

Writing a C Compiler, Part 7

Mar 14, 2018

Update 4/9

- There was a pretty big mistake in the original post - I forgot to deallocate local variables! I've added the "Deallocating Variables" section, and added the example from that section to the test suite.

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

In this post we're adding support for compound statements, which are a little weird because they don't *do* very much. We'll generate almost no new assembly in this post, but we'll be able to compile new and exciting programs at the end of it. How is this possible? Let's find out!

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

Part 7: Compound Statements

A compound statement is just a list of statements and declarations wrapped in curly braces.

They're normally used as substatements of `if`, `while`, and other control structures, like this¹:

```
if (flag) {  
    //this is a compound statement!  
    int a = 1;  
}
```

but they can also be free-standing, like this:

```
int main() {  
    int a;  
    {  
        //this is also a compound statement!  
        a = 4;  
    }  
}
```

You can have deeply nested compound statements:

```
int main() {  
    //compound statement #1 (function bodies are compound statements!)  
    int a = 1;  
    {  
        //compound statement #2  
        a = 2;  
        {  
            //compound statement #3  
            a = 3;  
            if (a) {  
                //compound statement #4  
                a = 4;  
            }  
        }  
    }  
}
```

Like I mentioned in the last post, a compound statement is one type of **block**, and I'm going to use the terms synonymously for the rest of this post. C uses **lexical scoping**; a variable's scope is dictated by the block where it's defined. (By "scope", I mean where in the program you're allowed to refer to it.) More precisely, a variable's scope starts at its definition, and ends when you exit the block where it's defined². Up until this point in the series, function bodies were the only blocks around, so a variable could be used at any point in `main` after it was defined. Now it's more complicated. I'm going to talk a bit about how scoping works in C; if you're already familiar with this, you can skip ahead to the next section.

If a variable is defined in an inner scope, it can't be accessed in an outer scope:

```
// here is the outer scope  
{  
    // here is the inner scope  
    int foo = 2;  
}  
  
// now we're back in the outer scope  
foo = 3; // ERROR - foo isn't defined in this scope!
```

However, code in an inner scope can access variables in an outer scope:

```
int a = 2;
{
    a = 4; // this is okay
}
return a; // returns 4 - changes made inside the inner scope are reflected
```

You can't have two variables with the same name in the same scope:

```
int foo = 0;
int foo = 1; //This will throw a compiler error
```

But you can have two variables with the same name in *different* scopes. Once the variable in the inner scope is declared, it will shadow the variable from the outer scope; the outer variable will be inaccessible until the inner variable goes out of scope.

```
int foo = 0;
{
    int foo; // this is a TOTALLY DIFFERENT foo, unrelated to foo from ear
    foo = 2; // this refers to the inner foo; outer foo is inaccessible
}
return foo; //this will return 0 - it refers to the original foo, which is
```

The key idea here is that the inner and outer `foo` variables are two totally unrelated variables that just happen to have the same name. When we're in the inner block, the outer variable `foo` still exists, but we have no way to refer to it, because `foo` now refers to the inner variable.

Note, however, that outer `foo` is accessible in the inner block before the point where it's shadowed:

```
int foo = 0;
{
    foo = 3; //changes outer foo
    int foo = 4; //defines inner foo, shadowing outer foo
}
return foo; //returns 3
```

Lexing

Compound statements don't require any new tokens, so we don't need to touch the lexing pass this week.

Parsing

Here's the current definition of statements in our AST:

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

We just need to add a `Compound` statement to this definition. Also recall that we added a `block_item` construct to the AST in our last post:

```
block_item = Statement(statement) | Declaration(declaration)
```

A compound statement is just a list of statements and declarations, so our new definition of statements will look like this:

```
statement = Return(exp)
           | Exp(exp)
           | Conditional(exp, statement, statement option) //exp is control
                                                         //first statement
                                                         //second statement
           | Compound(block_item list)
```

We'll parse conditional expressions and conditional statements totally differently. Statements are easier, so let's handle those first.

Now let's update our grammar. The rule for blocks is extremely simple:

```
"{" { <block-item> } "}
```

Note that `"{" "}"` are literal curly braces, and `{ }` indicates repetition. This is hard to read! But it just means we have an arbitrary number of block items wrapped in braces – if you refer back to the grammar for `<function>` you can see that we define function bodies exactly the same way.

Putting it all together, our updated grammar looks like this:

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

☑ Task:

Update the parsing pass to handle blocks. It should successfully parse all valid examples in stage 1-7. As in part 5, some invalid examples should fail during parsing and some should fail during code generation. At this point, your parsing pass should throw an appropriate error for all invalid stage 7 examples whose names start with `syntax_err`.

Code Generation

As we saw earlier, it's possible to have two different variables, in two different scopes, stored at two different locations on the stack, with the same name. Here's an example:

```
int foo = 3;
{
    int foo = 4;
}
```

So, whenever the program refers to variable `foo`, our generated code needs to access the correct `foo` on the stack – or raise an error if `foo` has gone out of scope. The code generation step this week is all about managing the variable map so we always look up the right `foo`.

The trick here is that **every block has a separate copy of the variable map**. That way, defining (or redefining) a variable in an inner scope won't interfere with an outer scope. And if you're using an immutable map (which you should be), every block will necessarily get its own variable map, so this approach is surprisingly easy.

Let's look at some pseudocode. After Part 5, your code to generate a function body probably looked something like this:

```
def generate_function_body(body):
    // initialize variable map and stack index
    var_map = Map()
    stack_index = -4

    //process statements one at a time
```

```
for statement in body:
    var_map, stack_index = generate_statement(statement, var_map, stack_in
```

Note that `generate_statement` has to return a new `var_map`. Every declaration updates the variable map (or, more precisely, creates a new variable map), and in part 5 `generate_statement` also handled declarations. Whenever we process a declaration, we need to return the latest, greatest variable map so future statements can reference the variable we just declared.

But in the last post, we separated statements from declarations in our AST, so you might have changed the last line to:

```
var_map, stack_index = generate_statement_or_declaration(statement, va
```

At this point, a declaration will create a new variable map, but a statement won't. Whatever happens in a statement – including a compound statement, which may itself contain declarations – has no impact on the variable map for the enclosing scope. Once you understand that point, handling nested scopes is easy:

```
def generate_function_body(body):
    // initialize variable map and stack index
    var_map = Map()
    stack_index = -4

    //process statements one at a time
    for block_item in body:
        if block_item is a declaration:
            //update the variable map
            var_map, stack_index = generate_declaration(statement, var_map, st
        else:
            //don't update the variable map
            generate_statement(statement, var_map, stack_index)
```

Of course you'll need to generalize `generate_function_body` into `generate_block`; the one difference between generating a function body and any other block is that you need to initialize your empty variable map and stack index at the start of the function body.

Now let's walk through a small example to see how this maintains the right variable maps for different scopes:

```

int main(){
    // 1) function body
    {    // 2) block
        int a = 2; // 3) variable declaration
        a = 3; // 4) variable reference
    }
    return a; // 5) return statement
}

```

1. We'll process the function body with `generate_block`. Right now we've got an empty variable map.
2. We call `generate_block` recursively to process the inner block. The variable map is still empty.
3. This is a declaration, so we add `a` to the variable map (technically, we create a copy of the variable map that contains `a`, because all these maps are immutable).
4. We look up `a`'s location on the stack in the variable map from step 3.
5. Back in the outer scope, `var_map` refers to the original, *empty* variable map. Since `a` isn't defined in this map, this will throw an error, as it should.

The code for handling declarations also needs to be changed. The pseudocode for processing declarations from part 5 included this line:

```

if var_map.contains("a"):
    fail() //shouldn't declare a var twice

```

This is now incorrect; it's legal to declare two variables with the same name, as long as the declarations aren't in the same scope. To solve this, we need a way to distinguish between variables defined in the current scope, and variables defined in an outer scope. My solution was to maintain a set of variables that are defined in the current scope, which means

`generate_block` now looks something like this:

```

def generate_block(block, var_map, stack_index):

    current_scope = Set()

    //process statements one at a time
    for block_item in block:
        if block_item is a declaration:
            //update the variable map
            var_map, stack_index, current_scope = generate_declaration(statement)
        else:

```

```
//don't update the variable map
generate_statement(statement, var_map, stack_index)
```

Finally, we check `current_scope`, rather than `var_map`, for duplicate variable declarations, and add the variable to both structures on success:

```
if current_scope.contains("a"):
    fail() //shouldn't declare a var twice in the same scope
else:
    //emit assembly, update stack_index and var_map as before...
    new_scope = current_scope.add("a")
    return (var_map, stack_index, current_scope)
```

This solution feels hacky, but I haven't come up with a better one. ▶

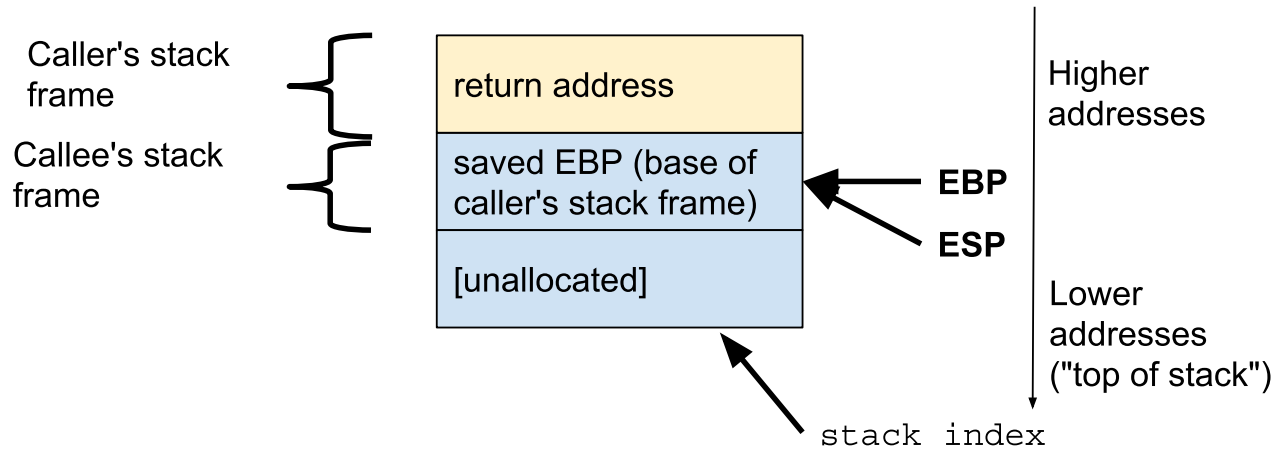
Now, if `a` is redefined in an inner scope, it just overwrites the old `a` in the variable map, so this scope and any inner ones will use the correct stack location, corresponding to the innermost definition of `a`. This won't affect the outer scope at all, because the outer scope is still using the original, unmodified variable map.

Deallocating Variables

We've carefully managed our variable map to prevent a block from interfering with any variable declarations in its enclosing scope. But there's one side effect we couldn't avoid: allocating a variable changes the stack pointer. This is a problem, because the stack pointer and our `stack_index` variable will get out of sync. Consider the following example:

```
int main() {
    {
        int i = 0;
    }
    int j = 1;
    return j;
}
```

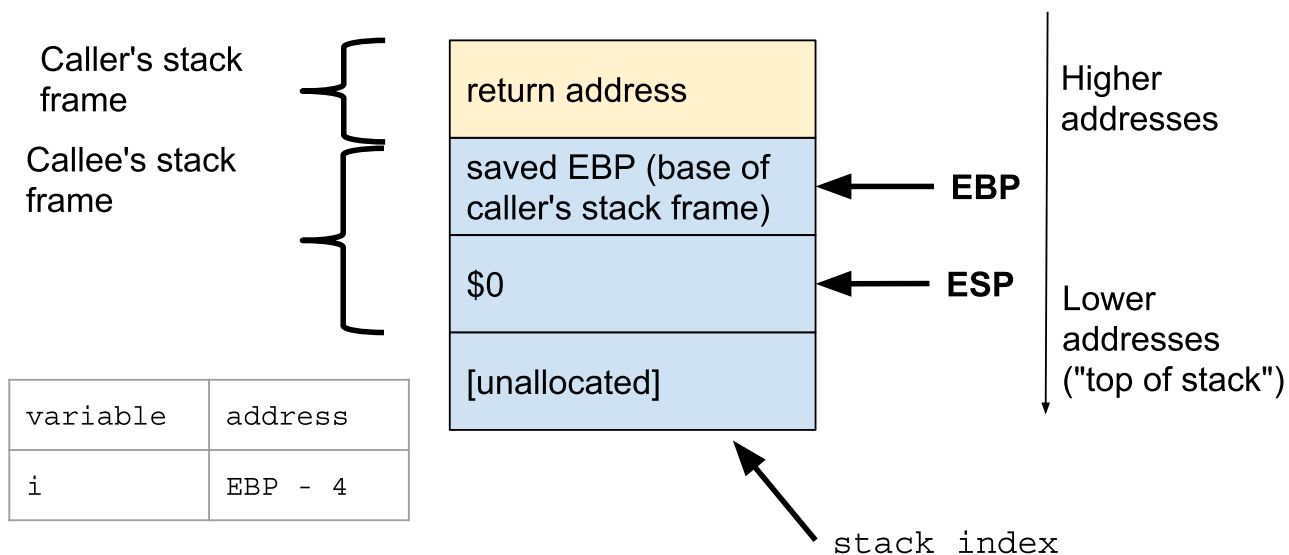
At first, the variable map is empty and `stack_index` is -4, because the first empty spot on the stack is four bytes below EBP:



When we process the block in this example with `generate_block`, we'll push `i` onto the stack:

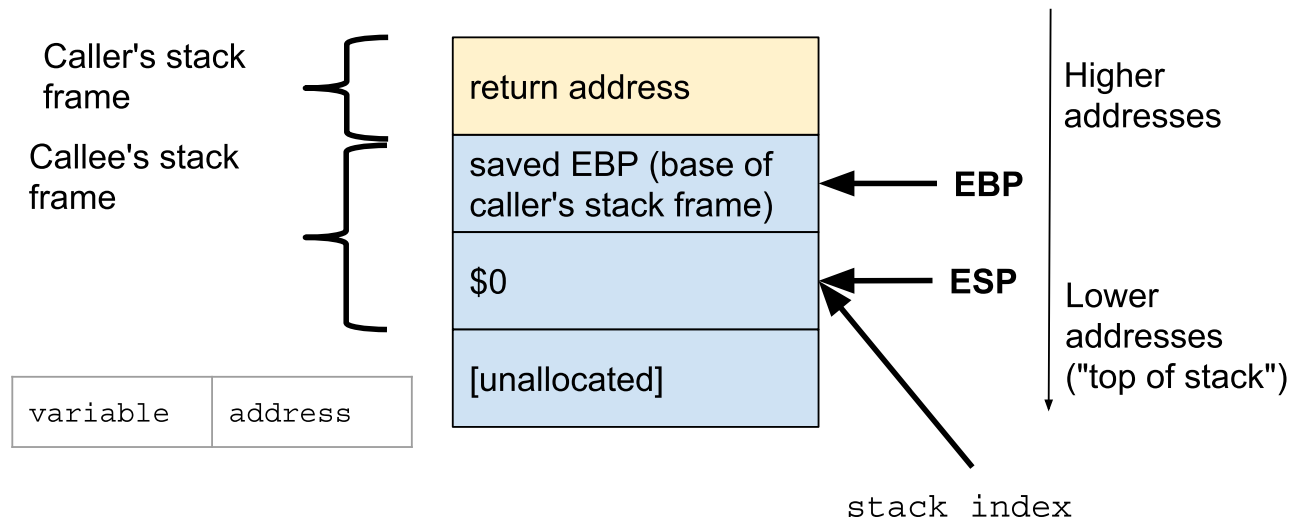
```
movl $0, %eax
push %eax
```

Now ESP is at EBP - 4, and `stack_index` is -8:



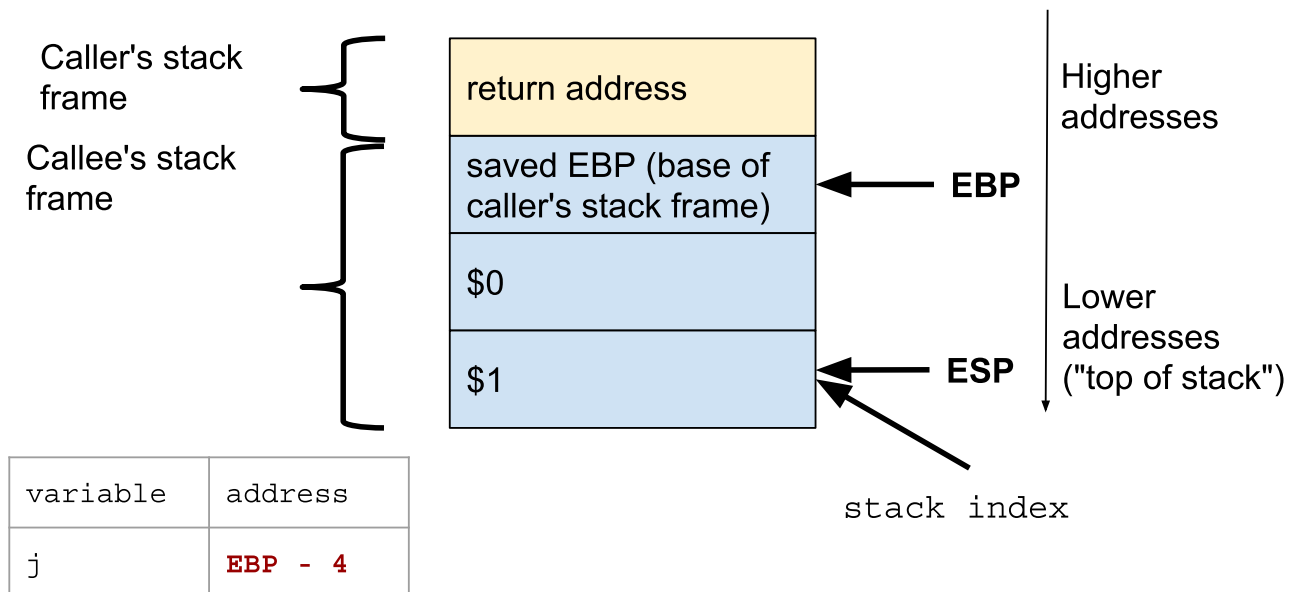
After we exit the block, we forget that we allocated `i`. That means `i` is no longer in our variable map, and we're still working with our original stack index of -4; remember that `generate_block` doesn't return a stack index. We *should* forget `i`, because it's out of scope.

The problem is, `i` is still there, because ESP is still pointing at it.



So when we push `j`, it will be just below `i`, at `EBP - 8`:

```
movl $1, %eax
push %eax
```



But because the stack index was -4, we'll add a mapping from `j` to -4 in our variable map. Any future references to `j` (like in the return statement) will incorrectly use the stack location of `i` instead.

We *could* solve this by having `generate_block` return a stack index, but it's probably better to just pop variables off the stack when we're done with them, right at the end of `generate_block`. Conveniently, the size of `current_scope` tells us how many variables we need to pop.

```
def generate_block(block, var_map, stack_index)

    current_scope = Set()
    ...as before...

    bytes_to_deallocate = 4 * current_scope.size()
    emit "    addl ${}, %esp".format(bytes_to_deallocate)
```


☑ Task:


Update the code-generation pass to correctly handle compound statements. It should succeed on all valid examples and fail on all invalid examples for stages 1-7.

Up Next

In the [next post](#), we'll add `for`, `do`, and `while` loops. See you then!

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

¹ I'll use comments to clarify the code snippets throughout this post, even though we haven't added support for comments yet. 

² Global variables work a bit differently but we haven't added those yet. 

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

© 2022 Nora Sandler.