Global Variables

Lecture #05

Charles Averill

Practical Compiler Design
The University of Texas at Dallas

Spring 2023

Recap from Last Week

We missed Wednesday last week due to the Winter storm. In short, we added the ability to include multiple statements in a program, like so:

```
print 2 + 3 * 5;
print 50 - 8;
```

Let's quickly look at what we did to accomplish this:

- Add PRINT and SEMICOLON tokens
- Added a scanner for multi-character tokens like `print`
- Replace most EOF checks with semicolon checks
- Move statement parsing function call inside of `generate_llvm`

Did anyone do the optional homework and would like to share what they did?

Today's Goal

Today we want to add variables, not only declarations, and assignmants, but also instances, like so:

```
int fred;
int jim;
fred = 5;
jim = 12;
print fred + jim;
```

To do this, we must:

- Implement a symbol table
- Add new special-purpose statement parsing functions
- Write some annoying File I/O thanks to the nature of LLVM globals

Symbol Tables

What is a symbol table and why do we need it?

A symbol table is a key-value data structure that maps identifiers (keys) to various pieces of information related to those identifiers (values), such as datatype, scope, etc.

We will be implementing ours using a custom Hash Table. You may NOT use python's built-in dictionary, or any data structure libraries for this part!

ecco/generation/symboltable.py

I've defined SymbolTableEntry, SymbolTableInterface, and HashTableSymbolTable classes.

SymbolTableEntries will be our symbol table values. SymbolTableInterface defines necessary functions for a symbol table implementation. I defined two implementations of a symbol table, although my recursive implementation doesn't yet work, so I won't be covering it. the del/set/getitem dunder functions allow us to access the table with square brackets, like a regular dictionary.

Hash Table Implementation

As a quick recap, a hash table passes our keys (identifiers are strings) through a "hash function", which generates a (mostly) unique "hash value" that we will use as an index into an array structure.

Designing hash functions can be scary; we want to minimize collisions without becoming cryptography experts, because that would take forever. That's why we'll be implementing a chained hash table: the values of our hash table are actually linked lists of SymbolTableEntries, so if we get a collision we just append to the linked list.

Let's look at our hash table implementation.

New Statement Parsing

In `ecco/parsing/statement.py`, we've moved our print statement logic to a `print_statement()` function. Now, our `parse_statements()` function contains a series of if-elses that determines which kind of statement we'll be parsing based on the first token of the statement.

For a print statement, it's clear that the `print` keyword should come first. However, for declaration statements we should match a type first (we will hardcode `int` for now). Finally, for assignment statements we should always see an identifier token first.

We will not yet implement declaration-assignment statements like `int x = 5;`, these will come later.

New Statement Parsing

Each of our statement parsing functions will match certain tokens, and perform specific logic for each situation to build an AST Node and interact with the symbol table.

- Print statements are trivial, construct an AST Node with token type PRINT, and a left child generated by `parse_binary_expression()`
- Declaration statements are slightly more complex. Match an `int`, then match an identifier and hold onto its string value. Create an entry for it in our global symbol table, and finally declare a global variable in LLVM. We will be generating these statements in a separate LLVM file that we will discuss soon
- Assignment statements require that we match an identifier (we will call it an LValue to distinguish it from instances of a variable). If the identifier does not exist in the GST, we will throw an error, because it has not been declared yet. We will match a new ASSIGN token ("="), then parse a binary expression. Our output ASTNode will have type ASSIGN with an LVALUE_IDENTIFIER child and a binary expression

LLVM Generation for Variables

One annoying thing about clang is that it puts global variable declarations at the top of LLVM files. They don't need to be at the top, but that's what clang does, so that's what we'll do.

We're going to generate the LLVM for our global variable declarations in a separate LLVM_GLOBALS_FILE, then insert that data into the top of our main LLVM file once we've completed all code generation.

Changes to our Code Generation

We've added a new parameter, `rvalue_register_number`, to `ast_to_llvm()`. This will store the register number of RValues (expressions on the right-hand side of an assign statement). We need to add this so we can transfer information from an ASSIGN node's left child (which contains a binary expression value) with its right child (an LVALUE_IDENTIFIER node) so that the expression value can be stored into the global variable. Now, when we traverse the left child in our DFT, we pass the output register number into our right child's traversal.

Generation Cases

If our AST's token type is an IDENTIFIER, the programmer is using a variable instance as a value in a binary expression (rvalue). So, let's load the value into a register and return that virtual register index.

If its type is an LVALUE_IDENTIFIER, we know we're at an assignment statement. So we'll store the contents of our `rvalue_register_number` into the corresponding global variable.

Generation Cases

If we've reached an ASSIGN node, remember that this is the same case as the LVALUE_IDENTIFIER node. However, the LVALUE case already handled our logic here, so we'll just return the VR that the value is loaded into.

Finally, if we've reached a PRINT node, call our already-existing print statement generation function.

When all of our code has been generated, we will insert our global variable declarations at the top of the file.

Expression Parsing Update

We are almost done. So far, what we've done will work for assignments, but we have to handle the case where a variable instance is used in an rvalue.

This is actually very easy to handle: we will update the conditional statements in `parse_terminal_nodes()` in `ecco/parsing/expression.py` to add a case for IDENTIFIER tokens.

And we are done! Reminder to please send me your repository links!