Milestone 2 Report

Group comp520-2017-01

Design Notes

Target Language

The language that our compiler is targeting is Javascript. This choice was made since Javascript is ubquitous, and for better or worse, is gaining traction in in many domains.

The pros and cons of targeting Javascript.

Pros

- Javascript is high level. We avoid the easily botched task of memory management. This also gives improved programmer productivity, compared to the often more tedious task of programming in a lower lever language.
- Javascript's language features have semantics that are similar, if not identical, to those found in Go/Golite. In most cases, this means generated code in Javascript is very similar to the original Go/Golite. This similarity allows us to be more confident that the generated code is correct.

Cons

- Performance. The choice of a lower level language would have resulted in improved performance.
- Type system. Javascript is dynamically typed, and has implicit type conversions. Even though the source program has already been type checked by the time Javascript code is generated, if the generated code was statically typed, we would have an additional method to verify the correctness of generated code.

3. Progress Report

The program responsible for code generation is completed. What remains to be done is testing and fixing bugs.

Code generator

This collection of functions implements code generation. It is a series of recursive functions, not much unlike the typed pretty printer, that generates code for a particular structure by calling functions responsible for generating code for each of its sub-componenets. This recursive generation continues until we arrive at some base case, literals or variables, for example. Code generation for certain language features involved some additional effort. These instances are breifly described below.

Scoping Rules

The canonical way to declare variables in Javascript, with the `var` keyword, results in variables

having global scope or function scope, depending if they are declared outside a function or inside one, respectively. If code was generated using the `var` keyword, the generated code would not behave as intended. This was an oversight that caused some confusion when running initial tests. Fortunately, this mistake was noticed and corrected by using the `let` keyword instead. This feature, available in modern implementations of Javascript, binds variables in the current block. By using `let` instead of `var`, the intended semantics are easily expresssed.

Variable initialization in control structures

While the `for` loop does allow for variable initialization, the `if` and the `switch` statements do not. To generate correct code in the instances where a short variable declaration is provided, we output the given declaration in each block of each structure. For example, the Go/Golite code

```
if x:=1, true {
  // do something
} else {
  // do something
}
```

gets translated to the following Javascript code

```
if (true) {
  let x = 1;
  // do something
} else {
  let x = 1;
  // do something
}
```

Initializing values for aliased variables

Variables get initialized to some default value on declaration if they are not assigned one. This is also the case with variables of a type aliased to some base type that has a default value. For example:

```
type foo int
var bar foo
// var = 0
```

This requires resolving the base type of all varaibles that of an aliased type. We accomplished this by retyping the variable. This approach was chosen since, for our team, it was the easiest and simplest way to implement such behavior.

Testing

Since the code generator itself is finished. What remains to do is testing. A bash script, as well as

run flags, have been implemented to aid in testing of generated code. We do this by comparing the output of running our generated code (via Node.js) to the output of running the original Go/Golite code (via the Go compiler).

4. Description of Programs

We decided to test the following five language constructs: * append * equality * identifiers * if statement initialization variable declaration * type casting

Append

For the `append` construct, we tested append with all four types that can be used with `println`. For each of the types, an empty slice was declared, and `append` applied to it. The behaviour was then verified by looping over the length of the slice and printing out what each value is, versus what we expect it to be.

Equality

For equality, we verified that all types that can be equated, including structs, work as intended. Again, a similar approach to the one used in verifing the `append` construct was used. Two variables of each type were declared and given values. Then the equality comparison operator `==` was applied to the variables. The expected result of the comparison and the actual result of the comparison are then printed to the screen.

Identifiers

The following behaviors of identifers were verifed. First, it was verified that type names (`int` etc.) were valid identifiers. Then, we verified that identifiers could be used in-place of values in expressions. Finally, it was verified that the value of identifiers can be reassigned. Again, testing involved printing the expected resutl versus the actual result.

If Statment Variable Initialization

The optional variable declaration in `if` statements was verified by checking that, first, the initialized variable was indeed accessible inside the body of the `if` statement. Secondly, it was verified that the variable declared in the initialization of the if statement could be shadowed by a variable of the same name declared inside the body of the `if` statement.

Type Casting

Type casting was verified by checking that all types could be casted to each other. This was done by declaring variables of each type and casting them to one another. In addition, the casting of aliased types was also verfied by declaring two aliased types, both capable of being cast to each other, and two variables, one of each aliased type. Then they were cast to each other and the outcomes compared.