



Writing a C Compiler, Part 5

Jan 8, 2018

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

We've spent the last two weeks adding binary primitives, and I don't know about you, but I'm starting to get kind of bored with it. This week, we'll do something completely different and add support for local variables. We'll finally be able to compile functions longer than one line! Hooray!

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

Week 5: Local Variables

We're adding variables this week! Programming without variables is hard, so this is very exciting. To keep things simple, we're going to support variables in a very restricted way for now:

- We only support local variables, which are declared in `main`. No global variables.
- We only support variables of type `int`.
- We don't support type modifiers like `short`, `long` or `unsigned`, storage-class specifiers like `static`, or type qualifiers like `const`. Just plain old `int`.
- You can only declare one variable per statement. We won't support statements like `int a, b;`

There are three things you can do with a variable:

- Declare it (`int a;`)
 - When you declare it, you can also optionally initialize it (`int a = 2;`)
- Assign to it (`a = 3;`)
- Reference it in an expression (`a + 2`)

We'll need to add support for these three things. We'll also add support for functions containing more than one statement.

Lexing

The only new token this week is the assignment operator, `=`. Here's our list of tokens, with the newest addition 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** `=`

☑ Task:

Update the `lex` function to handle the `=` token. It should work for all stage 1-5 examples in the test suite, including the invalid ones.

Parsing

We need to make a lot of changes to our AST this week. Let's look at a sample program we'd like to handle:

```
int main() {  
    int a = 1;  
    a = a + 1;  
}
```

```
    return a;
}
```

In this program, `main` contains three statements:

1. A variable declaration (`int a = 1;`)
2. A variable assignment (`a = a + 1;`)
3. A return statement (`return a;`)

We need to update the definition of `function_declaration` in the AST so a function can contain a list of statements, not just a single statement:

```
function_declaration = Function(string, statement list) //string is function name
```

Right now, the only statements we've defined are `return` statements. That's not right either. Let's add some more:

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

We've added `Decl` for variable declarations. We can use an option type (`Maybe` in Haskell) to represent that we may or may not have an initializer.

The AST for `int a;` might look like this:

```
decl = Declare("a", None) //None because we don't initialize it
```

And the AST for `int a = 3` might look like this:

```
init_exp = Const(3)
decl = Declare("a", Some(init_exp))
```

Note that we don't store the variable's type anywhere in our AST; we don't need to, because it can only have type `int`. We'll need to start tracking type information once we have multiple types

We've also added a standalone `Exp` statement, which means we can now write programs like this:

```
int main() {  
    2 + 2;  
    return 0;  
}
```

This is valid C; if you compile it with gcc, it will issue a warning but it won't fail.

However, `2+2;` isn't a very useful statement. The real reason to add an `Exp` statement is so we can write statements like this:

```
a = 2;
```

Variable assignment is just an expression! That's why you this statement is valid:

```
a = 2 * (b = 2);
```

In the code snippet above, the expression `b = 2` has the value `2`, and the side effect of updating `b` to have that value. This would be evaluated as:

```
a = 2 * (b = 2)  
a = 2 * 2 //also b is 2 now  
a = 4
```

Now we need to update `exp` in our AST definition to handle assignment operators. My first thought was to just add `=` as another binary operator – after all, `a = b` looks kind of like `a + b`. But that's totally wrong: the two operands of a binary operator can be arbitrary expressions, but the left side of an assignment operator can't. A statement like `2 = 2` doesn't make any sense, because you can't assign a new value to `2`.

Instead, we'll just define assignment as a new type of expression:

```
exp = Assign(string, exp) //string is variable, exp is value to assign  
    | BinOp(binary_operator, exp, exp)  
    | UnOp(unary_operator, exp)  
    | Constant(int)
```

Now we can write the AST for the statement `a = 2;` like this:

```
assign_exp = Assign("a", Const(2))  
assign_statement = Exp(assign_exp)
```

Now we can define variables and update their values, but that's not super helpful unless we can actually reference them. Let's add variable reference as another type of expression:

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

Now we can write the AST `return a;` like this:

```
return_exp = Var("a")
return_statement = Return(return_exp)
```

If we put it all together, here's our new AST, with changes bolded:

```
program = Program(function_declaration)
function_declaration = Function(string, statement list) //string is the fun

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

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

We also need to update our grammar. First, we need to update `<function>` to allow multiple statements.

Old definition:

```
<function> ::= "int" <id> "(" ")" "{" <statement> "}"
```

New definition:

```
<function> ::= "int" <id> "(" ")" "{" { <statement> } "}"
```

Thanks to the interspersed `{ / }`, indicating repetition, and `"{" / "}"`, indicating literal curly braces, this is almost completely unreadable. But it just means a function can have more than one statement now.

We need to handle multiple types of statement. We already have return statements:

```
"return" <exp> ";"
```

And standalone expressions are super easy:

```
<exp> ";"
```

A variable declaration needs a type specifier (`int`) followed by a name, optionally followed by an initializer. We use `[]` here to indicate something is optional:

```
"int" <id> [ = <exp> ] ";"
```

Let's put it all together to get a our new definition of `<statement>` :

```
<statement> ::= "return" <exp> ";"  
              | <exp> ";"  
              | "int" <id> [ = <exp> ] ";"
```

Finally, we need to update `<exp>`. Assignment is our lowest-precedence operator, so it becomes our top level `<exp>` expression. Also note that, unlike most of our other operators, it's right-associative, which makes it a bit simpler to express.

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

The grammar for all our binary operations (`<logical-and-exp>` on down to `<term>`) is unchanged. We just need to change `<factor>` so we can refer to variables as well as constants:

```
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int> | <id>
```

When you put it all together, here's our new grammar, with changes bolded:

```
<program> ::= <function>  
<function> ::= "int" <id> "(" ")" "{" { <statement> } "  
<statement> ::= "return" <exp> ";"
```

```

    | <exp> ";"
    | "int" <id> [ = <exp>] ";"
<exp> ::= <id> "=" <exp> | <logical-or-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-exp> }
<additive-exp> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/" ) <factor> }
<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int> | <id>
<unary_op> ::= "!" | "~" | "-"

```

☑ Task:

Update your expression-parsing code to handle variable declaration, assignment, and references. It should successfully parse all valid stage 1-5 examples in the test suite. The invalid examples are a little different this week. Some of them should fail during parsing; others can be parsed successfully but should cause errors during code generation (e.g. because they reference variables that haven't been declared.) I decided to deal with this in the laziest way possible; the names of the invalid examples that should fail during parsing all start with `syntax_err`.

Code Generation

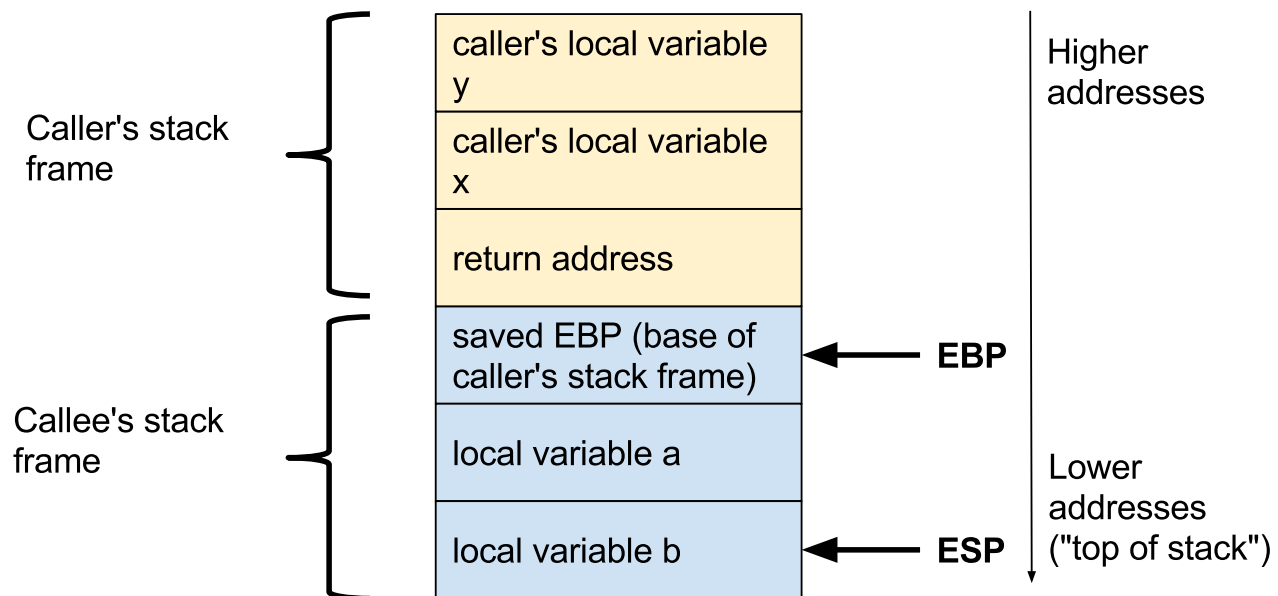
We need to save local variables somewhere, so we'll save them on the stack. We also need to remember exactly where on the stack each variable was saved, so we can refer to it later. To track this information, we'll create a map from variable names to locations.

But how are we supposed to know a variable's location at compile time? Absolute memory addresses aren't determined until runtime. We could store the variable's offset from ESP, except that the value of ESP changes whenever we push something onto the stack. The solution is to store the variable's offset from a different register, EBP. To understand why this will work, we need to know a little bit about stack frames.

Stack Frames

Whenever we call a function, we allocate a chunk of memory for it on top of the stack – this memory is called the *stack frame*. The stack frame holds function arguments, the address to jump to after the function returns, and of course local variables. We already know that ESP points to the top of stack, which is also the top of the current stack frame¹. The EBP (or base pointer) register points to the bottom of the current stack frame. Without EBP, we wouldn't

know where one stack frame ends and the other begins, and we wouldn't be able to find important values like a function's return address.



When a function (let's call it `f`) returns, its caller needs to be able to pick up where it left off. That means its stack frame, and the values in ESP and EBP, all need to be exactly the same as they were before `f` was called. The first thing `f` needs to do is set up a new stack frame for itself, using the following instructions:

```
push %ebp      ; save old value of EBP
movl %esp, %ebp ; current top of stack is bottom of new stack frame
```

These instructions are called the function prologue. Immediately before `f` returns, it executes the function epilogue to remove this stack frame, leaving everything just as it was before the function prologue:

```
movl %ebp, %esp ; restore ESP; now it points to old EBP
pop %ebp        ; restore old EBP; now ESP is where it was before prologue
ret
```

Up to this point, we could get away with not having a function prologue or epilogue, but now we need to add them. Adding them helps us in two ways:

- **We can store variable locations as offsets from EBP.** We know there's nothing above EBP (because we set up an empty stack frame in the function prologue), and we know that EBP won't change until the function epilogue.

- We can safely push local variables onto the stack without changing the caller's stack frame².

You should generate the function prologue at the start of the function definition, right after the function's label. You should generate the function epilogue as part of the return statement, right before `ret`.

Besides our variable map, we need to keep track of a *stack index*, which tells us the offset of the next available spot on the stack, relative to EBP. The next available spot is always the word right after ESP, at `ESP - 4`. Right after the function prologue, EBP and ESP are the same. That means the stack index will also be -4. Whenever we push a variable onto the stack, we'll decrement the stack index by 4³.

Now let's look at how we can handle declaring, assigning, and referring to variables.

Variable Declaration

When you encounter a variable declaration, just save the variable onto the stack and add it to the variable map⁴. Note that it's illegal to declare a variable twice in the same local scope⁵, as in the following code snippet:

```
int a;  
int a;
```

So your program should fail if the variable is already in the variable map. Here's how you might generate assembly for the statement `int a = expression`:

```
if var_map.contains("a"):  
    fail() //shouldn't declare a var twice  
generate_exp(expression)      // generate assembly to calculate e1 and m  
emit "    pushl %eax" // save initial value of "a" onto the stack  
var_map = var_map.put("a", stack_index) // record location of a in the v  
stack_index = stack_index - 4 // stack location of next address will be
```

A few points here:

- If a variable isn't initialized, you can just initialize it to 0. Or whatever you want, really.
- The variable map exists during code generation, not at runtime.
- You should **definitely use an immutable data structure** for your variable map. In the next post we'll add `if` statements, and then we'll have nested scopes; a variable declared inside an `if` block isn't accessible outside it. If you have to worry about code from an inner scope messing with the variable map in an outer scope, you will not be a happy camper.

Variable Assignment

We can look up a variable's location in memory in our map; to assign it a new value, just move that value to the right memory location. Here's how to handle `a = expression`:

```
generate_exp(expression) // generate assembly to calculate expression and
var_offset = var_map.find("a") //if "a" isn't in the map, fail b/c it has
emit "    movl %eax, {}(%ebp)".format(var_offset) //using python-style s
```

Note that the value of `expression` is still in EAX, so this assignment expression has the correct value.

Variable Reference

To refer to a variable in an expression, just copy it from the stack to EAX:

```
var_offset = var_map.find("a") //find location of variable "a" on the sta
                                //should fail if it hasn't been declared
emit "    movl {}(%ebp), %eax".format(var_offset) //retrieve value of va
```

Missing Return Statements

Now that we support multiple types of statements, we can successfully parse programs with no return statement at all:

```
int main() {
    int a = 2;
}
```

What's the expected behavior here? According to section 5.1.2.2.3 of the [C11 standard](#):

If the return type of the `main` function is a type compatible with `int`, a return from the initial call to the `main` function is equivalent to calling the `exit` function with the value returned by the `main` function as its argument; reaching the `}` that terminates the `main` function returns a value of 0.

So, `main` needs to return 0 if it's missing a return statement. Right now `main` is our only function, so that's the only case we need to handle.

Eventually, we'll need to deal with this problem in functions other than `main`. Here's what section 6.9.1 of the standard says about missing return statements in general:

If the `}` that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.

So this program has undefined behavior:

```
int foo() {
    1 + 1;
}

int main() {
    return foo();
}
```

You could technically handle this however you want – fail, continue silently, issue a **HALT AND CATCH FIRE** instruction.

This program, on the other hand, is perfectly valid, because the value returned from `foo()` is never used:

```
int foo() {
    1 + 1;
}

int main() {
    foo();
    return 0;
}
```

Honestly, the specification here seems really dumb to me. If I write a non-`void` function without a return statement, that is **WRONG** and I want the compiler to save me from myself, even if I haven't technically used it in an illegal way yet. I can't think of any situation where we'd want this behavior; if you can, please let me know.

However, that's the spec, so our functions have to return successfully even when they're missing a return statement. That means you need to issue the function epilogue and `ret` instruction even if the return statement is missing. It's probably easiest to handle `main` and all other functions uniformly, so you can just return 0 from any function without a return statement.

☑ **Task:**

Update your code-generation pass to:

- Generate function prologues and epilogues.
- Generate correct code for variable declarations, assignments, and references.
- Make `main` return 0 even if the return statement is missing.

Your code should succeed on all valid examples and fail on all invalid examples for stages 1-5.

Bonus features

At this point, there are a handful of other features you can implement pretty easily:

Compound Assignment Operators

- `+=`
- `-=`
- `/=`
- `*=`
- `%=`
- `<<=`
- `>>=`
- `&=`
- `|=`
- `^=`

Comma Operators

- `e1, e2`. The result is the value of `e2`; the value of `e1` is ignored.

Increment/Decrement Operators

- Prefix and postfix `++`
- Prefix and postfix `--`

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

Up Next


I'm going to switch to one blog post every two weeks. In the [next post](#), we'll add `if` statements and conditional operators (`a ? b : c`). See you then!


Update 1/12


- Corrected the “Missing Return Statements” section, which previously said that the behavior of `main` is undefined when it’s missing a return statement. Also updated the test suite accordingly.
- Clarified that declaring a variable multiple times is sometimes legal at file scope.


Thanks to [Olivier Gay](#) for pointing out both those things.

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

¹ Keep in mind that the stack grows *down* towards lower addresses; we decrement ESP whenever we push things onto the stack, and ESP will always hold a lower value than EBP. So the top of the stack is really...on the bottom `_()` 

² Even though `main` is the only function, it still has a caller: it’s called by the setup routine, `crt0`. 

³ We don’t really need to keep track of the stack index, since we can just derive it from the size of the variable map. However, the stack index will come in handy once we add types other than `int`, since at that point our variables won’t all be the same size. If you don’t want to keep track of it for now, that’s fine with me. 

⁴ This is not at all how real compilers work; they usually allocate space for local variables all at once in the function prologue, or just store them in registers. Our way is less effort, though. 

⁵ It’s sometimes legal to declare a variable at file scope, per section 6.9.2 of the C11 specification. 

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

© 2022 Nora Sandler.