COMP520 - Group 10
Final Report

David Herrera	Maurice Zhang	Adam Bognat
ID: 260422001	ID: 260583688	ID: 260550169

April 12, 2017

Contents

1 Introduction

Go is a free and open-source programming language developed by Rob Pike, Ken Thompson and Robert Griesemer while employed at Google in 2007. It is a compiled, statically-typed language featuring garbage collection and built-in concurrency support. It enjoys use by a number of companies, including Netflix, Google, Soundcloud, and Twitch.

The language for our compiler is a "subset" of Go, GoLite, a pedagogical language invented by Vincent Foley. Restrictions include a minimal type system, an expanded set of reserved words, declaration before use, limited means of abstraction (no modules or interfaces), no concurrency support (no channels or co-routines). Moreover, properly speaking, GoLite is not a subset as some features of GoLite are not supported by Go (such as unused declarations). Nevertheless, GoLite as enough structure to conveniently express common computational patterns and to serve as a genuine challenge to any compiler writer.

This report is organized as follows: Section 2 describes our group structure and the contributions of each member to the project. Section 3 discusses our choice of implementation language, external tools and target language. Sections 4 - 6 comprise the bulk of the paper and cover each stage of our compiler in detail, including design choices, implementation challenges, and testing strategies. Section 7 concludes the report with a discussion of the remaining issues with our compiler and possible areas of of improvement.

2 Group Structure

Our team has no hierarchy. We endeavor to divide the work equally, and there is often overlap between the tasks taken on. 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 contributed to the scanner and weeding phase, and wrote most of the milestone 1 and 2 test suites and some milestone 3 benchmarks. The test suites were refined with the help of the other members.
- Maurice, took charge of the parser and the implementation of the abstract syntax tree. Collaborated throughly with tests and decisions in the project. Implementation of valid programs.
- David, took over from Adam's scanner section and finalized the details of the scanner. Design decisions in terms of the AST and aid with it's implementation. Set up for project, Implementation of pretty printer.

3 Implementation Language and Tool Choices

We opted to use Flex+Bison and the C programming language to implement our GoLite compiler due to familiarity and flexibility. Separating the scanner from the parser allowed for greater modularity, and the precedence directives available in Bison simplified writing the grammar. C was then a natural choice to use with these tools. Also, we were all familiar with the tools since we had used them to implement the Minilang compiler. This allowed us to focus on the challenges of the problem rather than fighting with the tools.

In general, C was a good choice. It provided for a fast compiler and as mentioned, Flex and Bison were good tools to work with. However, C did have its complications. Having to deal with pointers in such a large project could be troublesome at times, especially when trying to debug segmentation faults. Additionally, C's lack of data structures such as arraylists could slow down our workflow, as we sometimes had to find solutions to problems that could easily be solved using those data structures. In the end, we are happy with our decision of going with C as our implementation language.

We chose the Javascript ES6 implementation, as the target language that our compiler would output. 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 made

implementing the GoLite scoping rules incredibly straightforward, given the ***let*** keyword from ES6. We discussed other target languages and found elegantly implementing said scoping rules was one of the major challenges of the code generation phase, although we still encountered some issues, as discussed in the code generation section. Moreoever, being written in a dynamically typed language, the target Javascript code is more concise, as we (mostly) didn'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.

Still, tradeoffs had 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 ensured that arrays were cloned when necessary, such as passing them to function calls and assigning them to variables. Some GoLite constructs are not supported in Java so auxiliary functions will have to be defined to reproduce the functionality (e.g. GoLite supports assignments of the form `a, b = b, a` while Javascript does not). Further details about the issues encountered during code generation are discussed in the code generation section below.

4 Scanning and Parsing

4.1 Scanner

Scanner for GoLite unlike Go would only accept ASCII characters. For the scanner the decisions were pretty straight forward. The following is a list of the main highlights for this part.

- To handle block comments the Regex from the O. Miller blog was used. [1]
- A tricky aspect of the scanner was handling the different cases for interpreted strings, runes, and raw strings. They each have different rules. For instance, escaped characters. Interpreted strings and runes, accept a list of given escaped charaters in both forms. i.e. \\t and \t. They, however, do not accept any other escaped characters not in that list, nor do they accept a literal \n, unless is in the form \\n. Raw strings on the other hand allow any combination of escaped charaters, even sequences that are not escaped charaters, moreover, raw strings allow a literal Enter (\n) in the source code as part of the string. Runes, although meant to represent one character, are actually two characters in some cases such as the one above with the new line. This brought us to the decision to keep the representation of a rune as a C string internally which in turn avoided having to translate the escape characters back and forward in future phases such as pretty printer, and type printer.

4.2 Parser

4.2.1 Short Assignments

For the parser, most rules were fairly straightforward to implement and were done so simply by following the Go specifications. The biggest issue that arose was a conflict between assignments and short assignments. On the left hand side of an assignment, expressions are allowed (only those that are addressable). This includes identifiers. For a short assignment, the left hand side must only comprise of identifiers. This created a conflict because if the first element on the left hand side of an assignment was an identifier followed by a comma, the parser would not know whether to reduce it to a list of identifiers, or to a list of expressions since they both could start with identifiers. For example,

```
varName, a, b := 2, 1, 4 // short assignment
varName, a[3], b.field = 2, 1, 4 // assignment
```

Therefore, to resolve this conflict, we allowed the left hand side of a short assignment to be expressions in the grammar. However, when constructing the abstract syntax tree, we would check and see if those expressions were only identifiers and throw an error if that wasn't the case.

Also, for structures that should be grouped together as lists, like a list of statements, linked lists were used. This was implemented with a field in each node type pointing to the next node. These links were created in the parser actions in reversed order to make for simpler, more efficient code. Thus, when traversing through the tree, we had to make sure that lists were traversed in a head recursive manner.

4.2.2 Left Values

Another noteworthy detail of our implementation is that we handled both blank identifiers and left value expressions in the parser. For left value expressions, it was done so with a nonterminal `leftexpr` that would produce either a parenthesized left value, an identifier, an expression indexed by an expression, and expression selected with an identifier.

4.3 Abstract Syntax Tree

The construction of the AST was a pretty straighforward procedure without any pecularities other than verifying short assignments as previously mentionned. For some nodes that had many fields, like the `if` statement, we defined a separate structure. Also, although a block is only a list of statements, we initially gave it its own structure type to make for clearer code. This decision turned out to be necessary later on, as we would attach a SymbolTable to this Block Node.

We also attached types for which we knew, such as declarations with an explicit type, function parameters, etc.

4.4 Weeding Phase

Several parsing details were deferred to a weeding phase:

1. Checking that the number of identifiers on the left-hand side of an assignment statement equals the number of expressions on the right-hand side. In hindsight, this could have been more easily implemented when constructing the assignment node during AST construction.
2. Ensuring a `switch` statement has no more than one `default` case.
3. Ensuring that a `continue` statement only appears within the body of a loop.
4. Ensuring that a `break` statement only appears within the body of a loop or `switch` statement.
5. Ensuring `return` statements would occur in every execution path given a function with a return value. This implementation followed the initial specification of return statements. In some cases it does not generate valid Go code, such as when having statements after a return. Additionally, this implemenation does not take into account break statements when looking at a return.

4.5 Testing

In this section, we made a variety of valid and invalid tests. In total for this part we had 97 invalid tests and 50 valid ones that tested different features of the language. We also try to let the person that was least familiar with the implementation of a particular section to write the tests. To test the lexer without waiting for the scanner, another file called testlexer.l was created. To test the programs for valid or invalid we created a script[1] that went through the folder For more information about the tests see: `https://github.com/Sable/comp520-2017-10/tree/master/src/test/ml1`.

5 Symbol Table

5.1 SymbolType vs. Symbol & Type

We made the design decision to separate the type-checker and the symbol table in a way that allowed each team member to work separately on an implementation, additionally, it allowed to give modularity and clarity to the code. Having said that, this decision had three consequences. Firstly, two passes of the tree had to be done done, one for the symbol, one for type-checker. Secondly, the symbol tables were attached to the pertinent nodes in the AST that opened a new scope for the type-checker to grab later on. Lastly, variables that might not typecheck are in the symbol table until the type checker removes them from it in case of an error.

[1] The script was heavily based off the scripts provided for minilang

5.2 Predeclared Types

All the predeclared types are of type alias, as per specification. The type node alias contains a field called actual_type, which reference the actual_node types, this is set by the type phase.

5.3 Shadowing

In order to shadow variables such as predefined types/variables a new scope was defined to mediate this process. This had the following consequences:

- There was a pre-program symbol table that contained the pre-declarations. The top-level, or program symbol table was a child of that pre-symbol table.
- To handle for, if, and switch statements there were two new symbol tables created, one for the simple/post statements and one for the actual block, in that way, it was possible to achieve the shadowing of variable/type declarations by simple statements and the shadowing of the block statements on the simple statements. This also meant that to print the symbol tables it was necessary to recurse to the next scope symbol table for those particular statements.

5.4 Block Nodes

Separating successfully the type and symbol parts meant that we had to make some modifications to the tree structure, instead of for, if, switch, pointing at statements, they now point to a new node *Block* that contains the statements and the symbol table. (Other nodes were also modified to include a type).

5.5 Struct Type Literals

To keep track of those struct fields, a temporary symbol table was made in order to make sure no field was repeated while checking the types using the upper symbol table.

5.6 Code Generation

To aid with the implementation of Code Generation, since the symbol section was already doing all the scope analysis for the compiler, it made sense to add code generation details of implementation to this section. For more information refer to Section 7.

6 Typechecker

Here are some noteworthy design decisions and implementation details that are part of the type-checking phase of our compiler.

6.1 Comparing Types

6.1.1 Base Types

We defined five constant TYPE nodes that represented the base types:

- `BASE_TYPE_INT`
- `BASE_TYPE_FLOAT`
- `BASE_TYPE_RUNE`
- `BASE_TYPE_BOOL`
- `BASE_TYPE_STRING`

These types were assigned to literals and to the predefined type identifiers (`int`, `float64`, `rune`, `bool`, `string`).

Consider the following case:

```
// valid
var foo int
foo = 3
```

In this case, the expression `3` is attached with the literal base type `BASE_TYPE_INT` and the variable `foo` is attached with the identifier type `int` who's actual type is `BASE_TYPE_INT`. In situations like these, where we compare an identifier type and a literal type, we must determine if the identifier type is an alias to the literal type. Thus, to make the comparison, we would get the actual type of the identifier (which can be another identifier type), check whether or not that actual type is a base type, and then compare between the base types. This sequence correctly invalidates the following situation:

```
// invalid
type int string
var foo int
foo = 3
```

In this situation, the variable `foo` has a type `int`, which is an identifier type who's actual type is `string` (also an identifier type). Thus, the actual type is not a literal base type and the typecheck for the assignment does not pass.

6.1.2 Arrays and Slices

When comparing array and slice types, we check that the inner types are equal. For an array, we also check that the array lengths are the same.

6.1.3 Structs

For structs, we had to check that both struct definitions had the same fields. This was a simple task since the fields had to be in the same order. Thus, we simply iterated through both struct fields and compared the names and types.

6.1.4 Type Aliases

When comparing type aliases, we simply compare the pointers of the `TYPE` nodes since those nodes are only created once per type declaration. This ensures that type aliases with the same name and same actual type are still considered different. For example, the following case is invalid:

```
type num int
var a num
{
  type num int
  a = num(23)
}
```

Although both type aliases have the same name and the same actual type, they are not the same and thus the assignment is invalid.

6.2 Resolving to a Type

While implementing the typechecker, a recurring task was to see if one type resolved to another. For example, when indexing an array, the index expression must resolve to an `int` or `rune`. To help with this, we created a function called `get_root_type(TYPE *type)` which would take a type as input and output it's deepest connected type that wasn't an alias. Consider the following type declarations:

```
type num int
type number num
type number1 number
```

With these type aliases, if we were to pass a `TYPE` node representing the `number1` type alias to `get_root_type()`, it would output the literal base type node representing an `int` (`BASE_TYPE_INT`).

6.3 Testing

In this section, we made a variety of valid and invalid tests. In total for this part we had 84 invalid tests and 67 valid ones that tested different features of the language. `https://github.com/Sable/comp520-2017-10/tree/master/src/test/ml2`.

7 Generating Code

Overall, the implementation of the code generator was very straightforward as it involved simply parsing through the AST and outputting the code segments. However, there were some discrepancies between Go and Javascript which is discussed below.

7.1 Naming Convention

To avoid any conflicts, in terms of naming and keywords with Javascript, every indentifier was named to have an _. An example, would be the pre-declared Go variables _true, and _false.

7.2 Handling Types

Since Javascript is not a strongly typed language and since the previous part of our GoLite compiler took care of the type semantics, there was absolutely nothing to do in Javascript code generation in terms of type aliases and types in variable declarations. This was an attractive feature that aid us to decide on Javascript as our target language. Table 1 contains the Go to Javascript types as were internally represented by Javascript.

Go	Javascript
int, float64	Number
rune, string, raw string	String
array, slice	Array
struct	Object
bool	Boolean

Table 1: Go to Javascript type translation

7.2.1 Bool

In Go, true and false are predeclared variables. To maintain the same semantics, we first wrapped the code around a block, and then created variables _true, _false in Javascript.

```
let _true = true;
let _false = false;
{
//Rest of Go Program
}
```

7.2.2 Arrays & Structs

To provide the correct Go semantics of arrays and structs in Javascript, we had to make sure the these types were passed and compared by value. This meant that any time an array or struct was part of an expression on the right hand side of an assignment or declaration, or simply as an argument to function call, we had to create a copy of the array or the object. Moreover, this also meant that evaluation of equality for arrays and structs had to be done by deep comparison and not pointer comparison as is implemented by Javascript. In order to do this, we created four functions in a utility module to maintain this semantics. i.e. `copyArray`, `copyObject`, `equalObject`, `equalArray`. The first two, as their names indicate, create a deep copy of arrays and objects, the last two compute a deep comparison of the objects/arrays.

7.2.3 Splices

To represent the Go splice type, we also use the arrays since arrays in javascript are dynamic. The only special consideration for this type was the `append` function. For this, we use a utility function `appendSlice`, this function similarly to the go function, creates a copy of the slice, adds the right expression and returns this newly created slice.

7.2.4 Numbers in Javascript

In contrast to Go, Javascript does not have separate numerical types such as integers, floats, etc. Instead, Javascript groups all numerical value types as a `Number` type. This was not very problematic as there was only one case where we needed special handling: integer division. To implement proper integer division, we simply checked whether or not the first expression of a binary expression was an integer type (`int` or `rune`). We could do this since GoLite only allows division between the same types (i.e. we could not divide an integer by a float). If that first expression was indeed an integer type, then we would wrap the division with the function `Math.trunc()`. Thus, the expression in Go `3/4` would convert to `Math.trunc(3/4)`.

For division assignments, we would add a statement underneath that would truncate the variable.

```
// in GoLite
var a int = 3
a /= 4

// converts to
let a = 3;
a /= 4;
a = Math.trunc(a);
```

7.2.5 Strings & Runes

For strings and runes, we use a Javascript String. There were a few considerations.

- In Javascript, all the strings used are interpreted by nature. This naturally created problem when attempting to translate escaped characters from Go to Javascript. More specifically, escaped characters that did not exist in Javascript, the first strategy was to use a .replace string function in Javascript for every string and let run-time take care of problem. This, however, did not work due to the interpreted nature of the strings in Javascript. To solve this problem, this had to be done in the codeGen c code at compiler time.
- To correctly use Go runes in Javascript expressions we had to use the string build in function for string `charCodeAt`
- For raw strings we use ES6 Javascript function `String.raw‘‘`

7.3 Assignments

To handle assignments in Javascript, there were various considerations to take into account.

7.3.1 Multiple Assignments and Swapping

```
a,b := b,a
    var a,b = a,b
```

To handle the swapping problem in multiple assignments, we first had to create temporary variables to make sure that the semantics of swapping were maintained. A temporary variable was created for every value, even literals. A future improvement of our compiler could look at creating temp variables only when necessary.

7.3.2 Short Declarations

```
var a int
a,b := 1,2
```

When implementing the case above in Javascript, since Javascript does not have the short variable declaration semantics, it was necessary to check whether a variable was already defined in scope for a short declaration, otherwise we would have obtained two let statements with the same variable name, which is an error in Javascript.

```
let _a = 0
    var temp_a = 1
    var temp_b = 2
    let _a = 1
    let _b = 2
```

To get around this, the symbol work in scoping was leveraged, a field is_defined_in_scope was added to the node of the AST, this field was later initialized by the Symbol table, and finally used in code generation to replace the above code to:

```
    let _a = 0
    var temp_a = 1
    var temp_b = 2
    _a = 1
    let _b = 2
```

7.3.3 Short Assignments

For short assignments, the translation to Javascript was equivalent except for the bit clear &∧ = short assignment. For this the following represents the Javascript translation:

```
\\GO
a&^=b
\\Javascript
temp_b &= (a ^ temp_b)
```

7.3.4 Different Variables With the Same Name

There are cases with nested scopes where a variable has the same name as another variable in the outer scope. For example, consider the following Go code segment: This code segment is handled as expected in Go. However, in Javascript, an error is thrown because `a` is used before its declaration in that scope. To solve this issue, we decided to keep a count of the variables with the same name, as we travelled down a path in the cactus stack of symbol tables. We then appended this count to the variable name in Javascript, thus ensuring that no newly declared variable would have the same name as another variable in an outer scope. Thus, the previous code segment would convert to the following in Javascript, with some small changes for clarity:

```
let a_0 = 98;
{
    console.log(a_0) // 98
    let a_1 = a_0;
    console.log(a_1) // 98
}
```

These count values were attached to the identifier symbols.

7.4 Run time errors

There are few runtime errors that were handled in Go which we had to handle in the Javascript source. The first one is division by zero which is disallowed in Go but not in Javascript. To handle this, we simply checked every division operation at runtime and determined whether or not the divisor was equal to 0. If so, then the code would throw an error. The second runtime error we

had to check was accessing an index in an array that is out of bounds. To do this, we created a function in Javascript that was passed the array and the index, determine whether or not the index was out of bounds, and throw an error if so. This function was called for every array or slice access expression.

7.5 For Loop Post Statements

The syntax for `for` loops in Go and in Javascript are very similar, both accepting an initializing statement, a terminating condition, and a post statement. However, because Go could accept a multivariable assignment in the post statement, we could not simply translate the post statement from Go to Javascript as the multivariable assignment was broken up into multiple assignments in Javascript, which would be disallowed in a `for` loop post statement. Our initial approach was to simply append the post statement at the end of the loop's block in our generated code. However, this approach failed whenever a `continue` statement was present. The problem was that as `continue` skips to the next iteration, it still calls the post statement. By appending the post statement at the end of the block, it would not be called, which is problematic. Conveniently, Javascript allows for functions to be defined anywhere. Since it also allows function calls as post statements, we simply defined a self-calling anonymous function in the `for` loop's post statement that would run the contents of the original post statement.

7.6 Binary & Unary Expressions

Binary and Unary expressions were almost identical in Javascript except for two cases: The first case was unary XOR/complement, for which Javascript provided `~`. The second case was the bit clear binary operation, for which the translation in Javascript was:

```
leftexp&(leftexp^rightexp)
```

7.7 Testing

To test the Javascript code generated by our compiler, we needed a more efficient and reliable way than to manually inspect the code and ensure that it was correct. We decided that we would use print statements and compare the output of our generated programs, against the output of the original Go program when ran with the Go compiler. To automate this process, we created a script called `testcodegen.sh` which would take as input a Go program, pass that Go program to our compiler which would generate Javascript code, run the original Go program with the Go compiler, run the generated program with the NodeJS interpreter, and compare the content of the standard outputs. If both outputs were the same (or if both programs threw errors), this would be considered to be a passing test, otherwise it would be a failing test. With this testing strategy, we had to ensure that our test programs would print whatever state we were testing for rather than assert it. In the end, we tested on 31 programs, one of which consisted of 550 lines of code.

8 Conclusion

We have completed a compiler that generates valid Javascript code for all of the GoLite constructs given the specifications. Since our last code submission, we discovered that our code generation for short assignments contained a bug, thus we fixed that along with a small performance issue. In fact, we went through every test case that was used for grading and fixed every issue.

For the code generation phase, we are very pleased with our decision to go with Javascript. In general, most translations from GoLite were easily made and for the edge cases, Javascript provided some constructs, such as closures, that made it easy to implement a workaround. Javascript's flexibility also allowed us to omit many details that would have been cumbersome to deal with in other languages.

With such a large project, we realized that decisions made during the early stages often had the most impact on the rest of the project. Unfortunately, since this is the first full compiler that we wrote, those were the stages during which we had the least knowledge. However, the decisions we made turned out mostly to be the right ones as there was no situation where a newly discovered issue caused us to rewrite a major part of the codebase, nor any other major hindrances. One of the most important decisions in those early stages is the design of the AST. We realized that a

well-designed AST can save a lot of work in the long run since it is the main data structure that we work with.

One of our main areas of focus as we were implementing the code was testing. From the start of the project, we wrote hundreds of tests which proved fruitful since we discovered many bugs through them. However, we still didn't cover all the test cases used during grading. This shows that you can never test too much.

In summary, this was a very large project that necessitated a lot of hard work but we are very proud of our endproduct. Other than some minor details in the implementation, there is not much that we would have changed. Playing around with Go and GoLite allowed us to learn a lot about language semantics and their nuances. It would be interesting to implement more complex constructs of the Go language such as function types. Additionally, the large scope of this project not only allowed us to learn about compilers, but also about project management and working well as a team.

References

[1] O. Miller. *Finding Comments in Source Code Using Regular Expressions*, 2010.