Statements

Lecture #04

Charles Averill

Practical Compiler Design
The University of Texas at Dallas

Spring 2023

The Current State of the Compiler

We can now compile arithmetic expressions to LLVM, which we can pass to the `clang` compiler and then execute as a binary.

```
(base) charles@nostromo:~/Desktop/ecco$ cat examples/test1
2 + 3 * 5 - 8 / 3
(base) charles@nostromo:~/Desktop/ecco$ ./scripts run examples/test1
-----RUN-----
(base) charles@nostromo:~/Desktop/ecco$ clang test1.ll -o test1
(base) charles@nostromo:~/Desktop/ecco$ ./test1
15
```

This is very cool, but there are some limitations. We can only have one arithmetic expression per file. Also, our expressions are auto-printed - this could be fine for some kinds of programming languages, but in C we usually want to be as explicit as possible in how we describe our program to the computer. Therefore, we should make an explicit print statement so that expressions don't have any "hidden print" meaning behind them. We'll also set up the groundwork for more complex statements in the future.

Planning our Changes

We're going to modify our language to support print statements that look like this: `print 1 + 2 * 3;`

That's not what C looks like! But we'll fix that later. This is going to let us lay the groundwork for complex statements, like variable assignments in the next lecture.

We'll need to add print and semicolon TokenTypes and modify our parser to support checking for multi-length tokens. We'll also need to tell the parser not to stop scanning/parsing at EOF, but rather at each semicolon.

Finally, we'll perform some small structural changes to combine statement parsing with continual LLVM generation. Let's do it!

Quick BNF Update

```
statements: statement
          | statement statements
statement: "print" expression ";"

expression: add_expression

add_expression: mul_expression
              | add_expression "+" mul_expression
              | add_expression "-" mul_expression

mul_expression: INTEGER_LITERAL
              | INTEGER_LITERAL '*' mul_expression
              | INTEGER_LITERAL '/' mul_expression
```

Scanner Updates

Previously, I've started out by saying "let's check out what changed in `ecco/ecco.py`." But we're pretty familiar with the structure of the project now, and we've already planned out our changes. So let's look right at the scanner changes in `ecco/scanning/ecco_scanner.py`.

Adding TokenTypes `PRINT` and `SEMICOLON` is trivial. However there are some fairly large changes in `Scanner.scan()`. I've moved our checking for token types into the `TokenType` class for coherence. If there is a match, store the type into our global token and continue; otherwise check for integer literals, or our new addition: **identifiers**.

Identifiers could be keywords like "for", "int", "struct", or they could be variable names like "x" or "count". In short, identifiers are multi-character Tokens. In our C implementation, identifiers can start with an alphabetic character or an underscore.

Scanner Updates

You can take a peek into `scan_identifier()` if you'd like, but this is another implementation detail, feel free to copy it. Remember that in C, identifiers can have numbers in them as long as they aren't the first character in the identifier. I've also added a length maximum to identifiers, this handles some bugs in the future.

Expression Parser Updates

Next, let's see the update for our expression parser. We add some checks for semicolons, either in place of or addition to checks for the EOF, but the logic remains the same.

Statement Parser

Now we have to parse statements. Because our statements are so simple for now, this is a really easy change. Take a look at `ecco/parsing/statement.py`. First, let's look into `parse_statements()`.

This implements the "`statements: statement | statement statements`" rule of our grammar. While we haven't reached the EOF, we're going to check for a PRINT Token, parse a binary expression, check for a SEMICOLON Token, then finally `yield` the expression's tree.

For those unfamiliar, `yield` is a Python keyword used in "generator functions". These functions return multiple values over time, using a kind of "lazy computation". This means that after a yield statement, the state of the generator function is preserved until it is "reawoken" to generate another value. `range()` is a common generator function.

Integration of Statement Parsing and LLVM Generation

Why use a generator, you are probably asking? In many cases, we want to perform lazy computation – rather than constructing a list of all the values we need at once – because the possible size of the resulting list might be very big. Consider a program in our current language that contains a million `print` statements. If we made a list of all of the ASTs in the program and THEN passed the list to our code generator, we would likely run out of memory. A generator function allows us to circumvent this and only store the data we need when we need it.

The consequence of this is that we must move our call to `parse_statements()` inside of `generate_llvm()`. Now, for each root AST of our program that we receive from `parse_statements()`, we perform stack allocation, we compile the AST to LLVM, and we generate a print instruction. Once we stop receiving ASTs, the loop terminates and we generate our postamble.

Wrapping up

Finally, we just have to update `ecco/ecco.py` so that we call `generate_llvm()` rather than explicitly parsing binary expressions or statements. And tada! We've added statement parsing to our compiler.

In the next lecture, we're going to be adding variable declarations and instances, allowing us to perform more complex computations without having to cram it into a line of constants.

Optional Homework

This week's optional homework is to implement some kind of complex mathematical statement of your choosing, and update your parser and compiler accordingly. The only requirements are that your statement has to take in an integer value as input, and that the AST generated by the statement must be a complex expression.

For example, you could write a statement like `fibonacci n;` that prints out the n^{th} fibonacci number. In this case I would suggest doing your fibonacci logic inside of the compiler and converting it to an AST we can already generate code for, rather than implementing more LLVM features than what we have now.

Your new statement has to be complex though! Don't give me `square n;` that prints out n^2. That's too easy! Give me something cool!