GoLite Milestone 3

James Brace, Paul-Andre Henegar, Youri Tamitegama

Intro

For the code generation process of our GoLite compiler, we decided we would compile to Javascript. The pros of compiling to a high-level language like Javascript is that its expressiveness will make it much faster to perform code generation for most of the semantics of GoLite compared to a lower-level language like LLVM. The downside of using a high-level language is that we can by accident use some feature without realizing that its semantics aren't exactly the same as in GoLite (This would trivially not happen with a very low-level language -- cannot misuse a feature when there are no features). We have initially considered LLVM for the experience, however we decided to use a higher-level language because we did not learn enough LLVM in time and we have very tight remaining time constraints.

We considered C, Python and JavaScript. C is one of the languages with the the closest semantics to Go. For example, in both C and Go, structs lie contiguously in memory, are pass by value and actually contain their fields, unlike languages like JavaScript, Python or Java. However, one big difference between C and Go is that C is much less flexible about where struct can be declared, unlike Go, which lets has anonymous structs and lets you declare struct in the middle of functions. We would have needed to hoist all structs into the global scope. We would also needed to generate code to do things like copying and comparisons. We have instead decided to use JavaScript, use object literals for everything and do most operations like comparisons and copying dynamically using prewritten functions. Some disadvantage of JavaScript is that it does not have operator overloading, though it is not necessary if we use dynamic function. Another disadvantage is that JavaScript does not have integers and we need to work around it. However we have selected JavaScript because of our familiarity with it and because object literals seemed to make dealing with struct easier.

Printing

In Go, the printing functions have a somewhat unusual behaviour.

Basic Types

When printing basic types, Go prints an interpretation of the input:

- **Runes**: the numeric ASCII value of the rune will be printed
- **Integers**: the initial format of the integer (octal/decimal/hexa) will be ignored, and the value will be printed in decimal
- **Float**: the float will have the format [sign][mantissa]e[sign of exponent][exponent], with 7 significant digits for the mantissa and 3 for the exponent

- **Strings**: interpreted strings will have their escaped characters interpreted, while raw strings will simply be outputted as is
- **Booleans**: the strings "true" or "false" will be printed depending on the value

To take care of this, we will generate and use separate functions to handle printing each type.

Nested Function Calls

When a function call is present in an argument of the print function, Go will execute it before outputting the arguments. This could potentially be tricky to implement if the function call is nested within an expression (eg. used as an index). One way to go about this is to make one variable for each function call, replace the function calls in the arguments by their variable, then evaluate all function calls and assign their return values to their respective variables. Once this is done, we can print the arguments. However this approach might run into problems if we have function calls that return pointers.

print() versus println()

Against all intuition, the print() and println() functions do not exhibit similar behaviors. Indeed, when provided a list of arguments, print() will output each argument without inserting spaces in between, while println() will output arguments separated by spaces, then insert a newline character at the very end. Since we will most likely need to separate the arguments of the print statement because of the issue of nested function calls, we will be able to add print statements for blank spaces between arguments if necessary.

Types

Basic Types

In GoLite, there are 5 basic types: int, rune (practically 32-bit ints), float64, string, bool. JavaScript has 64-bit floats, strings and bools.

JavaScript technically does not have integers, but 32-bit integers can be be represented using the 64-bit floating points. We can simulate integer numeric operations (with overflow) by bitwise-ORing the result with 0 for most operations and using Math.imul for multiplication. Floating point numeric operations should be exactly the same. All bitwise operations in Javascript work as if they were applied to 32-bit signed ints and operate the same as in GoLite (except &^ that does not exist in JavaScript). Strings in both JavaScript are as we would expect. String addition represents concatenation, and ordering operators of strings uses lexicographical order.

Struct Types

In GoLite, struct types can be either defined or anonymous. In GoLite, the only way to initialize a struct is to not provide any initialization and let it be initialize to the zero value. In JavaScript, there are no structs, but there are objects. T he simplest way to create objects in JavaScript is using object literals. Since it’s a dynamically typed language, we also don’t need to specify that a variable has a certain object type. Because of this we won’t include any type definitions and instead initialize all structs using object literals representing the zero value, doing it recursively if a struct contains other structs. (This could lead some pathological programs to blow-up exponentially after compilation, but we hope that only a small portion of the test programs will be like that)

Arrays Types

Arrays in GoLite are fixed-sized and are similar to structs. In JavaScript, through some weird turn of events arrays were defined as objects with elements indexed by strings representing numbers (“0”, “1”,”2”,”1000”) with an automatically updated “length” property. Anyway, we are going to represent GoLite arrays as normal JavaScript arrays whose size we will not modify. GoLite arrays are initialized by initializing all their elements. To initialize the arrays in JavaScript, we will use a makeArray function, that will create a new array of a specified size filled by deepCopied instances of a provided object.

Slice Types

Slices in GoLite represent pointers into an underlying array along with a size. When we append to a slice, we create a new slice, sometimes referring to the original underlying array and sometimes creating a new array and copying the previous values. There is no equivalence for this in JavaScript, so we’ll need to create our own. We’ll represent them using an object containing a reference to an underlying array and a size, and we’ll implement an append function following mimicking the behavior of the reference compiler.

Defined Types

In Go, two variables can have the same underlying type but with a different defined type. In JavaScript, there is no such thing. And we really don’t need to worry about that since we would have caught any such errors at compile time. So when compiling to JavaScript, we will simply ignore all type definitions.

Comparison Operators

In GoLite, we can use ordering operators on integers (also runes), floats and strings. In JavaScript, we can use them on numbers and strings and they work as expected. In GoLite, we can also use equality operators on structs and arrays that don’t contain slices, which compares all element one by one. If we try to use “==” or “===” on JavaScript objects or arrays, it will tell us whether the references point to the same object. To do actual comparison, we will implement a deepEq function that will dynamically and recursively test if two objects are equal.

Bound Checks

Go bound checks its arrays and slices at runtime and crashes if an index is not in bounds. Also, the bound that is checked for a slice is not for the size of the underlying array, but for the size of the slice itself. In JavaScript, if we try indexing out of bounds into an array, it will return an *undefined*. Hence we will need to check ourselves whether an index is in bounds before using it to index into an array or the underlying array of a slice. We can do that by wrapping the index in a *checkIfInBound* function.

Scoping Rules

Blocks

For the most part, Go's scoping conventions are nothing unusual. Regarding variable declarations, the scope of variable identifiers exists in its encompassing block and any nested blocks within that scope. Variables can only be declared once in a block, but are free to be redeclared in a nested block. In Go this is called shadowing. Javascript has the same semantics with it's `let` declaration, however using it will require close analysis in order to make sure that Go semantics are properly preserved and the code is doing what it is supposed to be doing. In order to allow shadowing, we will name our variables in an {id}_{#} fashion. Another exception to normal scoping conventions in Go is that type declarations are unique. However, as far as codegen is concerned, we won't be worrying about type declarations, for we will use our annotated AST to map variables to resolved types.

Functions

As with blocks, scoping rules of functions in GoLite are the norm. Parameters are only present within the body of the function and normal block scoping rules apply. Note that the scope of a variable in GoLite starts after the end of the varspec, this allows the valid use of `var x int = x` within a function block (or within any scope for that matter) when the RHS x is declared in a previous scope. The same does not occur in javascript, where `var x = x` will result in *undefined.*

Control Flow

Control flow in Go is a bit more complicated given that you can define simple statements in the conditionals of if statements. This means that you can add a variable to the scope of an if-statement's body before entering the body. This cannot be done in Javascript, so for this scenario we will need to execute statements before entering the conditional. The rest of control flow in Go follows conventions normally seen in programming languages and are reflected in Javascript (i.e loops & switch statements).

Status Report

Nothing.