

# C Compiler, Part 8: Loops

Apr 10, 2018

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

In this post we're going to add loops! Now we'll finally be able to compile FizzBuzz...except we won't, because we can't call `printf` yet. Still, it's progress!

If you've been following along, note that there was a mistake in [the last post](#). Make sure you read the "Deallocating Variables" section and update your compiler to pass the new stage 7 tests before you start on stage 8.

As usual, accompanying tests are [here](#).

## Part 8: Loops

In this post we're implementing what the [C11 standard](#) calls iteration statements; if you want to refer to the standard itself, they're in section 6.8.5. There are a few different iteration statements:

### `for` loops

First, some terminology. I'm going to call the three parts of a `for` loop header the *initial clause*, *controlling expression*, and *post-expression*, as in:

```
for (int i = 0; // initial clause
    i < 10;    // controlling expression
    i = i + 1 // post-expression
) {
    // do something
}
```

`for` loops come in two flavors: one where the initial statement is a variable declaration, and one where it's just an expression.

Flavor #1:

```
for (int i = 0; i < 10; i = i + 1) {  
    // do something  
}
```

Flavor #2:

```
int i;  
for (i = 0; i < 10; i = i + 1) {  
    //do something  
}
```

One interesting thing about `for` loops is that any of the expressions in the loop header can be empty:

```
for (;;) {  
    //do something  
}
```

But if the controlling expression is empty, the compiler needs to replace it with a constant nonzero expression<sup>1</sup>. So the example above is equivalent to:

```
for (;1;) {  
    //do something  
}
```

## `while` and `do` Loops

There's not a whole lot to say about these.

```
while (i < 10) {  
    i = i + 1;  
}
```

```
do {  
    i = i + 1;  
} while (i < 10); // ← the semicolon is required!
```

## `break` and `continue`

`break` and `continue` aren't loops, but they always appear inside loops, so it makes sense to add them now<sup>2</sup>. The C11 standard calls them "jump statements" and defines them in section 6.8.6.

A `break` statement inside a loop causes execution to jump to the end of the loop:

```
while (1) {  
    break; // go to end of loop  
}  
// break statement will go here
```

A `continue` statement causes execution to jump to the end of the loop body – immediately before the post expression in a for loop.

```
for (int i = 0; i < 10; i = i + 1) {  
    if (i % 2)  
        continue;  
    // do something  
  
    //continue statement will jump here  
}
```

In the example above, the loop will execute ten times, but only "do something" for odd values of *i*.

## Null statements

Sort of like you can have null expressions in a `for` loop, you can also have null statements<sup>3</sup>:

```
int a = 0;  
; // does nothing  
return a;
```

Null statements don't really have anything to do with loops, but they share a common feature with the expressions in a for loop: they're both defined in terms of optional expressions in the standard. Since we need to support optional expressions in for loops, it's pretty easy to add support for null expressions too.

As usual, we'll update the lexing, parsing, and code generation passes, in order.

## Lexing

We're adding five (!) keywords in this post: `for`, `do`, `while`, `break`, and `continue`.  
Here's all our tokens so far:

- `{`
- `}`
- `(`
- `)`
- `;`
- `int`
- `return`
- Identifier `[a-zA-Z]\w*`
- Integer literal `[0-9]+`
- `-`
- `~`
- `!`
- `+`
- `*`
- `/`
- `&&`
- `||`
- `=`
- `≠`
- `<`
- `≤`
- `>`
- `≥`
- `=`
- `if`
- `else`
- `:`
- `?`
- `for`
- `while`
- `do`
- `break`
- `continue`

### ☑ Task:

You know the drill here.

# Parsing

We're adding six kinds of statements: `do` loops, `while` loops, the two different kinds of `for` loop, `break` and `continue`. We're also changing the `Exp` statement; its argument is now optional, so we can use it to represent null statements. Now we can construct a null statement in the AST like this:

```
null_exp = Exp(None)
```

The initial expression and post-expression in a `for` loop are also optional.

Here's the updated definition of statements in the AST, with new and changed parts bolded:

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

           | Compound(block_item list)
           | For(exp option, exp, exp option, statement) // initial express
           | ForDecl(declaration, exp, exp option, statement) // initial de
           | While(expression, statement) // condition, body
           | Do(statement, expression) // body, condition
           | Break
           | Continue
```

Note that our AST lets `break` and `continue` statements appear outside of loops, even though that's illegal; we'll catch that error during code generation, not parsing.

The trickiest part of the grammar here is dealing with optional expressions. I dealt with this by defining an `<exp-option>` symbol:

```
<exp-option> ::= <exp> | ""
```

Once we've added that, updating the grammar for statements is pretty easy:

```
<statement> ::= "return" <exp> ";"
              | <exp-option> ";"
              | "if" "(" <exp> ")" <statement> [ "else" <statement> ]
              | "{" { <block-item> } "}"
              | "for" "(" <exp-option> ";" <exp-option> ";" <exp-option> "
```

```

| "for" "(" <declaration> <exp-option> ";" <exp-option> ")"
| "while" "(" <exp> ")" <statement>
| "do" <statement> "while" "(" <exp> ")" ";"
| "break" ";"
| "continue" ";"

```

If you're wondering why there's a semicolon after the initial `<exp-option>` in the first `for` rule, but not after the initial `<declaration>` in the second one, it's because the rule for `<declaration>` also includes a semicolon.

Parsing `<exp-option>` isn't entirely straightforward, because the empty string is not actually a token. I dealt with this by looking ahead to see if the next token was a close paren (after a post-expression) or a semicolon (after a statement, post-expression or controlling condition). If it was, the expression was empty; if not, not. I think this approach violates some formalisms about context-free grammars and LL parsers: in order to parse an `<exp-option>` symbol, you may have to look at a token that comes *after* that symbol. This isn't actually a problem, but if it bothers you, you can refactor the grammar to avoid it:

```

<exp-option-semicolon> ::= <exp> ";" | ";"
<exp-option-close-paren> ::= <exp> ")" | ")"
<statement> ::= ...
                | <exp-option-semicolon> // null statement
                | "for" "(" <declaration> <exp-option-semicolon> <exp-opti
                ...

```

Note that there's a discrepancy here between the grammar and the AST definition; the grammar allows controlling expressions in `for` loops to be empty, but the AST doesn't. That's because, as I mentioned earlier, an empty controlling expression needs to be replaced with a nonzero constant. So our approach to parsing controlling expressions in `for` loops will look something like this:

```

match parse_optional_exp(controlling_expression) with
| Some e → e
| None → Const(1) // construct a constant nonzero expression

```

You could do this during the code generation stage instead of the parsing stage, if you wanted.

### ☑ Task:

Update parsing to succeed on all valid stage 1-8 examples, and fail on all invalid stage 8 examples whose names start with `syntax_err`.

# Code Generation

## Null Statements

Don't emit any assembly for null statements. Easy!

### `while` loops

Given a `while` loop like this:

```
while (expression)
    statement
```

we can describe its control flow like this:

1. Evaluate `expression`.
2. If it's false, jump to step 5.
3. Execute `statement`.
4. Jump to step 1.
5. Finish.

I won't show you the exact assembly you need to generate here; by now you know enough to figure it out yourself. The main thing is labeling steps 1 and 5, so when we need a jump instruction we have somewhere to jump to. It's worth noting that the loop body is a new scope, and you need to reset your `current_scope` set accordingly.

### `do` Loops

These are basically the same as `while` loops; just evaluate the expression after the statement.

### `for` loops

Given a `for` loop like this:

```
for (init; condition; post-expression)
    statement
```

we can break it down in the same way as `while` loops above:

1. Evaluate `init`.
2. Evaluate `condition`.

3. If it's false, jump to step 7.
4. Execute `statement`.
5. Execute `post-expression`.
6. Jump to step 2.
7. Finish.

The init and post-expression might be empty, in which case we just don't emit any assembly for steps 1 and 5. Note that a `for` loop, including the header, is a block with its own scope, and the *body* of the `for` loop is *also* a block. That means you can have code like this:

```
int i = 100; // scope 1
for (int i = 0; i < 10; i = i + 1) { // scope 2 - variable i shadows previ
    int i; //scope 3 - this variable i shadows BOTH previous i's
}
```

The main gotcha here is that you need to pop the variable declared in `init` off the stack when you exit the block, just like you needed to handle deallocating other variables in the last post.

`break`   **and**   `continue`

We can implement each of these with a single `jmp` instruction – the trick is just figuring out where to jump *to*. A break statement “terminates execution of the smallest enclosing `switch` or iteration statement,” so we want to jump to the point right after the loop<sup>4</sup>. We already have an “end of loop” label, which we jump to when the controlling condition is false; we just need to pass that label around along with the variable map, stack index and current scope.

We also need to pass *another* label for `continue` to refer to. `continue` “causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body”<sup>5</sup> – that’s step 4 in the `while` loop or step 5 in the `for` loop above.

Unlike the stack index, variable map and so forth, the jump and continue labels can be null, if you’re not inside a loop. Hitting a `break` or `continue` statement when these labels are null should, of course, cause an error.


At this point, I was passing enough arguments around that I defined a `Context` type and wrapped it all up in that. You may want to do something similar, but you don’t have to.


## Up Next


In the [next post](#) we’re going to implement a pretty fundamental concept: **function calls**. I don’t know about you but I am VERY EXCITED for function calls. See you then!





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

<sup>1</sup> See section 6.8.5.3 of the C11 standard. 

<sup>2</sup> `break` can also appear in `switch` statements, but we haven't added those yet. 

<sup>3</sup> C11 standard, section 6.8.3. 

<sup>4</sup> C11 standard, section 6.8.6.3. 

<sup>5</sup> C11 standard, section 6.8.6.2. 

---

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

© 2022 Nora Sandler.