COMP520 - Group 10
Milestone #3

David Herrera `Id: 260422001`
Maurice Zhang `Id: 260583688`
Adam Bognat `Id: 260550169`

March 27, 2017

1 Target Language Decision

Our compiler will produce Javascript source code. We decided to settle on a high-level target language so that we could focus on the principles behind code generation without getting bogged down with the messy details of a lower level language. Most importantly, Javascript has lexically scoped functions and blocks, which makes implementing the GoLite scoping rules incredibly straightforward, given the ***let*** keyword from ECMA2015. We discussed other target languages and found elegantly implementing said scoping rules was one of the major challenges of the code generation phase. Moreoever, being written in a dynamically typed language, the target Javascript code is more concise, as we (mostly) don't have to worry about types at this stage. Furthermore, several higher-level operations on strings and dynamics arrays are included in the standard implementation, so code for most operations can be carried over directly from GoLite.

There's no free lunch, though, and some tradeoffs have to be made. In GoLite, arrays have pass-by-value call semantics and slices have pass-by-reference call semantics, but in Javascript both objects are passed by reference. Thus we distinguish arrays from slices by wrapping them in objects and ensure arrays are cloned when used in function calls. This is not the most elegant solution but it's preferable to making the symbol table explicit. Some GoLite constructs are not supported in Java so auxiliary functions will have to be defined to reproduce the functionality (e.g. GoLite supports tuple assignment of the form `a, b = b, a` while Javascript does not).

2 Implementation

So far we have implemented three constructs, the basic package declaration,variable initialization and function declaration.

2.1 Package Declaration

This was done in the way a NodeJS module is initialized:

```
var main = module.exports = {};
```

2.2 Variable Initialization

To implement Go variables correctly in Javascript, scoping was a major concern due to the fact that variables declared with the keyword *var* do not respect respect the Go rules for scoping. The following snippet returns the value of 3 for both print statements.

```
function scopes()
{
    var a = 4;
    {
        var a = 3;
        console.log(a);
    }
    console.log(a);
```

```
}
scopes();
```

Taking advantage of the ECMA2015 standard, it was possible to use the *let* keyword to defined a variable and achieve the desired result. In terms of actual initialization of variables, Go initializes variables to basic constructs, while Javascript defines all the variables that are not explicitly initialized to undefined. This means that we must initialize variables that are not initialized in *codegen* using recursion for variables that resolve to arrays, structs and slices, the following two snippets contain the initializations.

In Go Language:

```
package main
var b,c int
var k rune
type t struct{
                a int
                b float64
                c struct {
                        t []int
                }
}
var a [3]t
```

In Javascript:

```
var main = module.exports = {};
let b = 0;
let c = 0;
let k = 0;
let a = [{"a":0, "b":0.0, "c":{"t":[]}}, {"a":0, "b":0.0, "c":{"t":[]}},
        {"a":0, "b":0.0, "c":{"t":[]}}];
```

2.3 Functions

Function declarations are simpler since Javascript does not offer types, that implies that the arguments to the functions do not have a type and moreover that returning functions are not differentiable from non-returning in Javascript. The following snippets contain the translations.

In Go Language:

```
func foo(a int, b float64) bool{
        return true;
}
```

In Javascript:

```
function foo(a, b){
        return true;
}
```

3 Construct Tests

We chose to consider the following constructs.

3.1 Assignment

For the assignment construct, we are making sure that variables being assigned a value indeed contain the proper value after the assignment. This applies to variable declarations, short assignments, and assignments. Our tests emphasize assignments to multiple variables in one statement. They also make sure to test the tricky situation where a variable is being assigned to and its value is being assigned to another variable at the same time, for example:

```
var a, b, c = 3, 4, 5
a, b, c = c, a, b
println(a, b, c) // 5 3 4
```

3.2 Equality Comparison

For the equality construct, we are making sure that equality between expressions is properly implemented. In particular, we are making sure that equivalent expressions that are not represented in the same way, are being properly evaluated. For example, `3` should be equal to `03`. This notion is important when comparing a string literal to a raw string literal. Also, we are verifying that equality for structs and arrays checks that the contained values are the same. Here's an example for arrays:

```
var a1, a2 [4]string
a1[0] = "1"
a2[0] = "1"
println(a1 == a2) // true
a2[0] = "2"
println(a1 == a2) // false
```

3.3 Variable Initialization

For variable initialization, we are making sure that variables declared without a value contain the proper default value according the its type. For example,

```
var a int
println(a) // 0
```

We are also testing that the default values are properly applied for variables that are declared with an aliased type. The default value of an aliased type should be the default value its base type. For arrays and structs, we are checking that the inner values contain the proper default values according to its inner type. For example, an array of `int` should initialize all of its inner values to 0.

3.4 Passing Reference Types

For this construct, we are verifying that slices (and its aliases) are passed to functions as references and that arrays and structs are passed as values. For example,

```
func changeFirstArrayVal(arr [5]int) {
   arr[0] = -1
}

func changeFirstSliceVal(slice []int) {
   slice[0] = -1
}

func main() {
   var arr [5]int
   arr[0] = 1
   changeFirstArrayVal(arr)
   println(arr[0]) // 1

   var sli []int
   sli = append(sli, 1)
   changeFirstSliceVal(sli)
   println(sli[0]) // -1
}
```

3.5 Typecasts

For the typecasting construct, our tests verify that the subsequent expression following a typecast has the proper value. For example,

```
println(int(3.4)) // 3
println(rune(65.5)) // 65
println(bool(4)) // true
```

We also make sure to check that this is the case for type casts to type aliases.

4 Group Structure

Our team has no hierarchy. We validated each other's design decisions and code implementations and had great collaboration among all team members. The tasks were broken down roughly as follows:

- Adam: Wrote benchmark programs and wrote-up language decisions.
- Maurice: Created the valid tests for the constructs and wrote the accompanying section in the report
- David: Implementation of initial steps of the *codegen* phase.