LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

5 June 1989

Part XII: MISCELLANY

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 improtant 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.