Comparisons

Lecture #06

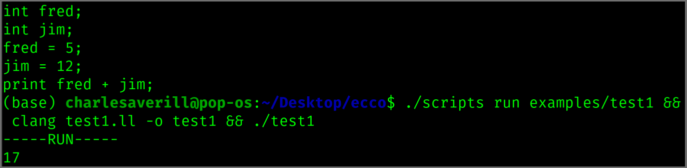
Charles Averill

Practical Compiler Design
The University of Texas at Dallas

Spring 2023

The Current State of the Compiler

In our last lecture we added variable declarations and assignments, and introduced our global symbol table:



```
int fred;
int jim;
fred = 5;
jim = 12;
print fred + jim;
(base) charlesaverill@pop-os:~/Desktop/ecco$ ./scripts run examples/test1 &&
 clang test1.ll -o test1 && ./test1
-----RUN-----
17
```

The Goal and Plan

Today, we'd like to add comparison operators to our compiler, and single-line comments for the heck of it:

```
print 7 != 9; // 1
print 7 > 9;  // 0
print 7 <= 9; // 1
```

This will require:

- Adding comparison operator tokens and precedence values
- Updating our scanner to look ahead and make sure we can tell the difference between == and =, <= and <, etc.
- Generating LLVM comparison code
- Adding extension and truncation generation functions, as LLVM's `icmp` returns `i1`s instead of the `i32`s we're used to by now

New Tokens

I've added the following new tokens with precedence values according to the C operator precedence table:

1. `EQ` ("=") - 10
2. `NEQ` ("`!=`") - 10
3. `LT` ("<") - 11
4. `LEQ` ("<=") - 11
5. `GT` (">") - 11
6. `GEQ` (">=") - 11

Comment Scanning

Comments are super easy to detect. I've only added single-lines right now but I will add multi-lines later.

We just have to check the current and next character; if they're "//" then grab the next character repeatedly until we see a newline.

Token Scanning Updates

Now that we have tokens like "<", which is a prefix of "<=", or "=" and "==", we need to look ahead at the next character to see if it's part of a length-2 token with another token as a prefix, or if it's just the prefix token.

I've added this logic inside of `TokenType.from_string()`.

Number Types

Before we can start writing more LLVM generation, we have an issue.

LLVM's `icmp` statement does not return `i32`s, it returns `i1`s! Our entire structure is built around `i32`s right now, so we need to modify it to support more types. This is why I've declared the `NumberType` class in `generation/llvmvalue.py`. LLVMValues and SymbolTableEntries now contain `NumberType`s, which store information about the bit width and type of values. Throughout `llvm.py` I have replaced all applicable `i32`s with calls to a register's number type.

Extending/Truncating Number Types

In order to use our `i1` values, we must first extend them into `i32`s so that we can put them inside variables. Where pertinent (storing and loading functions, arithmetic, etc.), I have added calls to `llvm_int_resize()`, which takes in a VR, a new NumberType, and uses LLVM's `zext` (zero-extend) and `trunc` (truncate) commands to modify the width of these numbers (lossy-ly, if truncating).

This logic will actually extend to other integer types like `short` and `long`, which ties into this week's optional homework.

Generating LLVM Comparisons

Comparisons are actually super easy to generate. Each of our comparison tokens can directly map to an `icmp` mode.

In `llvm_comparison()`, we will make sure our operands are the same width, then we will generate an `icmp` statement that gets stored into a new VR (which is loaded).

I'd like to take a second to emphasize how cool LLVM is. Our language didn't have a single thing to do with comparisons before, and now we've added two LLVM commands that handle all of that for us. LLVM is awesome!

Optional Homework

For this week's optional homework, I'd like you to parse and generate code for the remaining integer types EXCEPT for `char`, which we will do together later on.

Requirements:

- You should be able to parse `bool`, `short`, `int`, and `long` tokens
- Your compiler should enforce that assignments to these types fit within the type's range (if a programmer tries to assign 65536 to a short, there should be an error thrown)
- Your generated LLVM should use `i1, i16, i32, i64` for the widths of each of these types, for declarations, assignments, boolean expressions, etc.
- For assignments, you must truncate/extend rvalues to fit into lvalues (LLVM will enforce this anyways)

This is a big one but it's a lot of fun. Enjoy!