Writing a C Compiler, Part 4

Dec 28, 2017

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

This week we're adding some boolean operators (`&&`, `||`) and a whole bunch of relational operators (`<`, `==`, etc.). Since we already know how to handle binary operators, this week will be pretty straightforward. As always, you can find the accompanying tests here.

The test suite is slightly weird this week; the three tests whose names start with `skip_on_failure_` use local variables, which we haven't implemented yet. I've included them because otherwise the test suite can't validate that short-circuiting works correctly. When you run the test suite, they should show up as `NOT_IMPLEMENTED` rather than `FAIL` in the results, and they shouldn't count toward the total number of failures. Once you've implemented local variables, these tests should pass.

Week 4: Even More Binary Operators

We're adding eight new operators this week:

- Logical AND `&&`
- Logical OR `||`
- Equal to `==`
- Not equal to `!=`
- Less than `<`
- Less than or equal to `<=`
- Greater than `>`
- Greater than or equal to `>=`

As usual, we'll update our lexing, parsing, and code generation passes to support these operations.

Lexing

Each new operator corresponds to a new token. Here's the full list of tokens we need to support, with old tokens at the top and new tokens in bold 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 `/`
- **AND** `&&`
- **OR** `||`
- **Equal** `==`
- **Not Equal** `!=`
- **Less than** `<`
- **Less than or equal** `<=`
- **Greater than** `>`
- **Greater than or equal** `>=`

☑ Task:

Update the *lex* function to handle the new tokens. It should work for all valid and invalid stage 1-4 examples in the test suite, except the `skip_on_failure_` ones.

Parsing

Last week, we found that we needed one production rule in our grammar for each operator precedence level. This week we have a lot more precedence levels, which means our grammar will grow a lot. However, our parsing strategy hasn't changed at all; we'll handle our new production rules exactly the same way as the old rules for `exp` and `term`. Honestly, this is going to be pretty tedious, but I hope it will help solidify all the stuff about parsing from last week.

Here are our all binary operators, from highest to lowest precedence[1]:

- Multiplication & division (`*`, `/`)

- Addition & subtraction (`+` , `-`)
- Relational less than/greater than/less than or equal/greater than or equal (`<` , `>` , `<=` , `>=`)
- Relational equal/not equal (`==` , `!=`)
- Logical AND (`&&`)
- Logical OR (`||`)

We handled the first two bullet points last week; the last four are new. We'll add a production rule for each of the last four bullet points. The new grammar is below, with changed/added rules bolded.

```
<program> ::= <function>
<function> ::= "int" <id> "(" ")" "{" <statement> "}"
<statement> ::= "return" <exp> ";"
<exp> ::= <logical-and-exp> { "||" <logical-and-exp> }
<logical-and-exp> ::= <equality-exp> { "&&" <equality-exp> }
<equality-exp> ::= <relational-exp> { ("!=" | "==") <relational-exp> }
<relational-exp> ::= <additive-exp> { ("<" | ">" | "<=" | ">=") <additive-
<additive-exp> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/") <factor> }
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int>
<unary_op> ::= "!" | "~" | "-"
```

`<additive-exp>` is the same as `<exp>` from last week. We had to rename it because `<exp>` now refers to logical OR expressions, which now have lowest precedence.

Last week you wrote `parse_exp` and `parse_term`; now you'll need `parse_relational_exp`, `parse_equality_exp`, etc. Other than handling different operators, these functions will all be identical.

And for the sake of completeness, here's our AST definition:

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

This is identical to last week, except we've added more possible values of `binary_operator`.

☑ Task:

Update your expression-parsing code to handle this week's new binary operators. It should successfully parse all valid stage 1-4 examples in the test suite (except the `skip_on_failure` ones), and fail on all invalid stage 1-4 examples. The test suite doesn't directly verify that your program generates the correct AST, so you'll need to manually inspect the AST for each example to make sure it's right.

Code Generation

Our general approach to code generation for binary operations is the same as last week:

1. Calculate `e1`
2. Push it onto the stack
3. Calculate `e2`
4. Pop `e1` from the stack back into a register
5. Perform the operation on `e1` and `e2`.

All the new stuff will be in step 5.

Relational Operators

Let's handle the relational operators first. Like the logical NOT operator (`!`) in week 2, these return 1 for true results and 0 for false results. These operators are almost identical to `!` except that they compare two expressions to each other, instead of comparing an expression to zero.

Here's the assembly we generated for `!` in week 2:

```
    <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
```

We can modify this slightly to implement `==`:

```
    <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 - e2 is already
    cmpl   %eax, %ecx    ;set ZF on if e1 == e2, 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
```

The `sete` instruction is just one of a whole slew of conditional set instructions. There's also `setne` (set if not equal), `setge` (set if greater than or equal), and so on. To implement `<`, `>`, and the other relational operators, we can generate exactly the same assembly as we used for `==` above, just replacing `sete` with the appropriate conditional set instruction. Easy!

In week 2, we talked about testing for equality with the zero flag (ZF). But we can't use ZF to determine which operand is larger. For that, we need the sign flag (SF), which is set if the result of an operation is negative, like so:

```
    movl $0, %eax ;zero out EAX
    movl $2, %ecx ;ECX = 2
    cmpl $3, %ecx ;compute 2 - 3, set flags
    setl %al      ;set AL if 2 < 3, i.e. if 2 - 3 is negative
```

Now let's talk about `&&` and `||`. I'll use `&` and `|` to indicate bitwise AND and OR, respectively.

Short-Circuit Evaluation

The C11 standard guarantees that evaluation of `&&` and `||` will short-circuit: if we know the result after evaluating the first clause, we don't evaluate the second clause[2]. For example, consider the following line of code:

```
return 0 && foo();
```

Because the first clause is false, we don't need to know the return value of `foo`, so we won't call `foo` at all. Whether `foo` is called won't change the return value on this line, but it could perform I/O, update global variables, or have other important side effects. So making sure that `&&` and `||` short-circuit isn't just a performance optimization; it's required for some programs to execute correctly.

Logical OR

To guarantee that logical OR short-circuits, we'll need to jump over clause 2 when clause 1 is true. We'll follow these steps to calculate `e1 || e2`:

1. Calculate `e1`
2. If the result is 0, jump to the step 4.
3. Set EAX to 1 and jump to the end.
4. Calculate `e2`.
5. If the result is 0, set EAX to 0. Otherwise set EAX to 1.

Step 2 will require a new type of instruction called **conditional jumps**. These are similar to the conditional set instructions, like `sete` and `setne`, that we've already used. The only difference is that instead of setting a byte to 1, they jump to a specific point in the assembly code, which we specify with a label. Here's an example of `je`, the "jump if equal" instruction, in action:

```
    cmpl $0, %eax ; set ZF if EAX == 0
    je _there     ; if ZF is set, go to _there
    movl $1, %eax
    ret
_there:
    movl $2, %eax
    ret
```

If EAX is 0 at the start of this code snippet, it will return 2; otherwise it will return 1. Let's look at exactly what instructions will execute in each case.

First consider the case where EAX is 0 at the start:

1. `cmpl $0, %eax` Because EAX is 0, this will set the zero flag (ZF) to true.
2. `je _there` Because ZF is true, it **will** jump.
3. `movl $2, %eax` This executes next because it's the first instruction after `_there`. It sets EAX to 2.
4. `ret` The return value will be 2.

Now consider the case where EAX isn't zero:

1. `cmpl $0, %eax` Because EAX isn't 0, this will set ZF to false.
2. `je _there` Because ZF is false, it **will not** jump, so this instruction is a no-op.
3. `movl $1, %eax` Since we didn't jump, control passes to the next instruction as usual. It sets EAX to 1.
4. `ret` The return value will be 1.

We'll also need the `jmp` instruction, which performs an unconditional jump. Here's an example of `jmp` in action:

```
    movl $0, %eax ; zero out EAX
    jmp _there    ; go to _there label
    movl $5 %eax  ; this will never execute, we always jump over it
_there:
    ret           ; will always return zero
```

Now that we're familiar with `jmp` and `je`, here's the assembly for `e1 || e2`:

```
    <CODE FOR e1 GOES HERE>
    cmpl $0, %eax            ; check if e1 is true
    je _clause2              ; e1 is 0, so we need to evaluate clause 2
    movl $1, %eax            ; we didn't jump, so e1 is true and therefore
    jmp _end
_clause2:
    <CODE FOR e2 GOES HERE>
    cmpl $0, %eax            ; check if e2 is true
    movl $0, %eax            ; zero out EAX without changing ZF
    setne %al                ; set AL register (the low byte of EAX) to 1
_end:
```

Note that labels have to be unique. This means you can't actually use `_clause2` or `_end` as labels, because you'll have duplicate labels if your program includes more than one logical OR. You should probably write a utility function to generate unique labels. It doesn't have to be fancy; the label generator in nqcc just includes an incrementing counter in every label.

The `_end` label here may look odd, since it doesn't appear to label anything. Actually, it labels whatever comes right after this expression; it just gives us a target to jump over `_clause2`.

Logical AND

Almost identical to logical OR, except we short-circuit if `e1` is 0. We use the `jne` (jump if not equal) instruction. In that case we don't need to move anything into EAX, since 0 is the result we want. Here's the assembly:

```
    <CODE FOR e1 GOES HERE>
    cmpl $0, %eax            ; check if e1 is true
    jne _clause2             ; e1 isn't 0, so we need to evaluate clause 2
    jmp _end
_clause2:
    <CODE FOR e2 GOES HERE>
    cmpl $0, %eax            ; check if e2 is true
    movl $0, %eax            ; zero out EAX without changing ZF
    setne %al                ; set AL register (the low byte of EAX) to 1
_end:
```

As with logical OR, we need to make sure the labels are unique.

☑ Task:

Update your code-generation pass to emit correct code for `&&`, `||`, `==`, `!=`, `<`, `<=`, `>`, and `>=`. It should succeed on all valid examples (except the `skip_on_failure_` ones) and fail on all invalid examples for stages 1-4.

Other Binary Operators

We still haven't implemented all the binary operators! We can't implement assignment operators yet (like `+=` and `-=`), because we don't have support for local variables. But there are other operators you should be able to implement on your own now:

- Modulo `%`
- Bitwise AND `&`
- Bitwise OR `|`
- Bitwise XOR `^`
- Bitwise shift left `<<`
- Bitwise shift right `>>`

This week's tests don't cover these, so it's up to you whether to implement them or skip them.

Up Next

Next week we'll add local variables! That means we'll finally be able to write programs that aren't just return statements. See you then!

Update 2/2/2019

- Updated code generation for logical AND and OR to short-circuit correctly.

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

1 You can find a complete C operator precedence table here. ↩

2 See section 6.5.13, paragraph 4, for logical AND and 6.5.14, paragraph 4 for logical OR. ↩

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