Function Declarations and the `char` Type

Lecture #08

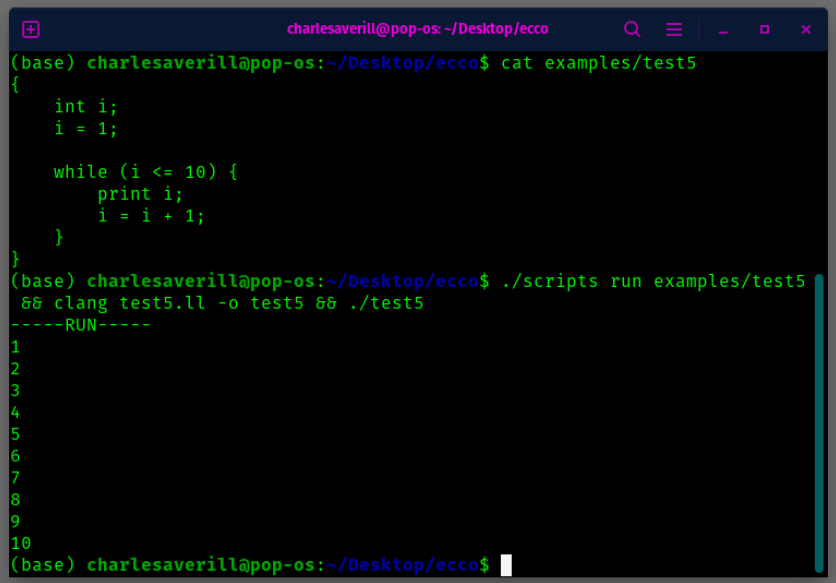
Charles Averill

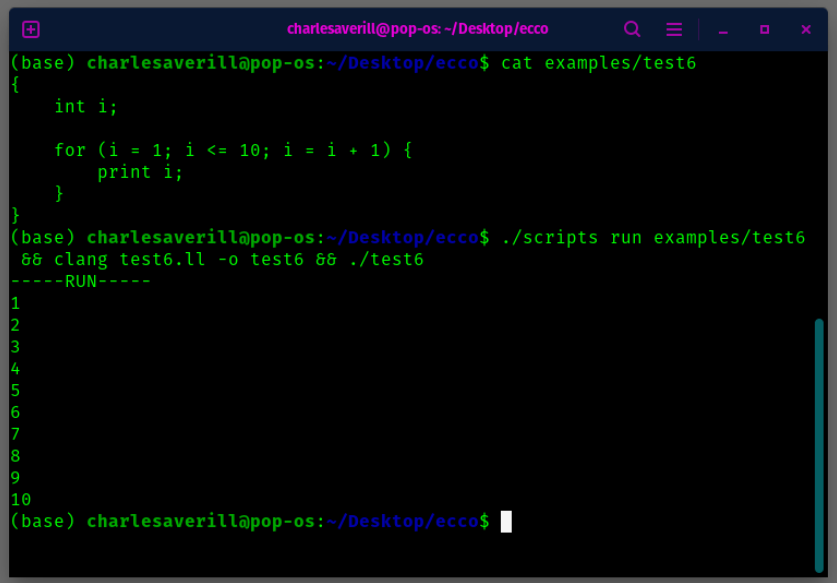
Practical Compiler Design
The University of Texas at Dallas

Spring 2023

The Current State of the Compiler

In our previous lecture, we introduced if statements, while loops, and for loops into our language:





The Goal

Today we have a fairly small goal that will require a fair amount of refactoring: function declarations (actually function declaration, we're only supporting 1 function right now):

```
void main() {
    int fred;
    int jim;
}
```

(I finally get to use C syntax highlighting! Yay!)

Previously I explained that all of our compound statements would be surrounded by braces. This is why; the vast majority of statements in C must be made inside of a function definition. We *could* just add a specific parser for the function declaration grammar, but we should probably work ahead to make sure our type system can handle more complex definitions in the future.

The Plan

- Add a function declaration parser
- Modify our type system so that we can store function return types
- Update our preamble/postamble generators to support multiple function declarations in the future
- Add the `char` type and the framework for adding more types in the future

Function Declaration Parser

We have to hardcode some behavior for now. Specifically, our only function type will be `void`, as we haven't implemented return statements yet. Also, we won't support function arguments for a while, so we'll just match an open and close parenthesis. After the function header we can just call `parse_statements()` and store the result into a Function ASTNode.

You might notice we've updated the return type of `match_token`; this ties into the char type updates that we'll see later.

Type System Updates

Take a look at `generation/types.py`. I've added Number, Function, and Type classes, and relocated NumberType. These functions are going to better store information about variables and functions to make sure we're not mixing and matching where we shouldn't.

The Type class is actually going to store a value as well, which gets a bit confusing as we'll see, but that's just what happens when you're dealing with type systems.

We've updated the Symbol Table class to accept a Type object as a value now. This will allow us to register both variables and functions in our table in the future.

Generator Updates

We actually don't *have* to update our generator that much, because we're only adding support for a main function right now, which is sort of what we've had the whole time. But let's work ahead and ensure that we'll be safe when we start generating code for multiple functions.

In `generate/translate.py`, we've added a case in our "special token types" section for `TokenType.FUNCTION`. We see that we now generate a preamble here, then we do stack allocation, then the code within the function, then its postamble, then return. Previously, we were generating preambles and postambles in the main generator loop.

Funny Story

While writing this lecture, I referenced my Purple function generation code. I still generated stack allocation statements in the main generator loop, which would cause my `alloca` statements to be printed **before** the function preamble.

To solve this problem, I created a "buffered_stack_allocations" array to hang onto any allocation statements that tried to generate before a function had its preamble generated. This worked and I moved on.

Looking back on that code while I wrote this lecture, I wondered why I didn't just move the allocation generation function call to right after the preamble generation call. That's what ECCO does, and it works! Don't overengineer if you don't have to.

`char` type

Let's add in another datatype to see what the process is like. We won't add in char parsing yet, but we will do that pretty soon. A few steps:

1. Add a char token
2. Un-hardcode `declaration_statement`
3. Minor generator updates

declaration_statements

I mentioned that we updated the return type of `match_token`. Now it's returning both the previously scanned token's value AND its token type. But aren't we matching the type we expect? Kind of.

Now we're matching multiple types, any of the data types we could expect during a variable declaration. So, we'll pass in a list containing the INT and CHAR token types, and then receive the type that it matched to.

Generator updates

A lot of our LLVMValues expected integers, so there are a few places we need to update by passing in NumberTypes.

And that's it!