

Writing a C Compiler, Part 3

Dec 15, 2017

This is the third post in a series. Read part 1 [here](#).

This week we'll add binary operations to support basic arithmetic. We'll figure out how to correctly handle operator precedence and associativity. You can find the accompanying tests [here](#).

Week 3: Binary Operators

This week we're adding several binary operations (operators that take two values):

- Addition `+`
- Subtraction `-`
- Multiplication `*`
- Division `/`

As usual, we'll update each stage of the compiler to support these operations.

Lexing

Each of the operators above will require a new token, except for subtraction – we already have a `-` token. It gets tokenized the same way whether it's a subtraction or negation operator; we'll figure out how to interpret it during the parsing stage. Arithmetic expressions can also contain parentheses, but we already have tokens for those too, so we don't need to change our lexer at all to handle them.

Here's the full list of tokens we need to support. Tokens from previous weeks are at the top, new tokens are bolded at the bottom:

- Open brace `{`
- Close brace `}`
- Open parenthesis `(`
- Close parenthesis `)`
- Semicolon `;`

- Int keyword `int`
- Return keyword `return`
- Identifier `[a-zA-Z]\w*`
- Integer literal `[0-9]+`
- Minus `-`
- Bitwise complement `~`
- Logical negation `!`
- **Addition** `+`
- **Multiplication** `*`
- **Division** `/`

☑ Task:

Update the *lex* function to handle the new tokens. It should work for all stage 1, 2, and 3 examples in the test suite, including the invalid ones.

Parsing

This week we'll need to add another expression type to our AST: binary operations. Here's the latest set of definitions for our AST nodes; only the definition of `exp` has changed:

```
program = Program(function_declaration)
function_declaration = Function(string, statement) //string is the function name
statement = Return(exp)
exp = BinOp(binary_operator, exp, exp)
    | UnOp(unary_operator, exp)
    | Constant(int)
```

Note that we now distinguish between binary and unary operators in our AST definition. For example, `-` as in negation and `-` as in subtraction would be different types/classes/whatever in our AST. Here's how we might construct the AST for `2 - (-3)`:

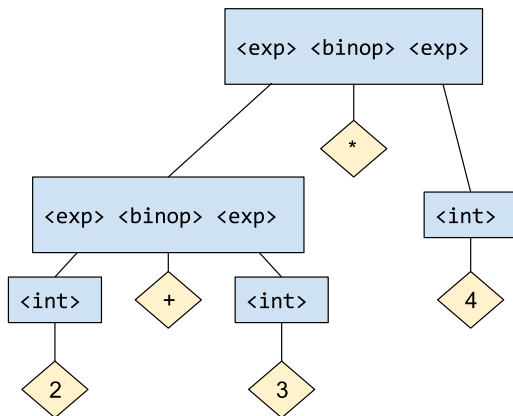
```
two = Const(2)
three = Const(3)
neg_three = UnOp(NEG, three)
exp = BinOp(MINUS, two, neg_three)
```

We also need to change the definition of `<exp>` in our grammar. The most obvious definition is something like this:

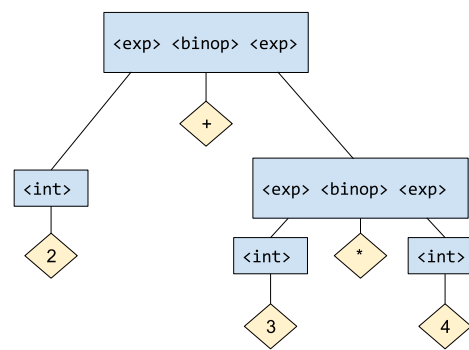
```
<exp> ::= <exp> <binary_op> <exp> | <unary_op> <exp> | "(" <exp> ")" | <int>
```

But there are several related problems with this grammar:

1. **It doesn't handle operator precedence.** Consider the expression `2 + 3 * 4`. Using the grammar above, you can construct two possible parse trees:



Tree #1



Tree #2

Using the first parse tree, this expression evaluates to `(2 + 3) * 4 = 24`. Using the second, it's `2 + (3 * 4) = 14`. According to the C standard and mathematical convention, `*` has higher precedence than `+`, so the second parse tree is correct. Our grammar has to encode this precedence somehow.

This is a problem with our unary operations too – according to this grammar, `~2 + 3` could be parsed as `~(2 + 3)`, which is of course wrong.

2. **It doesn't handle associativity.** Operations at the same precedence level should be evaluated left-to-right¹. For example `1 - 2 - 3` should be parsed as `(1 - 2) - 3`. But, according to the grammar above, parsing it as `1 - (2 - 3)` is also valid.

3. **It's left-recursive.** In the grammar above, one of the production rules for `<exp>` is:

```
<exp> <binary_op> <exp>
```

In this production rule, the left-most (i.e. first) symbol is also `<exp>` — that's what left-recursive means. Left-recursive grammars aren't *incorrect*, but recursive descent (RD) parsers can't handle them². We'll talk about *why* this is a problem later in this post.

Let's start by tackling problem #1, precedence³. We'll handle unary operators first – they always have higher precedence than binary operators. A unary operator should only be applied to a whole expression if:

- the expression is a single integer (e.g. `~4`)
- the expression is wrapped in parentheses (e.g. `~(1+1)`), or
- the expression is itself a unary operation (e.g. `~!8`, `~(2+2)`).

To express this, we're going to need another symbol in our grammar to refer to "an expression a unary operator can be applied to". We'll call it a factor. We'll rewrite our grammar like this:

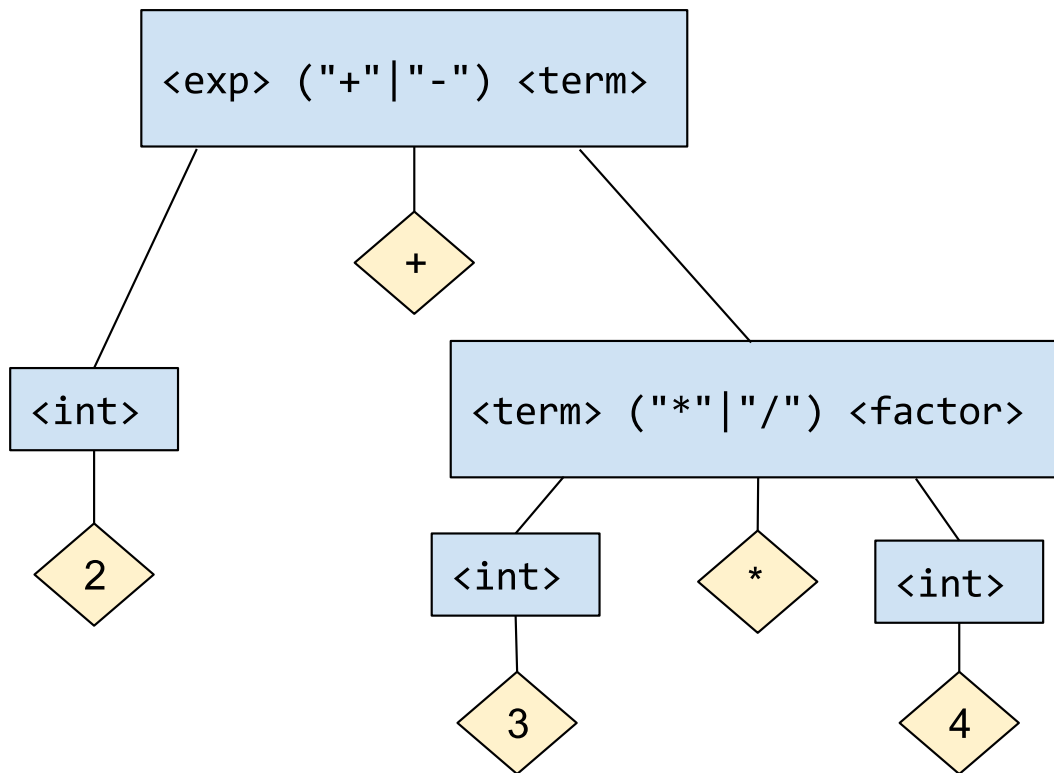
```
<exp> ::= <exp> <binary_op> <exp> | <factor>
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int>
```

We've now created two levels of precedence: one for binary operations, one for unary operations. We're also handling parentheses correctly – putting an expression inside parentheses forces higher precedence.

We can make a similar change to force `*` and `/` to be higher precedence than `+` and `-`. We added a `<factor>` symbol before, representing the operands of unary operations. Now we'll add a `<term>` symbol, representing the operands of multiplication and division⁴.

```
<exp> ::= <exp> ("+" | "-") <exp> | <term>
<term> ::= <term> ("*" | "/") <term> | <factor>
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int>
```

This grammar encodes operator precedence correctly. There's now only one possible parse tree for `2 + 3 * 4`:



That's problem #1 solved. Problem #2 was that this grammar didn't handle associativity. *If* you're not using an RD parser, you generally use left recursive production rules for left-associative operations, and right recursive rules for right-associative operations. In that case, we could rewrite the rule for `<exp>` like so:

```
<exp> ::= <exp> ("+" | "-") <term> | <term>
```

This would force addition and subtraction to be left-associative; you can't parse `1 - 2 - 3` as `1 - (2 - 3)` because `2 - 3` isn't a term.

But we are using an RD parser, so we can't handle this left recursive rule. To understand why this won't work, let's try writing a function to parse expressions according to this rule.

```
def parse_expression(tokens):
    //determine which of two production rules applies:
    // * <exp> ("+" | "-") <term>
    // * <term>
    if is_term(tokens): //how do we figure this out???
        return parse_term(tokens)
    else:
```

```
//recursively call parse_expression to handle it
e1 = parse_expression(tokens) //recurse forever 🐼
```

To figure out which production rule to use, we can't just look at the first token or two – we need to know if there's a `+` or `-` operation anywhere in this expression. And if we determine that this expression is a sum or difference, we're going to call `parse_expression` recursively forever. In order to not do that, we'd need to find the last `<term>` at the end of the expression, parse and remove *that*, then go back and parse the rest. Both of these problems (figuring out which production rule to use, and parsing the last term at the end) require us to look ahead an arbitrary number of tokens until we hit the end of the expression. You might be able to get this approach to work – I'm not sure if there are existing parsing algorithms similar to this – but it will be complicated, and it definitely won't be a recursive descent parser. So we're not going to do that.

If, on the other hand, we just switched around `<term>` and `<exp>` to avoid left recursion, we'd have this rule:

```
<exp> ::= <term> ("+" | "-") <exp> | <term>
```

This is easy to parse but wrong – it's *right*-associative. Using this grammar, you would have to parse `1 - 2 - 3` as `1 - (2 - 3)`.

So our options seem to be an unparseable left recursive grammar, or an incorrect right recursive grammar. Luckily, there's another solution. We'll introduce repetition into our grammar, so we can define an expression as a term, possibly plus or minus a term, possibly plus or minus another term...and so on forever. In EBNF notation, wrapping something in curly braces (`{ }`) means it can be repeated zero or more times. Here's the **final** grammar we'll be using for expressions this week:

```
<exp> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/" ) <factor> }
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int>
```

This grammar handles precedence correctly, it isn't left recursive and it isn't right-associative. However, it's not really left-associative either. Before, an `<exp>` was a binary operation with two terms – now it has an arbitrary number of terms. If you have a bunch of operations at the same precedence level (like in `1 - 2 - 3`), this grammar doesn't provide any way to group them into sub-expressions.

That's okay, though! Our grammar doesn't need to correspond perfectly with our AST. We can still build our AST in a left-associative way. We'll parse the first term, and then, if there are any

more terms, we process them in a loop, constructing a new BinOp node at each iteration. Here's the pseudocode for this:

```
def parse_expression(toks):
    term = parse_term(toks) //pops off some tokens
    next = toks.peek() //check the next token, but don't pop it off the list
    while next == PLUS or next == MINUS: //there's another term!
        op = convert_to_op(toks.next())
        next_term = parse_term(toks) //pops off some more tokens
        term = BinOp(op, term, next_term)
        next = toks.peek()

    return t1
```

We can use exactly the same approach in `parse_term`.

`parse_factor` is straightforward, since it doesn't have to handle associativity. We'll look at the first token to identify which production rule to use; pop off constants and make sure they have the value we expect; and call other functions to handle non-terminal symbols.

```
def parse_factor(toks)
    next = toks.next()
    if next == OPEN_PAREN:
        //<factor> ::= "(" <exp> ")"
        exp = parse_exp(toks) //parse expression inside parens
        if toks.next() != CLOSE_PAREN: //make sure parens are balanced
            fail()
        return exp
    else if is_unop(next)
        //<factor> ::= <unary_op> <factor>
        op = convert_to_op(next)
        factor = parse_factor(toks)
        return UnOp(op, factor)
    else if next.type == "INT":
        //<factor> ::= <int>
        return Const(convert_to_int(next))
    else:
        fail()
```

Just for the sake of completeness, here's this week's complete grammar, including the new stuff above (expressions, terms, factors) and stuff that hasn't changed since last week (functions, statements, etc.):

```

<program> ::= <function>
<function> ::= "int" <id> "(" ")" "{" <statement> "}"
<statement> ::= "return" <exp> ";"
<exp> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/" ) <factor> }
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int>

```

☑ Task:

Update your expression-parsing code to handle addition, subtraction, multiplication, and division. It should successfully parse all valid stage 1, 2, and 3 examples in the test suite, and fail on all invalid stage 1, 2, and 3 examples. Manually inspect the AST for each example to make sure it handles associativity and operator precedence correctly. If you haven't written a pretty printer yet, you'll probably need to do that now.

Code Generation

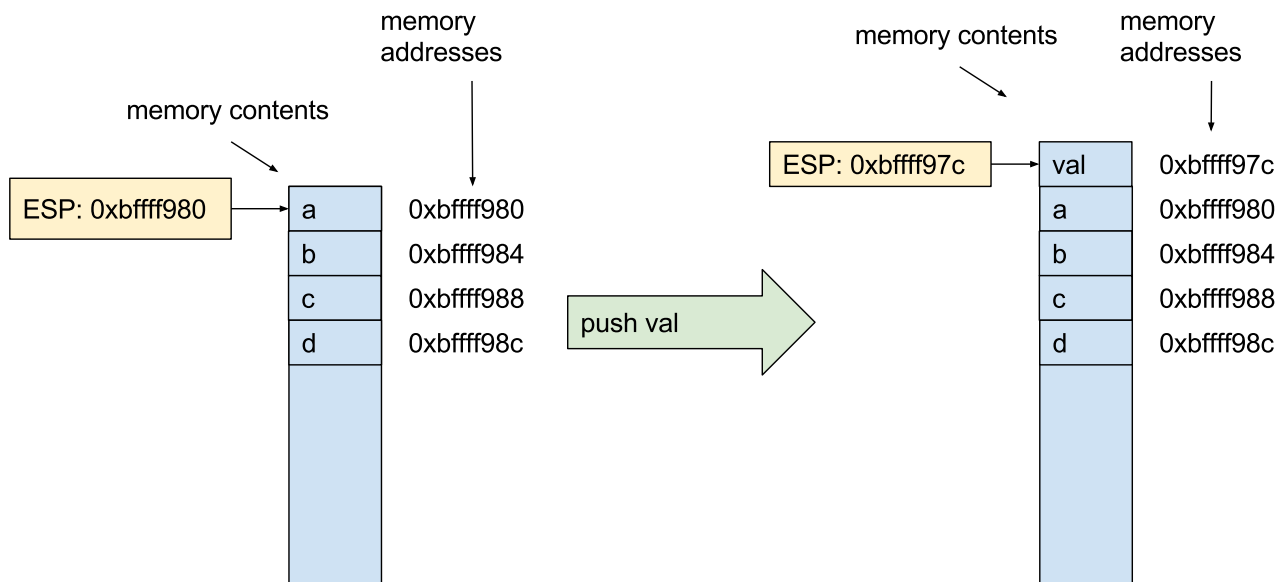
There's a new challenge in the code generation stage this week. To handle a binary expression, like `e1 + e2`, our generated assembly needs to:

- Calculate `e1` and save it somewhere.
- Calculate `e2`.
- Add `e1` to `e2`, and store the result in EAX.

So, we need somewhere to save the first operand. Saving it in a register would be complicated; the second operand can itself contain subexpressions, so it might also need to save intermediate results in a register, potentially overwriting `e1`⁵. Instead, we'll save the first operand on the stack.

Let's talk about the stack briefly. Every process on a computer has some memory. This memory is divided into several segments, one of which is the **call stack**, or just the stack. The address of the top of the stack is stored in the ESP register, aka the stack pointer. Like with most stacks, you can push things onto the top, or pop things off the top; x86 includes `push` and `pop` instructions to do just that. One confusing thing about the stack is that it grows towards *lower* memory addresses – when you push something onto the stack, you *decrement* ESP. The processor relies on ESP to figure out where the top of the stack is. So, `pushl val` does the following⁶:

- Writes `val` to the next empty spot on the stack (i.e. ESP - 4)
- Decrements ESP by 4, so it contains the memory address of `val`.



Along the same lines, `popl dest` does the following:

- Reads value from the top of the stack (i.e. the value at the memory address in ESP).
- Copies that value into `dest`, which is a register or other memory location
- Increments ESP by 4, so it points to the value just below `val`.

But right now, all you really need to know is that there's a stack, `push` puts things on it and `pop` takes things off it. So, here's our assembly for `e1 + e2`:

```
<CODE FOR e1 GOES HERE>
push %eax ; save value of e1 on the stack
<CODE FOR e2 GOES HERE>
pop %ecx ; pop e1 from the stack into ecx
addl %ecx, %eax ; add e1 to e2, save results in eax
```

You can handle `e1 * e2` exactly the same way, using `imul` instead of `addl`. Subtraction is a bit more complicated because order of operands matters; `subl src, dst` computes `dst - src`, and saves the result in `dst`. You'll need to make sure `e1` is in `dst` and `e2` is in `src` – and, of course, that the result ends up in EAX. Division is even trickier: `idivl dst` treats EDX and EAX as a single, 64-bit register and calculates `[EDX:EAX] / dst`. It then stores the quotient in EAX and the remainder in EDX. To make it work, you'll need to first move `e1` into EAX, and then sign-extend it into EDX using the `cdq` instruction before issuing the `idivl` instruction⁷. You can check the [Intel Software Developer's Manual](#) for more details on any of these instructions.

☑ **Task:**

Update your code-generation pass to emit correct code for addition, subtraction, division, and multiplication. It should succeed on all valid examples and fail on all invalid examples for stages 1, 2, and 3.

Using GDB

If you run into trouble during code generation, you may want to step through it with GDB or LLDB. I'm not going to cover how to use GDB here (I list some tutorials under "Further Reading"), but here are a few tips specifically for stepping through assembly without a symbol table:

- You can use `nexti` and `stepi` to step through one assembly instruction at a time.
- In GDB, but not LLDB, `layout asm` displays the assembly as you step through it, and `layout regs` displays the registers.
- You can set breakpoints at functions (e.g. `b main`) even when your binary doesn't have debug symbols.

Also, running GDB on OS X is a pain in the butt. You can find instructions about how to get it working [here](#), or you can use LLDB instead. I kind of hate LLDB but maybe I'm just not used to it.


Up Next


Next week, I'll be on vacation. [The week after that](#), we'll add more binary operators to support comparisons and boolean logic. See you then!

If you have any questions, corrections, or other feedback, you can [email me](#) or [open an issue](#).

Further Reading

- [Intel® 64 and IA-32 Architectures Software Developer's Manual](#) – the definitive reference on all x86 instructions, which will be useful for the rest of the series. Also a good way to impress people at parties when they ask what you're reading.
- [Some problems of recursive descent parsers](#) – a nice explanation of how RD parsers handle associativity. Several other articles on the same blog are also worth a read:
 - [Recursive descent, LL and predictive parsers](#) – an overview of RD parsing.
 - [Top-Down operator precedence parsing](#) – an intro to Pratt parsers.
 - [Parsing expressions by precedence climbing](#) – an intro to precedence climbing.
- [An Introduction to GDB](#) – a helpful reference if you haven't used GDB before.

¹ Not all operations in C are left-associative, but all of this week's binary operations are. 

² Other types of parsers, like [LALR parsers](#), can handle left recursive grammars just fine. 

³ Alternatively, you can handle operator precedence and associativity is [precedence climbing](#) or [Pratt parsing](#) – they’re [basically the same thing](#). Precedence climbing is pretty easy to understand, and it’s a nice option because it requires less boilerplate code and makes it easier to add more operators later on. I didn’t use it in this blog post because I wanted to present the simplest, most vanilla possible solution, but I will probably use it the next time I write a recursive descent parser. If you decide to use precedence parsing in your own compiler, it shouldn’t impact your ability to follow along with the rest of this series. ➡

⁴ Don’t let the notation trip you up here. `()` indicates a grouping of tokens, so `("*" | "/")` means “either a plus token or a minus token.” `"("` or `")"`, in quotes, means a literal parenthesis token. ➡

⁵ Real compilers usually do save intermediate values in registers, because it’s much faster than saving them on the stack. We won’t do that because allocating registers is complicated and saving them onto the stack is simple. ➡

⁶ This all assumes you’re working with 4-byte words. If you were pushing 8-byte words, for example (either because you were compiling for a 64-bit architecture or because you specified the word size with the `pushq` instruction), you would push it to `ESP - 8` and then decrement `ESP` by 8. ➡

⁷ An earlier version of this post incorrectly said that you should zero out `EDX` before the `idivl` instruction. Thanks to Borna Cafuk for pointing out the error! ➡

Want to become a better programmer? [Join the Recurse Center!](#)

© 2022 Nora Sandler.