# LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

29 May, 1995

## Part 16: UNIT CONSTRUCTION



## 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:

o   The KISS philosophy -- Never do things the hard way without a reason

o   Lazy coding -- Never put off until tomorrow what you can put of forever (with credits to P.J. Plauger)

o   Skepticism -- Stubborn refusal to do something just because that's the way it's always been done.

o   Acceptance of inefficient code

o   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 his 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 IFs, 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 he 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),DO   (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),DO    ; load a
 MOVE D0,-(SP)      ; push it
 MOVE B(PC),DO      ; load b
 EOR #-1,D0     ; not it
 AND (SP)+,D0       ; and with a
 MOVE D0,-(SP)      ; push result
 MOVE A(PC),DO      ; load a
 EOR #-1,D0     ; not it
 MOVE D0,-(SP)      ; push it
 MOVE B(PC),DO      ; 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.