Writing a C Compiler, Part 2

Dec 5, 2017

This is the second post in a series. Read part 1 here.

In the last post, we learned to compile programs that return integers. This week we'll do math to those integers. Same as last week, you can find the accompanying tests here.

Week 2: Unary Operators

This week, we're adding three unary operators[1] (i.e. operators that only take one value):

Negation (`-`)

-5 = 0 - 5. In other words, it's a regular negative number.

Bitwise complement (`~`)

This flips every bit in a number[2]. For example:

- 4 is written as 100 in binary.
- The bitwise complement of 100 is 011.
- 011 in decimal is 3.
- So ~4 = 3.

Logical negation (`!`)

The boolean "not" operator. This treats 0 as "false" and everything else as "true".

- !0 = 1
- !(anything else) = 0

Last week we created a compiler with three stages: a lexer, a parser, and a code generator. Now we'll update each stage to handle these new operators.

Lexing

We just need to add each of these operators to our list of tokens. Here's the list of tokens: tokens from last week are at the top, and 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]+`
- **Negation** `-`
- **Bitwise complement** `~`
- **Logical negation** `!`

We can process these new tokens exactly the same way as the other single-character tokens, like braces and parentheses.

☑ `Task:`

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

`Parsing`

Last week we defined several AST nodes, including expressions. We only defined one type of expression: constants. This week, we'll add another type of expression, unary operations. The latest set of definitions is below. Only the definition of `exp` has changed.

```
program = Program(function_declaration)
function_declaration = Function(string, statement) //string is the functio
statement = Return(exp)
exp = UnOp(operator, exp) | Constant(int)
```

Now, an expression can take one of two forms - it can be either a constant, or a unary operation. A unary operation consists of the operator (e.g. `~`), and the operand, which is itself an expression. For example, here's how we could construct the expression `~3`:

```
c = Const(3)
exp = UnOp(COMPLEMENT, c)
```

Our definition of expressions is recursive - expressions can contain other expressions!

This is an expression: `!3`

So is this: `!~-4`

So is this: `!!!!!!!-~~-!3`

We need to update our formal grammar too:

```
<program> ::= <function>
<function> ::= "int" <id> "(" ")" "{" <statement> "}"
<statement> ::= "return" <exp> ";"
<exp> ::= <unary_op> <exp> | <int>
<unary_op> ::= "!" | "~" | "-"
```

Because the definition of an expression is recursive, the function to parse an expression should also be recursive. Here's the pseudocode for parsing an expression:

```
def parse_expression(tokens):
    tok = tokens.next()
    if tok.type == "INT":
        //parse this the same way as before, return a Const node
    else:
        op = get_operator(tok) //convert token to unary_op AST element - f
        inner_exp  = parse_expression(tokens) //HOORAY, RECURSION - this w
        return UnOp(op, inner_exp)
```

☑ Task:

Update your expression-parsing function to handle unary operations. It should successfully parse all valid stage 1 and 2 examples in the test suite, and fail on all invalid stage 1 and 2 examples.

Code Generation

Negation and bitwise complement are super easy; each of them can be accomplished with a single assembly instruction.

`neg` negates the value of its operand[3]. Here's an example:

```
    movl    $3, %eax    ;EAX register contains 3
```

```
    neg     %eax        ;now EAX register contains -3
```

Of course, we need to calculate a value before we can negate it, so we need to recursively generate code for the inner expression, then emit the `neg` instruction.

`not` replaces a value with its bitwise complement. We can use it exactly the same way as `neg`.

Logical negation is a little more complicated. Remember, `return !exp` is equivalent to:

```
if (exp == 0) {
    return 1;
} else {
    return 0;
}
```

Unlike the other operations, which were straightforward bit manipulation, this requires some conditional logic. We can implement it using `cmpl`[4], which compares two values, and `sete` ("set if equal"). `sete` sets its operand to 1 if the result of the last comparison was equal, and 0 otherwise.

Comparision and conditonal instructions like `cmpl` and `sete` are a little weird; `cmpl` doesn't explicitly store the result of the comparison, and `sete` doesn't explicitly refer to that result, or to the values being compared. Both of these instructions - and all comparison and conditional instructions - implicitly refer to the FLAGS register[5]. As the name suggests, the contents of this register are interpreted as an array of one-bit flags, rather than a single integer. These flags are automatically set after every arithmetic operation. The only flag we care about right now is the zero flag (ZF), which is set on if the result of an operation is 0, and set off otherwise.

`cmpl a, b` computes (b - a) and sets FLAGS accordingly. If two values are equal, their difference is 0, so ZF will be set on if and only if the operands to `cmpl` are equal. The `sete` instruction uses ZF to test for equality; it sets its operand to 1 if ZF is on, and 0 if ZF is off. In fact, `setz` ("set if zero") is another mnemonic for the same instruction.

The last gotcha here is that `sete` can only set a *byte*, not an entire word. We'll have it set the AL register, which is just the least-significant byte of EAX. We just need to zero out EAX first[6]; since the result of `!` is always 0 or 1, we dont want to leave any stray higher bits set.

So that was a long explanation, but you can actually implement `!` in just three lines of assembly:

```
    <CODE FOR exp GOES HERE>
    cmpl    $0, %eax    ;set ZF on if exp == 0, set it off otherwise
```

```
    movl    $0, %eax    ;zero out EAX (doesn't change FLAGS)
    sete    %al         ;set AL register (the lower byte of EAX) to 1 iff Z
```

To convince ourselves that this is correct, let's work through this with the expression `!5`:

1. First we move 5 into the EAX register.
2. We compare 0 to 5. 5 != 0, so the ZF flag is set to 0.
3. The EAX register is zeroed out, so we can set it in the next step.
4. We conditionaly set AL; because ZF is 0, we set AL to 0. AL refers to the lower bytes of EAX; the upper bytes were also zeroed in step 3, so now EAX contains 0.

☑ Task:

Update your code-generation pass to emit correct code for `!`, `~` and `-` operations. It should generate correct assembly for all valid stage 1 and stage 2 examples.

Up Next

Next week we'll find out whether 2+2 really does equal 4, by adding some binary operations: addition, subtraction, and more. See you then!

If you have any questions, corrections, or other feedback, you can email me or open an issue.

[1] These are defined on page 89 of the C11 standard. ↩

[2] As remcob correctly pointed out on Hacker News, the bitwise complement of a number really depends on how many bits you're using to represent it. The example above is correct if we assume all integers are 3 bits. But if an integer is 8 bits, then 4 is 00000100, so ~4 is 11111011, or 251. To make things even MORE complicated, 11111011 is only 251 if you're dealing with an unsigned integer - if it's signed, then 11111011 is actually the two's complement representation of -5. ↩

[3] To be really pedantic, `neg` calculates the two's complement of its operand. ↩

[4] Some documentation mentions `mov` and `cmp` instead of `movl` and `cmpl`; later you'll see that "l" suffix on a lot of other instructions too. The "l" stands for "long", and indicates that the operand is 32 bits - a throwback to the time when a word was 16 bits and a long was 32 bits. More here. ↩

[5] Actually, it's FLAGS on 16-bit machines, EFLAGS on 32-bit machines, and RFLAGS on 64-bit machines, so your computer almost definitely doesn't have a FLAGS register. But the flags we care about are all in the lower 16 bits so it doesn't really matter. ↩

[6] `mov $0, %eax` is the most obvious way to zero out a register, but it isn't the quickest. `xor %eax, %eax` is better, for reasons you can read about here. There's a whole slew of ways to zero a register; in your compiler you can use whichever one you want. ↩

Want to become a better programmer? Join the Recurse Center!