSub;
begin
   EmitLn('SUB (SP)+,D0');
   EmitLn('NEG D0');
end;


{---------------------------------------------------------------}
{ Multiply Top of Stack by Primary }

procedure PopMul;
begin
   EmitLn('MULS (SP)+,D0');
end;


{---------------------------------------------------------------}
{ Divide Top of Stack by Primary }

procedure PopDiv;
begin
   EmitLn('MOVE (SP)+,D7');
   EmitLn('EXT.L D7');
   EmitLn('DIVS D0,D7');
   EmitLn('MOVE D7,D0');
end;


{---------------------------------------------------------------}
{ AND Top of Stack with Primary }

procedure PopAnd;
begin
   EmitLn('AND (SP)+,D0');
end;


{---------------------------------------------------------------}
{ OR Top of Stack with Primary }

procedure PopOr;
```

```
begin
   EmitLn('OR (SP)+,D0');
end;


{---------------------------------------------------------------}
{ XOR Top of Stack with Primary }

procedure PopXor;
begin
   EmitLn('EOR (SP)+,D0');
end;


{---------------------------------------------------------------}
{ Compare Top of Stack with Primary }

procedure PopCompare;
begin
   EmitLn('CMP (SP)+,D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was = }

procedure SetEqual;
begin
   EmitLn('SEQ D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was != }

procedure SetNEqual;
begin
   EmitLn('SNE D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was > }

procedure SetGreater;
begin
   EmitLn('SLT D0');
   EmitLn('EXT D0');
end;


{---------------------------------------------------------------}
{ Set D0 If Compare was < }

procedure SetLess;
begin
   EmitLn('SGT D0');
   EmitLn('EXT D0');
end;
```

```
{-----------------------------------------------------------------}
{ Set D0 If Compare was <= }

procedure SetLessOrEqual;
begin
   EmitLn('SGE D0');
   EmitLn('EXT D0');
end;


{-----------------------------------------------------------------}
{ Set D0 If Compare was >= }

procedure SetGreaterOrEqual;
begin
   EmitLn('SLE D0');
   EmitLn('EXT D0');
end;


{-----------------------------------------------------------------}
{ Store Primary to Variable }

procedure Store(Name: string);
begin
   EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)')
end;


{-----------------------------------------------------------------}
{ Branch Unconditional  }

procedure Branch(L: string);
begin
   EmitLn('BRA ' + L);
end;


{-----------------------------------------------------------------}
{ Branch False }

procedure BranchFalse(L: string);
begin
   EmitLn('TST D0');
   EmitLn('BEQ ' + L);
end;


{-----------------------------------------------------------------}
{ Read Variable to Primary Register }

procedure ReadIt(Name: string);
begin
   EmitLn('BSR READ');
   Store(Name);
end;
```

```
{ Write from Primary Register }

procedure WriteIt;
begin
   EmitLn('BSR WRITE');
end;


{---------------------------------------------------------------}
{ Write Header Info }

procedure Header;
begin
   WriteLn('WARMST', TAB, 'EQU $A01E');
end;


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

procedure Prolog;
begin
   PostLabel('MAIN');
end;


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

procedure Epilog;
begin
   EmitLn('DC WARMST');
   EmitLn('END MAIN');
end;


{---------------------------------------------------------------}
{ Allocate Storage for a Static Variable }

procedure Allocate(Name, Val: string);
begin
   WriteLn(Name, ':', TAB, 'DC ', Val);
end;


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

procedure BoolExpression; Forward;

procedure Factor;
begin
   if Token = '(' then begin
      Next;
      BoolExpression;
      MatchString(')');
      end
   else begin
      if Token = 'x' then
         LoadVar(Value)
      else if Token = '#' then
```

```
         LoadConst(Value)
      else Expected('Math Factor');
      Next;
   end;
end;


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

procedure Multiply;
begin
   Next;
   Factor;
   PopMul;
end;


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

procedure Divide;
begin
   Next;
   Factor;
   PopDiv;
end;


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

procedure Term;
begin
   Factor;
   while IsMulop(Token) do begin
      Push;
      case Token of
       '*': Multiply;
       '/': Divide;
      end;
   end;
end;


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

procedure Add;
begin
   Next;
   Term;
   PopAdd;
end;


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

procedure Subtract;
begin
```

```
   Next;
   Term;
   PopSub;
end;


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

procedure Expression;
begin
   if IsAddop(Token) then
      Clear
   else
      Term;
   while IsAddop(Token) do begin
      Push;
      case Token of
       '+': Add;
       '-': Subtract;
      end;
   end;
end;


{-----------------------------------------------------------}
{ Get Another Expression and Compare }

procedure CompareExpression;
begin
   Expression;
   PopCompare;
end;


{-----------------------------------------------------------}
{ Get The Next Expression and Compare }

procedure NextExpression;
begin
   Next;
   CompareExpression;
end;


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

procedure Equal;
begin
   NextExpression;
   SetEqual;
end;


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

procedure LessOrEqual;
begin
   NextExpression;
```

```
   SetLessOrEqual;
end;


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

procedure NotEqual;
begin
   NextExpression;
   SetNEqual;
end;


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

procedure Less;
begin
   Next;
   case Token of
     '=': LessOrEqual;
     '>': NotEqual;
   else begin
           CompareExpression;
           SetLess;
        end;
   end;
end;


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

procedure Greater;
begin
   Next;
   if Token = '=' then begin
      NextExpression;
      SetGreaterOrEqual;
      end
   else begin
      CompareExpression;
      SetGreater;
   end;
end;


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


procedure Relation;
begin
   Expression;
   if IsRelop(Token) then begin
      Push;
      case Token of
       '=': Equal;
       '<': Less;
       '>': Greater;
```

```
      end;
   end;
end;


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

procedure NotFactor;
begin
   if Token = '!' then begin
      Next;
      Relation;
      NotIt;
      end
   else
      Relation;
end;


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

procedure BoolTerm;
begin
   NotFactor;
   while Token = '&' do begin
      Push;
      Next;
      NotFactor;
      PopAnd;
   end;
end;


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

procedure BoolOr;
begin
   Next;
   BoolTerm;
   PopOr;
end;


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

procedure BoolXor;
begin
   Next;
   BoolTerm;
   PopXor;
end;


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

procedure BoolExpression;
```

```
begin
   BoolTerm;
   while IsOrOp(Token) do begin
      Push;
      case Token of
       '|': BoolOr;
       '~': BoolXor;
      end;
   end;
end;


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

procedure Assignment;
var Name: string;
begin
   CheckTable(Value);
   Name := Value;
   Next;
   MatchString('=');
   BoolExpression;
   Store(Name);
end;


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

procedure Block; Forward;

procedure DoIf;
var L1, L2: string;
begin
   Next;
   BoolExpression;
   L1 := NewLabel;
   L2 := L1;
   BranchFalse(L1);
   Block;
   if Token = 'l' then begin
      Next;
      L2 := NewLabel;
      Branch(L2);
      PostLabel(L1);
      Block;
   end;
   PostLabel(L2);
   MatchString('ENDIF');
end;


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

procedure DoWhile;
var L1, L2: string;
begin
   Next;
   L1 := NewLabel;
```

```
   L2 := NewLabel;
   PostLabel(L1);
   BoolExpression;
   BranchFalse(L2);
   Block;
   MatchString('ENDWHILE');
   Branch(L1);
   PostLabel(L2);
end;


{--------------------------------------------------------------}
{ Read a Single Variable }

procedure ReadVar;
begin
   CheckIdent;
   CheckTable(Value);
   ReadIt(Value);
   Next;
end;


{--------------------------------------------------------------}
{ Process a Read Statement }

procedure DoRead;
begin
   Next;
   MatchString('(');
   ReadVar;
   while Token = ',' do begin
      Next;
      ReadVar;
   end;
   MatchString(')');
end;


{--------------------------------------------------------------}
{ Process a Write Statement }

procedure DoWrite;
begin
   Next;
   MatchString('(');
   Expression;
   WriteIt;
   while Token = ',' do begin
      Next;
      Expression;
      WriteIt;
   end;
   MatchString(')');
end;


{--------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure Block;
```

```
begin
   Scan;
   while not(Token in ['e', 'l']) do begin
      case Token of
       'i': DoIf;
       'w': DoWhile;
       'R': DoRead;
       'W': DoWrite;
      else Assignment;
      end;
      Scan;
   end;
end;


{---------------------------------------------------------------}
{ Allocate Storage for a Variable }

procedure Alloc;
begin
   Next;
   if Token <> 'x' then Expected('Variable Name');
   CheckDup(Value);
   AddEntry(Value, 'v');
   Allocate(Value, '0');
   Next;
end;


{---------------------------------------------------------------}
{ Parse and Translate Global Declarations }

procedure TopDecls;
begin
   Scan;
   while Token = 'v' do
      Alloc;
      while Token = ',' do
         Alloc;
end;


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

procedure Init;
begin
   GetChar;
   Next;
end;


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

begin
   Init;
   MatchString('PROGRAM');
   Header;
   TopDecls;
   MatchString('BEGIN');
```

```
   Prolog;
   Block;
   MatchString('END');
   Epilog;
end.
{-------------------------------------------------------------}
```

Chapter 12

Part XII: Miscellany - 5 June 1989

Introduction

This installment is another one of those excursions into side alleys that don't seem to fit into the mainstream of this tutorial series. As I mentioned last time, it was while I was writing this installment that I realized some changes had to be made to the compiler structure. So I had to digress from this digression long enough to develop the new structure and show it to you.

Now that that's behind us, I can tell you what I set out to in the first place. This shouldn't take long, and then we can get back into the mainstream.

Several people have asked me about things that other languages provide, but so far I haven't addressed in this series. The two biggies are semicolons and comments. Perhaps you've wondered about them, too, and wondered how things would change if we had to deal with them. Just so you can proceed with what's to come, without being bothered by that nagging feeling that something is missing, we'll address such issues here.

Semicolons

Ever since the introduction of Algol, semicolons have been a part of almost every modern language. We've all used them to the point that they are taken for granted. Yet I suspect that more compilation errors have occurred due to misplaced or missing semicolons than any other single cause. And if we had a penny for every extra keystroke programmers have used to type the little rascals, we could pay off the national debt.

Having been brought up with FORTRAN, it took me a long time to get used to using semicolons, and to tell the truth I've never quite understood why they were necessary. Since I program in Pascal, and since the use of semicolons in Pascal is particularly tricky, that one little character is still by far my biggest source of errors.

When I began developing KISS, I resolved to question *every* construct in other languages, and to try to avoid the most common problems that occur with them. That puts the semicolon very high on my hit list.

To understand the role of the semicolon, you have to look at a little history.

Early programming languages were line-oriented. In FORTRAN, for example, various parts of the statement had specific columns or fields that they had to appear in. Since some statements were too long for one line, the "continuation card" mechanism was provided to let the compiler know that a given card was still part of the previous line. The mechanism survives to this day, even though punched cards are now things of the distant past.

When other languages came along, they also adopted various mechanisms for dealing with multiple-line statements. BASIC is a good example. It's important to recognize, though, that the FORTRAN mechanism was not so much required by the line orientation of that language, as by the column-orientation. In those versions of FORTRAN where free-form input is permitted, it's no longer needed.

When the fathers of Algol introduced that language, they wanted to get away from line-oriented programs like FORTRAN and BASIC, and allow for free-form input. This included the possibility of stringing multiple statements on a single line, as in `a=b; c=d; e=e+1;`. In cases like this, the semicolon is almost *required*. The same line, without the semicolons, just looks "funny": `a=b c= d e=e+1`. I suspect that this is the major ... perhaps *only* ... reason for semicolons: to keep programs from looking funny.

But the idea of stringing multiple statements together on a single line is a dubious one at best. It's not very good programming style, and harks back to the days when it was considered important to conserve cards. In these days of CRT's and indented code, the clarity of programs is far better served by keeping statements separate. It's still nice to have the *option* of multiple statements, but it seems a shame to keep programmers in slavery to the semicolon, just to keep that one rare case from "looking funny."

When I started in with KISS, I tried to keep an open mind. I decided that I would use semicolons when it became necessary for the parser, but not until then. I figured this would happen just about the time I added

the ability to spread statements over multiple lines. But, as you can see, that never happened. The TINY compiler is perfectly happy to parse the most complicated statement, spread over any number of lines, without semicolons.

Still, there are people who have used semicolons for so long, they feel naked without them. I'm one of them. Once I had KISS defined sufficiently well, I began to write a few sample programs in the language. I discovered, somewhat to my horror, that I kept putting semicolons in anyway. So now I'm facing the prospect of a *new* rash of compiler errors, caused by *unwanted* semicolons. Phooey!

Perhaps more to the point, there are readers out there who are designing their own languages, which may include semicolons, or who want to use the techniques of these tutorials to compile conventional languages like C. In either case, we need to be able to deal with semicolons.

Syntactic Sugar

This whole discussion brings up the issue of "syntactic sugar" ... constructs that are added to a language, not because they are needed, but because they help make the programs look right to the programmer. After all, it's nice to have a small, simple compiler, but it would be of little use if the resulting language were cryptic and hard to program. The language FORTH comes to mind (a premature OUCH! for the barrage I know that one's going to fetch me). If we can add features to the language that make the programs easier to read and understand, and if those features help keep the programmer from making errors, then we should do so. Particularly if the constructs don't add much to the complexity of the language or its compiler.

The semicolon could be considered an example, but there are plenty of others, such as the `THEN` in a `IF`-statement, the `DO` in a `WHILE`-statement, and even the `PROGRAM` statement, which I came within a gnat's eyelash of leaving out of TINY. None of these tokens add much to the syntax of the language ... the compiler can figure out what's going on without them. But some folks feel that they `DO` add to the readability of programs, and that can be very important.

There are two schools of thought on this subject, which are well represented by two of our most popular languages, C and Pascal.

To the minimalists, all such sugar should be left out. They argue that it clutters up the language and adds to the number of keystrokes programmers must type. Perhaps more importantly, every extra token or keyword represents a trap laying in wait for the inattentive programmer. If you leave out a token, misplace it, or misspell it, the compiler will get you. So these people argue that the best approach is to get rid of such things. These folks tend to like C, which has a minimum of unnecessary keywords and punctuation.

Those from the other school tend to like Pascal. They argue that having to type a few extra characters is a small price to pay for legibility. After all, humans have to read the programs, too. Their best argument is that each such construct is an opportunity to tell the compiler that you really mean for it to do what you said to. The sugary tokens serve as useful landmarks to help you find your way.

The differences are well represented by the two languages. The most oft-heard complaint about C is that it is too forgiving. When you make a mistake in C, the erroneous code is too often another legal C construct. So the compiler just happily continues to compile, and leaves you to find the error during debug. I guess that's why debuggers are so popular with C programmers.

On the other hand, if a Pascal program compiles, you can be pretty sure that the program will do what you told it. If there is an error at run time, it's probably a design error.

The best example of useful sugar is the semicolon itself. Consider the code fragment `a=1+(2*b+c) b...` . Since there is no operator connecting the token `b` with the rest of the statement, the compiler will conclude that the expression ends with the `)`, and the `b` is the beginning of a new statement. But suppose I have simply left out the intended operator, and I really want to say `a=1+(2*b+c)*b...`.

In this case the compiler will get an error, all right, but it won't be very meaningful since it will be expecting an `=` sign after the `b` that really shouldn't be there.

If, on the other hand, I include a semicolon after the `b`, *then* there can be no doubt where I intend the statement to end. Syntactic sugar, then, can serve a very useful purpose by providing some additional insurance that we remain on track.

I find myself somewhere in the middle of all this. I tend to favor the Pascal-ers' view ... I'd much rather find my bugs at compile time rather than run time. But I also hate to just throw verbosity in for no apparent reason, as in COBOL. So far I've consistently left most of the Pascal sugar out of KISS/TINY. But I certainly have no strong feelings either way, and I also can see the value of sprinkling a little sugar around just for the extra insurance that it brings. If you like this latter approach, things like that are easy to add. Just remember that, like the semicolon, each item of sugar is something that can potentially cause a compile error by its omission.

Dealing with Semicolons

There are two distinct ways in which semicolons are used in popular languages. In Pascal, the semicolon is regarded as an statement SEPARATOR. No semicolon is required after the last statement in a block. The syntax is:

```
<block> ::= <statement> ( ';' <statement>)*

<statement> ::= <assignment> | <if> | <while> ... | null
```

(The null statement is *important!*)

Pascal also defines some semicolons in other places, such as after the PROGRAM statement.

In C and Ada, on the other hand, the semicolon is considered a statement *terminator*, and follows all statements (with some embarrassing and confusing exceptions). The syntax for this is simply:

```
<block> ::= ( <statement> ';')*
```

Of the two syntaxes, the Pascal one seems on the face of it more rational, but experience has shown that it leads to some strange difficulties. People get so used to typing a semicolon after every statement that they tend to type one after the last statement in a block, also. That usually doesn't cause any harm ... it just gets treated as a null statement. Many Pascal programmers, including yours truly, do just that. But there is one place you absolutely *cannot* type a semicolon, and that's right before an `ELSE`. This little gotcha has cost me many an extra compilation, particularly when the `ELSE` is added to existing code. So the C/Ada choice turns out to be better. Apparently Nicklaus Wirth thinks so, too: In his Modula 2, he abandoned the Pascal approach.

Given either of these two syntaxes, it's an easy matter (now that we've reorganized the parser!) to add these features to our parser. Let's take the last case first, since it's simpler.

To begin, I've made things easy by introducing a new recognizer:

```
{-------------------------------------------------------------}
{ Match a Semicolon }

procedure Semi;
begin
   MatchString(';');
end;
{-------------------------------------------------------------}
```

This procedure works very much like our old `Match`. It insists on finding a semicolon as the next token. Having found it, it skips to the next one.

Since a semicolon follows a statement, procedure `Block` is almost the only one we need to change:

```
{-------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure Block;
begin
   Scan;
   while not(Token in ['e', 'l']) do begin
      case Token of
       'i': DoIf;
       'w': DoWhile;
       'R': DoRead;
       'W': DoWrite;
       'x': Assignment;
      end;
      Semi;
      Scan;
   end;
end;
{-------------------------------------------------------------}
```

Note carefully the subtle change in the case statement. The call to `Assignment` is now guarded by a test on `Token`. This is to avoid calling `Assignment` when the token is a semicolon (which could happen if the statement is null).

Since declarations are also statements, we also need to add a call to `Semi` within procedure `TopDecls`:

```
{--------------------------------------------------------------}
{ Parse and Translate Global Declarations }

procedure TopDecls;
begin
   Scan;
   while Token = 'v' do begin
      Alloc;
      while Token = ',' do
         Alloc;
      Semi;
   end;
end;
{--------------------------------------------------------------}
```

Finally, we need one for the `PROGRAM` statement:

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

begin
   Init;
   MatchString('PROGRAM');
   Semi;
   Header;
   TopDecls;
   MatchString('BEGIN');
   Prolog;
   Block;
   MatchString('END');
   Epilog;
end.
{--------------------------------------------------------------}
```

It's as easy as that. Try it with a copy of TINY and see how you like it.

The Pascal version is a little trickier, but it still only requires minor changes, and those only to procedure `Block`. To keep things as simple as possible, let's split the procedure into two parts. The following procedure handles just one statement:

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

procedure Statement;
begin
   Scan;
   case Token of
    'i': DoIf;
    'w': DoWhile;
    'R': DoRead;
    'W': DoWrite;
    'x': Assignment;
   end;
end;
```

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

Using this procedure, we can now rewrite `Block` like this:

```
{-------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure Block;
begin
   Statement;
   while Token = ';' do begin
      Next;
      Statement;
   end;
end;
{-------------------------------------------------------------}
```

That sure didn't hurt, did it? We can now parse semicolons in Pascal-like fashion.

A Compromise

Now that we know how to deal with semicolons, does that mean that I'm going to put them in KISS/TINY? Well, yes and no. I like the extra sugar and the security that comes with knowing for sure where the ends of statements are. But I haven't changed my dislike for the compilation errors associated with semicolons.

So I have what I think is a nice compromise: Make them *optional*!

Consider the following version of Semi:

```
{-------------------------------------------------------------}
{ Match a Semicolon }

procedure Semi;
begin
   if Token = ';' then Next;
end;
{-------------------------------------------------------------}
```

This procedure will *accept* a semicolon whenever it is called, but it won't *insist* on one. That means that when you choose to use semicolons, the compiler will use the extra information to help keep itself on track. But if you omit one (or omit them all) the compiler won't complain. The best of both worlds.

Put this procedure in place in the first version of your program (the one for C/Ada syntax), and you have the makings of TINY Version 1.2.

Comments

Up until now I have carefully avoided the subject of comments. You would think that this would be an easy subject ... after all, the compiler doesn't have to deal with comments at all; it should just ignore them. Well, sometimes that's true.

Comments can be just about as easy or as difficult as you choose to make them. At one extreme, we can arrange things so that comments are intercepted almost the instant they enter the compiler. At the other, we can treat them as lexical elements. Things tend to get interesting when you consider things like comment delimiters contained in quoted strings.

Single-Character Delimiters

Here's an example. Suppose we assume the Turbo Pascal standard and use curly braces for comments. In this case we have single-character delimiters, so our parsing is a little easier.

One approach is to strip the comments out the instant we encounter them in the input stream; that is, right in procedure `GetChar`. To do this, first change the name of `GetChar` to something else, say `GetCharX`. (For the

record, this is going to be a *temporary* change, so best not do this with your only copy of TINY. I assume you understand that you should always do these experiments with a working copy.)

Now, we're going to need a procedure to skip over comments. So key in the following one:

```
{--------------------------------------------------------------}
{ Skip A Comment Field }

procedure SkipComment;
begin
   while Look <> '}' do
      GetCharX;
   GetCharX;
end;
{--------------------------------------------------------------}
```

Clearly, what this procedure is going to do is to simply read and discard characters from the input stream, until it finds a right curly brace. Then it reads one more character and returns it in Look.

Now we can write a new version of `GetChar` that `SkipComment` to strip out comments:

```
{--------------------------------------------------------------}
{ Get Character from Input Stream }
{ Skip Any Comments }

procedure GetChar;
begin
   GetCharX;
   if Look = '{' then SkipComment;
end;
{--------------------------------------------------------------}
```

Code this up and give it a try. You'll find that you can, indeed, bury comments anywhere you like. The comments never even get into the parser proper ... every call to `GetChar` just returns any character that's *not* part of a comment.

As a matter of fact, while this approach gets the job done, and may even be perfectly satisfactory for you, it does its job a little *too* well. First of all, most programming languages specify that a comment should be treated like a space, so that comments aren't allowed to be embedded in, say, variable names. This current version doesn't care *where* you put comments.

Second, since the rest of the parser can't even receive a `{` character, you will not be allowed to put one in a quoted string.

Before you turn up your nose at this simplistic solution, though, I should point out that as respected a compiler as Turbo Pascal also won't allow a `{` in a quoted string. Try it. And as for embedding a comment in an identifier, I can't imagine why anyone would want to do such a thing, anyway, so the question is moot. For 99% of all applications, what I've just shown you will work just fine.

But, if you want to be picky about it and stick to the conventional treatment, then we need to move the interception point downstream a little further.

To do this, first change `GetChar` back to the way it was and change the name called in `SkipComment`. Then, let's add the left brace as a possible whitespace character:

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

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

Now, we can deal with comments in procedure `SkipWhite`:

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

procedure SkipWhite;
begin
   while IsWhite(Look) do begin
      if Look = '{' then
         SkipComment
      else
         GetChar;
   end;
end;
{-------------------------------------------------------------}
```

Note that `SkipWhite` is written so that we will skip over any combination of whitespace characters and comments, in one call.

OK, give this one a try, too. You'll find that it will let a comment serve to delimit tokens. It's worth mentioning that this approach also gives us the ability to handle curly braces within quoted strings, since within such strings we will not be testing for or skipping over whitespace.

There's one last item to deal with: `Nested` comments. Some programmers like the idea of nesting comments, since it allows you to comment out code during debugging. The code I've given here won't allow that and, again, neither will Turbo Pascal.

But the fix is incredibly easy. All we need to do is to make `SkipComment` recursive:

```
{-------------------------------------------------------------}
{ Skip A Comment Field }

procedure SkipComment;
begin
   while Look <> '}' do begin
      GetChar;
      if Look = '{' then SkipComment;
   end;
   GetChar;
end;
{-------------------------------------------------------------}
```

That does it. As sophisticated a comment-handler as you'll ever need.

Multi-Character Delimiters

That's all well and good for cases where a comment is delimited by single characters, but what about the cases such as C or standard Pascal, where two characters are required? Well, the principles are still the same, but we have to change our approach quite a bit. I'm sure it won't surprise you to learn that things get harder in this case.

For the multi-character situation, the easiest thing to do is to intercept the left delimiter back at the `GetChar` stage. We can "tokenize" it right there, replacing it by a single character.

Let's assume we're using the C delimiters `/*` and `*/`. First, we need to go back to the `GetCharX` approach. In yet another copy of your compiler, rename `GetChar` to `GetCharX` and then enter the following new procedure `GetChar`:

```
{-------------------------------------------------------------}
{ Read New Character.  Intercept '/*' }

procedure GetChar;
begin
   if TempChar <> ' ' then begin
      Look := TempChar;
      TempChar := ' ';
      end
```

```
   else begin
      GetCharX;
      if Look = '/' then begin
         Read(TempChar);
         if TempChar = '*' then begin
            Look := '{';
            TempChar := ' ';
         end;
      end;
   end;
end;
{--------------------------------------------------------------}
```

As you can see, what this procedure does is to intercept every occurrence of `/`. It then examines the *next* character in the stream. If the character is a `*`, then we have found the beginning of a comment, and `GetChar` will return a single character replacement for it. (For simplicity, I'm using the same `{` character as I did for Pascal. If you were writing a C compiler, you'd no doubt want to pick some other character that's not used elsewhere in C. Pick anything you like ... even `$FF`, anything that's unique.)

If the character following the `/` is *not* a `*`, then `GetChar` tucks it away in the new global TempChar, and returns the `/`.

Note that you need to declare this new variable and initialize it to . I like to do things like that using the Turbo "typed constant" construct:

```
const TempChar: char = ' ';
```

Now we need a new version of `SkipComment`:

```
{--------------------------------------------------------------}
{ Skip A Comment Field }

procedure SkipComment;
begin
   repeat
      repeat
         GetCharX;
      until Look = '*';
      GetCharX;
   until Look = '/';
   GetChar;
end;
{--------------------------------------------------------------}
```

A few things to note: first of all, function `IsWhite` and procedure `SkipWhite` don't need to be changed, since `GetChar` returns the `{` token. If you change that token character, then of course you also need to change the character in those two routines.

Second, note that `SkipComment` doesn't call `GetChar` in its loop, but `GetCharX`. That means that the trailing `/` is not intercepted and is seen by `SkipComment`. Third, although `GetChar` is the procedure doing the work, we can still deal with the comment characters embedded in a quoted string, by calling `GetCharX` instead of `GetChar` while we're within the string. Finally, note that we can again provide for nested comments by adding a single statement to `SkipComment`, just as we did before.

One-Sided Comments

So far I've shown you how to deal with any kind of comment delimited on the left and the right. That only leaves the one-sided comments like those in assembler language or in Ada, that are terminated by the end of the line. In a way, that case is easier. The only procedure that would need to be changed is `SkipComment`, which must now terminate at the newline characters:

```
{--------------------------------------------------------------}
{ Skip A Comment Field }
```

```
procedure SkipComment;
begin
   repeat
      GetCharX;
   until Look = CR;
   GetChar;
end;
{--------------------------------------------------------------}
```

If the leading character is a single one, as in the `;` of assembly language, then we're essentially done. If it's a two-character token, as in the `--` of Ada, we need only modify the tests within `GetChar`. Either way, it's an easier problem than the balanced case.

Conclusion

At this point we now have the ability to deal with both comments and semicolons, as well as other kinds of syntactic sugar. I've shown you several ways to deal with each, depending upon the convention desired. The only issue left is: which of these conventions should we use in KISS/TINY?

For the reasons that I've given as we went along, I'm choosing the following:

1. Semicolons are *terminators*, not separators
2. Semicolons are *optional*
3. Comments are delimited by curly braces
4. Comments *may* be nested

Put the code corresponding to these cases into your copy of TINY. You now have TINY Version 1.2.

Now that we have disposed of these sideline issues, we can finally get back into the mainstream. In the next installment, we'll talk about procedures and parameter passing, and we'll add these important features to TINY. See you then.

Chapter 13

Part XIII: Procedures - 27 August 1989

Introduction

At last we get to the good part!

At this point we've studied almost all the basic features of compilers and parsing. We have learned how to translate arithmetic expressions, Boolean expressions, control constructs, data declarations, and I/O statements. We have defined a language, TINY 1.3, that embodies all these features, and we have written a rudimentary compiler that can translate them. By adding some file I/O we could indeed have a working compiler that could produce executable object files from programs written in TINY. With such a compiler, we could write simple programs that could read integer data, perform calculations with it, and output the results.

That's nice, but what we have is still only a toy language. We can't read or write even a single character of text, and we still don't have procedures.

It's the features to be discussed in the next couple of installments that separate the men from the toys, so to speak. "Real" languages have more than one data type, and they support procedure calls. More than any others, it's these two features that give a language much of its character and personality. Once we have provided for them, our languages, TINY and its successors, will cease to become toys and will take on the character of real languages, suitable for serious programming jobs.

For several installments now, I've been promising you sessions on these two important subjects. Each time, other issues came up that required me to digress and deal with them. Finally, we've been able to put all those issues to rest and can get on with the mainstream of things. In this installment, I'll cover procedures. Next time, we'll talk about the basic data types.

One Last Digression

This has been an extraordinarily difficult installment for me to write. The reason has nothing to do with the subject itself ... I've known what I wanted to say for some time, and in fact I presented most of this at Software Development '89, back in February. It has more to do with the approach. Let me explain.

When I first began this series, I told you that we would use several "tricks" to make things easy, and to let us learn the concepts without getting too bogged down in the details. Among these tricks was the idea of looking at individual pieces of a compiler at a time, i.e. performing experiments using the Cradle as a base. When we studied expressions, for example, we dealt with only that part of compiler theory. When we studied control structures, we wrote a different program, still based on the Cradle, to do that part. We only incorporated these concepts into a complete language fairly recently. These techniques have served us very well indeed, and led us to the development of a compiler for TINY version 1.3.

When I first began this session, I tried to build upon what we had already done, and just add the new features to the existing compiler. That turned out to be a little awkward and tricky ... much too much to suit me.

I finally figured out why. In this series of experiments, I had abandoned the very useful techniques that had allowed us to get here, and without meaning to I had switched over into a new method of working, that involved incremental changes to the full TINY compiler.

You need to understand that what we are doing here is a little unique. There have been a number of articles, such as the Small C articles by Cain and Hendrix, that presented finished compilers for one language or another. This is different. In this series of tutorials, you are watching me design and implement both a language and a compiler, in real time.

In the experiments that I've been doing in preparation for this article, I was trying to inject the changes into the TINY compiler in such a way that, at every step, we still had a real, working compiler. In other words, I was

attempting an incremental enhancement of the language and its compiler, while at the same time explaining to you what I was doing.

That's a tough act to pull off! I finally realized that it was dumb to try. Having gotten this far using the idea of small experiments based on single-character tokens and simple, special-purpose programs, I had abandoned them in favor of working with the full compiler. It wasn't working.

So we're going to go back to our roots, so to speak. In this installment and the next, I'll be using single-character tokens again as we study the concepts of procedures, unfettered by the other baggage that we have accumulated in the previous sessions. As a matter of fact, I won't even attempt, at the end of this session, to merge the constructs into the TINY compiler. We'll save that for later.

After all this time, you don't need more buildup than that, so let's waste no more time and dive right in.

The Basics

All modern CPU's provide direct support for procedure calls, and the 68000 is no exception. For the 68000, the call is a `BSR` (PC-relative version) or `JSR`, and the return is `RTS`. All we have to do is to arrange for the compiler to issue these commands at the proper place.

Actually, there are really *three* things we have to address. One of them is the call/return mechanism. The second is the mechanism for *defining* the procedure in the first place. And, finally, there is the issue of passing parameters to the called procedure. None of these things are really very difficult, and we can of course borrow heavily on what people have done in other languages ... there's no need to reinvent the wheel here. Of the three issues, that of parameter passing will occupy most of our attention, simply because there are so many options available.

A Basis for Experiments

As always, we will need some software to serve as a basis for what we are doing. We don't need the full TINY compiler, but we do need enough of a program so that some of the other constructs are present. Specifically, we need at least to be able to handle statements of some sort, and data declarations.

The program shown below is that basis. It's a vestigial form of TINY, with single-character tokens. It has data declarations, but only in their simplest form ... no lists or initializers. It has assignment statements, but only of the kind `<ident> = <ident>`.

In other words, the only legal expression is a single variable name. There are no control constructs ... the only legal statement is the assignment.

Most of the program is just the standard Cradle routines. I've shown the whole thing here, just to make sure we're all starting from the same point:

```
{--------------------------------------------------------------}
program Calls;

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

const TAB = ^I;
      CR  = ^M;
      LF  = ^J;

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

var Look: char;               { Lookahead Character }

var ST: Array['A'..'Z'] of char;


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


{---------------------------------------------------------------}
{ Report an Undefined Identifier }

procedure Undefined(n: string);
begin
   Abort('Undefined Identifier ' + n);
end;


{---------------------------------------------------------------}
{ Report an Duplicate Identifier }

procedure Duplicate(n: string);
begin
     Abort('Duplicate Identifier ' + n);
end;


{---------------------------------------------------------------}
{ Get Type of Symbol }

function TypeOf(n: char): char;
begin
     TypeOf := ST[n];
end;


{---------------------------------------------------------------}
{ Look for Symbol in Table }
```

```
function InTable(n: char): Boolean;
begin
   InTable := ST[n] <> ' ';
end;


{-------------------------------------------------------------}
{ Add a New Symbol to Table }

procedure AddEntry(Name, T: char);
begin
     if Intable(Name) then Duplicate(Name);
     ST[Name] := T;
end;


{-------------------------------------------------------------}
{ Check an Entry to Make Sure It's a Variable }

procedure CheckVar(Name: char);
begin
     if not InTable(Name) then Undefined(Name);
     if  TypeOf(Name)  <>  'v'  then    Abort(Name  +  ' is not a
variable');
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 Character }

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 a Mulop }

function IsMulop(c: char): boolean;
begin
   IsMulop := c in ['*', '/'];
end;


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

function IsOrop(c: char): boolean;
begin
   IsOrop := c in ['|', '~'];
end;


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

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


{---------------------------------------------------------------}
{ Skip Over an End-of-Line }

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

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

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


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

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


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

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


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

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

```
{---------------------------------------------------------------}
{ Load a Variable to the Primary Register }

procedure LoadVar(Name: char);
begin
     CheckVar(Name);
     EmitLn('MOVE ' + Name + '(PC),D0');
end;


{---------------------------------------------------------------}
{ Store the Primary Register }

procedure StoreVar(Name: char);
begin
     CheckVar(Name);
     EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)')
end;


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

procedure Init;
var i: char;
begin
     GetChar;
     SkipWhite;
     for i := 'A' to 'Z' do
          ST[i] := ' ';
end;


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

procedure Expression;
begin
     LoadVar(GetName);
end;


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

procedure Assignment;
var Name: char;
begin
     Name := GetName;
     Match('=');
     Expression;
     StoreVar(Name);
end;


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

```
{ Parse and Translate a Block of Statements }

procedure DoBlock;
begin
     while not(Look in ['e']) do begin
          Assignment;
          Fin;
   end;
end;


{--------------------------------------------------------------}
{ Parse and Translate a Begin-Block }

procedure BeginBlock;
begin
     Match('b');
     Fin;
     DoBlock;
     Match('e');
     Fin;
end;


{--------------------------------------------------------------}
{ Allocate Storage for a Variable }

procedure Alloc(N: char);
begin
     if InTable(N) then Duplicate(N);
   ST[N] := 'v';
     WriteLn(N, ':', TAB, 'DC 0');
end;


{--------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
var Name: char;
begin
   Match('v');
     Alloc(GetName);
end;


{--------------------------------------------------------------}
{ Parse and Translate Global Declarations }

procedure TopDecls;
begin
     while Look <> 'b' do begin
      case Look of
        'v': Decl;
```

```
      else Abort('Unrecognized Keyword ' + Look);
          end;
          Fin;
     end;
end;


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

begin
     Init;
     TopDecls;
     BeginBlock;
end.
{--------------------------------------------------------------}
```

Note that we DO have a symbol table, and there is logic to check a variable name to make sure it's a legal one. It's also worth noting that I have included the code you've seen before to provide for white space and newlines. Finally, note that the main program is delimited, as usual, by BEGIN-END brackets.

Once you've copied the program to Turbo, the first step is to compile it and make sure it works. Give it a few declarations, and then a begin-block. Try something like:

```
va              (for VAR A)
vb              (for VAR B)
vc              (for VAR C)
b               (for BEGIN)
a=b
b=c
e.              (for END.)
```

As usual, you should also make some deliberate errors, and verify that the program catches them correctly.

Declaring a Procedure

If you're satisfied that our little program works, then it's time to deal with the procedures. Since we haven't talked about parameters yet, we'll begin by considering only procedures that have no parameter lists.

As a start, let's consider a simple program with a procedure, and think about the code we'd like to see generated for it:

```
PROGRAM FOO;
.
.
PROCEDURE BAR;                      BAR:
BEGIN                                    .
.                                        .
.                                        .
END;                                     RTS

BEGIN { MAIN PROGRAM }              MAIN:
.                                        .
.                                        .
FOO;                                     BSR BAR
.                                        .
.                                        .
END.                                     END MAIN
```

Here I've shown the high-order language constructs on the left, and the desired assembler code on the right. The first thing to notice is that we certainly don't have much code to generate here! For the great bulk of both the procedure and the main program, our existing constructs take care of the code to be generated.

The key to dealing with the body of the procedure is to recognize that although a procedure may be quite long, declaring it is really no different than declaring a variable. It's just one more kind of declaration. We can write the BNF:

```
<declaration> ::= <data decl> | <procedure>
```

This means that it should be easy to modify `TopDecl` to deal with procedures. What about the syntax of a procedure? Well, here's a suggested syntax, which is essentially that of Pascal:

```
<procedure> ::= PROCEDURE <ident> <begin-block>
```

There is practically no code generation required, other than that generated within the begin-block. We need only emit a label at the beginning of the procedure, and an `RTS` at the end.

Here's the required code:

```
{--------------------------------------------------------------}
{ Parse and Translate a Procedure Declaration }

procedure DoProc;
var N: char;
begin
     Match('p');
     N := GetName;
     Fin;
     if InTable(N) then Duplicate(N);
     ST[N] := 'p';
     PostLabel(N);
     BeginBlock;
     Return;
end;
{--------------------------------------------------------------}
```

Note that I've added a new code generation routine, Return, which merely emits an `RTS` instruction. The creation of that routine is "left as an exercise for the student."

To finish this version, add the following line within the Case statement in DoBlock:

```
'p': DoProc;
```

I should mention that this structure for declarations, and the BNF that drives it, differs from standard Pascal. In the Jensen & Wirth definition of Pascal, variable declarations, in fact *all* kinds of declarations, must appear in a specific sequence, i.e. labels, constants, types, variables, procedures, and main program. To follow such a scheme, we should separate the two declarations, and have code in the main program something like

```
DoVars;
DoProcs;
DoMain;
```

However, most implementations of Pascal, including Turbo, don't require that order and let you freely mix up the various declarations, as long as you still don't try to refer to something before it's declared. Although it may be more aesthetically pleasing to declare all the global variables at the top of the program, it certainly doesn't do any *harm* to allow them to be sprinkled around. In fact, it may do some *good*, in the sense that it gives you the opportunity to do a little rudimentary information hiding. Variables that should be accessed only by the main program, for example, can be declared just before it and will thus be inaccessible by the procedures.

OK, try this new version out. Note that we can declare as many procedures as we choose (as long as we don't run out of single-character names!), and the labels and `RTS`s all come out in the right places.

It's worth noting here that I do *not* allow for nested procedures. In TINY, all procedures must be declared at the global level, the same as in C. There has been quite a discussion about this point in the Computer Language Forum of CompuServe. It turns out that there is a significant penalty in complexity that must be paid for the luxury of nested procedures. What's more, this penalty gets paid at *run time*, because extra code must be added and executed every time a procedure is called. I also happen to believe that nesting is not a good idea, simply on the grounds that I have seen too many abuses of the feature. Before going on to the next step, it's also worth noting that the "main program" as it stands is incomplete, since it doesn't have the label and `END` statement. Let's fix that little oversight:

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

procedure DoMain;
begin
     Match('b');
     Fin;
     Prolog;
     DoBlock;
     Epilog;
end;
{--------------------------------------------------------------}
.
.
.
{--------------------------------------------------------------}
{ Main Program }

begin
     Init;
     TopDecls;
     DoMain;
end.
{--------------------------------------------------------------}
```

Note that `DoProc` and `DoMain` are not quite symmetrical. `DoProc` uses a call to `BeginBlock`, whereas `DoMain` cannot. That's because a procedure is signaled by the keyword `PROCEDURE` (abbreviated by a `p` here), while the main program gets no keyword other than the `BEGIN` itself.

And *that* brings up an interesting question: *why*?

If we look at the structure of C programs, we find that all functions are treated just alike, except that the main program happens to be identified by its name, `main`. Since C functions can appear in any order, the main program can also be anywhere in the compilation unit.

In Pascal, on the other hand, all variables and procedures must be declared before they're used, which means that there is no point putting anything after the main program ... it could never be accessed. The "main program" is not identified at all, other than being that part of the code that comes after the global `BEGIN`. In other words, if it ain't anything else, it must be the main program.

This causes no small amount of confusion for beginning programmers, and for big Pascal programs sometimes it's difficult to find the beginning of the main program at all. This leads to conventions such as identifying it in comments:

```
BEGIN { of MAIN }
```

This has always seemed to me to be a bit of a kludge. The question comes up: Why should the main program be treated so much differently than a procedure? In fact, now that we've recognized that procedure declarations are just that ... part of the global declarations ... isn't the main program just one more declaration, also?

The answer is yes, and by treating it that way, we can simplify the code and make it considerably more orthogonal. I propose that we use an explicit keyword, `PROGRAM`, to identify the main program (Note that this means that we can't start the file with it, as in Pascal). In this case, our BNF becomes:

```
<declaration> ::= <data decl> | <procedure> | <main program>

<procedure> ::= PROCEDURE <ident> <begin-block>

<main program> ::= PROGRAM <ident> <begin-block>
```

The code also looks much better, at least in the sense that `DoMain` and `DoProc` look more alike:

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

procedure DoMain;
```

```
var N: char;
begin
     Match('P');
     N := GetName;
     Fin;
     if InTable(N) then Duplicate(N);
     Prolog;
     BeginBlock;
end;
{--------------------------------------------------------------}
.
.
.
{--------------------------------------------------------------}
{ Parse and Translate Global Declarations }

procedure TopDecls;
begin
     while Look <> '.' do begin
      case Look of
            'v': Decl;
            'p': DoProc;
            'P': DoMain;
          else Abort('Unrecognized Keyword ' + Look);
          end;
          Fin;
     end;
end;


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

begin
     Init;
     TopDecls;
     Epilog;
end.
{--------------------------------------------------------------}
```

Since the declaration of the main program is now within the loop of `TopDecl`, that does present some difficulties. How do we ensure that it's the last thing in the file? And how do we ever exit from the loop? My answer for the second question, as you can see, was to bring back our old friend the period. Once the parser sees that, we're done.

To answer the first question: it depends on how far we're willing to go to protect the programmer from dumb mistakes. In the code that I've shown, there's nothing to keep the programmer from adding code after the main program ... even another main program. The code will just not be accessible. However, we *could* access it via a `FORWARD` statement, which we'll be providing later. As a matter of fact, many assembler language programmers like to use the area just after the program to declare large, uninitialized data blocks, so there may indeed be some value in not requiring the main program to be last. We'll leave it as it is.

If we decide that we should give the programmer a little more help than that, it's pretty easy to add some logic to kick us out of the loop once the main program has been processed. Or we could at least flag an error if someone tries to include two mains.

Calling the Procedure

If you're satisfied that things are working, let's address the second half of the equation ... the call.

Consider the BNF for a procedure call:

```
<proc_call> ::= <identifier>
```

for an assignment statement, on the other hand, the BNF is:

```
<assignment> ::= <identifier> '=' <expression>
```

At this point we seem to have a problem. The two BNF statements both begin on the right-hand side with the token `<identifier>`. How are we supposed to know, when we see the identifier, whether we have a procedure call or an assignment statement? This looks like a case where our parser ceases being predictive, and indeed that's exactly the case. However, it turns out to be an easy problem to fix, since all we have to do is to look at the type of the identifier, as recorded in the symbol table. As we've discovered before, a minor local violation of the predictive parsing rule can be easily handled as a special case.

Here's how to do it:

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

procedure Assignment(Name: char);
begin
     Match('=');
     Expression;
     StoreVar(Name);
end;


{-------------------------------------------------------------}
{ Decide if a Statement is an Assignment or Procedure Call }

procedure AssignOrProc;
var Name: char;
begin
     Name := GetName;
     case TypeOf(Name) of
          ' ': Undefined(Name);
          'v': Assignment(Name);
          'p': CallProc(Name);
          else Abort('Identifier ' + Name +
                                ' Cannot Be Used Here');
     end;
end;


{-------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure DoBlock;
begin
     while not(Look in ['e']) do begin
          AssignOrProc;
          Fin;
   end;
end;
{-------------------------------------------------------------}
```

As you can see, procedure `Block` now calls `AssignOrProc` instead of `Assignment`. The function of this new procedure is to simply read the identifier, determine its type, and then call whichever procedure is appropriate for that type. Since the name has already been read, we must pass it to the two procedures, and modify `Assignment` to match. Procedure `CallProc` is a simple code generation routine:

```
{-------------------------------------------------------------}
{ Call a Procedure }

procedure CallProc(N: char);
begin
     EmitLn('BSR ' + N);
end;
{-------------------------------------------------------------}
```

Well, at this point we have a compiler that can deal with procedures. It's worth noting that procedures can call procedures to any depth. So even though we don't allow nested *declarations*, there is certainly nothing to keep us from nesting *calls*, just as we would expect to do in any language. We're getting there, and it wasn't too hard, was it?

Of course, so far we can only deal with procedures that have no parameters. The procedures can only operate on the global variables by their global names. So at this point we have the equivalent of BASIC's `GOSUB` construct. Not too bad ... after all lots of serious programs were written using `GOSUB`s, but we can do better, and we will. That's the next step.

Passing Parameters

Again, we all know the basic idea of passed parameters, but let's review them just to be safe.

In general the procedure is given a parameter list, for example `PROCEDURE FOO(X, Y, Z)`. In the declaration of a procedure, the parameters are called formal parameters, and may be referred to in the body of the procedure by those names. The names used for the formal parameters are really arbitrary. Only the position really counts. In the example above, the name `X` simply means "the first parameter" wherever it is used.

When a procedure is called, the "actual parameters" passed to it are associated with the formal parameters, on a one-for-one basis.

The BNF for the syntax looks something like this:

```
<procedure> ::= PROCEDURE <ident>
                '(' <param-list> ')' <begin-block>


<param_list> ::= <parameter> ( ',' <parameter> )* | null
```

Similarly, the procedure call looks like:

```
<proc call> ::= <ident> '(' <param-list> ')'
```

Note that there is already an implicit decision built into this syntax. Some languages, such as Pascal and Ada, permit parameter lists to be optional. If there are no parameters, you simply leave off the parens completely. Other languages, like C and Modula 2, require the parens even if the list is empty. Clearly, the example we just finished corresponds to the former point of view. But to tell the truth I prefer the latter. For procedures alone, the decision would seem to favor the "listless" approach. The statement `Initialize;`, standing alone, can only mean a procedure call. In the parsers we've been writing, we've made heavy use of parameterless procedures, and it would seem a shame to have to write an empty pair of parens for each case.

But later on we're going to be using functions, too. And since functions can appear in the same places as simple scalar identifiers, you can't tell the difference between the two. You have to go back to the declarations to find out. Some folks consider this to be an advantage. Their argument is that an identifier gets replaced by a value, and what do you care whether it's done by substitution or by a function? But we sometimes *do* care, because the function may be quite time-consuming. If, by writing a simple identifier into a given expression, we can incur a heavy run-time penalty, it seems to me we ought to be made aware of it.

Anyway, Niklaus Wirth designed both Pascal and Modula 2. I'll give him the benefit of the doubt and assume that he had a good reason for changing the rules the second time around!

Needless to say, it's an easy thing to accommodate either point of view as we design a language, so this one is strictly a matter of personal preference. Do it whichever way you like best.

Before we go any further, let's alter the translator to handle a (possibly empty) parameter list. For now we won't generate any extra code ... just parse the syntax. The code for processing the declaration has very much the same form we've seen before when dealing with `VAR`-lists:

```
{--------------------------------------------------------------}
{ Process the Formal Parameter List of a Procedure }

procedure FormalList;
begin
     Match('(');
     if Look <> ')' then begin
          FormalParam;
          while Look = ',' do begin
               Match(',');
               FormalParam;
          end;
     end;
     Match(')');
end;
{--------------------------------------------------------------}
```

Procedure `DoProc` needs to have a line added to call `FormalList`:

```
{--------------------------------------------------------------}
{ Parse and Translate a Procedure Declaration }

procedure DoProc;
var N: char;
begin
     Match('p');
     N := GetName;
     FormalList;
     Fin;
     if InTable(N) then Duplicate(N);
     ST[N] := 'p';
     PostLabel(N);
     BeginBlock;
     Return;
end;
{--------------------------------------------------------------}
```

For now, the code for `FormalParam` is just a dummy one that simply skips the parameter name:

```
{--------------------------------------------------------------}
{ Process a Formal Parameter }

procedure FormalParam;
var Name:  char;
begin
     Name := GetName;
end;
{--------------------------------------------------------------}
```

For the actual procedure call, there must be similar code to process the actual parameter list:

```
{--------------------------------------------------------------}
{ Process an Actual Parameter }

procedure Param;
var Name:  char;
begin
     Name := GetName;
end;
```

```
{-------------------------------------------------------------}
{ Process the Parameter List for a Procedure  Call }

procedure ParamList;
begin
     Match('(');
     if Look <> ')' then begin
          Param;
          while Look = ',' do begin
               Match(',');
               Param;
          end;
     end;
     Match(')');
end;


{-------------------------------------------------------------}
{ Process a Procedure Call }

procedure CallProc(Name: char);
begin
     ParamList;
     Call(Name);
end;
{-------------------------------------------------------------}
```

Note here that `CallProc` is no longer just a simple code generation routine. It has some structure to it. To handle this, I've renamed the code generation routine to just Call, and called it from within `CallProc`.

OK, if you'll add all this code to your translator and try it out, you'll find that you can indeed parse the syntax properly. I'll note in passing that there is *no* checking to make sure that the number (and, later, types) of formal and actual parameters match up. In a production compiler, we must of course do this. We'll ignore the issue now if for no other reason than that the structure of our symbol table doesn't currently give us a place to store the necessary information. Later on, we'll have a place for that data and we can deal with the issue then.

The Semantics of Parameters

So far we've dealt with the *syntax* of parameter passing, and we've got the parsing mechanisms in place to handle it. Next, we have to look at the *semantics*, i.e., the actions to be taken when we encounter parameters. This brings us square up against the issue of the different ways parameters can be passed.

There is more than one way to pass a parameter, and the way we do it can have a profound effect on the character of the language. So this is another of those areas where I can't just give you my solution. Rather, it's important that we spend some time looking at the alternatives so that you can go another route if you choose to.

There are two main ways parameters are passed:

- By value
- By reference (address)

The differences are best seen in the light of a little history.

The old FORTRAN compilers passed all parameters by reference. In other words, what was actually passed was the address of the parameter. This meant that the called subroutine was free to either read or write that parameter, as often as it chose to, just as though it were a global variable. This was actually quite an efficient way to do things, and it was pretty simple since the same mechanism was used in all cases, with one exception that I'll get to shortly.

There were problems, though. Many people felt that this method created entirely too much coupling between the called subroutine and its caller. In effect, it gave the subroutine complete access to all variables that appeared in the parameter list.

Many times, we didn't want to actually change a parameter, but only use it as an input. For example, we might pass an element count to a subroutine, and wish we could then use that count within a `DO`-loop. To avoid

changing the value in the calling program, we had to make a local copy of the input parameter, and operate only on the copy. Some FORTRAN programmers, in fact, made it a practice to copy ALL parameters except those that were to be used as return values. Needless to say, all this copying defeated a good bit of the efficiency associated with the approach.

There was, however, an even more insidious problem, which was not really just the fault of the "pass by reference" convention, but a bad convergence of several implementation decisions.

Suppose we have a subroutine `SUBROUTINE FOO(X, Y, N)`, where N is some kind of input count or flag. Many times, we'd like to be able to pass a literal or even an expression in place of a variable, such as `CALL FOO( A, B, J + 1)`. Here the third parameter is not a variable, and so it has no address. The earliest FORTRAN compilers did not allow such things, so we had to resort to subterfuges like:

```
K = J + 1
CALL FOO(A, B, K)
```

Here again, there was copying required, and the burden was on the programmer to do it. Not good.

Later FORTRAN implementations got rid of this by allowing expressions as parameters. What they did was to assign a compiler-generated variable, store the value of the expression in the variable, and then pass the address of the expression.

So far, so good. Even if the subroutine mistakenly altered the anonymous variable, who was to know or care? On the next call, it would be recalculated anyway.

The problem arose when someone decided to make things more efficient. They reasoned, rightly enough, that the most common kind of "expression" was a single integer value, as in `CALL FOO(A, B, 4)`.

It seemed inefficient to go to the trouble of "computing" such an integer and storing it in a temporary variable, just to pass it through the calling list. Since we had to pass the address of the thing anyway, it seemed to make lots of sense to just pass the address of the literal integer, 4 in the example above.

To make matters more interesting, most compilers, then and now, identify all literals and store them separately in a "literal pool," so that we only have to store one value for each unique literal. That combination of design decisions: passing expressions, optimization for literals as a special case, and use of a literal pool, is what led to disaster.

To see how it works, imagine that we call subroutine `FOO` as in the example above, passing it a literal 4. Actually, what gets passed is the address of the literal 4, which is stored in the literal pool. This address corresponds to the formal parameter, K, in the subroutine itself.

Now suppose that, unbeknownst to the programmer, subroutine `FOO` actually modifies `K` to be, say, `-7`. Suddenly, that literal 4 in the literal pool gets *changed*, to a `-7`. From then on, every expression that uses a 4 and every subroutine that passes a 4 will be using the value of `-7` instead! Needless to say, this can lead to some bizarre and difficult-to-find behavior. The whole thing gave the concept of pass-by-reference a bad name, although as we have seen, it was really a combination of design decisions that led to the problem.

In spite of the problem, the FORTRAN approach had its good points. Chief among them is the fact that we don't have to support multiple mechanisms. The same scheme, passing the address of the argument, works for *every* case, including arrays. So the size of the compiler can be reduced.

Partly because of the FORTRAN gotcha, and partly just because of the reduced coupling involved, modern languages like C, Pascal, Ada, and Modula 2 generally pass scalars by value.

This means that the value of the scalar is COPIED into a separate value used only for the call. Since the value passed is a copy, the called procedure can use it as a local variable and modify it any way it likes. The value in the caller will not be changed.

It may seem at first that this is a bit inefficient, because of the need to copy the parameter. But remember that we're going to have to fetch *some* value to pass anyway, whether it be the parameter itself or an address for it. Inside the subroutine, using pass-by-value is definitely more efficient, since we eliminate one level of indirection. Finally, we saw earlier that with FORTRAN, it was often necessary to make copies within the subroutine anyway, so pass-by-value reduces the number of local variables. All in all, pass-by-value is better.

Except for one small little detail: if all parameters are passed by value, there is no way for a called to procedure to return a result to its caller! The parameter passed is *not* altered in the caller, only in the called procedure. Clearly, that won't get the job done.

There have been two answers to this problem, which are equivalent. In Pascal, Wirth provides for `VAR` parameters, which are passed-by-reference. What a `VAR` parameter is, in fact, is none other than our old friend the FORTRAN parameter, with a new name and paint job for disguise. Wirth neatly gets around the "changing a literal" problem as well as the "address of an expression" problem, by the simple expedient of allowing only a variable to be the actual parameter. In other words, it's the same restriction that the earliest FORTRANs imposed.

C does the same thing, but explicitly. In C, *all* parameters are passed by value. One kind of variable that C supports, however, is the pointer. So by passing a pointer by value, you in effect pass what it points to by reference. In some ways this works even better yet, because even though you can change the variable pointed

to all you like, you still *can't* change the pointer itself. In a function such as strcpy, for example, where the pointers are incremented as the string is copied, we are really only incrementing copies of the pointers, so the values of those pointers in the calling procedure still remain as they were. To modify a pointer, you must pass a pointer to the pointer.

Since we are simply performing experiments here, we'll look at *both* pass-by-value and pass-by-reference. That way, we'll be able to use either one as we need to. It's worth mentioning that it's going to be tough to use the C approach to pointers here, since a pointer is a different type and we haven't studied types yet!

Pass-by-Value

Let's just try some simple-minded things and see where they lead us. Let's begin with the pass-by-value case. Consider the procedure call `FOO(X, Y)`.

Almost the only reasonable way to pass the data is through the CPU stack. So the code we'd like to see generated might look something like this:

```
MOVE X(PC),-(SP)     ; Push X
MOVE Y(PC),-(SP)     ; Push Y
BSR FOO              ; Call FOO
```

That certainly doesn't seem too complex!

When the `BSR` is executed, the CPU pushes the return address onto the stack and jumps to `FOO`. At this point the stack will look like this:

```
          .
          .
          Value of X (2 bytes)
          Value of Y (2 bytes)
 SP -->   Return Address (4 bytes)
```

So the values of the parameters have addresses that are fixed offsets from the stack pointer. In this example, the addresses are:

- X: 6(SP)
- Y: 4(SP)

Now consider what the called procedure might look like:

```
PROCEDURE FOO(A, B)
BEGIN
     A = B
END
```

(Remember, the names of the formal parameters are arbitrary ... only the positions count.)

The desired output code might look like:

```
FOO: MOVE 4(SP),D0
     MOVE D0,6(SP)
     RTS
```

Note that, in order to address the formal parameters, we're going to have to know which position they have in the parameter list. This means some changes to the symbol table stuff. In fact, for our single-character case it's best to just create a new symbol table for the formal parameters.

Let's begin by declaring a new table:

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

We also will need to keep track of how many parameters a given procedure has:

```
var NumParams: integer;
```

And we need to initialize the new table. Now, remember that the formal parameter list will be different for each procedure that we process, so we'll need to initialize that table anew for each procedure. Here's the initializer:

```
{-------------------------------------------------------------}
{ Initialize Parameter Table to Null }

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

We'll put a call to this procedure in `Init`, and also at the end of `DoProc`:

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

procedure Init;
var i: char;
begin
     GetChar;
     SkipWhite;
     for i := 'A' to 'Z' do
          ST[i] := ' ';
     ClearParams;
end;
{-------------------------------------------------------------}
.
.
.
{-------------------------------------------------------------}
{ Parse and Translate a Procedure Declaration }

procedure DoProc;
var N: char;
begin
     Match('p');
     N := GetName;
     FormalList;
     Fin;
     if InTable(N) then Duplicate(N);
     ST[N] := 'p';
     PostLabel(N);
     BeginBlock;
     Return;
     ClearParams;
end;
{-------------------------------------------------------------}
```

Note that the call within `DoProc` ensures that the table will be clear when we're in the main program.

OK, now we need a few procedures to work with the table. The next few functions are essentially copies of `InTable`, `TypeOf`, etc.:

```
{-------------------------------------------------------------}
{ Find the Parameter Number }
```

```
function ParamNumber(N: char): integer;
begin
     ParamNumber := Params[N];
end;


{---------------------------------------------------------------}
{ See if an Identifier is a Parameter }

function IsParam(N: char): boolean;
begin
     IsParam := Params[N] <> 0;
end;


{---------------------------------------------------------------}
{ Add a New Parameter to Table }

procedure AddParam(Name: char);
begin
     if IsParam(Name) then Duplicate(Name);
     Inc(NumParams);
     Params[Name] := NumParams;
end;
{---------------------------------------------------------------}
```

Finally, we need some code generation routines:

```
{---------------------------------------------------------------}
{ Load a Parameter to the Primary Register }

procedure LoadParam(N: integer);
var Offset: integer;
begin
     Offset := 4 + 2 * (NumParams - N);
     Emit('MOVE ');
     WriteLn(Offset, '(SP),D0');
end;


{---------------------------------------------------------------}
{ Store a Parameter from the Primary Register }

procedure StoreParam(N: integer);
var Offset: integer;
begin
     Offset := 4 + 2 * (NumParams - N);
     Emit('MOVE D0,');
     WriteLn(Offset, '(SP)');
end;


{---------------------------------------------------------------}
{ Push The Primary Register to the Stack }

procedure Push;
begin
     EmitLn('MOVE D0,-(SP)');
end;
{---------------------------------------------------------------}
```

(The last routine is one we've seen before, but it wasn't in this vestigial version of the program.)

With those preliminaries in place, we're ready to deal with the semantics of procedures with calling lists (remember, the code to deal with the syntax is already in place).

Let's begin by processing a formal parameter. All we have to do is to add each parameter to the parameter symbol table:

```
{-------------------------------------------------------------}
{ Process a Formal Parameter }

procedure FormalParam;
begin
     AddParam(GetName);
end;
{-------------------------------------------------------------}
```

Now, what about dealing with a formal parameter when it appears in the body of the procedure? That takes a little more work. We must first determine that it *is* a formal parameter. To do this, I've written a modified version of `TypeOf`:

```
{-------------------------------------------------------------}
{ Get Type of Symbol }

function TypeOf(n: char): char;
begin
     if IsParam(n) then
          TypeOf := 'f'
     else
          TypeOf := ST[n];
end;
{-------------------------------------------------------------}
```

(Note that, since `TypeOf` now calls `IsParam`, it may need to be relocated in your source.)

We also must modify `AssignOrProc` to deal with this new type:

```
{-------------------------------------------------------------}
{ Decide if a Statement is an Assignment or Procedure Call }

procedure AssignOrProc;
var Name: char;
begin
     Name := GetName;
     case TypeOf(Name) of
          ' ': Undefined(Name);
          'v', 'f': Assignment(Name);
          'p': CallProc(Name);
          else  Abort('Identifier ' + Name +  '  Cannot  Be  Used
Here');
     end;
end;
{-------------------------------------------------------------}
```

Finally, the code to process an assignment statement and an expression must be extended:

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

procedure Expression;
var Name: char;
begin
     Name := GetName;
     if IsParam(Name) then
```

```
          LoadParam(ParamNumber(Name))
     else
          LoadVar(Name);
end;


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

procedure Assignment(Name: char);
begin
     Match('=');
     Expression;
     if IsParam(Name) then
          StoreParam(ParamNumber(Name))
     else
          StoreVar(Name);
end;
{---------------------------------------------------------------}
```

As you can see, these procedures will treat every variable name encountered as either a formal parameter or a global variable, depending on whether or not it appears in the parameter symbol table. Remember that we are using only a vestigial form of `Expression`. In the final program, the change shown here will have to be added to `Factor`, not `Expression`.

The rest is easy. We need only add the semantics to the actual procedure call, which we can do with one new line of code:

```
{---------------------------------------------------------------}
{ Process an Actual Parameter }

procedure Param;
begin
     Expression;
     Push;
end;
{---------------------------------------------------------------}
```

That's it. Add these changes to your program and give it a try. Try declaring one or two procedures, each with a formal parameter list. Then do some assignments, using combinations of global and formal parameters. You can call one procedure from within another, but you cannot *declare* a nested procedure. You can even pass formal parameters from one procedure to another. If we had the full syntax of the language here, you'd also be able to do things like read or write formal parameters or use them in complicated expressions.

What's Wrong?

At this point, you might be thinking: Surely there's more to this than a few pushes and pops. There must be more to passing parameters than this.

You'd be right. As a matter of fact, the code that we're generating here leaves a lot to be desired in several respects.

The most glaring oversight is that it's wrong! If you'll look back at the code for a procedure call, you'll see that the caller pushes each actual parameter onto the stack before it calls the procedure. The procedure *uses* that information, but it doesn't change the stack pointer. That means that the stuff is still there when we return. *Somebody* needs to clean up the stack, or we'll soon be in very hot water!

Fortunately, that's easily fixed. All we have to do is to increment the stack pointer when we're finished.

Should we do that in the calling program, or the called procedure? Some folks let the called procedure clean up the stack, since that requires less code to be generated per call, and since the procedure, after all, knows how many parameters it's got. But that means that it must do something with the return address so as not to lose it.

I prefer letting the caller clean up, so that the callee need only execute a return. Also, it seems a bit more balanced, since the caller is the one who "messed up" the stack in the first place. But *that* means that the caller

must remember how many items it pushed. To make things easy, I've modified the procedure `ParamList` to be a function instead of a procedure, returning the number of bytes pushed:

```
{--------------------------------------------------------------}
{ Process the Parameter List for a Procedure  Call }

function ParamList: integer;
var N: integer;
begin
     N := 0;
     Match('(');
     if Look <> ')' then begin
          Param;
          inc(N);
          while Look = ',' do begin
               Match(',');
               Param;
               inc(N);
          end;
     end;
     Match(')');
     ParamList := 2 * N;
end;
{--------------------------------------------------------------}
```

Procedure `CallProc` then uses this to clean up the stack:

```
{--------------------------------------------------------------}
{ Process a Procedure Call }

procedure CallProc(Name: char);
var N: integer;
begin
     N := ParamList;
     Call(Name);
     CleanStack(N);
end;
{--------------------------------------------------------------}
```

Here I've created yet another code generation procedure:

```
{--------------------------------------------------------------}
{ Adjust the Stack Pointer Upwards by N Bytes }

procedure CleanStack(N: integer);
begin
     if N > 0 then begin
          Emit('ADD #');
          WriteLn(N, ',SP');
     end;
end;
{--------------------------------------------------------------}
```

OK, if you'll add this code to your compiler, I think you'll find that the stack is now under control.

The next problem has to do with our way of addressing relative to the stack pointer. That works fine in our simple examples, since with our rudimentary form of expressions nobody else is messing with the stack. But consider a different example as simple as:

```
PROCEDURE FOO(A, B)
BEGIN
     A = A + B
END
```

The code generated by a simple-minded parser might be:

```
FOO: MOVE 6(SP),D0       ; Fetch A
     MOVE D0,-(SP)       ; Push it
     MOVE 4(SP),D0       ; Fetch B
     ADD (SP)+,D0        ; Add A
     MOVE D0,6(SP)       : Store A
     RTS
```

This would be wrong. When we push the first argument onto the stack, the offsets for the two formal parameters are no longer 4 and 6, but are 6 and 8. So the second fetch would fetch A again, not B.

This is not the end of the world. I think you can see that all we really have to do is to alter the offset every time we do a push, and that in fact is what's done if the CPU has no support for other methods.

Fortunately, though, the 68000 does have such support. Recognizing that this CPU would be used a lot with high-order language compilers, Motorola decided to add direct support for this kind of thing.

The problem, as you can see, is that as the procedure executes, the stack pointer bounces up and down, and so it becomes an awkward thing to use as a reference to access the formal parameters. The solution is to define some *other* register, and use it instead. This register is typically set equal to the original stack pointer, and is called the frame pointer.

The 68000 instruction set `LINK` lets you declare such a frame pointer, and sets it equal to the stack pointer, all in one instruction. As a matter of fact, it does even more than that. Since this register may have been in use for something else in the calling procedure, `LINK` also pushes the current value of that register onto the stack. It can also add a value to the stack pointer, to make room for local variables.

The complement of `LINK` is `UNLK`, which simply restores the stack pointer and pops the old value back into the register.

Using these two instructions, the code for the previous example becomes:

```
FOO: LINK A6,#0
     MOVE 10(A6),D0      ; Fetch A
     MOVE D0,-(SP)       ; Push it
     MOVE 8(A6),D0       ; Fetch B
     ADD (SP)+,D0        ; Add A
     MOVE D0,10(A6)      : Store A
     UNLK A6
     RTS
```

Fixing the compiler to generate this code is a lot easier than it is to explain it. All we need to do is to modify the code generation created by `DoProc`. Since that makes the code a little more than one line, I've created new procedures to deal with it, paralleling the `Prolog` and `Epilog` procedures called by `DoMain`:

```
{---------------------------------------------------------------}
{ Write the Prolog for a Procedure }

procedure ProcProlog(N: char);
begin
     PostLabel(N);
     EmitLn('LINK A6,#0');
end;


{---------------------------------------------------------------}
{ Write the Epilog for a Procedure }

procedure ProcEpilog;
begin
     EmitLn('UNLK A6');
     EmitLn('RTS');
end;
```

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

Procedure `DoProc` now just calls these:

```
{-------------------------------------------------------------}
{ Parse and Translate a Procedure Declaration }

procedure DoProc;
var N: char;
begin
     Match('p');
     N := GetName;
     FormalList;
     Fin;
     if InTable(N) then Duplicate(N);
     ST[N] := 'p';
     ProcProlog(N);
     BeginBlock;
     ProcEpilog;
     ClearParams;
end;
{-------------------------------------------------------------}
```

Finally, we need to change the references to `SP` in procedures `LoadParam` and `StoreParam`:

```
{-------------------------------------------------------------}
{ Load a Parameter to the Primary Register }

procedure LoadParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 2 * (NumParams - N);
     Emit('MOVE ');
     WriteLn(Offset, '(A6),D0');
end;


{-------------------------------------------------------------}
{ Store a Parameter from the Primary Register }

procedure StoreParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 2 * (NumParams - N);
     Emit('MOVE D0,');
     WriteLn(Offset, '(A6)');
end;
{-------------------------------------------------------------}
```

(Note that the `Offset` computation changes to allow for the extra push of `A6`.)

That's all it takes. Try this out and see how you like it.

At this point we are generating some relatively nice code for procedures and procedure calls. Within the limitation that there are no local variables (yet) and that no procedure nesting is allowed, this code is just what we need.

There is still just one little small problem remaining:

WE HAVE NO WAY TO RETURN RESULTS TO THE CALLER!

But that, of course, is not a limitation of the code we're generating, but one inherent in the call-by-value protocol. Notice that we *can* use formal parameters in any way inside the procedure. We can calculate new values for them, use them as loop counters (if we had loops, that is!), etc. So the code is doing what it's supposed to. To get over this last problem, we need to look at the alternative protocol.

Call-by-Reference

This one is easy, now that we have the mechanisms already in place. We only have to make a few changes to the code generation. Instead of pushing a value onto the stack, we must push an address. As it turns out, the 68000 has an instruction, `PEA`, that does just that.

We'll be making a new version of the test program for this. Before we do anything else,

MAKE A COPY of the program as it now stands, because we'll be needing it again later.

Let's begin by looking at the code we'd like to see generated for the new case. Using the same example as before, we need the call

```
FOO(X, Y)
```

to be translated to:

```
PEA X(PC)           ; Push the address of X
PEA Y(PC)           ; Push Y the address of Y
BSR FOO             ; Call FOO
```

That's a simple matter of a slight change to Param:

```
{---------------------------------------------------------------}
{ Process an Actual Parameter }

procedure Param;
begin
     EmitLn('PEA ' + GetName + '(PC)');
end;
{---------------------------------------------------------------}
```

(Note that with pass-by-reference, we can't have expressions in the calling list, so `Param` can just read the name directly.)

At the other end, the references to the formal parameters must be given one level of indirection:

```
FOO: LINK A6,#0
     MOVE.L 12(A6),A0    ; Fetch the address of A
     MOVE (A0),D0        ; Fetch A
     MOVE D0,-(SP)       ; Push it
     MOVE.L 8(A6),A0     ; Fetch the address of B
     MOVE (A0),D0        ; Fetch B
     ADD (SP)+,D0        ; Add A
     MOVE.L 12(A6),A0    ; Fetch the address of A
     MOVE D0,(A0)        : Store A
     UNLK A6
     RTS
```

All of this can be handled by changes to `LoadParam` and `StoreParam`:

```
{---------------------------------------------------------------}
{ Load a Parameter to the Primary Register }

procedure LoadParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 4 * (NumParams - N);
     Emit('MOVE.L ');
     WriteLn(Offset, '(A6),A0');
     EmitLn('MOVE (A0),D0');
end;


{---------------------------------------------------------------}
{ Store a Parameter from the Primary Register }
```

```
procedure StoreParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 4 * (NumParams - N);
     Emit('MOVE.L ');
     WriteLn(Offset, '(A6),A0');
     EmitLn('MOVE D0,(A0)');
end;
{--------------------------------------------------------------}
```

To get the count right, we must also change one line in `ParamList`:

```
ParamList := 4 * N;
```

That should do it. Give it a try and see if it's generating reasonable-looking code. As you will see, the code is hardly optimal, since we reload the address register every time a parameter is needed. But that's consistent with our KISS approach here, of just being sure to generate code that works. We'll just make a little note here, that here's yet another candidate for optimization, and press on.

Now we've learned to process parameters using pass-by-value and pass-by-reference. In the real world, of course, we'd like to be able to deal with *both* methods. We can't do that yet, though, because we have not yet had a session on types, and that has to come first.

If we can only have *one* method, then of course it has to be the good ol' FORTRAN method of pass-by-reference, since that's the only way procedures can ever return values to their caller.

This, in fact, will be one of the differences between TINY and KISS. In the next version of TINY, we'll use pass-by-reference for all parameters. KISS will support both methods.

Local Variables

So far, we've said nothing about local variables, and our definition of procedures doesn't allow for them. Needless to say, that's a big gap in our language, and one that needs to be corrected.

Here again we are faced with a choice: Static or dynamic storage?

In those old FORTRAN programs, local variables were given static storage just like global ones. That is, each local variable got a name and allocated address, like any other variable, and was referenced by that name.

That's easy for us to do, using the allocation mechanisms already in place. Remember, though, that local variables can have the same names as global ones. We need to somehow deal with that by assigning unique names for these variables.

The characteristic of static storage, of course, is that the data survives a procedure call and return. When the procedure is called again, the data will still be there. That can be an advantage in some applications. In the FORTRAN days we used to do tricks like initialize a flag, so that you could tell when you were entering a procedure for the first time and could do any one-time initialization that needed to be done.

Of course, the same "feature" is also what makes recursion impossible with static storage. Any new call to a procedure will overwrite the data already in the local variables.

The alternative is dynamic storage, in which storage is allocated on the stack just as for passed parameters. We also have the mechanisms already for doing this. In fact, the same routines that deal with passed (by value) parameters on the stack can easily deal with local variables as well ... the code to be generated is the same. The purpose of the offset in the 68000 `LINK` instruction is there just for that reason: we can use it to adjust the stack pointer to make room for locals. Dynamic storage, of course, inherently supports recursion.

When I first began planning TINY, I must admit to being prejudiced in favor of static storage. That's simply because those old FORTRAN programs were pretty darned efficient ... the early FORTRAN compilers produced a quality of code that's still rarely matched by modern compilers. Even today, a given program written in FORTRAN is likely to outperform the same program written in C or Pascal, sometimes by wide margins. (Whew! Am I going to hear about *that* statement!)

I've always supposed that the reason had to do with the two main differences between FORTRAN implementations and the others: static storage and pass-by-reference. I know that dynamic storage supports recursion, but it's always seemed to me a bit peculiar to be willing to accept slower code in the 95% of cases that don't need recursion, just to get that feature when you need it. The idea is that, with static storage, you can use absolute addressing rather than indirect addressing, which should result in faster code.

More recently, though, several folks have pointed out to me that there really is no performance penalty associated with dynamic storage. With the 68000, for example, you shouldn't use absolute addressing anyway

... most operating systems require position independent code. And the 68000 instruction `MOVE 8(A6),D0` has exactly the same timing as `MOVE X(PC),D0`.

So I'm convinced, now, that there is no good reason *not* to use dynamic storage.

Since this use of local variables fits so well into the scheme of pass-by-value parameters, we'll use that version of the translator to illustrate it. (I *sure* hope you kept a copy!)

The general idea is to keep track of how many local parameters there are. Then we use the integer in the `LINK` instruction to adjust the stack pointer downward to make room for them. Formal parameters are addressed as positive offsets from the frame pointer, and locals as negative offsets. With a little bit of work, the same procedures we've already created can take care of the whole thing.

Let's start by creating a new variable, Base:

```
var Base: integer;
```

We'll use this variable, instead of NumParams, to compute stack offsets. That means changing the two references to `NumParams` in `LoadParam` and `StoreParam`:

```
{---------------------------------------------------------------}
{ Load a Parameter to the Primary Register }

procedure LoadParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 2 * (Base - N);
     Emit('MOVE ');
     WriteLn(Offset, '(A6),D0');
end;


{---------------------------------------------------------------}
{ Store a Parameter from the Primary Register }

procedure StoreParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 2 * (Base - N);
     Emit('MOVE D0,');
     WriteLn(Offset, '(A6)');
end;
{---------------------------------------------------------------}
```

The idea is that the value of Base will be frozen after we have processed the formal parameters, and won't increase further as the new, local variables, are inserted in the symbol table. This is taken care of at the end of `FormalList`:

```
{---------------------------------------------------------------}
{ Process the Formal Parameter List of a Procedure }

procedure FormalList;
begin
     Match('(');
     if Look <> ')' then begin
          FormalParam;
          while Look = ',' do begin
               Match(',');
               FormalParam;
          end;
     end;
     Match(')');
     Fin;
     Base := NumParams;
     NumParams := NumParams + 4;
end;
```

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

(We add four words to make allowances for the return address and old frame pointer, which end up between the formal parameters and the locals.)

About all we need to do next is to install the semantics for declaring local variables into the parser. The routines are very similar to `Decl` and `TopDecls`:

```
{--------------------------------------------------------------}
{ Parse and Translate a Local Data Declaration }

procedure LocDecl;
var Name: char;
begin
   Match('v');
     AddParam(GetName);
     Fin;
end;


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


{ Parse and Translate Local Declarations }

function LocDecls: integer;
var n: integer;
begin
     n := 0;
     while Look = 'v' do begin
          LocDecl;
          inc(n);
     end;
     LocDecls := n;
end;
{--------------------------------------------------------------}
```

Note that `LocDecls` is a `FUNCTION`, returning the number of locals to `DoProc`.

Next, we modify `DoProc` to use this information:

```
{--------------------------------------------------------------}
{ Parse and Translate a Procedure Declaration }

procedure DoProc;
var N: char;
      k: integer;
begin
     Match('p');
     N := GetName;
     if InTable(N) then Duplicate(N);
     ST[N] := 'p';
     FormalList;
     k := LocDecls;
     ProcProlog(N, k);
     BeginBlock;
     ProcEpilog;
     ClearParams;
end;
{--------------------------------------------------------------}
```

(I've made a couple of changes here that weren't really necessary. Aside from rearranging things a bit, I moved the call to `Fin` to within `FormalList`, and placed one inside `LocDecls` as well. Don't forget to put one at the end of `FormalList`, so that we're together here.)

Note the change in the call to `ProcProlog`. The new argument is the number of `WORDS` (not bytes) to allocate space for. Here's the new version of ProcProlog:

```
{-------------------------------------------------------------}
{ Write the Prolog for a Procedure }

procedure ProcProlog(N: char; k: integer);
begin
     PostLabel(N);
     Emit('LINK A6,#');
     WriteLn(-2 * k)
end;
{-------------------------------------------------------------}
```

That should do it. Add these changes and see how they work.

Conclusion

At this point you know how to compile procedure declarations and procedure calls, with parameters passed by reference and by value. You can also handle local variables. As you can see, the hard part is not in providing the mechanisms, but in deciding just which mechanisms to use. Once we make these decisions, the code to translate the constructs is really not that difficult. I didn't show you how to deal with the combination of local parameters and pass-by-reference parameters, but that's a straightforward extension to what you've already seen. It just gets a little more messy, that's all, since we need to support both mechanisms instead of just one at a time. I'd prefer to save that one until after we've dealt with ways to handle different variable types.

That will be the next installment, which will be coming soon to a Forum near you. See you then.

Chapter 14

Part XIV: Types - 26 May 1990

Introduction

In the last installment (Part XIII: PROCEDURES) I mentioned that in that part and this one, we would cover the two features that tend to separate the toy language from a real, usable one. We covered procedure calls in that installment. Many of you have been waiting patiently, since August '89, for me to drop the other shoe. Well, here it is.

In this installment, we'll talk about how to deal with different data types. As I did in the last segment, I will *not* incorporate these features directly into the TINY compiler at this time. Instead, I'll be using the same approach that has worked so well for us in the past: using only fragments of the parser and single-character tokens. As usual, this allows us to get directly to the heart of the matter without having to wade through a lot of unnecessary code. Since the major problems in dealing with multiple types occur in the arithmetic operations, that's where we'll concentrate our focus.

A few words of warning: First, there are some types that I will NOT be covering in this installment. Here we will *only* be talking about the simple, predefined types. We won't even deal with arrays, pointers or strings in this installment; I'll be covering them in the next few.

Second, we also will not discuss user-defined types. That will not come until much later, for the simple reason that I still haven't convinced myself that user-defined types belong in a language named KISS. In later installments, I do intend to cover at least the general concepts of user-defined types, records, etc., just so that the series will be complete. But whether or not they will be included as part of KISS is still an open issue. I am open to comments or suggestions on this question.

Finally, I should warn you: what we are about to do *can* add considerable extra complication to both the parser and the generated code. Handling variables of different types is straightforward enough. The complexity comes in when you add rules about conversion between types. In general, you can make the compiler as simple or as complex as you choose to make it, depending upon the way you define the type-conversion rules. Even if you decide not to allow *any* type conversions (as in Ada, for example) the problem is still there, and is built into the mathematics. When you multiply two short numbers, for example, you can get a long result.

I've approached this problem very carefully, in an attempt to Keep It Simple. But we can't avoid the complexity entirely. As has so often has happened, we end up having to trade code quality against complexity, and as usual I will tend to opt for the simplest approach.

What's Coming Next?

Before diving into the tutorial, I think you'd like to know where we are going from here ... especially since it's been so long since the last installment.

I have not been idle in the meantime. What I've been doing is reorganizing the compiler itself into Turbo Units. One of the problems I've encountered is that as we've covered new areas and thereby added features to the TINY compiler, it's been getting longer and longer. I realized a couple of installments back that this was causing trouble, and that's why I've gone back to using only compiler fragments for the last installment and this one. The problem is that it just seems dumb to have to reproduce the code for, say, processing boolean exclusive ORs, when the subject of the discussion is parameter passing.

The obvious way to have our cake and eat it, too, is to break up the compiler into separately compilable modules, and of course the Turbo Unit is an ideal vehicle for doing this. This allows us to hide some fairly complex code (such as the full arithmetic and boolean expression parsing) into a single unit, and just pull it in whenever it's needed. In that way, the only code I'll have to reproduce in these installments will be the code that actually relates to the issue under discussion.

I've also been toying with Turbo 5.5, which of course includes the Borland object-oriented extensions to Pascal. I haven't decided whether to make use of these features, for two reasons. First of all, many of you who have been following this series may still not have 5.5, and I certainly don't want to force anyone to have to go out and buy a new compiler just to complete the series. Secondly, I'm not convinced that the O-O extensions have all that much value for this application. We've been having some discussions about that in CompuServe's CLM forum, and so far we've not found any compelling reason to use O-O constructs. This is another of those areas where I could use some feedback from you readers. Anyone want to vote for Turbo 5.5 and O-O?

In any case, after the next few installments in the series, the plan is to upload to you a complete set of Units, and complete functioning compilers as well. The plan, in fact, is to have *three* compilers: One for a single-character version of TINY (to use for our experiments), one for TINY and one for KISS. I've pretty much isolated the differences between TINY and KISS, which are these:

- TINY will support only two data types: The character and the 16-bit integer. I may also try to do something with strings, since without them a compiler would be pretty useless. KISS will support all the usual simple types, including arrays and even floating point.
- TINY will only have two control constructs, the `IF` and the `WHILE`. KISS will support a very rich set of constructs, including one we haven't discussed here before ... the `CASE`.
- KISS will support separately compilable modules.

One caveat: Since I still don't know much about 80x86 assembler language, all these compiler modules will still be written to support 68000 code. However, for the programs I plan to upload, all the code generation has been carefully encapsulated into a single unit, so that any enterprising student should be able to easily retarget to any other processor. This task is "left as an exercise for the student." I'll make an offer right here and now: For the person who provides us the first robust retarget to 80x86, I will be happy to discuss shared copyrights and royalties from the book that's upcoming.

But enough talk. Let's get on with the study of types. As I said earlier, we'll do this one as we did in the last installment: by performing experiments using single-character tokens.

The Symbol Table

It should be apparent that, if we're going to deal with variables of different types, we're going to need someplace to record what those types are. The obvious vehicle for that is the symbol table, and we've already used it that way to distinguish, for example, between local and global variables, and between variables and procedures.

The symbol table structure for single-character tokens is particularly simple, and we've used it several times before. To deal with it, we'll steal some procedures that we've used before.

First, we need to declare the symbol table itself:

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

var Look: char;              { Lookahead Character }

    ST: Array['A'..'Z'] of char;   {  *** ADD THIS LINE ***}
{--------------------------------------------------------------}
```

Next, we need to make sure it's initialized as part of procedure `Init`:

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

procedure Init;
var i: char;
begin
   for i := 'A' to 'Z' do
      ST[i] := '?';
   GetChar;
end;
{--------------------------------------------------------------}
```

We don't really need the next procedure, but it will be helpful for debugging. All it does is to dump the contents of the symbol table:

```
{--------------------------------------------------------------}
{ Dump the Symbol Table }

procedure DumpTable;
var i: char;
begin
   for i := 'A' to 'Z' do
      WriteLn(i, ' ', ST[i]);
end;
{--------------------------------------------------------------}
```

It really doesn't matter much where you put this procedure ... I plan to cluster all the symbol table routines together, so I put mine just after the error reporting procedures.

If you're the cautious type (as I am), you might want to begin with a test program that does nothing but initializes, then dumps the table. Just to be sure that we're all on the same wavelength here, I'm reproducing the entire program below, complete with the new procedures. Note that this version includes support for white space:

```
{--------------------------------------------------------------}
program Types;

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

const TAB = ^I;
      CR  = ^M;
      LF  = ^J;

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

var Look: char;              { Lookahead Character }

    ST: Array['A'..'Z'] of char;


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


{--------------------------------------------------------------}
{ Dump the Symbol Table }

procedure DumpTable;
var i: char;
begin
   for i := 'A' to 'Z' do
        WriteLn(i, ' ', ST[i]);
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 Character }

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 a Mulop }

function IsMulop(c: char): boolean;
begin
   IsMulop := c in ['*', '/'];
end;


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

function IsOrop(c: char): boolean;
begin
   IsOrop := c in ['|', '~'];
end;


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

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


{---------------------------------------------------------------}
{ Skip Over an End-of-Line }

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


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

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


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

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


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

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


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

procedure Init;
var i: char;
begin
   for i := 'A' to 'Z' do
      ST[i] := '?';
   GetChar;
   SkipWhite;
```

```
end;


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

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

OK, run this program. You should get a (very fast) printout of all the letters of the alphabet (potential identifiers), each followed by a question mark. Not very exciting, but it's a start.

Of course, in general we only want to see the types of the variables that have been defined. We can eliminate the others by modifying `DumpTable` with an `IF` test. Change the loop to read:

```
for i := 'A' to 'Z' do
   if ST[i] <> '?' then
       WriteLn(i, ' ', ST[i]);
```

Now, run the program again. What did you get?

Well, that's even more boring than before! There was no output at all, since at this point *none* of the names have been declared. We can spice things up a bit by inserting some statements declaring some entries in the main program. Try these:

```
ST['A'] := 'a';
ST['P'] := 'b';
ST['X'] := 'c';
```

This time, when you run the program, you should get an output showing that the symbol table is working right.

Adding Entries

Of course, writing to the table directly is pretty poor practice, and not one that will help us much later. What we need is a procedure to add entries to the table. At the same time, we know that we're going to need to test the table, to make sure that we aren't redeclaring a variable that's already in use (easy to do with only 26 choices!). To handle all this, enter the following new procedures:

```
{-------------------------------------------------------------}
{ Report Type of a Variable }


function TypeOf(N: char): char;
begin
   TypeOf := ST[N];
end;


{-------------------------------------------------------------}
{ Report if a Variable is in the Table }


function InTable(N: char): boolean;
begin
   InTable := TypeOf(N) <> '?';
end;


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

```
{ Check for a Duplicate Variable Name }

procedure CheckDup(N: char);
begin
   if InTable(N) then Abort('Duplicate Name ' + N);
end;


{---------------------------------------------------------------}
{ Add Entry to Table }

procedure AddEntry(N, T: char);
begin
   CheckDup(N);
   ST[N] := T;
end;
{---------------------------------------------------------------}
```

Now change the three lines in the main program to read:

```
AddEntry('A', 'a');
AddEntry('P', 'b');
AddEntry('X', 'c');
```

and run the program again. Did it work? Then we have the symbol table routines needed to support our work on types. In the next section, we'll actually begin to use them.

Allocating Storage

In other programs like this one, including the TINY compiler itself, we have already addressed the issue of declaring global variables, and the code generated for them. Let's build a vestigial version of a "compiler" here, whose only function is to allow us declare variables. Remember, the syntax for a declaration is:

```
<data decl> ::= VAR <identifier>
```

Again, we can lift a lot of the code from previous programs. The following are stripped-down versions of those procedures. They are greatly simplified since I have eliminated niceties like variable lists and initializers. In procedure `Alloc`, note that the new call to `AddEntry` will also take care of checking for duplicate declarations:

```
{---------------------------------------------------------------}
{ Allocate Storage for a Variable }

procedure Alloc(N: char);
begin
   AddEntry(N, 'v');
   WriteLn(N, ':', TAB, 'DC 0');
end;


{---------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
var Name: char;
begin
   Match('v');
   Alloc(GetName);
end;


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

```
{ Parse and Translate Global Declarations }

procedure TopDecls;
begin
   while Look <> '.' do begin
      case Look of
        'v': Decl;
      else Abort('Unrecognized Keyword ' + Look);
      end;
      Fin;
   end;
end;
{--------------------------------------------------------------}
```

Now, in the main program, add a call to `TopDecls` and run the program. Try allocating a few variables, and note the resulting code generated. This is old stuff for you, so the results should look familiar. Note from the code for `TopDecls` that the program is ended by a terminating period.

While you're at it, try declaring two variables with the same name, and verify that the parser catches the error.

Declaring Types

Allocating storage of different sizes is as easy as modifying procedure `TopDecls` to recognize more than one keyword. There are a number of decisions to be made here, in terms of what the syntax should be, etc., but for now I'm going to duck all the issues and simply declare by executive fiat that our syntax will be:

```
<data decl> ::= <typename>  <identifier>

<typename> ::= BYTE | WORD | LONG
```

(By an amazing coincidence, the first letters of these names happen to be the same as the 68000 assembly code length specifications, so this choice saves us a little work.)

We can create the code to take care of these declarations with only slight modifications. In the routines below, note that I've separated the code generation parts of `Alloc` from the logic parts. This is in keeping with our desire to encapsulate the machine-dependent part of the compiler.

```
{--------------------------------------------------------------}
{ Generate Code for Allocation of a Variable }

procedure AllocVar(N, T: char);
begin
   WriteLn(N, ':', TAB, 'DC.', T, ' 0');
end;


{--------------------------------------------------------------}
{ Allocate Storage for a Variable }

procedure Alloc(N, T: char);
begin
   AddEntry(N, T);
   AllocVar(N, T);
end;


{--------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
var Typ: char;
```

```
begin
   Typ := GetName;
   Alloc(GetName, Typ);
end;


{---------------------------------------------------------------}
{ Parse and Translate Global Declarations }

procedure TopDecls;
begin
   while Look <> '.' do begin
      case Look of
        'b', 'w', 'l': Decl;
      else Abort('Unrecognized Keyword ' + Look);
      end;
      Fin;
   end;
end;
{---------------------------------------------------------------}
```

Make the changes shown to these procedures, and give the thing a try. Use the single characters `b`, `w`, and `l` for the keywords (they must be lower case, for now). You will see that in each case, we are allocating the proper storage size. Note from the dumped symbol table that the sizes are also recorded for later use. What later use? Well, that's the subject of the rest of this installment.

Assignments

Now that we can declare variables of different sizes, it stands to reason that we ought to be able to do something with them. For our first trick, let's just try loading them into our working register, `D0`. It makes sense to use the same idea we used for Alloc; that is, make a load procedure that can load more than one size. We also want to continue to encapsulate the machine-dependent stuff. The load procedure looks like this:

```
{---------------------------------------------------------------}
{ Load a Variable to Primary Register }

procedure LoadVar(Name, Typ: char);
begin
   Move(Typ, Name + '(PC)', 'D0');
end;
{---------------------------------------------------------------}
```

On the 68000, at least, it happens that many instructions turn out to be `MOVE`s. It turns out to be useful to create a separate code generator just for these instructions, and then call it as needed:

```
{---------------------------------------------------------------}
{ Generate a Move Instruction }

procedure Move(Size: char; Source, Dest: String);
begin
   EmitLn('MOVE.' + Size + ' ' + Source + ',' + Dest);
end;
{---------------------------------------------------------------}
```

Note that these two routines are strictly code generators; they have no error-checking or other logic. To complete the picture, we need one more layer of software that provides these functions.

First of all, we need to make sure that the type we are dealing with is a loadable type. This sounds like a job for another recognizer:

```
{---------------------------------------------------------------}
{ Recognize a Legal Variable Type }

function IsVarType(c: char): boolean;
begin
   IsVarType := c in ['B', 'W', 'L'];
end;
{---------------------------------------------------------------}
```

Next, it would be nice to have a routine that will fetch the type of a variable from the symbol table, while checking it to make sure it's valid:

```
{---------------------------------------------------------------}
{ Get a Variable Type from the Symbol Table }

function VarType(Name: char): char;
var Typ: char;
begin
   Typ := TypeOf(Name);
   if not IsVarType(Typ) then Abort('Identifier ' + Name +
                                        ' is not a variable');
   VarType := Typ;
end;
{---------------------------------------------------------------}
```

Armed with these tools, a procedure to cause a variable to be loaded becomes trivial:

```
{---------------------------------------------------------------}
{ Load a Variable to the Primary Register }

procedure Load(Name: char);
begin
     LoadVar(Name, VarType(Name));
end;
{---------------------------------------------------------------}
```

(**Note** to the concerned: I know, I know, all this is all very inefficient. In a production program, we probably would take steps to avoid such deep nesting of procedure calls. Don't worry about it. This is an *exercise*, remember? It's more important to get it right and understand it, than it is to make it get the wrong answer, quickly. If you get your compiler completed and find that you're unhappy with the speed, feel free to come back and hack the code to speed it up!)

It would be a good idea to test the program at this point. Since we don't have a procedure for dealing with assignments yet, I just added the lines:

```
Load('A');
Load('B');
Load('C');
Load('X');
```

to the main program. Thus, after the declaration section is complete, they will be executed to generate code for the loads. You can play around with this, and try different combinations of declarations to see how the errors are handled.

I'm sure you won't be surprised to learn that storing variables is a lot like loading them. The necessary procedures are shown next:

```
{---------------------------------------------------------------}
{ Store Primary to Variable }

procedure StoreVar(Name, Typ: char);
begin
   EmitLn('LEA ' + Name + '(PC),A0');
```

```
   Move(Typ, 'D0', '(A0)');
end;


{---------------------------------------------------------------}
{ Store a Variable from the Primary Register }

procedure Store(Name: char);
begin
   StoreVar(Name, VarType(Name));
end;
{---------------------------------------------------------------}
```

You can test this one the same way as the loads.

Now, of course, it's a *rather* small step to use these to handle assignment statements. What we'll do is to create a special version of procedure `Block` that supports only assignment statements, and also a special version of `Expression` that only supports single variables as legal expressions. Here they are:

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

procedure Expression;
var Name: char;
begin
   Load(GetName);
end;


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

procedure Assignment;
var Name: char;
begin
   Name := GetName;
   Match('=');
   Expression;
   Store(Name);
end;


{---------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure Block;
begin
   while Look <> '.' do begin
      Assignment;
      Fin;
   end;
end;
{---------------------------------------------------------------}
```

(It's worth noting that, if anything, the new procedures that permit us to manipulate types are, if anything, even simpler and cleaner than what we've seen before. This is mostly thanks to our efforts to encapsulate the code generator procedures.)

There is one small, nagging problem. Before, we used the Pascal terminating period to get us out of procedure `TopDecls`. This is now the wrong character ... it's used to terminate `Block`. In previous programs, we've used the `BEGIN` symbol (abbreviated `b`) to get us out. But that is now used as a type symbol.

The solution, while somewhat of a kludge, is easy enough. We'll use an upper-case `B` to stand for the `BEGIN`. So change the character in the `WHILE` loop within `TopDecls`, from `.` to `B`, and everything will be fine.

Now, we can complete the task by changing the main program to read:

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

begin
   Init;
   TopDecls;
   Match('B');
   Fin;
   Block;
   DumpTable;
end.
{-----------------------------------------------------------------}
```

(Note that I've had to sprinkle a few calls to `Fin` around to get us out of Newline troubles.)

OK, run this program. Try the input:

```
ba        { byte a }   *** DON'T TYPE THE COMMENTS!!! ***
wb        { word b }
lc        { long c }
B         { begin  }
a=a
a=b
a=c
b=a
b=b
b=c
c=a
c=b
c=c
.
```

For each declaration, you should get code generated that allocates storage. For each assignment, you should get code that loads a variable of the correct size, and stores one, also of the correct size.

There's only one small little problem: The generated code is *wrong*!

Look at the code for a=c above. The code is:

```
MOVE.L    C(PC),D0
LEA       A(PC),A0
MOVE.B    D0,(A0)
```

This code is correct. It will cause the lower eight bits of C to be stored into A, which is a reasonable behavior. It's about all we can expect to happen.

But now, look at the opposite case. For `c=a`, the code generated is:

```
MOVE.B A(PC),D0
LEA  C(PC),A0
MOVE.L D0,(A0)
```

This is *not* correct. It will cause the byte variable `A` to be stored into the lower eight bits of `D0`. According to the rules for the 68000 processor, the upper 24 bits are unchanged. This means that when we store the entire 32 bits into C, whatever garbage that was in those high bits will also get stored. Not good.

So what we have run into here, early on, is the issue of *type conversion*, or *coercion*.

Before we do anything with variables of different types, even if it's just to copy them, we have to face up to the issue. It is not the most easy part of a compiler. Most of the bugs I have seen in production compilers have had to do with errors in type conversion for some obscure combination of arguments. As usual, there is a tradeoff between compiler complexity and the potential quality of the generated code, and as usual, we will take the path that keeps the compiler simple. I think you'll find that, with this approach, we can keep the potential complexity in check rather nicely.

The Coward's Way Out

Before we get into the details (and potential complexity) of type conversion, I'd like you to see that there is one super-simple way to solve the problem: simply promote every variable to a long integer when we load it!

This takes the addition of only one line to `LoadVar`, although if we are not going to *completely* ignore efficiency, it should be guarded by an `IF` test. Here is the modified version:

```
{---------------------------------------------------------------}
{ Load a Variable to Primary Register }

procedure LoadVar(Name, Typ: char);
begin
   if Typ <> 'L' then
      EmitLn('CLR.L D0');
   Move(Typ, Name + '(PC)', 'D0');
end;
{---------------------------------------------------------------}
```

(Note that `StoreVar` needs no similar change.)

If you run some tests with this new version, you will find that everything works correctly now, albeit sometimes inefficiently. For example, consider the case `a=b` (for the same declarations shown above). Now the generated code turns out to be:

```
CLR.L D0
MOVE.W B(PC),D0
LEA  A(PC),A0
MOVE.B D0,(A0)
```

In this case, the `CLR` turns out not to be necessary, since the result is going into a byte-sized variable. With a little bit of work, we can do better. Still, this is not bad, and it typical of the kinds of inefficiencies that we've seen before in simple-minded compilers.

I should point out that, by setting the high bits to zero, we are in effect treating the numbers as *unsigned* integers. If we want to treat them as signed ones instead (the more likely case) we should do a sign extension after the load, instead of a clear before it. Just to tie this part of the discussion up with a nice, red ribbon, let's change `LoadVar` as shown below:

```
{---------------------------------------------------------------}
{ Load a Variable to Primary Register }

procedure LoadVar(Name, Typ: char);
begin
   if Typ = 'B' then
      EmitLn('CLR.L D0');
   Move(Typ, Name + '(PC)', 'D0');
   if Typ = 'W' then
      EmitLn('EXT.L D0');
end;
{---------------------------------------------------------------}
```

With this version, a byte is treated as unsigned (as in Pascal and C), while a word is treated as signed.

A More Reasonable Solution

As we've seen, promoting every variable to long while it's in memory solves the problem, but it can hardly be called efficient, and probably wouldn't be acceptable even for those of us who claim be unconcerned about efficiency. It will mean that all arithmetic operations will be done to 32-bit accuracy, which will *double* the run time for most operations, and make it even worse for multiplication and division. For those operations, we would need to call subroutines to do them, even if the data were byte or word types. The whole thing is sort of a cop-out, too, since it ducks all the real issues.

OK, so that solution's no good. Is there still a relatively easy way to get data conversion? Can we still Keep It Simple?

Yes, indeed. All we have to do is to make the conversion at the other end ... that is, we convert on the way *out*, when the data is stored, rather than on the way in.

But, remember, the storage part of the assignment is pretty much independent of the data load, which is taken care of by procedure `Expression`. In general the expression may be arbitrarily complex, so how can procedure `Assignment` know what type of data is left in register `D0`?

Again, the answer is simple: We'll just *ask* procedure `Expression`! The answer can be returned as a function value.

All of this requires several procedures to be modified, but the mods, like the method, are quite simple. First of all, since we aren't requiring `LoadVar` to do all the work of conversion, let's go back to the simple version:

```
{-----------------------------------------------------------------}
{ Load a Variable to Primary Register }

procedure LoadVar(Name, Typ: char);
begin
   Move(Typ, Name + '(PC)', 'D0');
end;
{-----------------------------------------------------------------}
```

Next, let's add a new procedure that will convert from one type to another:

```
{-----------------------------------------------------------------}
{ Convert a Data Item from One Type to Another }


procedure Convert(Source, Dest: char);
begin
   if Source <> Dest then begin
      if Source  = 'B' then
         EmitLn('AND.W #$FF,D0');
      if Dest = 'L' then
         EmitLn('EXT.L D0');
   end;
end;
{-----------------------------------------------------------------}
```

Next, we need to do the logic required to load and store a variable of any type. Here are the routines for that:

```
{-----------------------------------------------------------------}
{ Load a Variable to the Primary Register }

function Load(Name: char): char;
var Typ : char;
begin
   Typ := VarType(Name);
   LoadVar(Name, Typ);
   Load := Typ;
end;


{-----------------------------------------------------------------}
{ Store a Variable from the Primary Register }

procedure Store(Name, T1: char);
var T2: char;
begin
   T2 := VarType(Name);
   Convert(T1, T2);
   StoreVar(Name, T2);
end;
```

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

Note that `Load` is a function, which not only emits the code for a load, but also returns the variable type. In this way, we always know what type of data we are dealing with. When we execute a `Store`, we pass it the current type of the variable in `D0`. Since `Store` also knows the type of the destination variable, it can convert as necessary.

Armed with all these new routines, the implementation of our rudimentary assignment statement is essentially trivial. Procedure `Expression` now becomes a function, which returns its type to procedure `Assignment`:

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

function Expression: char;
begin
   Expression := Load(GetName);
end;


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

procedure Assignment;
var Name: char;
begin
   Name := GetName;
   Match('=');
   Store(Name, Expression);
end;
{--------------------------------------------------------------}
```

Again, note how incredibly simple these two routines are. We've encapsulated all the type logic into `Load` and `Store`, and the trick of passing the type around makes the rest of the work extremely easy. Of course, all of this is for our special, trivial case of `Expression`. Naturally, for the general case it will have to get more complex. But you're looking now at the *final* version of procedure `Assignment`!

All this seems like a very simple and clean solution, and it is indeed. Compile this program and run the same test cases as before. You will see that all types of data are converted properly, and there are few if any wasted instructions. Only the byte-to-long conversion uses two instructions where one would do, and we could easily modify `Convert` to handle this case, too.

Although we haven't considered unsigned variables in this case, I think you can see that we could easily fix up procedure `Convert` to deal with these types as well. This is "left as an exercise for the student."

Literal Arguments

Sharp-eyed readers might have noticed, though, that we don't even have a proper form of a simple factor yet, because we don't allow for loading literal constants, only variables. Let's fix that now.

To begin with, we'll need a `GetNum` function. We've seen several versions of this, some returning only a single character, some a string, and some an integer. The one needed here will return a `LongInt`, so that it can handle anything we throw at it. Note that no type information is returned here: `GetNum` doesn't concern itself with how the number will be used:

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

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

```
   end;
   GetNum := Val;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

Now, when dealing with literal data, we have one little small problem. With variables, we know what type things should be because they've been declared to be that type. We have no such type information for literals. When the programmer says, `-1`, does that mean a byte, word, or longword version? We have no clue. The obvious thing to do would be to use the largest type possible, i.e. a longword. But that's a bad idea, because when we get to more complex expressions, we'll find that it will cause every expression involving literals to be promoted to long, as well.

A better approach is to select a type based upon the value of the literal, as shown next:

```
{--------------------------------------------------------------}
{ Load a Constant to the Primary Register }

function LoadNum(N: LongInt): char;
var Typ : char;
begin
   if abs(N) <= 127 then
      Typ := 'B'
   else if abs(N) <= 32767 then
      Typ := 'W'
   else Typ := 'L';
   LoadConst(N, Typ);
   LoadNum := Typ;
end;
{--------------------------------------------------------------}
```

(I know, I know, the number base isn't really symmetric. You can store `-128` in a single byte, and `-32768` in a word. But that's easily fixed, and not worth the time or the added complexity to fool with it here. It's the thought that counts.)

Note that `LoadNum` calls a new version of the code generator routine `LoadConst`, which has an added argument to define the type:

```
{--------------------------------------------------------------}
{ Load a Constant to the Primary Register }

procedure LoadConst(N: LongInt; Typ: char);
var temp:string;
begin
   Str(N, temp);
   Move(Typ, '#' + temp, 'D0');
end;
{--------------------------------------------------------------}
```

Now we can modify procedure `Expression` to accommodate the two possible kinds of factors:

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

function Expression: char;
begin
   if IsAlpha(Look) then
      Expression := Load(GetName)
   else
      Expression := LoadNum(GetNum);
end;
```

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

(Wow, that sure didn't hurt too bad! Just a few extra lines do the job.)

OK, compile this code into your program and give it a try. You'll see that it now works for either variables or constants as valid expressions.

Additive Expressions

If you've been following this series from the beginning, I'm sure you know what's coming next: We'll expand the form for an expression to handle first additive expressions, then multiplicative, then general expressions with parentheses.

The nice part is that we already have a pattern for dealing with these more complex expressions. All we have to do is to make sure that all the procedures called by `Expression` (`Term`, `Factor`, etc.) always return a type identifier. If we do that, the program structure gets changed hardly at all.

The first step is easy: We can rename our existing function `Expression` to `Term`, as we've done so many times before, and create the new version of `Expression`:

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

function Expression: char;
var Typ: char;
begin
   if IsAddop(Look) then
      Typ := Unop
   else
      Typ := Term;
   while IsAddop(Look) do begin
      Push(Typ);
      case Look of
       '+': Typ := Add(Typ);
       '-': Typ := Subtract(Typ);
      end;
   end;
   Expression := Typ;
end;
{---------------------------------------------------------------}
```

Note in this routine how each procedure call has become a function call, and how the local variable Typ gets updated at each pass.

Note also the new call to a function `Unop`, which lets us deal with a leading unary minus. This change is not necessary ... we could still use a form more like what we've done before. I've chosen to introduce UnOp as a separate routine because it will make it easier, later, to produce somewhat better code than we've been doing. In other words, I'm looking ahead to optimization issues.

For this version, though, we'll retain the same dumb old code, which makes the new routine trivial:

```
{---------------------------------------------------------------}
{ Process a Term with Leading Unary Operator }

function Unop: char;
begin
   Clear;
   Unop := 'W';
end;
{---------------------------------------------------------------}
```

Procedure `Push` is a code-generator routine, and now has a type argument:

```
{-------------------------------------------------------------}
{ Push Primary onto Stack }

procedure Push(Size: char);
begin
   Move(Size, 'D0', '-(SP)');
end;
{-------------------------------------------------------------}
```

Now, let's take a look at functions `Add` and `Subtract`. In the older versions of these routines, we let them call code generator routines `PopAdd` and `PopSub`. We'll continue to do that, which makes the functions themselves extremely simple:

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

function Add(T1: char): char;
begin
   Match('+');
   Add := PopAdd(T1, Term);
end;


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

function Subtract(T1: char): char;
begin
   Match('-');
   Subtract := PopSub(T1, Term);
end;
{-------------------------------------------------------------}
```

The simplicity is deceptive, though, because what we've done is to defer all the logic to `PopAdd` and `PopSub`, which are no longer just code generation routines. They must also now take care of the type conversions required.

And just what conversion is that? Simple: Both arguments must be of the same size, and the result is also of that size. The smaller of the two arguments must be "promoted" to the size of the larger one.

But this presents a bit of a problem. If the argument to be promoted is the second argument (i.e. in the primary register D0), we are in great shape. If it's not, however, we're in a fix: we can't change the size of the information that's already been pushed onto the stack.

The solution is simple but a little painful: We must abandon that lovely "pop the data and do something with it" instructions thoughtfully provided by Motorola.

The alternative is to assign a secondary register, which I've chosen to be `R7`. (Why not `R1`? Because I have later plans for the other registers.)

The first step in this new structure is to introduce a `Pop` procedure analogous to the `Push`. This procedure will always `Pop` the top element of the stack into `D7`:

```
{-------------------------------------------------------------}
{ Pop Stack into Secondary Register }

procedure Pop(Size: char);
begin
   Move(Size, '(SP)+', 'D7');
end;
{-------------------------------------------------------------}
```

The general idea is that all the "Pop-Op" routines can call this one. When this is done, we will then have both operands in registers, so we can promote whichever one we need to. To deal with this, procedure `Convert` needs another argument, the register name:

```
{-------------------------------------------------------------}
{ Convert a Data Item from One Type to Another }

procedure Convert(Source, Dest: char; Reg: String);
begin
   if Source <> Dest then begin
      if Source  = 'B' then
         EmitLn('AND.W #$FF,' + Reg);
      if Dest = 'L' then
         EmitLn('EXT.L ' + Reg);
   end;
end;
{-------------------------------------------------------------}
```

The next function does a conversion, but only if the current type `T1` is smaller in size than the desired type `T2`. It is a function, returning the final type to let us know what it decided to do:

```
{-------------------------------------------------------------}
{ Promote the Size of a Register Value }

function Promote(T1, T2: char; Reg: string): char;
var Typ: char;
begin
   Typ := T1;
   if T1 <> T2 then
      if (T1 = 'B') or ((T1 = 'W') and (T2 = 'L')) then begin
         Convert(T1, T2, Reg);
         Typ := T2;
      end;
   Promote := Typ;
end;
{-------------------------------------------------------------}
```

Finally, the following function forces the two registers to be of the same type:

```
{-------------------------------------------------------------}
{ Force both Arguments to Same Type }

function SameType(T1, T2: char): char;
begin
   T1 := Promote(T1, T2, 'D7');
   SameType := Promote(T2, T1, 'D0');
end;
{-------------------------------------------------------------}
```

These new routines give us the ammunition we need to flesh out `PopAdd` and `PopSub`:

```
{-------------------------------------------------------------}
{ Generate Code to Add Primary to the Stack }

function PopAdd(T1, T2: char): char;
begin
   Pop(T1);
   T2 := SameType(T1, T2);
   GenAdd(T2);
   PopAdd := T2;
end;


{-------------------------------------------------------------}
{ Generate Code to Subtract Primary from the Stack }
```

```
function PopSub(T1, T2: char): char;
begin
   Pop(T1);
   T2 := SameType(T1, T2);
   GenSub(T2);
   PopSub := T2;
end;
{--------------------------------------------------------------}
```

After all the buildup, the final results are almost anticlimactic. Once again, you can see that the logic is quite simple. All the two routines do is to pop the top-of-stack into `D7`, force the two operands to be the same size, and then generate the code.

Note the new code generator routines `GenAdd` and `GenSub`. These are vestigial forms of the *original* `PopAdd` and `PopSub`. That is, they are pure code generators, producing a register-to-register add or subtract:

```
{--------------------------------------------------------------}
{ Add Top of Stack to Primary }

procedure GenAdd(Size: char);
begin
   EmitLn('ADD.' + Size + ' D7,D0');
end;


{--------------------------------------------------------------}
{ Subtract Primary from Top of Stack }

procedure GenSub(Size: char);
begin
   EmitLn('SUB.' + Size + ' D7,D0');
   EmitLn('NEG.' + Size + ' D0');
end;
{--------------------------------------------------------------}
```

OK, I grant you: I've thrown a lot of routines at you since we last tested the code. But you have to admit that each new routine is pretty simple and transparent. If you (like me) don't like to test so many new routines at once, that's OK. You can stub out routines like `Convert`, `Promote`, and `SameType`, since they don't read any inputs. You won't get the correct code, of course, but things should work. Then flesh them out one at a time.

When testing the program, don't forget that you first have to declare some variables, and then start the "body" of the program with an upper-case `B` (for `BEGIN`). You should find that the parser will handle any additive expressions. Once all the conversion routines are in, you should see that the correct code is generated, with type conversions inserted where necessary. Try mixing up variables of different sizes, and also literals. Make sure that everything's working properly. As usual, it's a good idea to try some erroneous expressions and see how the compiler handles them.

Why So Many Procedures?

At this point, you may think I've pretty much gone off the deep end in terms of deeply nested procedures. There is admittedly a lot of overhead here. But there's a method in my madness. As in the case of `UnOp`, I'm looking ahead to the time when we're going to want better code generation. The way the code is organized, we can achieve this without major modifications to the program. For example, in cases where the value pushed onto the stack does *not* have to be converted, it's still better to use the "pop and add" instruction. If we choose to test for such cases, we can embed the extra tests into `PopAdd` and `PopSub` without changing anything else much.

Multiplicative Expressions

The procedure for dealing with multiplicative operators is much the same. In fact, at the first level, they are almost identical, so I'll just show them here without much fanfare. The first one is our general form for `Factor`, which includes parenthetical subexpressions:

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

function Expression: char; Forward;

function Factor: char;
begin
   if Look = '(' then begin
      Match('(');
      Factor := Expression;
      Match(')');
      end
   else if IsAlpha(Look) then
      Factor := Load(GetName)
   else
      Factor := LoadNum(GetNum);
end;


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

Function Multiply(T1: char): char;
begin
   Match('*');
   Multiply := PopMul(T1, Factor);
end;


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

function Divide(T1: char): char;
begin
   Match('/');
   DIvide := PopDiv(T1, Factor);
end;


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

function Term: char;
var Typ: char;
begin
   Typ := Factor;
   while IsMulop(Look) do begin
      Push(Typ);
      case Look of
       '*': Typ := Multiply(Typ);
       '/': Typ := Divide(Typ);
      end;
   end;
   Term := Typ;
end;
{--------------------------------------------------------------}
```

These routines parallel the additive ones almost exactly. As before, the complexity is encapsulated within `PopMul` and `PopDiv`. If you'd like to test the program before we get into that, you can build dummy versions of them, similar to `PopAdd` and `PopSub`. Again, the code won't be correct at this point, but the parser should handle expressions of arbitrary complexity.

Multiplication

Once you've convinced yourself that the parser itself is working properly, we need to figure out what it will take to generate the right code. This is where things begin to get a little sticky, because the rules are more complex.

Let's take the case of multiplication first. This operation is similar to the "addops" in that both operands should be of the same size. It differs in two important respects:

- The type of the product is typically not the same as that of the two operands. For the product of two words, we get a longword result.
- The 68000 does not support a 32 x 32 multiply, so a call to a software routine is needed. This routine will become part of the run-time library.
- It also does not support an 8 x 8 multiply, so all byte operands must be promoted to words.

The actions that we have to take are best shown in the following table:

T1 --> / T2 V	B	W	L
B	Convert D0 to W Convert D7 to W MULS Result = W	Convert D0 to W MULS Result = L	Convert D0 to L JSR MUL32 Result = L
W	Convert D7 to W MULS Result = L	MULS Result = L	Convert D0 to L JSR MUL32 Result = L
L	Convert D7 to L JSR MUL32 Result = L	Convert D7 to L JSR MUL32 Result = L	JSR MUL32 Result = L

This table shows the actions to be taken for each combination of operand types. There are three things to note: First, we assume a library routine `MUL32` which performs a 32 x 32 multiply, leaving a **32-bit** (not 64-bit) product. If there is any overflow in the process, we choose to ignore it and return only the lower 32 bits.

Second, note that the table is symmetric ... the two operands enter in the same way. Finally, note that the product is *always* a longword, except when both operands are bytes. (It's worth noting, in passing, that this means that many expressions will end up being longwords, whether we like it or not. Perhaps the idea of just promoting them all up front wasn't all that outrageous, after all!)

Now, clearly, we are going to have to generate different code for the 16-bit and 32-bit multiplies. This is best done by having separate code generator routines for the two cases:

```
{--------------------------------------------------------------}
{ Multiply Top of Stack by Primary (Word) }

procedure GenMult;
begin
   EmitLn('MULS D7,D0')
end;


{--------------------------------------------------------------}
{ Multiply Top of Stack by Primary (Long) }
```

```
procedure GenLongMult;
begin
   EmitLn('JSR MUL32');
end;
{--------------------------------------------------------------}
```

An examination of the code below for `PopMul` should convince you that the conditions in the table are met:

```
{--------------------------------------------------------------}
{ Generate Code to Multiply Primary by Stack }

function PopMul(T1, T2: char): char;
var T: char;
begin
   Pop(T1);
   T := SameType(T1, T2);
   Convert(T, 'W', 'D7');
   Convert(T, 'W', 'D0');
   if T = 'L' then
      GenLongMult
   else
      GenMult;
   if T = 'B' then
      PopMul := 'W'
   else
      PopMul:= 'L';
end;
{--------------------------------------------------------------}
```

As you can see, the routine starts off just like `PopAdd`. The two arguments are forced to the same type. The two calls to `Convert` take care of the case where both operands are bytes. The data themselves are promoted to words, but the routine remembers the type so as to assign the correct type to the result. Finally, we call one of the two code generator routines, and then assign the result type. Not too complicated, really.

At this point, I suggest that you go ahead and test the program. Try all combinations of operand sizes.

Division

The case of division is not nearly so symmetric. I also have some bad news for you:

All modern 16-bit CPU's support integer divide. The manufacturer's data sheet will describe this operation as a 32 x 16-bit divide, meaning that you can divide a 32-bit dividend by a 16-bit divisor. Here's the bad news:

THEY'RE LYING TO YOU!!!

If you don't believe it, try dividing any large 32-bit number (meaning that it has non-zero bits in the upper 16 bits) by the integer `1`. You are guaranteed to get an overflow exception.

The problem is that the instruction really requires that the resulting quotient fit into a 16-bit result. This won't happen *unless* the divisor is sufficiently large. When any number is divided by unity, the quotient will of course be the same as the dividend, which had better fit into a 16-bit word.

Since the beginning of time (well, computers, anyway), CPU architects have provided this little gotcha in the division circuitry. It provides a certain amount of symmetry in things, since it is sort of the inverse of the way a multiply works. But since unity is a perfectly valid (and rather common) number to use as a divisor, the division as implemented in hardware needs some help from us programmers.

The implications are as follows:

- The type of the quotient must always be the same as that of the dividend. It is independent of the divisor.
- In spite of the fact that the CPU supports a longword dividend, the hardware-provided instruction can only be trusted for byte and word dividends. For longword dividends, we need another library routine that can return a long result.

This looks like a job for another table, to summarize the required actions:

```
-------------------------------------------------------------------
  T1 -->  |                 |                 |                 |
          |                 |                 |                 |
      |   |        B        |        W        |       L         |
  T2  V   |                 |                 |                 |
-------------------------------------------------------------------
          |                 |                 |                 |
     B    | Convert D0 to W | Convert D0 to W | Convert D0 to L |
          | Convert D7 to L | Convert D7 to L |                 |
          | DIVS            | DIVS            | JSR DIV32       |
          | Result = B      | Result = W      | Result = L      |
          |                 |                 |                 |
-------------------------------------------------------------------
          |                 |                 |                 |
     W    | Convert D7 to L | Convert D7 to L | Convert D0 to L |
          | DIVS            | DIVS            | JSR DIV32       |
          | Result = B      | Result = W      | Result = L      |
          |                 |                 |                 |
-------------------------------------------------------------------
          |                 |                 |                 |
     L    | Convert D7 to L | Convert D7 to L |                 |
          | JSR DIV32       | JSR DIV32       | JSR DIV32       |
          | Result = B      | Result = W      | Result = L      |
          |                 |                 |                 |
-------------------------------------------------------------------
```

(You may wonder why it's necessary to do a 32-bit division, when the dividend is, say, only a byte in the first place. Since the number of bits in the result can only be as many as that in the dividend, why bother? The reason is that, if the divisor is a longword, and there are any high bits set in it, the result of the division must be zero. We might not get that if we only use the lower word of the divisor.)

The following code provides the correct function for PopDiv:

```
{-------------------------------------------------------------------}
{ Generate Code to Divide Stack by the Primary }

function PopDiv(T1, T2: char): char;
begin
   Pop(T1);
   Convert(T1, 'L', 'D7');
   if (T1 = 'L') or (T2 = 'L') then begin
      Convert(T2, 'L', 'D0');
      GenLongDiv;
      PopDiv := 'L';
      end
   else begin
      Convert(T2, 'W', 'D0');
      GenDiv;
      PopDiv := T1;
   end;
end;
{-------------------------------------------------------------------}
```

The two code generation procedures are:

```
{-------------------------------------------------------------------}
{ Divide Top of Stack by Primary  (Word) }

procedure GenDiv;
begin
   EmitLn('DIVS D0,D7');
   Move('W', 'D7', 'D0');
```

```
end;


{-----------------------------------------------------------------}
{ Divide Top of Stack by Primary (Long) }

procedure GenLongDiv;
begin
   EmitLn('JSR DIV32');
end;
{-----------------------------------------------------------------}
```

Note that we assume that `DIV32` leaves the (longword) result in `D0`.

OK, install the new procedures for division. At this point you should be able to generate code for any kind of arithmetic expression. Give it a whirl!

Beginning to Wind Down

At last, in this installment, we've learned how to deal with variables (and literals) of different types. As you can see, it hasn't been too tough. In fact, in some ways most of the code looks even more simple than it does in earlier programs. Only the multiplication and division operators require a little thinking and planning.

The main concept that made things easy was that of converting procedures such as `Expression` into functions that return the type of the result. Once this was done, we were able to retain the same general structure of the compiler.

I won't pretend that we've covered every single aspect of the issue. I conveniently ignored unsigned arithmetic. From what we've done, I think you can see that to include them adds no new challenges, just extra possibilities to test for.

I've also ignored the logical operators And, Or, etc. It turns out that these are pretty easy to handle. All the logical operators are bitwise operations, so they are symmetric and therefore work in the same fashion as `PopAdd`. There is one difference, however: if it is necessary to extend the word length for a logical variable, the extension should be done as an *unsigned* number. Floating point numbers, again, are straightforward to handle ... just a few more procedures to be added to the run-time library, or perhaps instructions for a math chip.

Perhaps more importantly, I have also skirted the issue of type *checking*, as opposed to conversion. In other words, we've allowed for operations between variables of all combinations of types. In general this will not be true ... certainly you don't want to add an integer, for example, to a string. Most languages also don't allow you to mix up character and integer variables.

Again, there are really no new issues to be addressed in this case. We are already checking the types of the two operands ... much of this checking gets done in procedures like SameType. It's pretty straightforward to include a call to an error handler, if the types of the two operands are incompatible.

In the general case, we can think of every single operator as being handled by a different procedure, depending upon the type of the two operands. This is straightforward, though tedious, to implement simply by implementing a jump table with the operand types as indices. In Pascal, the equivalent operation would involve nested Case statements. Some of the called procedures could then be simple error routines, while others could effect whatever kind of conversion we need. As more types are added, the number of procedures goes up by a square-law rule, but that's still not an unreasonably large number of procedures.

What we've done here is to collapse such a jump table into far fewer procedures, simply by making use of symmetry and other simplifying rules.

To Coerce or Not to Coerce

In case you haven't gotten this message yet, it sure appears that TINY and KISS will probably *not* be strongly typed languages, since I've allowed for automatic mixing and conversion of just about any type. Which brings up the next issue:

Is this really what we want to do?

The answer depends on what kind of language you want, and the way you'd like it to behave. What we have not addressed is the issue of when to allow and when to deny the use of operations involving different data types. In other words, what should be the *semantics* of our compiler? Do we want automatic type conversion for all cases, for some cases, or not at all?

Let's pause here to think about this a bit more. To do so, it will help to look at a bit of history.

FORTRAN II supported only two simple data types: `Integer` and `Real`. It allowed implicit type conversion between real and integer types during assignment, but not within expressions. All data items (including literal constants) on the right-hand side of an assignment statement had to be of the same type. That made things pretty easy ... much simpler than what we've had to do here.

This was changed in FORTRAN IV to support "mixed-mode" arithmetic. If an expression had any real data items in it, they were all converted to reals and the expression itself was real. To round out the picture, functions were provided to explicitly convert from one type to the other, so that you could force an expression to end up as either type.

This led to two things: code that was easier to write, and code that was less efficient. That's because sloppy programmers would write expressions with simple constants like 0 and 1 in them, which the compiler would dutifully compile to convert at execution time. Still, the system worked pretty well, which would tend to indicate that implicit type conversion is a Good Thing.

C is also a weakly typed language, though it supports a larger number of types. C won't complain if you try to add a character to an integer, for example. Partly, this is helped by the C convention of promoting every char to integer when it is loaded, or passed through a parameter list. This simplifies the conversions quite a bit. In fact, in subset C compilers that don't support long or float types, we end up back where we were in our earlier, simple-minded first try: every variable has the same representation, once loaded into a register. Makes life pretty easy!

The ultimate language in the direction of automatic type conversion is PL/I. This language supports a large number of data types, and you can mix them all freely. If the implicit conversions of FORTRAN seemed good, then those of PL/I should have been Heaven, but it turned out to be more like Hell! The problem was that with so many data types, there had to be a large number of different conversions, *and* a correspondingly large number of rules about how mixed operands should be converted. These rules became so complex that no one could remember what they were! A lot of the errors in PL/I programs had to do with unexpected and unwanted type conversions. Too much of a Good Thing can be bad for you!

Pascal, on the other hand, is a language which is "strongly typed," which means that in general you can't mix types, even if they differ only in *name*, and yet have the same base type! Niklaus Wirth made Pascal strongly typed to help keep programmers out of trouble, and the restrictions have indeed saved many a programmer from themself, because the compiler kept them from doing something dumb. Better to find the bug in compilation rather than the debug phase. The same restrictions can also cause frustration when you really *want* to mix types, and they tend to drive an ex-C-programmer up the wall.

Even so, Pascal does permit some implicit conversions. You can assign an integer to a real value. You can also mix integer and real types in expressions of type `Real`. The integers will be automatically coerced to real, just as in FORTRAN (and with the same hidden cost in run-time overhead).

You can't, however, convert the other way, from real to integer, without applying an explicit conversion function, `Trunc`. The theory here is that, since the numerical value of a real number is necessarily going to be changed by the conversion (the fractional part will be lost), you really shouldn't do it in "secret."

In the spirit of strong typing, Pascal will not allow you to mix `Char` and `Integer` variables, without applying the explicit coercion functions Chr and Ord.

Turbo Pascal also includes the types `Byte`, `Word`, and `LongInt`. The first two are basically the same as unsigned integers. In Turbo, these can be freely intermixed with variables of type `Integer`, and Turbo will automatically handle the conversion. There are run-time checks, though, to keep you from overflowing or otherwise getting the wrong answer. Note that you still can't mix `Byte` and `Char` types, even though they are stored internally in the same representation.

The ultimate in a strongly-typed language is Ada, which allows *no* implicit type conversions at all, and also will not allow mixed-mode arithmetic. Jean Ichbiah's position is that conversions cost execution time, and you shouldn't be allowed to build in such cost in a hidden manner. By forcing the programmer to explicitly request a type conversion, you make it more apparent that there could be a cost involved.

I have been using another strongly-typed language, a delightful little language called Whimsical, by John Spray. Although Whimsical is intended as a systems programming language, it also requires explicit conversion *every* time. There are *never* any automatic conversions, even the ones supported by Pascal.

This approach does have certain advantages: The compiler never has to guess what to do: the programmer always tells it precisely what they want. As a result, there tends to be a more nearly one-to-one correspondence between source code and compiled code, and John's compiler produces *very* tight code.

On the other hand, I sometimes find the explicit conversions to be a pain. If I want, for example, to add one to a character, or AND it with a mask, there are a lot of conversions to make. If I get it wrong, the only error message is "Types are not compatible." As it happens, John's particular implementation of the language in his compiler doesn't tell you exactly *which* types are not compatible ... it only tells you which *line* the error is in.

I must admit that most of my errors with this compiler tend to be errors of this type, and I've spent a lot of time with the Whimsical compiler, trying to figure out just *where* in the line I've offended it. The only real way to fix the error is to keep trying things until something works.

So what should we do in TINY and KISS? For the first one, I have the answer: TINY will support only the types `Char` and `Integer`, and we'll use the C trick of promoting Chars to Integers internally. That means that the TINY compiler will be *much* simpler than what we've already done. Type conversion in expressions is sort of moot, since none will be required! Since longwords will not be supported, we also won't need the `MUL32` and `DIV32` run-time routines, nor the logic to figure out when to call them. I *like* it!

KISS, on the other hand, will support the type Long.

Should it support both signed and unsigned arithmetic? For the sake of simplicity I'd rather not. It does add quite a bit to the complexity of type conversions. Even Niklaus Wirth has eliminated unsigned (Cardinal) numbers from his new language Oberon, with the argument that 32-bit integers should be long enough for anybody, in either case.

But KISS is supposed to be a systems programming language, which means that we should be able to do whatever operations that can be done in assembler. Since the 68000 supports both flavors of integers, I guess KISS should, also. We've seen that logical operations need to be able to extend integers in an unsigned fashion, so the unsigned conversion procedures are required in any case.

Conclusion

That wraps up our session on type conversions. Sorry you had to wait so long for it, but hope you feel that it was worth the wait.

In the next few installments, we'll extend the simple types to include arrays and pointers, and we'll have a look at what to do about strings. That should pretty well wrap up the mainstream part of the series. After that, I'll give you the new versions of the TINY and KISS compilers, and then we'll start to look at optimization issues.

See you then.

Chapter 15

Part XV: Back to the Future - 5 March 1994

Introduction

Can it really have been four years since I wrote installment fourteen of this series? Is it really possible that six long years have passed since I began it? Funny how time flies when you're having fun, isn't it?

I won't spend a lot of time making excuses; only point out that things happen, and priorities change. In the four years since installment fourteen, I've managed to get laid off, get divorced, have a nervous breakdown, begin a new career as a writer, begin another one as a consultant, move, work on two real-time systems, and raise fourteen baby birds, three pigeons, six possums, and a duck. For awhile there, the parsing of source code was not high on my list of priorities. Neither was writing stuff for free, instead of writing stuff for pay. But I do try to be faithful, and I do recognize and feel my responsibility to you, the reader, to finish what I've started. As the tortoise said in one of my son's old stories, I may be slow, but I'm sure. I'm sure that there are people out there anxious to see the last reel of this film, and I intend to give it to them. So, if you're one of those who's been waiting, more or less patiently, to see how this thing comes out, thanks for your patience. I apologize for the delay. Let's move on.

New Starts, Old Directions

Like many other things, programming languages and programming styles change with time. In 1994, it seems a little anachronistic to be programming in Turbo Pascal, when the rest of the world seems to have gone bananas over C++. It also seems a little strange to be programming in a classical style when the rest of the world has switched to object-oriented methods. Still, in spite of the four-year hiatus, it would be entirely too wrenching a change, at this point, to switch to, say, C++ with object-orientation. Anyway, Pascal is still not only a powerful programming language (more than ever, in fact), but it's a wonderful medium for teaching. C is a notoriously difficult language to read ... it's often been accused, along with Forth, of being a "write-only language." When I program in C++, I find myself spending at least 50% of my time struggling with language syntax rather than with concepts. A stray `&` or `*` can not only change the functioning of the program, but its correctness as well. By contrast, Pascal code is usually quite transparent and easy to read, even if you don't know the language. What you see is almost always what you get, and we can concentrate on concepts rather than implementation details. I've said from the beginning that the purpose of this tutorial series was not to generate the world's fastest compiler, but to teach the fundamentals of compiler technology, while spending the least amount of time wrestling with language syntax or other aspects of software implementation. Finally, since a lot of what we do in this course amounts to software experimentation, it's important to have a compiler and associated environment that compiles quickly and with no fuss. In my opinion, by far the most significant time measure in software development is the speed of the edit/compile/test cycle. In this department, Turbo Pascal is king. The compilation speed is blazing fast, and continues to get faster in every release (how do they keep doing that?). Despite vast improvements in C compilation speed over the years, even Borland's fastest C/C++ compiler is still no match for Turbo Pascal. Further, the editor built into their IDE, the make facility, and even their superb smart linker, all complement each other to produce a wonderful environment for quick turnaround. For all of these reasons, I intend to stick with Pascal for the duration of this series. We'll be using Turbo Pascal for Windows, one of the compilers provided Borland Pascal with Objects, version 7.0. If you don't have this compiler, don't worry ... nothing we do here is going to count on your having the latest version. Using the Windows version helps me a lot, by allowing me to use the Clipboard to copy code from the compiler's editor into these documents. It should also help you at least as much, copying the code in the other direction.

I've thought long and hard about whether or not to introduce objects to our discussion. I'm a big advocate of object-oriented methods for all uses, and such methods definitely have their place in compiler technology. In fact, I've written papers on just this subject [1] [2] [3]. But the architecture of a compiler which is based on object-oriented approaches is vastly different than that of the more classical compiler we've been building. Again, it would seem to be entirely too much to change these horses in mid-stream. As I said, programming styles change. Who knows, it may be another six years before we finish this thing, and if we keep changing the code every time programming style changes, we may *never* finish.

So for now, at least, I've determined to continue the classical style in Pascal, though we might indeed discuss objects and object orientation as we go. Likewise, the target machine will remain the Motorola 68000 family. Of all the decisions to be made here, this one has been the easiest. Though I know that many of you would like to see code for the 80x86, the 68000 has become, if anything, even more popular as a platform for embedded systems, and it's to that application that this whole effort began in the first place. Compiling for the PC, MSDOS platform, we'd have to deal with all the issues of DOS system calls, DOS linker formats, the PC file system and hardware, and all those other complications of a DOS environment. An embedded system, on the other hand, must run standalone, and it's for this kind of application, as an alternative to assembly language, that I've always imagined that a language like KISS would thrive. Anyway, who wants to deal with the 80x86 architecture if they don't have to?

The one feature of Turbo Pascal that I'm going to be making heavy use of is units. In the past, we've had to make compromises between code size and complexity, and program functionality. A lot of our work has been in the nature of computer experimentation, looking at only one aspect of compiler technology at a time. We did this to avoid to avoid having to carry around large programs, just to investigate simple concepts. In the process, we've re-invented the wheel and re-programmed the same functions more times than I'd like to count. Turbo units provide a wonderful way to get functionality and simplicity at the same time: You write reusable code, and invoke it with a single line. Your test program stays small, but it can do powerful things.

One feature of Turbo Pascal units is their initialization block. As with an Ada package, any code in the main begin-end block of a unit gets executed as the program is initialized. As you'll see later, this sometimes gives us neat simplifications in the code. Our procedure `Init`, which has been with us since Installment 1, goes away entirely when we use units. The various routines in the Cradle, another key features of our approach, will get distributed among the units.

The concept of units, of course, is no different than that of C modules. However, in C (and C++), the interface between modules comes via preprocessor include statements and header files. As someone who's had to read a lot of other people's C programs, I've always found this rather bewildering. It always seems that whatever data structure you'd like to know about is in some other file. Turbo units are simpler for the very reason that they're criticized by some: The function interfaces and their implementation are included in the same file. While this organization may create problems with code security, it also reduces the number of files by half, which isn't half bad. Linking of the object files is also easy, because the Turbo compiler takes care of it without the need for make files or other mechanisms.

Starting Over?

Four years ago, in Installment 14, I promised you that our days of re-inventing the wheel, and recoding the same software over and over for each lesson, were over, and that from now on we'd stick to more complete programs that we would simply add new features to. I still intend to keep that promise; that's one of the main purposes for using units. However, because of the long time since Installment 14, it's natural to want to at least do some review, and anyhow, we're going to have to make rather sweeping changes in the code to make the transition to units. Besides, frankly, after all this time I can't remember all the neat ideas I had in my head four years ago. The best way for me to recall them is to retrace some of the steps we took to arrive at Installment 14. So I hope you'll be understanding and bear with me as we go back to our roots, in a sense, and rebuild the core of the software, distributing the routines among the various units, and bootstrapping ourselves back up to the point we were at lo, those many moons ago. As has always been the case, you're going to get to see me make all the mistakes and execute changes of direction, in real time. Please bear with me ... we'll start getting to the new stuff before you know it.

Since we're going to be using multiple modules in our new approach, we have to address the issue of file management. If you've followed all the other sections of this tutorial, you know that, as our programs evolve, we're going to be replacing older, more simple-minded units with more capable ones. This brings us to an issue of version control. There will almost certainly be times when we will overlay a simple file (unit), but later wish we had the simple one again. A case in point is embodied in our predilection for using single-character variable names, keywords, etc., to test concepts without getting bogged down in the details of a lexical scanner. Thanks to the use of units, we will be doing much less of this in the future. Still, I not only suspect, but am certain that we will need to save some older versions of files, for special purposes, even though they've been replaced by newer, more capable ones.

To deal with this problem, I suggest that you create different directories, with different versions of the units as needed. If we do this properly, the code in each directory will remain self-consistent. I've tentatively created four directories: SINGLE (for single-character experimentation), MULTI (for, of course, multi-character versions), TINY, and KISS.

Enough said about philosophy and details. Let's get on with the resurrection of the software.

The Input Unit

A key concept that we've used since Day 1 has been the idea of an input stream with one lookahead character. All the parsing routines examine this character, without changing it, to decide what they should do next. (Compare this approach with the C/Unix approach using `getchar` and `unget`, and I think you'll agree that our approach is simpler). We'll begin our hike into the future by translating this concept into our new, unit-based organization. The first unit, appropriately called `Input`, is shown below:

```
{--------------------------------------------------------------}
unit Input;
{--------------------------------------------------------------}
interface
var Look: char;               { Lookahead character }
procedure GetChar;            { Read new character  }

{--------------------------------------------------------------}
implementation

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

procedure GetChar;
begin
Read(Look);
end;

{--------------------------------------------------------------}
{ Unit Initialization }
begin
GetChar;
end.
{--------------------------------------------------------------}
```

As you can see, there's nothing very profound, and certainly nothing complicated, about this unit, since it consists of only a single procedure. But already, we can see how the use of units gives us advantages. Note the executable code in the initialization block. This code "primes the pump" of the input stream for us, something we've always had to do before, by inserting the call to `GetChar` in line, or in procedure `Init`. This time, the call happens without any special reference to it on our part, except within the unit itself. As I predicted earlier, this mechanism is going to make our lives much simpler as we proceed. I consider it to be one of the most useful features of Turbo Pascal, and I lean on it heavily.

Copy this unit into your compiler's IDE, and compile it. To test the software, of course, we always need a main program. I used the following, really complex test program, which we'll later evolve into the `Main` for our compiler:

```
{--------------------------------------------------------------}
program Main;
uses WinCRT, Input;
begin
WriteLn(Look);
end.
{--------------------------------------------------------------}
```

Note the use of the Borland-supplied unit, WinCRT. This unit is necessary if you intend to use the standard Pascal I/O routines, Read, ReadLn, Write, and WriteLn, which of course we intend to do. If you forget to include this unit in the `uses` clause, you will get a really bizarre and indecipherable error message at run time.

Note also that we can access the lookahead character, even though it's not declared in the main program. All variables declared within the interface section of a unit are global, but they're hidden from prying eyes; to that extent, we get a modicum of information hiding. Of course, if we were writing in an object-oriented fashion, we should not allow outside modules to access the units internal variables. But, although Turbo units have a lot in common with objects, we're not doing object-oriented design or code here, so our use of Look is appropriate.

Go ahead and save the test program as `Main.pas`. To make life easier as we get more and more files, you might want to take this opportunity to declare this file as the compiler's Primary file. That way, you can execute the program from any file. Otherwise, if you press Cntl-F9 to compile and run from one of the units, you'll get an error message. You set the primary file using the main submenu, "Compile," in the Turbo IDE.

I hasten to point out, as I've done before, that the function of unit `Input` is, and always has been, considered to be a dummy version of the real thing. In a production version of a compiler, the input stream will, of course, come from a file rather than from the keyboard. And it will almost certainly include line buffering, at the very least, and more likely, a rather large text buffer to support efficient disk I/O. The nice part about the unit approach is that, as with objects, we can modify the code in the unit to be as simple or as sophisticated as we like. As long as the interface, as embodied in the public procedures and the lookahead character, don't change, the rest of the program is totally unaffected. And since units are compiled, rather than merely included, the time required to link with them is virtually nil. Again, the result is that we can get all the benefits of sophisticated implementations, without having to carry the code around as so much baggage.

In later installments, I intend to provide a full-blown IDE for the KISS compiler, using a true Windows application generated by Borland's OWL applications framework. For now, though, we'll obey my #1 rule to live by: Keep It Simple.

The Output Unit

Of course, every decent program should have output, and ours is no exception. Our output routines included the `Emit` functions. The code for the corresponding output unit is shown next:

```
{--------------------------------------------------------------}
unit Output;
{--------------------------------------------------------------}
interface
procedure Emit(s: string); { Emit an instruction  }
procedure EmitLn(s: string); { Emit an instruction line }

{--------------------------------------------------------------}
implementation
const TAB = ^I;

{--------------------------------------------------------------}
{ Emit an Instruction }

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

{--------------------------------------------------------------}
{ Emit an Instruction, Followed By a Newline }

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

end.
{--------------------------------------------------------------}
```

(Notice that this unit has no initialization clause, so it needs no begin-block.)

Test this unit with the following main program:

```
{--------------------------------------------------------------}
program Test;
uses WinCRT, Input, Output, Scanner, Parser;
begin
WriteLn('MAIN:");
EmitLn('Hello, world!');
end.
{--------------------------------------------------------------}
```

Did you see anything that surprised you? You may have been surprised to see that you needed to type something, even though the main program requires no input. That's because of the initialization in unit `Input`, which still requires something to put into the lookahead character. Sorry, there's no way out of that box, or rather, we don't *want* to get out. Except for simple test cases such as this, we will always want a valid lookahead character, so the right thing to do about this "problem" is ... nothing.

Perhaps more surprisingly, notice that the TAB character had no effect; our line of "instructions" begins at column 1, same as the fake label. That's right: WinCRT doesn't support tabs. We have a problem.

There are a few ways we can deal with this problem. The one thing we can't do is to simply ignore it. Every assembler I've ever used reserves column 1 for labels, and will rebel to see instructions starting there. So, at the very least, we must space the instructions over one column to keep the assembler happy. . That's easy enough to do: Simply change, in procedure `Emit`, the line `Write(TAB, s);` by `Write(' ', s);`.

I must admit that I've wrestled with this problem before, and find myself changing my mind as often as a chameleon changes color. For the purposes we're going to be using, 99% of which will be examining the output code as it's displayed on a CRT, it would be nice to see neatly blocked out "object" code. The line:

```
SUB1: MOVE #4,D0
```

just plain looks neater than the different, but functionally identical code,

```
SUB1:
 MOVE #4,D0
```

In test versions of my code, I included a more sophisticated version of the procedure `PostLabel`, that avoids having labels on separate lines, but rather defers the printing of a label so it can end up on the same line as the associated instruction. As recently as an hour ago, my version of unit `Output` provided full support for tabs, using an internal column count variable and software to manage it. I had, if I do say so myself, some rather elegant code to support the tab mechanism, with a minimum of code bloat. It was awfully tempting to show you the "prettyprint" version, if for no other reason than to show off the elegance.

Nevertheless, the code of the "elegant" version was considerably more complex and larger. Since then, I've had second thoughts. In spite of our desire to see pretty output, the inescapable fact is that the two versions of the `MAIN:` code fragment shown above are functionally identical; the assembler, which is the ultimate destination of the code, couldn't care less which version it gets, except that the prettier version will contain more characters, therefore will use more disk space and take longer to assemble. but the prettier one not only takes more code to generate, but will create a larger output file, with many more space characters than the minimum needed. When you look at it that way, it's not very hard to decide which approach to use, is it?

What finally clinched the issue for me was a reminder to consider my own first commandment: KISS. Although I was pretty proud of all my elegant little tricks to implement tabbing, I had to remind myself that, to paraphrase Senator Barry Goldwater, elegance in the pursuit of complexity is no virtue. Another wise man once wrote, "Any idiot can design a Rolls-Royce. It takes a genius to design a VW." So the elegant, tab-friendly version of `Output` is history, and what you see is the simple, compact, VW version.

The Error Unit

Our next set of routines are those that handle errors. To refresh your memory, we take the approach, pioneered by Borland in Turbo Pascal, of halting on the first error. Not only does this greatly simplify our code, by completely avoiding the sticky issue of error recovery, but it also makes much more sense, in my opinion, in an interactive environment. I know this may be an extreme position, but I consider the practice of reporting all errors in a program to be an anachronism, a holdover from the days of batch processing. It's time to scuttle the practice. So there.

In our original Cradle, we had two error-handling procedures: `Error`, which didn't halt, and `Abort`, which did. But I don't think we ever found a use for the procedure that didn't halt, so in the new, lean and mean unit `Errors`, shown next, procedure `Error` takes the place of `Abort`.

```
{-------------------------------------------------------------}
unit Errors;
{-------------------------------------------------------------}
interface
procedure Error(s: string);
procedure Expected(s: string);

{-------------------------------------------------------------}
implementation

{-------------------------------------------------------------}
{ Write error Message and Halt }

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

{-------------------------------------------------------------}
{ Write "<something> Expected" }

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

end.
{-------------------------------------------------------------}
```

As usual, here's a test program:

```
{-------------------------------------------------------------}
program Test;
uses WinCRT, Input, Output, Errors;

begin
Expected('Integer');
end.
{-------------------------------------------------------------}
```

Have you noticed that the `uses` line in our main program keeps getting longer? That's OK. In the final version, the main program will only call procedures in our parser, so its use clause will only have a couple of entries. But for now, it's probably best to include all the units so we can test procedures in them.

Scanning and Parsing

The classical compiler architecture consists of separate modules for the lexical scanner, which supplies tokens in the language, and the parser, which tries to make sense of the tokens as syntax elements. If you can still remember what we did in earlier installments, you'll recall that we didn't do things that way. Because we're using a predictive parser, we can almost always tell what language element is coming next, just by examining the lookahead character. Therefore, we found no need to prefetch tokens, as a scanner would do.

But, even though there is no functional procedure called `Scanner`, it still makes sense to separate the scanning functions from the parsing functions. So I've created two more units called, amazingly enough, `Scanner` and `Parser`. The `Scanner` unit contains all of the routines known as recognizers. Some of these, such as `IsAlpha`, are pure boolean routines which operate on the lookahead character only. The other routines are those which collect tokens, such as identifiers and numeric constants. The `Parser` unit will contain all of the routines making up the recursive-descent parser. The general rule should be that unit Parser contains all of the information that is language-specific; in other words, the syntax of the language should be wholly contained in `Parser`. In

an ideal world, this rule should be true to the extent that we can change the compiler to compile a different language, merely by replacing the single unit, `Parser`.

In practice, things are almost never this pure. There's always a small amount of "leakage" of syntax rules into the scanner as well. For example, the rules concerning what makes up a legal identifier or constant may vary from language to language. In some languages, the rules concerning comments permit them to be filtered by the scanner, while in others they do not. So in practice, both units are likely to end up having language-dependent components, but the changes required to the scanner should be relatively trivial.

Now, recall that we've used two versions of the scanner routines: One that handled only single-character tokens, which we used for a number of our tests, and another that provided full support for multi-character tokens. Now that we have our software separated into units, I don't anticipate getting much use out of the single-character version, but it doesn't cost us much to provide for both. I've created two versions of the `Scanner` unit. The first one, called `Scanner1`, contains the single-digit version of the recognizers:

```
{-------------------------------------------------------------}
unit Scanner1;
{-------------------------------------------------------------}
interface
uses Input, Errors;

function IsAlpha(c: char): boolean;
function IsDigit(c: char): boolean;
function IsAlNum(c: char): boolean;
function IsAddop(c: char): boolean;
function IsMulop(c: char): boolean;

procedure Match(x: char);
function GetName: char;
function GetNumber: char;

{-------------------------------------------------------------}
implementation

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

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

{-------------------------------------------------------------}
{ Recognize a Numeric Character }

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

{-------------------------------------------------------------}
{ Recognize an Alphanumeric Character }

function IsAlnum(c: char): boolean;
begin
IsAlnum := IsAlpha(c) or IsDigit(c);
end;

{-------------------------------------------------------------}
{ Recognize an Addition Operator }

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

```
IsAddop := c in ['+','-'];
end;

{---------------------------------------------------------------}
{ Recognize a Multiplication Operator }

function IsMulop(c: char): boolean;
begin
IsMulop := c in ['*','/'];
end;

{---------------------------------------------------------------}
{ Match One Character }

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

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

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

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

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

end.
{---------------------------------------------------------------}
```

The following code fragment of the main program provides a good test of the scanner. For brevity, I'll only include the executable code here; the rest remains the same. Don't forget, though, to add the name `Scanner1` to the `uses` clause.

```
Write(GetName);
Match('=');
Write(GetNumber);
Match('+');
WriteLn(GetName);
```

This code will recognize all sentences of the form `x=0+y` where `x` and `y` can be any single-character variable names, and `0` any digit. The code should reject all other sentences, and give a meaningful error message. If it did, you're in good shape and we can proceed.

The Scanner Unit

The next, and by far the most important, version of the scanner is the one that handles the multi-character tokens that all real languages must have. Only the two functions, `GetName` and `GetNumber`, change between the

two units, but just to be sure there are no mistakes, I've reproduced the entire unit here. This is unit `Scanner`:

```
{--------------------------------------------------------------}
unit Scanner;
{--------------------------------------------------------------}
interface
uses Input, Errors;

function IsAlpha(c: char): boolean;
function IsDigit(c: char): boolean;
function IsAlNum(c: char): boolean;
function IsAddop(c: char): boolean;
function IsMulop(c: char): boolean;

procedure Match(x: char);
function GetName: string;
function GetNumber: longint;

{--------------------------------------------------------------}
implementation

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

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

{--------------------------------------------------------------}
{ Recognize a Numeric Character }

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

{--------------------------------------------------------------}
{ Recognize an Alphanumeric Character }

function IsAlnum(c: char): boolean;
begin
IsAlnum := IsAlpha(c) or IsDigit(c);
end;

{--------------------------------------------------------------}
{ Recognize an Addition Operator }

function IsAddop(c: char): boolean;
begin
IsAddop := c in ['+','-'];
end;

{--------------------------------------------------------------}
{ Recognize a Multiplication Operator }

function IsMulop(c: char): boolean;
begin
IsMulop := c in ['*','/'];
end;
```

```
{--------------------------------------------------------------}
{ Match One Character }

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

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

function GetName: string;
var n: string;
begin
n := '';
if not IsAlpha(Look) then Expected('Name');
while IsAlnum(Look) do begin
n := n + Look;
GetChar;
end;
GetName := n;
end;

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

function GetNumber: string;
var n: string;
begin
n := '';
if not IsDigit(Look) then Expected('Integer');
while IsDigit(Look) do begin
n := n + Look;
GetChar;
end;
GetNumber := n;
end;

end.
{--------------------------------------------------------------}
```

The same test program will test this scanner, also. Simply change the `uses` clause to use `Scanner` instead of `Scanner1`. Now you should be able to type multi-character names and numbers.

Decisions, Decisions

In spite of the relative simplicity of both scanners, a lot of thought has gone into them, and a lot of decisions had to be made. I'd like to share those thoughts with you now so you can make your own educated decision, appropriate for your application. First, note that both versions of `GetName` translate the input characters to upper case. Obviously, there was a design decision made here, and this is one of those cases where the language syntax splatters over into the scanner. In the C language, the case of characters in identifiers is significant. For such a language, we obviously can't map the characters to upper case. The design I'm using assumes a language like Pascal, where the case of characters doesn't matter. For such languages, it's easier to go ahead and map all identifiers to upper case in the scanner, so we don't have to worry later on when we're comparing strings for equality.

We could have even gone a step further, and map the characters to upper case right as they come in, in `GetChar`. This approach works too, and I've used it in the past, but it's too confining. Specifically, it will also map characters that may be part of quoted strings, which is not a good idea. So if you're going to map to upper case at all, `GetName` is the proper place to do it.

Note that the function `GetNumber` in this scanner returns a string, just as `GetName` does. This is another one of those things I've oscillated about almost daily, and the last swing was all of ten minutes ago. The alternative approach, and one I've used many times in past installments, returns an integer result.

Both approaches have their good points. Since we're fetching a number, the approach that immediately comes to mind is to return it as an integer. But bear in mind that the eventual use of the number will be in a write statement that goes back to the outside world. Someone -- either us or the code hidden inside the write statement -- is going to have to convert the number back to a string again. Turbo Pascal includes such string conversion routines, but why use them if we don't have to? Why convert a number from string to integer form, only to convert it right back again in the code generator, only a few statements later?

Furthermore, as you'll soon see, we're going to need a temporary storage spot for the value of the token we've fetched. If we treat the number in its string form, we can store the value of either a variable or a number in the same string. Otherwise, we'll have to create a second, integer variable.

On the other hand, we'll find that carrying the number as a string virtually eliminates any chance of optimization later on. As we get to the point where we are beginning to concern ourselves with code generation, we'll encounter cases in which we're doing arithmetic on constants. For such cases, it's really foolish to generate code that performs the constant arithmetic at run time. Far better to let the parser do the arithmetic at compile time, and merely code the result. To do that, we'll wish we had the constants stored as integers rather than strings.

What finally swung me back over to the string approach was an aggressive application of the KISS test, plus reminding myself that we've studiously avoided issues of code efficiency. One of the things that makes our simple-minded parsing work, without the complexities of a "real" compiler, is that we've said up front that we aren't concerned about code efficiency. That gives us a lot of freedom to do things the easy way rather than the efficient one, and it's a freedom we must be careful not to abandon voluntarily, in spite of the urges for efficiency shouting in our ear. In addition to being a big believer in the KISS philosophy, I'm also an advocate of "lazy programming," which in this context means, don't program anything until you need it. As P.J. Plauger says, "Never put off until tomorrow what you can put off indefinitely." Over the years, much code has been written to provide for eventualities that never happened. I've learned that lesson myself, from bitter experience. So the bottom line is: We won't convert to an integer here because we don't need to. It's as simple as that.

For those of you who still think we may need the integer version (and indeed we may), here it is:

```
{--------------------------------------------------------------}
{ Get a Number (integer version) }

function GetNumber: longint;
var n: longint;
begin
n := 0;
if not IsDigit(Look) then Expected('Integer');
while IsDigit(Look) do begin
n := 10 * n + (Ord(Look) - Ord('0'));
GetChar;
end;
GetNumber := n;
end;
{--------------------------------------------------------------}
```

You might file this one away, as I intend to, for a rainy day.

Parsing

At this point, we have distributed all the routines that made up our Cradle into units that we can draw upon as we need them. Obviously, they will evolve further as we continue the process of bootstrapping ourselves up again, but for the most part their content, and certainly the architecture that they imply, is defined. What remains is to embody the language syntax into the parser unit. We won't do much of that in this installment, but I do want to do a little, just to leave us with the good feeling that we still know what we're doing. So before we go, let's generate just enough of a parser to process single factors in an expression. In the process, we'll also, by necessity, find we have created a code generator unit, as well.

Remember the very first installment of this series? We read an integer value, say n, and generated the code to load it into the `D0` register via an immediate move: `MOVE #n,D0`. Shortly afterwards, we repeated the process for a variable, `MOVE X(PC),D0` and then for a factor that could be either constant or variable. For old times sake, let's revisit that process. Define the following new unit:

```
{-----------------------------------------------------------}
unit Parser;
{-----------------------------------------------------------}
interface
uses Input, Scanner, Errors, CodeGen;
procedure Factor;

{-----------------------------------------------------------}
implementation

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

procedure Factor;
begin
LoadConstant(GetNumber);
end;

end.
{-----------------------------------------------------------}
```

As you can see, this unit calls a procedure, `LoadConstant`, which actually effects the output of the assembly-language code. The unit also uses a new unit, `CodeGen`. This step represents the last major change in our architecture, from earlier installments: The removal of the machine-dependent code to a separate unit. If I have my way, there will not be a single line of code, outside of `CodeGen`, that betrays the fact that we're targeting the 68000 CPU. And this is one place I think that having my way is quite feasible.

For those of you who wish I were using the 80x86 architecture (or any other one) instead of the 68000, here's your answer: Merely replace `CodeGen` with one suitable for your CPU of choice.

So far, our code generator has only one procedure in it. Here's the unit:

```
{-----------------------------------------------------------}
unit CodeGen;

{-----------------------------------------------------------}
interface
uses Output;
procedure LoadConstant(n: string);

{-----------------------------------------------------------}
implementation

{-----------------------------------------------------------}
{ Load the Primary Register with a Constant }

procedure LoadConstant(n: string);
begin
EmitLn('MOVE #' + n + ',D0' );
end;

end.
{-----------------------------------------------------------}


Copy and compile this unit, and execute the following main
program:

{-----------------------------------------------------------}
program Main;
uses WinCRT, Input, Output, Errors, Scanner, Parser;
begin
Factor;
```

```
end.
{--------------------------------------------------------------}
```

There it is, the generated code, just as we hoped it would be.

Now, I hope you can begin to see the advantage of the unit-based architecture of our new design. Here we have a main program that's all of five lines long. That's all of the program we need to see, unless we choose to see more. And yet, all those units are sitting there, patiently waiting to serve us. We can have our cake and eat it too, in that we have simple and short code, but powerful allies. What remains to be done is to flesh out the units to match the capabilities of earlier installments. We'll do that in the next installment, but before I close, let's finish out the parsing of a factor, just to satisfy ourselves that we still know how. The final version of `CodeGen` includes the new procedure, `LoadVariable`:

```
{--------------------------------------------------------------}
unit CodeGen;

{--------------------------------------------------------------}
interface
uses Output;
procedure LoadConstant(n: string);
procedure LoadVariable(Name: string);

{--------------------------------------------------------------}
implementation

{--------------------------------------------------------------}
{ Load the Primary Register with a Constant }

procedure LoadConstant(n: string);
begin
EmitLn('MOVE #' + n + ',D0' );
end;

{--------------------------------------------------------------}
{ Load a Variable to the Primary Register }

procedure LoadVariable(Name: string);
begin
EmitLn('MOVE ' + Name + '(PC),D0');
end;


end.
{--------------------------------------------------------------}
```

The parser unit itself doesn't change, but we have a more complex version of procedure `Factor`:

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

procedure Factor;
begin
if IsDigit(Look) then
LoadConstant(GetNumber)
else if IsAlpha(Look)then
LoadVariable(GetName)
else
Error('Unrecognized character ' + Look);
end;
{--------------------------------------------------------------}
```

Now, without altering the main program, you should find that our program will process either a variable or a constant factor. At this point, our architecture is almost complete; we have units to do all the dirty work,

and enough code in the parser and code generator to demonstrate that everything works. What remains is to flesh out the units we've defined, particularly the parser and code generator, to support the more complex syntax elements that make up a real language. Since we've done this many times before in earlier installments, it shouldn't take long to get us back to where we were before the long hiatus. We'll continue this process in Installment 16, coming soon. See you then.

[1]Crenshaw, J.W., "Object-Oriented Design of Assemblers and Compilers," Proc. Software Development '91 Conference, Miller Freeman, San Francisco, CA, February 1991, pp. 143-155.

[2]Crenshaw, J.W., "A Perfect Marriage," Computer Language, Volume 8, #6, June 1991, pp. 44-55.

[3]Crenshaw, J.W., "Syntax-Driven Object-Oriented Design," Proc. 1991 Embedded Systems Conference, Miller Freeman, San Francisco, CA, September 1991, pp. 45-60.

Chapter 16

Part XVI: Unit Construction - 29 May 1995

Introduction

This series of tutorials promises to be perhaps one of the longest-running mini-series in history, rivalled only by the delay in Volume IV of Knuth. Begun in 1988, the series ran into a four-year hiatus in 1990 when the "cares of this world," changes in priorities and interests, and the need to make a living seemed to stall it out after Installment 14. Those of you with loads of patience were finally rewarded, in the spring of last year, with the long-awaited Installment 15. In it, I began to try to steer the series back on track, and in the process, to make it easier to continue on to the goal, which is to provide you with not only enough understanding of the difficult subject of compiler theory, but also enough tools, in the form of canned subroutines and concepts, so that you would be able to continue on your own and become proficient enough to build your own parsers and translators. Because of that long hiatus, I thought it appropriate to go back and review the concepts we have covered so far, and to redo some of the software, as well. In the past, we've never concerned ourselves much with the development of production-quality software tools ... after all, I was trying to teach (and learn) concepts, not production practice. To do that, I tended to give you, not complete compilers or parsers, but only those snippets of code that illustrated the particular point we were considering at the moment.

I still believe that's a good way to learn any subject; no one wants to have to make changes to 100,000 line programs just to try out a new idea. But the idea of just dealing with code snippets, rather than complete programs, also has its drawbacks in that we often seemed to be writing the same code fragments over and over. Although repetition has been thoroughly proven to be a good way to learn new ideas, it's also true that one can have too much of a good thing. By the time I had completed Installment 14 I seemed to have reached the limits of my abilities to juggle multiple files and multiple versions of the same software functions. Who knows, perhaps that's one reason I seemed to have run out of gas at that point.

Fortunately, the later versions of Borland's Turbo Pascal allow us to have our cake and eat it too. By using their concept of separately compilable units, we can still write small subroutines and functions, and keep our main programs and test programs small and simple. But, once written, the code in the Pascal units will always be there for us to use, and linking them in is totally painless and transparent.

Since, by now, most of you are programming in either C or C++, I know what you're thinking: Borland, with their Turbo Pascal (TP), certainly didn't invent the concept of separately compilable modules. And of course you're right. But if you've not used TP lately, or ever, you may not realize just how painless the whole process is. Even in C or C++, you still have to build a make file, either manually or by telling the compiler how to do so. You must also list, using `extern` statements or header files, the functions you want to import. In TP, you don't even have to do that. You need only name the units you wish to use, and all of their procedures automatically become available.

It's not my intention to get into a language-war debate here, so I won't pursue the subject any further. Even I no longer use Pascal on my job ... I use C at work and C++ for my articles in Embedded Systems Programming and other magazines. Believe me, when I set out to resurrect this series, I thought long and hard about switching both languages and target systems to the ones that we're all using these days, C/C++ and PC architecture, and possibly object-oriented methods as well. In the end, I felt it would cause more confusion than the hiatus itself has. And after all, Pascal still remains one of the best possible languages for teaching, not to mention production programming. Finally, TP still compiles at the speed of light, much faster than competing C/C++ compilers. And Borland's smart linker, used in TP but not in their C++ products, is second to none. Aside from being much faster than Microsoft-compatible linkers, the Borland smart linker will cull unused procedures and data items, even to the extent of trimming them out of defined objects if they're not needed. For one of the few times in our lives, we don't have to compromise between completeness and efficiency.

When we're writing a TP unit, we can make it as complete as we like, including any member functions and data items we may think we will ever need, confident that doing so will not create unwanted bloat in the compiled and linked executable.

The point, really, is simply this: By using TP's unit mechanism, we can have all the advantages and convenience of writing small, seemingly stand-alone test programs, without having to constantly rewrite the support functions that we need. Once written, the TP units sit there, quietly waiting to do their duty and give us the support we need, when we need it.

Using this principle, in Installment 15 I set out to minimize our tendency to re-invent the wheel by organizing our code into separate Turbo Pascal units, each containing different parts of the compiler. We ended up with the following units:

- `Input`
- `Output`
- `Errors`
- `Scanner`
- `Parser`
- `CodeGen`

Each of these units serves a different function, and encapsulates specific areas of functionality. The `Input` and `Output` units, as their name implies, provide character stream I/O and the all-important lookahead character upon which our predictive parser is based. The Errors unit, of course, provides standard error handling. The Scanner unit contains all of our boolean functions such as `IsAlpha`, and the routines `GetName` and `GetNumber`, which process multi-character tokens.

The two units we'll be working with the most, and the ones that most represent the personality of our compiler, are Parser and CodeGen. Theoretically, the `Parser` unit should encapsulate all aspects of the compiler that depend on the syntax of the compiled language (though, as we saw last time, a small amount of this syntax spills over into `Scanner`). Similarly, the code generator unit, `CodeGen`, contains all of the code dependent upon the target machine. In this installment, we'll be continuing with the development of the functions in these two all-important units.

Just Like Classical?

Before we proceed, however, I think I should clarify the relationship between, and the functionality of these units. Those of you who are familiar with compiler theory as taught in universities will, of course, recognize the names, Scanner, Parser, and CodeGen, all of which are components of a classical compiler implementation. You may be thinking that I've abandoned my commitment to the KISS philosophy, and drifted towards a more conventional architecture than we once had. A closer look, however, should convince you that, while the names are similar, the functionalities are quite different.

Together, the scanner and parser of a classical implementation comprise the so-called "front end," and the code generator, the back end. The front end routines process the language-dependent, syntax-related aspects of the source language, while the code generator, or back end, deals with the target machine-dependent parts of the problem. In classical compilers, the two ends communicate via a file of instructions written in an intermediate language (IL).

Typically, a classical scanner is a single procedure, operating as a co-procedure with the parser. It "tokenizes" the source file, reading it character by character, recognizing language elements, translating them into tokens, and passing them along to the parser. You can think of the parser as an abstract machine, executing "op codes," which are the tokens. Similarly, the parser generates op codes of a second abstract machine, which mechanizes the IL. Typically, the IL file is written to disk by the parser, and read back again by the code generator.

Our organization is quite different. We have no lexical scanner, in the classical sense; our unit `Scanner`, though it has a similar name, is not a single procedure or co-procedure, but merely a set of separate subroutines which are called by the parser as needed.

Similarly, the classical code generator, the back end, is a translator in its own right, reading an IL "source" file, and emitting an object file. Our code generator doesn't work that way. In our compiler, there IS no intermediate language; every construct in the source language syntax is converted into assembly language as it is recognized by the parser. Like Scanner, the unit `CodeGen` consists of individual procedures which are called by the parser as needed.

This "code 'em as you find 'em" philosophy may not produce the world's most efficient code -- for example, we haven't provided (yet!) a convenient place for an optimizer to work its magic -- but it sure does simplify the compiler, doesn't it?

And that observation prompts me to reflect, once again, on how we have managed to reduce a compiler's functions to such comparatively simple terms. I've waxed eloquent on this subject in past installments, so I won't belabor the point too much here. However, because of the time that's elapsed since those last soliloquies, I hope you'll grant me just a little time to remind myself, as well as you, how we got here. We got here by applying several principles that writers of commercial compilers seldom have the luxury of using. These are:

- The KISS philosophy: Never do things the hard way without a reason
- Lazy coding: Never put off until tomorrow what you can put of forever (with credits to P.J. Plauger)
- Skepticism: Stubborn refusal to do something just because that's the way it's always been done.
- Acceptance of inefficient code
- Rejection of arbitrary constraints

As I've reviewed the history of compiler construction, I've learned that virtually every production compiler in history has suffered from pre-imposed conditions that strongly influenced its design. The original FORTRAN compiler of John Backus, et al, had to compete with assembly language, and therefore was constrained to produce extremely efficient code. The IBM compilers for the minicomputers of the 70's had to run in the very small RAM memories then available -- as small as 4k. The early Ada compiler had to compile itself. Per Brinch Hansen decreed that his Pascal compiler developed for the IBM PC must execute in a 64k machine. Compilers developed in Computer Science courses had to compile the widest variety of languages, and therefore required LALR parsers.

In each of these cases, these preconceived constraints literally dominated the design of the compiler.

A good example is Brinch Hansen's compiler, described in his excellent book, "Brinch Hansen on Pascal Compilers" (highly recommended). Though his compiler is one of the most clear and un-obscure compiler implementations I've seen, that one decision, to compile large files in a small RAM, totally drives the design, and he ends up with not just one, but many intermediate files, together with the drivers to write and read them.

In time, the architectures resulting from such decisions have found their way into computer science lore as articles of faith. In this one man's opinion, it's time that they were re-examined critically. The conditions, environments, and requirements that led to classical architectures are not the same as the ones we have today. There's no reason to believe the solutions should be the same, either.

In this tutorial, we've followed the leads of such pioneers in the world of small compilers for Pcs as Leor Zolman, Ron Cain, and James Hendrix, who didn't know enough compiler theory to know that they "couldn't do it that way." We have resolutely refused to accept arbitrary constraints, but rather have done whatever was easy. As a result, we have evolved an architecture that, while quite different from the classical one, gets the job done in very simple and straightforward fashion.

I'll end this philosophizing with an observation re the notion of an intermediate language. While I've noted before that we don't have one in our compiler, that's not exactly true; we *do* have one, or at least are evolving one, in the sense that we are defining code generation functions for the parser to call. In essence, every call to a code generation procedure can be thought of as an instruction in an intermediate language. Should we ever find it necessary to formalize an intermediate language, this is the way we would do it: emit codes from the parser, each representing a call to one of the code generator procedures, and then process each code by calling those procedures in a separate pass, implemented in a back end. Frankly, I don't see that we'll ever find a need for this approach, but there is the connection, if you choose to follow it, between the classical and the current approaches.

Fleshing out the Parser

Though I promised you, somewhere along about Installment 14, that we'd never again write every single function from scratch, I ended up starting to do just that in Installment 15. One reason: that long hiatus between the two installments made a review seem eminently justified ... even imperative, both for you and for me. More importantly, the decision to collect the procedures into modules (units), forced us to look at each one yet again, whether we wanted to or not. And, finally and frankly, I've had some new ideas in the last four years that warranted a fresh look at some old friends. When I first began this series, I was frankly amazed, and pleased, to learn just how simple parsing routines can be made. But this last time around, I've surprised myself yet again, and been able to make them just that last little bit simpler, yet.

Still, because of this total rewrite of the parsing modules, I was only able to include so much in the last installment. Because of this, our hero, the parser, when last seen, was a shadow of its former self, consisting of only enough code to parse and process a factor consisting of either a variable or a constant. The main effort of this current installment will be to help flesh out the parser to its former glory. In the process, I hope you'll bear with me if we sometimes cover ground we've long since been over and dealt with.

First, let's take care of a problem that we've addressed before: Our current version of procedure `Factor`, as we left it in Installment 15, can't handle negative arguments. To fix that, we'll introduce the procedure `SignedFactor`:

```
{---------------------------------------------------------------}
{ Parse and Translate a Factor with Optional Sign }

procedure SignedFactor;
var Sign: char;
begin
Sign := Look;
if IsAddop(Look) then
GetChar;
Factor;
if Sign = '-' then Negate;
end;
{---------------------------------------------------------------}
```

Note that this procedure calls a new code generation routine, `Negate`:

```
{---------------------------------------------------------------}
{ Negate Primary }

procedure Negate;
begin
EmitLn('NEG D0');
end;
{---------------------------------------------------------------}
```

(Here, and elsewhere in this series, I'm only going to show you the new routines. I'm counting on you to put them into the proper unit, which you should normally have no trouble identifying. Don't forget to add the procedure's prototype to the interface section of the unit.)

In the main program, simply change the procedure called from `Factor` to `SignedFactor`, and give the code a test. Isn't it neat how the Turbo linker and make facility handle all the details?

Yes, I know, the code isn't very efficient. If we input a number, `-3`, the generated code is:

```
MOVE #3,D0
NEG D0
```

which is really, really dumb. We can do better, of course, by simply pre-appending a minus sign to the string passed to `LoadConstant`, but it adds a few lines of code to `SignedFactor`, and I'm applying the KISS philosophy very aggressively here. What's more, to tell the truth, I think I'm subconsciously enjoying generating "really, really dumb" code, so I can have the pleasure of watching it get dramatically better when we get into optimization methods.

Most of you have never heard of John Spray, so allow me to introduce him to you here. John's from New Zealand, and used to teach computer science at one of its universities. John wrote a compiler for the Motorola 6809, based on a delightful, Pascal-like language of his own design called "Whimsical." He later ported the compiler to the 68000, and for awhile it was the only compiler I had for my homebrewed 68000 system.

For the record, one of my standard tests for any new compiler is to see how the compiler deals with a null program like:

```
program main;
begin
end.
```

My test is to measure the time required to compile and link, and the size of the object file generated. The undisputed *loser* in the test is the DEC C compiler for the VAX, which took 60 seconds to compile, on a VAX 11/780, and generated a 50k object file. John's compiler is the undisputed, once, future, and forever king in the code size department. Given the null program, Whimsical generates precisely two bytes of code, implementing the one instruction, `RET`.

By setting a compiler option to generate an include file rather than a standalone program, John can even cut this size, from two bytes to zero! Sort of hard to beat a null object file, wouldn't you say?

Needless to say, I consider John to be something of an expert on code optimization, and I like what he has to say: "The best way to optimize is not to have to optimize at all, but to produce good code in the first place." Words to live by. When we get started on optimization, we'll follow John's advice, and our first step will not be to add a peephole optimizer or other after-the-fact device, but to improve the quality of the code emitted before optimization. So make a note of `SignedFactor` as a good first candidate for attention, and for now we'll leave it be.

Terms and Expressions

I'm sure you know what's coming next: We must, yet again, create the rest of the procedures that implement the recursive-descent parsing of an expression. We all know that the hierarchy of procedures for arithmetic expressions is:

```
expression
term
factor
```

However, for now let's continue to do things one step at a time, and consider only expressions with additive terms in them. The code to implement expressions, including a possibly signed first term, is shown next:

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

procedure Expression;
begin
SignedFactor;
while IsAddop(Look) do
case Look of
'+': Add;
'-': Subtract;
end;
end;
{--------------------------------------------------------------}
```

This procedure calls two other procedures to process the operations:

```
{--------------------------------------------------------------}
{ Parse and Translate an Addition Operation }

procedure Add;
begin
Match('+');
Push;
Factor;
PopAdd;
end;


{--------------------------------------------------------------}
{ Parse and Translate a Subtraction Operation }

procedure Subtract;
begin
Match('-');
Push;
Factor;
PopSub;
end;
{--------------------------------------------------------------}
```

The three procedures `Push`, `PopAdd`, and `PopSub` are new code generation routines. As the name implies, procedure `Push` generates code to push the primary register (`D0`, in our 68000 implementation) to the stack. `PopAdd` and `PopSub` pop the top of the stack again, and add it to, or subtract it from, the primary register. The code is shown next:

```
{-------------------------------------------------------------}
{ Push Primary to Stack }

procedure Push;
begin
EmitLn('MOVE D0,-(SP)');
end;

{-------------------------------------------------------------}
{ Add TOS to Primary }

procedure PopAdd;
begin
EmitLn('ADD (SP)+,D0');
end;

{-------------------------------------------------------------}
{ Subtract TOS from Primary }

procedure PopSub;
begin
EmitLn('SUB (SP)+,D0');
Negate;
end;
{-------------------------------------------------------------}
```

Add these routines to `Parser` and `CodeGen`, and change the `main` program to call `Expression`. Voila!

The next step, of course, is to add the capability for dealing with multiplicative terms. To that end, we'll add a procedure `Term`, and code generation procedures `PopMul` and `PopDiv`. These code generation procedures are shown next:

```
{-------------------------------------------------------------}
{ Multiply TOS by Primary }

procedure PopMul;
begin
EmitLn('MULS (SP)+,D0');
end;

{-------------------------------------------------------------}
{ Divide Primary by TOS }

procedure PopDiv;
begin
EmitLn('MOVE (SP)+,D7');
EmitLn('EXT.L D7');
EmitLn('DIVS D0,D7');
EmitLn('MOVE D7,D0');
end;
{-------------------------------------------------------------}
```

I admit, the division routine is a little busy, but there's no help for it. Unfortunately, while the 68000 CPU allows a division using the top of stack (TOS), it wants the arguments in the wrong order, just as it does for subtraction. So our only recourse is to pop the stack to a scratch register (`D7`), perform the division there, and then move the result back to our primary register, `D0`. Note the use of signed multiply and divide operations.

This follows an implied, but unstated, assumption, that all our variables will be signed 16-bit integers. This decision will come back to haunt us later, when we start looking at multiple data types, type conversions, etc.

Our procedure `Term` is virtually a clone of `Expression`, and looks like this:

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

procedure Term;
begin
Factor;
while IsMulop(Look) do
case Look of
'*': Multiply;
'/': Divide;
end;
end;
{--------------------------------------------------------------}
```

Our next step is to change some names. `SignedFactor` now becomes `SignedTerm`, and the calls to `Factor` in `Expression`, `Add`, `Subtract` and `SignedTerm` get changed to call `Term`:

```
{--------------------------------------------------------------}
{ Parse and Translate a Term with Optional Leading Sign }

procedure SignedTerm;
var Sign: char;
begin
Sign := Look;
if IsAddop(Look) then
GetChar;
Term;
if Sign = '-' then Negate;
end;
{--------------------------------------------------------------}
...
{--------------------------------------------------------------}
{ Parse and Translate an Expression }

procedure Expression;
begin
SignedTerm;
while IsAddop(Look) do
case Look of
'+': Add;
'-': Subtract;
end;
end;
{--------------------------------------------------------------}
```

If memory serves me correctly, we once had *both* a procedure `SignedFactor` and a procedure `SignedTerm`. I had reasons for doing that at the time ... they had to do with the handling of Boolean algebra and, in particular, the Boolean "not" function. But certainly, for arithmetic operations, that duplication isn't necessary. In an expression like `-x*y`, it's very apparent that the sign goes with the whole *term*, `x*y`, and not just the factor `x`, and that's the way `Expression` is coded.

Test this new code by executing `Main`. It still calls `Expression`, so you should now be able to deal with expressions containing any of the four arithmetic operators.

Our last bit of business, as far as expressions goes, is to modify procedure `Factor` to allow for parenthetical expressions. By using a recursive call to `Expression`, we can reduce the needed code to virtually nothing. Five lines added to `Factor` do the job:

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

procedure Factor;
begin
if Look ='(' then begin
Match('(');
Expression;
Match(')');
end
else if IsDigit(Look) then
LoadConstant(GetNumber)
else if IsAlpha(Look)then
LoadVariable(GetName)
else
Error('Unrecognized character ' + Look);
end;
{---------------------------------------------------------------}
```

At this point, your "compiler" should be able to handle any legal expression you can throw at it. Better yet, it should reject all illegal ones!

Assignments

As long as we're this close, we might as well create the code to deal with an assignment statement. This code needs only to remember the name of the target variable where we are to store the result of an expression, call `Expression`, then store the number. The procedure is shown next:

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

procedure Assignment;
var Name: string;
begin
Name := GetName;
Match('=');
Expression;
StoreVariable(Name);
end;
{---------------------------------------------------------------}
```

The assignment calls for yet another code generation routine:

```
{---------------------------------------------------------------}
{ Store the Primary Register to a Variable }

procedure StoreVariable(Name: string);
begin
EmitLn('LEA ' + Name + '(PC),A0');
EmitLn('MOVE D0,(A0)');
end;
{---------------------------------------------------------------}
```

Now, change the call in `Main` to call `Assignment`, and you should see a full assignment statement being processed correctly. Pretty neat, eh? And painless, too.

In the past, we've always tried to show BNF relations to define the syntax we're developing. I haven't done that here, and it's high time I did. Here's the BNF:

```
<factor>      ::= <variable> | <constant> | '(' <expression> ')'
<signed_term> ::= [<addop>] <term>
<term>        ::= <factor> (<mulop> <factor>)*
<expression>  ::= <signed_term> (<addop> <term>)*
<assignment>  ::= <variable> '=' <expression>
```

Booleans

The next step, as we've learned several times before, is to add Boolean algebra. In the past, this step has at least doubled the amount of code we've had to write. As I've gone over this step in my mind, I've found myself diverging more and more from what we did in previous installments. To refresh your memory, I noted that Pascal treats the Boolean operators pretty much identically to the way it treats arithmetic ones. A Boolean "and" has the same precedence level as multiplication, and the "or" as addition. C, on the other hand, sets them at different precedence levels, and all told has a whopping 17 levels. In our earlier work, I chose something in between, with seven levels. As a result, we ended up with things called Boolean expressions, paralleling in most details the arithmetic expressions, but at a different precedence level. All of this, as it turned out, came about because I didn't like having to put parentheses around the Boolean expressions in statements like `IF (c >= 'A') and (c <= 'Z') then ...`.

In retrospect, that seems a pretty petty reason to add many layers of complexity to the parser. Perhaps more to the point, I'm not sure I was even able to avoid the parens.

For kicks, let's start anew, taking a more Pascal-ish approach, and just treat the Boolean operators at the same precedence level as the arithmetic ones. We'll see where it leads us. If it seems to be down the garden path, we can always backtrack to the earlier approach.

For starters, we'll add the "addition-level" operators to `Expression`. That's easily done; first, modify the function `IsAddop` in unit `Scanner` to include two extra operators: `|` for "or," and `~` for "exclusive or":

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

Next, we must include the parsing of the operators in procedure `Expression`:

```
{---------------------------------------------------------------}
procedure Expression;
begin
SignedTerm;
while IsAddop(Look) do
case Look of
'+': Add;
'-': Subtract;
'|': _Or;
'~': _Xor;
end;
{---------------------------------------------------------------}
end;
```

(The underscores are needed, of course, because `or` and `xor` are reserved words in Turbo Pascal.)

Next, the procedures `_Or` and `_Xor`:

```
{---------------------------------------------------------------}
{ Parse and Translate a Subtraction Operation }

procedure _Or;
begin
Match('|');
Push;
Term;
```

```
PopOr;
end;

{---------------------------------------------------------------}
{ Parse and Translate a Subtraction Operation }

procedure _Xor;
begin
Match('~');
Push;
Term;
PopXor;
end;
{---------------------------------------------------------------}
```

And, finally, the new code generator procedures:

```
{---------------------------------------------------------------}
{ Or TOS with Primary }

procedure PopOr;
begin
EmitLn('OR (SP)+,D0');
end;

{---------------------------------------------------------------}
{ Exclusive-Or TOS with Primary }

procedure PopXor;
begin
EmitLn('EOR (SP)+,D0');
end;
{---------------------------------------------------------------}
```

Now, let's test the translator (you might want to change the call in `Main` back to a call to `Expression`, just to avoid having to type `x=` for an assignment every time).

So far, so good. The parser nicely handles expressions of the form `x|y~z`.

Unfortunately, it also does nothing to protect us from mixing Boolean and arithmetic algebra. It will merrily generate code for `(a+b)*(c~d)`.

We've talked about this a bit, in the past. In general the rules for what operations are legal or not cannot be enforced by the parser itself, because they are not part of the syntax of the language, but rather its semantics. A compiler that doesn't allow mixed-mode expressions of this sort must recognize that c and d are Boolean variables, rather than numeric ones, and balk at multiplying them in the next step. But this "policing" can't be done by the parser; it must be handled somewhere between the parser and the code generator. We aren't in a position to enforce such rules yet, because we haven't got either a way of declaring types, or a symbol table to store the types in. So, for what we've got to work with at the moment, the parser is doing precisely what it's supposed to do.

Anyway, are we sure that we *don't* want to allow mixed-type operations? We made the decision some time ago (or, at least, I did) to adopt the value `0000` as a Boolean `false`, and `-1`, or `FFFFh`, as a Boolean `true`. The nice part about this choice is that bitwise operations work exactly the same way as logical ones. In other words, when we do an operation on one bit of a logical variable, we do it on all of them. This means that we don't need to distinguish between logical and bitwise operations, as is done in C with the operators `&` and `&&`, and `|` and `||`. Reducing the number of operators by half certainly doesn't seem all bad.

From the point of view of the data in storage, of course, the computer and compiler couldn't care less whether the number FFFFh represents the logical `TRUE`, or the numeric `-1`. Should we? I sort of think not. I can think of many examples (though they might be frowned upon as "tricky" code) where the ability to mix the types might come in handy. Example, the Dirac delta function, which could be coded in one simple line, `-(x=0)`, or the absolute value function (*definitely* tricky code!), `x*(1+2*(x<0))`.

Please note, I'm not advocating coding like this as a way of life. I'd almost certainly write these functions in more readable form, using `IF`s, just to keep from confusing later maintainers. Still, a moral question arises: Do we have the right to *enforce* our ideas of good coding practice on the programmer, but writing the language

so they can't do anything else? That's what Nicklaus Wirth did, in many places in Pascal, and Pascal has been criticized for it -- for not being as "forgiving" as C.

An interesting parallel presents itself in the example of the Motorola 68000 design. Though Motorola brags loudly about the orthogonality of their instruction set, the fact is that it's far from orthogonal. For example, you can read a variable from its address:

```
MOVE X,D0 (where X is the name of a variable)
```

but you can't write in the same way. To write, you must load an address register with the address of `X`. The same is true for PC-relative addressing:

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

When you begin asking how such non-orthogonal behavior came about, you find that someone in Motorola had some theories about how software should be written. Specifically, in this case, they decided that self-modifying code, which you can implement using PC-relative writes, is a Bad Thing. Therefore, they designed the processor to prohibit it. Unfortunately, in the process they also prohibited *all* writes of the forms shown above, however benign. Note that this was not something done by default. Extra design work had to be done, and extra gates added, to destroy the natural orthogonality of the instruction set.

One of the lessons I've learned from life: If you have two choices, and can't decide which one to take, sometimes the best thing to do is nothing. Why add extra gates to a processor to enforce some stranger's idea of good programming practice? Leave the instructions in, and let the programmers debate what good programming practice is. Similarly, why should we add extra code to our parser, to test for and prevent conditions that the user might prefer to do, anyway? I'd rather leave the compiler simple, and let the software experts debate whether the practices should be used or not.

All of which serves as rationalization for my decision as to how to prevent mixed-type arithmetic: I won't. For a language intended for systems programming, the fewer rules, the better. If you don't agree, and want to test for such conditions, we can do it once we have a symbol table.

Boolean "and"

With that bit of philosophy out of the way, we can press on to the "and" operator, which goes into procedure `Term`. By now, you can probably do this without me, but here's the code, anyway:

In `Scanner`,

```
{-------------------------------------------------------------}
function IsMulop(c: char): boolean;
begin
IsMulop := c in ['*','/', '&'];
end;
{-------------------------------------------------------------}
```

In `Parser`,

```
{-------------------------------------------------------------}
procedure Term;
begin
Factor;
while IsMulop(Look) do
case Look of
'*': Multiply;
'/': Divide;
'&': _And;
end;
end;

{-------------------------------------------------------------}
{ Parse and Translate a Boolean And Operation }

procedure _And;
```

```
begin
Match('&');
Push;
Factor;
PopAnd;
end;
{---------------------------------------------------------------}
```

and in `CodeGen`,

```
{---------------------------------------------------------------}
{ And Primary with TOS }

procedure PopAnd;
begin
EmitLn('AND (SP)+,D0');
end;
{---------------------------------------------------------------}
```

Your parser should now be able to process almost any sort of logical expression, and (should you be so inclined), mixed-mode expressions as well.

Why not "all sorts of logical expressions"? Because, so far, we haven't dealt with the logical "not" operator, and this is where it gets tricky. The logical "not" operator seems, at first glance, to be identical in its behavior to the unary minus, so my first thought was to let the exclusive or operator, `~`, double as the unary "not." That didn't work. In my first attempt, procedure `SignedTerm` simply ate my `~`, because the character passed the test for an addop, but `SignedTerm` ignores all addops except `-`. It would have been easy enough to add another line to `SignedTerm`, but that would still not solve the problem, because note that `Expression` only accepts a signed term for the *first* argument.

Mathematically, an expression like `-a * -b` makes little or no sense, and the parser should flag it as an error. But the same expression, using a logical "not," makes perfect sense: `not a and not b`.

In the case of these unary operators, choosing to make them act the same way seems an artificial force fit, sacrificing reasonable behavior on the altar of implementational ease. While I'm all for keeping the implementation as simple as possible, I don't think we should do so at the expense of reasonableness. Patching like this would be missing the main point, which is that the logical "not" is simply NOT the same kind of animal as the unary minus. Consider the exclusive or, which is most naturally written as `a~b ::= (a and not b) or (not a and b)`.

If we allow the "not" to modify the whole term, the last term in parentheses would be interpreted as `not(a and b)`.

which is not the same thing at all. So it's clear that the logical "not" must be thought of as connected to the FACTOR, not the term.

The idea of overloading the `~` operator also makes no sense from a mathematical point of view. The implication of the unary minus is that it's equivalent to a subtraction from zero: `-x <=> 0-x`.

In fact, in one of my more simple-minded versions of `Expression`, I reacted to a leading addop by simply preloading a zero, then processing the operator as though it were a binary operator. But a "not" is not equivalent to an exclusive or with zero ... that would just give back the original number. Instead, it's an exclusive or with `FFFFh`, or `-1`.

In short, the seeming parallel between the unary "not" and the unary minus falls apart under closer scrutiny. "not" modifies the factor, not the term, and it is not related to either the unary minus nor the exclusive or. Therefore, it deserves a symbol to call its own. What better symbol than the obvious one, also used by C, the `!` character? Using the rules about the way we think the "not" should behave, we should be able to code the exclusive or (assuming we'd ever need to), in the very natural form: `a & !b | !a & b`.

Note that no parentheses are required -- the precedence levels we've chosen automatically take care of things.

If you're keeping score on the precedence levels, this definition puts the `!` at the top of the heap. The levels become:

1. `!`
2. `-` (unary)
3. `*`, `/`, `&`
4. `+`, `-`, `|`, `~`

Looking at this list, it's certainly not hard to see why we had trouble using ~ as the "not" symbol!

So how do we mechanize the rules? In the same way as we did with `SignedTerm`, but at the factor level. We'll define a procedure `NotFactor`:

```
{---------------------------------------------------------------}
{ Parse and Translate a Factor with Optional "Not" }

procedure NotFactor;
begin
if Look ='!' then begin
Match('!');
Factor;
Notit;
end
else
Factor;
end;
{---------------------------------------------------------------}
```

and call it from all the places where we formerly called `Factor`, i.e., from `Term`, `Multiply`, `Divide`, and `_And`. Note the new code generation procedure:

```
{---------------------------------------------------------------}
{ Bitwise Not Primary }

procedure NotIt;
begin
EmitLn('EOR #-1,D0');
end;

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

Try this now, with a few simple cases. In fact, try that exclusive or example, `a&!b|!a&b`.

You should get the code (without the comments, of course):

```
MOVE A(PC),D0 ; load a
MOVE D0,-(SP) ; push it
MOVE B(PC),D0 ; load b
EOR #-1,D0 ; not it
AND (SP)+,D0 ; and with a
MOVE D0,-(SP) ; push result
MOVE A(PC),D0 ; load a
EOR #-1,D0 ; not it
MOVE D0,-(SP) ; push it
MOVE B(PC),D0 ; load b
AND (SP)+,D0 ; and with !a
OR (SP)+,D0 ; or with first term
```

That's precisely what we'd like to get. So, at least for both arithmetic and logical operators, our new precedence and new, slimmer syntax hang together. Even the peculiar, but legal, expression with leading addop, `~x`, makes sense. `SignedTerm` ignores the leading `~`, as it should, since the expression is equivalent to `0~x`, which is equal to x.

When we look at the BNF we've created, we find that our boolean algebra now adds only one extra line:

```
<not_factor>  ::= [!] <factor>
<factor>      ::= <variable> | <constant> | '(' <expression> ')'
<signed_term>  ::= [<addop>] <term>
<term>        ::= <not_factor> (<mulop> <not_factor>)*
<expression>   ::= <signed_term> (<addop> <term>)*
<assignment>   ::= <variable> '=' <expression>
```

That's a big improvement over earlier efforts. Will our luck continue to hold when we get to relational operators? We'll find out soon, but it will have to wait for the next installment. We're at a good stopping place, and I'm anxious to get this installment into your hands. It's already been a year since the release of Installment 15. I blush to admit that all of this current installment has been ready for almost as long, with the exception of relational operators. But the information does you no good at all, sitting on my hard disk, and by holding it back until the relational operations were done, I've kept it out of your hands for that long. It's time for me to let go of it and get it out where you can get value from it. Besides, there are quite a number of serious philosophical questions associated with the relational operators, as well, and I'd rather save them for a separate installment where I can do them justice.

Have fun with the new, leaner arithmetic and logical parsing, and I'll see you soon with relationals.