Keith Strickling
260674699
COMP 520

Milestone 3 Write-Up

Target Language

I have elected to have my GoLite compiler generate C++ code. The reasons are simple. Firstly, C++ data types are very similar to GoLite data types. For example, C++ and GoLite structs are essentially identical in nature, as are C++ vectors and GoLite slices. Secondly, the reference compiler generates C++ code, so I can easily verify the correctness of my generated code by comparing it to the code generated by the reference compiler. Additionally, C++ has the following advantages and disadvantages:

Advantages

- C++ is a high-level language. Consequently, I do not need additional passes in my compiler for the computation of resources such as offsets and local stack sizes.
- C++ code executes quickly (in comparison to many other languages). Thus, my compiled GoLite programs should also execute quickly.

Disadvantages

- There is no construct in C++ that matches the type declaration construct in GoLite. Additionally, you cannot declare structs "on the fly" as you can in GoLite. Consequently, I need an additional pass prior to code generation to aggregate struct declarations (and also array declarations) appearing in the GoLite code so that I can declare them prior to any of their uses in my generated C++ code.
- Again pertaining to the lack of a type declaration construct in C++, I need to piggy-back on the additional pass described in the above bullet to resolve all GoLite types to their equivalent C++ types before generating the code.
- I do not know C++! I'm learning it for the purposes of this project.

Summary of what I have implemented so far

At this stage, I have implemented most of the code generator! The following has been implemented in full:

- A pass to aggregate structs and arrays and also to resolve GoLite types to their equivalent C++ types.
- Variable declarations
- Type declarations (handled by the extra pass)
- Function declarations
- Code generation for the following statements has been implemented in full

    - Empty statements
    - Expression statements
    - Increment/decrement statements
    - Regular/short assignment statements
    - Variable/type declaration statements
    - Return statements
    - If statements
    - If/else statements
    - While loops
    - Infinite loops
    - Break statements
    - Continue statements
    - Block statements
- All expressions minus those listed below

And the following hasn't:

- Code generation for the following statements has not been implemented
    - Binary operation assignment statements
    - Print statements
    - Println statements
    - Switch statements
    - Three-part for loops
- Code generation for the following expressions has not been implemented
    - Unary XOR
    - General equality/comparison testing (as of right now, the "==", ">=", "<=", "!=", ">", "<" operators are treated as the correct comparison operators for all types)
    - Bit clear expression (&^)
- Handling of raw string literals

Additionally, I still need to thoroughly test the code generation that I have written and the code generation that I have yet to write. Finally,

Outline of test programs

1. functionTest.go
   In this test, I check that GoLite function signatures are generated correctly. In order to thoroughly test function declarations, the test file includes a function with no parameters and no return, a function with parameters but no return, a function with a return value but no parameters, and a function with both parameters and a return value. Additionally, this file includes the main function in order to test that the generated code treats this function properly. For example, the point of entry in C++ is the main function, which has an int return type. Thus, the code generator should take the main function in GoLite,

which may or may not have a return value of type int (and can have no parameters), and generate a function with a return value of type int.

2. ifStatementTest.go
   In this test, I check the code generation of if statements. The test is thorough, covering if statements with an init statement and condition, if statements with just a condition, if/else statements with an init statement and condition, and if/else statements with just a condition.
3. typeDefTest.go
   This test is interesting. In general, it tests that code generation of type declarations works properly. It includes the declaration of an alias to type int, as well as two struct types, Person and SuperHero. SuperHero contains a field of type Person. Lastly, it includes an array of SuperHeros. As an added bonus, this test tests the assigning of default values to various types. This test is especially important to me because, as described previously, C++ type declarations cannot occur on the fly, and C++ does not allow you to reassign keywords such as “int” to a new type. So, for my purposes, this file also tests that I properly identify all the structs (and arrays) in the GoLite code and create corresponding C++ typedefs prior to any function declarations.
4. varDeclTest.go
   This test focuses on variable declarations. It includes variable declarations both inside and outside functions. It also includes variable declaration lists, and distributed variable declarations. It has variables declared with no type, variables declared with a type but no expression, and variables declared with both a type and expression. Lastly, it covers short variable declarations, including a short variable declaration with only new variables and a short variable declaration including a redeclaration.
5. forStatementTest.go
   This file covers the testing of all varieties of GoLite for loops. It includes a “while” loop (a for loop with just a condition) and an infinite loop (a for loop with no condition, init statement, or post statement). It also includes all eight possible ways to write a three-part for loop (since each of the init statement, condition, and post statement are optional). Note however that my code generator does not yet deal with three-part for loops correctly.