Writing a C Compiler, Part 6

Feb 25, 2018

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

Hi, this blog isn't dead! It was just, uh, resting. I've been swamped with non-blog things for the past few weeks but I'm back on track now, probably, I hope.

Today we'll implement conditional statements and expressions. As usual, accompanying tests are here.

Part 6: Conditionals

In this post we'll add support for two types of conditional constructs:

1. Conditional statements, a.k.a. `if` statements
2. Ternary conditional expressions, which have the form `a ? b : c`. I'll sometimes just call these "conditional expressions".

If Statements

An `if` statement consists of a condition, a substatement that executes if the condition is true, and maybe another substatement that executes if the condition is false. Either of these substatements can be a single statement, like this:

```
if (flag)
  return 0;
```

or a compound statement, like this:

```
if (flag) {
  int a = 1;
  return a*2;
}
```

Adding support for compound statements is a distinct task that we're not going to handle in this post. So for now, we'll only support the first of the examples above, and not the second.

We say a condition is **false** if it evaluates to zero, and **true** otherwise, just like when we implemented boolean operators in earlier posts.

Else If

Note that C doesn't have an explicit `else if` construct. If an `if` keyword immediately follows an `else` keyword, the whole `if` statement gets parsed as the `else` branch. In other words, the following code snippets are equivalent:

```
if (flag)
    return 0;
else if (other_flag)
    return 1;
else
    return 2;
```

```
if (flag)
    return 0;
else {
    if (other_flag)
        return 1;
    else
        return 2;
}
```

Conditional Expressions

These expressions take the following form:

```
a ? b : c
```

If `a` is true, the expression will evaluate to `b`; otherwise it will evaluate to `c`.

Note that we should only execute the expression we actually need. For example, in the following code snippet:

```
0 ? foo() : bar()
```

the function `foo` should never be called. You might be tempted to call both `foo` and `bar`, then discard the result from `foo`, but that would be wrong; `foo` could print to the console, make a network call, or dereference a null pointer and crash the program. Obviously this point is also true of `if` statements – we should execute the `if` branch or the `else` branch but definitely not both.

Conditional expressions and `if` statements might seem very similar, but it's important to remember that statements and expressions are used in totally different ways. For example, an expression has a value, but a statement doesn't. So this is legal:

```
int a = flag ? 2 : 3;
```

but this isn't[1]:

```
//this is bogus
int a = if (flag)
            2;
        else
            3;
```

On the other hand, a statement can contain other statements, but an expression can't contain statements. For example, you can nest a `return` statement inside an `if` statement:

```
if (flag)
    return 0;
```

but you can't have a `return` statement inside a conditional expression:

```
//this is also bogus
flag ? return 1 : return 2;
```

Lexing

We need to define a few more tokens: `if` and `else` keywords for `if` statements, plus `:` and `?` operators for conditional expressions. Here's the full list of tokens, with 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 `>=`
- Assignment `=`
- **If keyword** `if`
- **Else keyword** `else`
- **Colon** `:`
- **Question mark** `?`

☑ Task:

Update the *lex* function to handle the new tokens. It should work for all stage 1-6 examples in the test suite, including the invalid ones.

Parsing

We'll parse conditional expressions and `if` statements totally differently. Let's handle `if` statements first.

If Statements

So far, we've defined three types of statements in our AST: return statements, expressions, and variable declarations. Right now the definition looks like this:

```
statement = Return(exp)
          | Declare(string, exp option) //string is variable name
                                        //exp is optional initializer
          | Exp(exp)
```

We need to add an `If` statement, which has three parts: an expression (the controlling condition), an `if` branch and an optional `else` branch. Here's our updated AST definition for statements:

```
statement = Return(exp)
          | Declare(string, exp option) //string is variable name
                                        //exp is optional initializer
          | Exp(exp)
          | If(exp, statement, statement option) //exp is controlling cond
                                                 //first statement is 'if'
                                                 //second statement is opt
```

Now let's update our grammar. The rule for `if` statements consists of:

- The `if` keyword
- An expression wrapped in parentheses (the condition)
- A statement (executed if the condition is true)
- Optionally, the `else` keyword, followed by another statement (executed if the condition is false)

```
"if" "(" <exp> ")" <statement> [ "else" <statement> ]
```

So the updated grammar for statements looks like this:

```
<statement> ::= "return" <exp> ";"
              | <exp> ";"
              | "int" <id> [ = <exp> ] ";"
              | "if" "(" <exp> ")" <statement> [ "else" <statement> ]
```

Our definition of statements is recursive! But it's not left-recursive, so it's not a problem.

But we have another problem. We defined variable declarations as a type of statement, but declarations in C **aren't statements**. For example, this code snippet isn't valid:

```
//this will throw a compiler error!
if (flag)
```

```
  int i = 0;
```

When we added variable declarations in the last post, it didn't matter whether or not we defined them as statements; we could parse the same subset of C and generate the same assembly either way. Now that we're dealing with more complex structures like `if` statements, that simplification impacts what we can and can't parse, so we need to fix it.

So we need to move `Declare` out of the `statement` type and into its own type. But this introduces a new problem: we've defined a function body as a list of statements, but if declarations aren't statements, then you can't have declarations in a function body. To fix this, we'll need to tweak how we define functions in our AST. Let's introduce some terminology:

- A **block item** is a statement or declaration.
- A **block** or **compound statement** is a list of block items wrapped in curly braces[2].

Function bodies are just a special case of blocks; they contain a list of declarations and statements. To represent them, we'll introduce a new `block_item` type that can hold either a statement or a declaration. This will also come in handy when we add support for blocks in general in the next post. With those changes, the relevant parts of our AST will look like this:

```
statement = Return(exp)
          | Exp(exp)
          | Conditional(exp, statement, statement option) //exp is control
                                                          //first statemen
                                                          //second stateme

declaration = Declare(string, exp option) //string is variable name
                                          //exp is optional initializer

block_item = Statement(statement) | Declaration(declaration)

function_declaration = Function(string, block_item list) //string is the f
```

And here's the updated grammar:

```
<statement> ::= "return" <exp> ";"
              | <exp> ";"
              | "if" "(" <exp> ")" <statement> [ "else" <statement> ]
<declaration> ::= "int" <id> [ = <exp> ] ";"
<block-item> ::= <statement> | <declaration>
<function> ::= "int" <id> "(" ")" "{" { <block-item> } "}"
```

Now that we have our AST and grammar, you should be able to update your compiler to parse conditional statements. You may want to do that before we move on to conditional expressions.

☑ Task:

Update the parsing pass to handle conditional statements. It should successfully parse all valid stage 6 examples in `write_a_c_compiler/stage_6/valid/statement`, and throw an error for all invalid stage 6 examples in `write_a_c_compiler/stage_6/invalid/statement`.

Conditional Expressions

Now let's add ternary conditional expressions. Here's how we've defined our AST for expressions so far:

```
exp = Assign(string, exp)
    | Var(string) //string is variable name
    | BinOp(binary_operator, exp, exp)
    | UnOp(unary_operator, exp)
    | Constant(int)
```

It's straightforward to add a `Conditional` form:

```
exp = Assign(string, exp)
    | Var(string) //string is variable name
    | BinOp(binary_operator, exp, exp)
    | UnOp(unary_operator, exp)
    | Constant(int)
    | Conditional(exp, exp, exp) //the three expressions are the condition
```

We also need to update the grammar rules for expressions, which currently look like this:

```
<exp> ::= <id> "=" <exp> | <logical-or-exp>
<logical-or-exp> ::= <logical-and-exp> { "||" <logical-and-exp> }
...more rules...
```

The conditional operator has lower precedence than assignment (`=`) but higher precedence than logical OR (`||`), and it's right-associative. We can take its grammar rule straight from section 6.5.15 of the C11 standard:

```
<conditional-exp> ::= <logical-or-exp> "?" <exp> ":" <conditional-exp>
```

Let's think about why it's defined this way. I'll refer to the three sub-expressions as **e1**, **e2**, and **e3**, such that a conditional expression has the form `e1 ? e2 : e3`. Expression **e1** has to be a `<logical-or-exp>` because it can't be an assignment expression or a conditional expression. It can't be an assignment expression because assignment has lower precedence than the conditional operator. In other words:

```
a = 1 ? 2 : 3;
```

must be parsed as:

```
a = (1 ? 2 : 3);
```

In our current grammar this is specified unambiguously, but if we instead defined a conditional expression as:

```
<conditional-exp> ::= <exp> "?" <exp> ":" <conditional-exp>
```

then it would be ambiguous; the statement above could also be parsed as:

```
(a = 1) ? 2 : 3;
```

Note that `(a = 1) ? 2 : 3;` is a valid statement, but you need the parentheses in order to parse it that way.

So that's why **e1** can't be an assignment expression. It can't be a conditional expression because `?` is right-associative. In other words:

```
flag1 ? 4 : flag2 ? 6 : 7
```

must be parsed as

```
flag1 ? 4 : (flag2 ? 6 : 7)
```

If we had defined a conditional expression as:

```
<conditional-exp> ::= <conditional-exp> "?" <exp> ":" <conditional-exp>
```

then the example above could also be parsed as:

```
(flag1 ? 4 : flag2) ? 6 : 7
```

and the grammar would be ambiguous.

Expression **e2** in our ternary conditional can take any form; safely fenced in by `?` and `:`, it can't introduce any grammatical ambiguity. You can think of implicit parentheses wrapping everything between `?` and `:`.

Expression **e3** can be another ternary conditional, as in the example `a > b ? 4 : flag ? 6 : 7`. But it *can't* be an assignment statement – why not? Let's look at the following example:

```
flag ? a = 1 : a = 0
```

If we try to compile this with gcc, we'll get something like the following error message:

```
error: expression is not assignable
    flag ? a = 1 : a = 0;
    ~~~~~~~~~~~~~~~~ ^
```

In other words, gcc tried to parse the expression like this:

```
(flag ? a = 1 : a) = 0
```

This obviously doesn't work because the expression on the left isn't a variable[3]. You might wonder why we can't use the following grammar rule:

```
<conditional-exp> ::= <logical-or-exp> "?" <exp> ":" <exp>
```

Then gcc could just parse it like this:

```
flag ? a = 1 : (a = 0)
```

That grammar rule would work fine; in fact, that's how conditional expressions are defined in C++[4]. I don't know why it's different in C, but if *you* know I'd like to hear from you.

We also need a way to specify expressions that aren't conditionals, so we'll make the 'conditional' part of this grammar rule optional[5]:

```
<conditional-exp> ::= <logical-or-exp> [ "?" <exp> ":" <conditional-exp> ]
```

Anyway, we now know the correct grammar. Here are all the new and updated grammar rules concerning expressions:

```
<exp> ::= <id> "=" <exp> | <conditional-exp>
<conditional-exp> ::= <logical-or-exp> [ "?" <exp> ":" <conditional-exp> ]
<logical-or-exp> ::= <logical-and-exp> { "||" <logical-and-exp> }
...
```

☑ Task:

Update the parsing pass to handle ternary conditional expressions. At this point, it should successfully parse all valid stage 6 examples, and throw an error for all invalid examples.

Put It All Together

For the sake of completeness, here's our full AST definition and grammar, with new and changed parts bolded:

AST:

```
program = Program(function_declaration)

function_declaration = Function(string, block_item list) //string is the f

block_item = Statement(statement) | Declaration(declaration)

declaration = Declare(string, exp option) //string is variable name
                                          //exp is optional initializer

statement = Return(exp)
          | Exp(exp)
          | Conditional(exp, statement, statement option) //exp is control
                                                          //first statemen
                                                          //second stateme

exp = Assign(string, exp)
    | Var(string) //string is variable name
    | BinOp(binary_operator, exp, exp)
    | UnOp(unary_operator, exp)
    | Constant(int)
    | CondExp(exp, exp, exp) //the three expressions are the condition, 'i
```

Grammar:

```
<program> ::= <function>
<function> ::= "int" <id> "(" ")" "{" { <block-item> } "}"
<block-item> ::= <statement> | <declaration>
<declaration> ::= "int" <id> [ = <exp> ] ";"
<statement> ::= "return" <exp> ";"
              | <exp> ";"
              | "if" "(" <exp> ")" <statement> [ "else" <statement> ]


<exp> ::= <id> "=" <exp> | <conditional-exp>
<conditional-exp> ::= <logical-or-exp> [ "?" <exp> ":" <conditional-exp> ]
<logical-or-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> | <id>
<unary_op> ::= "!" | "~" | "-"
```

Code Generation

To generate the assembly for `if` statements and conditional expressions, we're going to need conditional and unconditional jumps, which we introduced in part 4. We can generate assembly for the conditional expression `e1 ? e2 : e3` as follows:

```
    <CODE FOR e1 GOES HERE>
    cmpl $0, %eax
    je   _e3                  ; if e1 == 0, e1 is false so execute e3
    <CODE FOR e2 GOES HERE>  ; we're still here so e1 must be true. execut
    jmp  _post_conditional    ; jump over e3
_e3:
    <CODE FOR e3 GOES HERE>  ; we jumped here because e1 was false. execut
_post_conditional:            ; we need this label to jump over e3
```

The assembly for `if` statements is quite similar, although it's slightly complicated by the optional `else` clause. I'll let you figure it out yourself.

As in the assembly for `&&` and `||` we saw earlier, labels have to be unique.

☑ Task:

Update the code-generation pass to correctly handle ternary conditional expressions and `if` statements. It should success on all valid examples and fail on all invalid examples for stages 1-6.

Up Next

In the next post, we'll add compound statements, so brace yourself (pun intended) for an exciting discussion of lexical scope! I **hope** that will be two weeks from now and not two months. See you then!

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

[1] But the `if` construct in many functional languages *is* an expression, and works just like C's ternary conditionals. This is valid OCaml, for instance:

```
let a = if b then 1 else 2
```

↩

[2] The terms "block" and "compound statement" aren't 100% synonymous; compound statements are a subset of blocks. But the terms are similar enough that it's fine to treat them as synonyms for now. ↩

[3] Actually, any "modifiable lvalue" is allowed on the left side of an assignment statement, not just variables. `*x`, `&x`, `++x`, and `x++` are all examples of modifiable lvalues. Conditional expressions aren't, though. ↩

[4] See this Stack Overflow answer and the C++11 standard. ↩

[5] Thanks to Stephen Bastians for pointing out a mistake in this grammar rule in an earlier verson of this post. ↩

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