LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

5 March 1994

Part 15: BACK TO THE FUTURE

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 (Refs. 1-3). 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.

REFERENCES

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.