LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

24 July 1988

Part II: EXPRESSION PARSING

GETTING STARTED

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

The purpose of this article is for us to learn how to parse and translate mathematical expressions. What we would like to see as output is a series of assembler-language statements that perform the desired actions. For purposes of definition, an expression is the right-hand side of an equation, as in

```
x = 2*y + 3/(4*z)
```

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

SINGLE DIGITS

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

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

```
//Parse and Translate a Math Expression

void Expression()
```

```
{
   EmitLn("MOVE #" + GetNum() + ",D0")
}
```

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

```
int main()
{
   Init();
   Expression();
   return 0;
}
```

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

CONGRATULATIONS! You have just written a working translator!

OK, I grant you that it's pretty limited. But don't brush it off too lightly. This little "compiler" does, on a very limited scale, exactly what any larger compiler does: it correctly recognizes legal statements in the input "language" that we have defined for it, and it produces correct, executable assembler code, suitable for assembling into object format. Just as importantly, it correctly recognizes statements that are NOT legal, and gives a meaningful error message. Who could ask for more? As we expand our parser, we'd better make sure those two characteristics always hold true.

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

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

BINARY EXPRESSIONS

Now that we have that under our belt, let's branch out a bit. Admittedly, an "expression" consisting of only one character is not going to meet our needs for long, so let's see what we can do to extend it. Suppose we want to handle expressions of the form:

```
                    1+2
 or                 4-3
 or, in general, <term> +/- <term>
```

(That's a bit of Backus-Naur Form, or BNF.)

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

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

{————————————————————————————} { Parse and Translate an Expression }

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

Next, just above Expression enter these two procedures:

{————————————————————————————} { Recognize and Translate an Add }

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

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

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

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

o Term (The OLD Expression) o Add o Subtract o Expression

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

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

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

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

There is a message here: code generated by our parser is less efficient than the code we would write by hand. Get used to it. That's going to be true

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

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

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

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

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

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

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

GENERAL EXPRESSIONS

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

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

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

{————————————————————————————} { Parse and Translate an Expression }

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

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

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

USING THE STACK

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

```
1+(2-(3+(4-5)))
```

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

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

```
-(SP)
```

and a pop, (SP)+ .

So let’s change the EmitLn in Expression to read:

```
EmitLn('MOVE D0,-(SP)');
```

and the two lines in Add and Subtract to

```
EmitLn('ADD (SP)+,D0')
```

and EmitLn(‘SUB (SP)+,D0’),

respectively. Now try the parser again and make sure we haven’t broken it.

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

MULTIPLICATION AND DIVISION

Now let’s get down to some REALLY serious business. As you all know, there are other math operators than “addops” ... expressions can also have multiply and divide operations. You also know that there is an implied operator PRECEDENCE, or hierarchy, associated with expressions, so that in an expression like

```
2 + 3 * 4,
```

we know that we’re supposed to multiply FIRST, then add. (See why we needed the stack?)

In the early days of compiler technology, people used some rather complex techniques to insure that the operator precedence rules were obeyed. It turns out, though, that none of this is necessary ... the rules can be accommodated quite nicely by our top-down parsing technique. Up till now, the only form that we’ve considered for a term is that of a single decimal digit.

More generally, we can define a term as a PRODUCT of FACTORS; i.e.,

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

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

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

{————————————————————} { Parse and Translate a Math Factor }

procedure Factor; begin EmitLn(‘MOVE #’ + GetNum + ‘,D0’) end;

{————————————————————} { Recognize and Translate a Multiply }

procedure Multiply; begin Match(’*‘); Factor; EmitLn(’MULS (SP)+,D0’); end;

{————————————————————-} { Recognize and Translate a Divide }

procedure Divide; begin Match(‘/’); Factor; EmitLn(‘MOVE (SP)+,D1’); EmitLn(‘DIVS D1,D0’); end;

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

procedure Term; begin Factor; while Look in [’*‘,’/’] do begin EmitLn(‘MOVE D0,-(SP)’); case Look of ’*‘: Multiply;’/‘: Divide; else Expected(’Mulop’); end; end; end;

{————————————————————} { Recognize and Translate an Add }

procedure Add; begin Match(‘+’); Term; EmitLn(‘ADD (SP)+,D0’); end;

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

procedure Subtract; begin Match(‘-’); Term; EmitLn(‘SUB (SP)+,D0’); EmitLn(‘NEG D0’); end;

{————————————————————} { Parse and Translate an Expression }

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

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

PARENTHESES

We can wrap up this part of the parser with the addition of parentheses with math expressions. As you know, parentheses are a mechanism to force a desired operator precedence. So, for example, in the expression

```
2*(3+4) ,
```

the parentheses force the addition before the multiply. Much more importantly, though, parentheses give us a mechanism for defining expressions of any degree of complexity, as in

```
(1+2)/((3+4)+(5-6))
```

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

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

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

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

{————————————————————————} { Parse and Translate a Math Factor }

procedure Expression; Forward;

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

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

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

UNARY MINUS

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

```
-1
```

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

```
-(3-2) .
```

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

{————————————————————————} { Parse and Translate an Expression }

procedure Expression; begin if IsAddop(Look) then EmitLn('CLR D0') else Term; while IsAddop(Look) do begin EmitLn('MOVE D0,-(SP)'); case Look of '+': Add; '-': Subtract; else Expected('Addop'); end; end; end; {————————————————————————}

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

{————————————————————————} { Recognize an Addop }

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

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

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

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

A WORD ABOUT OPTIMIZATION

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

There are two basic approaches we can take:

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

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

though, in both the speed, size, and complexity of the compiler. Looking for all those combinations calls for a lot of IF tests, each one of which is a source of error. And, of course, it takes time.

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

o Try to generate better code in the first place

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

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

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

Well, the 68000 has eight data registers. Why not use them as a privately managed stack? The key is to recognize that, at any point in its processing, the parser KNOWS how many items are on the stack, so it can indeed manage it properly. We can define a private "stack pointer" that keeps track of which stack level we're at, and addresses the corresponding register. Procedure Factor, for example, would not cause data to be loaded into register D0, but into whatever

the current "top-of-stack" register happened to be.

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

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

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

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