McGill University

COMP 520 - Compiler Design

Winter 2018

GoLite

Milestone 3 - Group 02

Name: Jérémie Poisson
McGill ID: 260627104
E-Mail: jeremie.poisson@mail.mcgill.ca

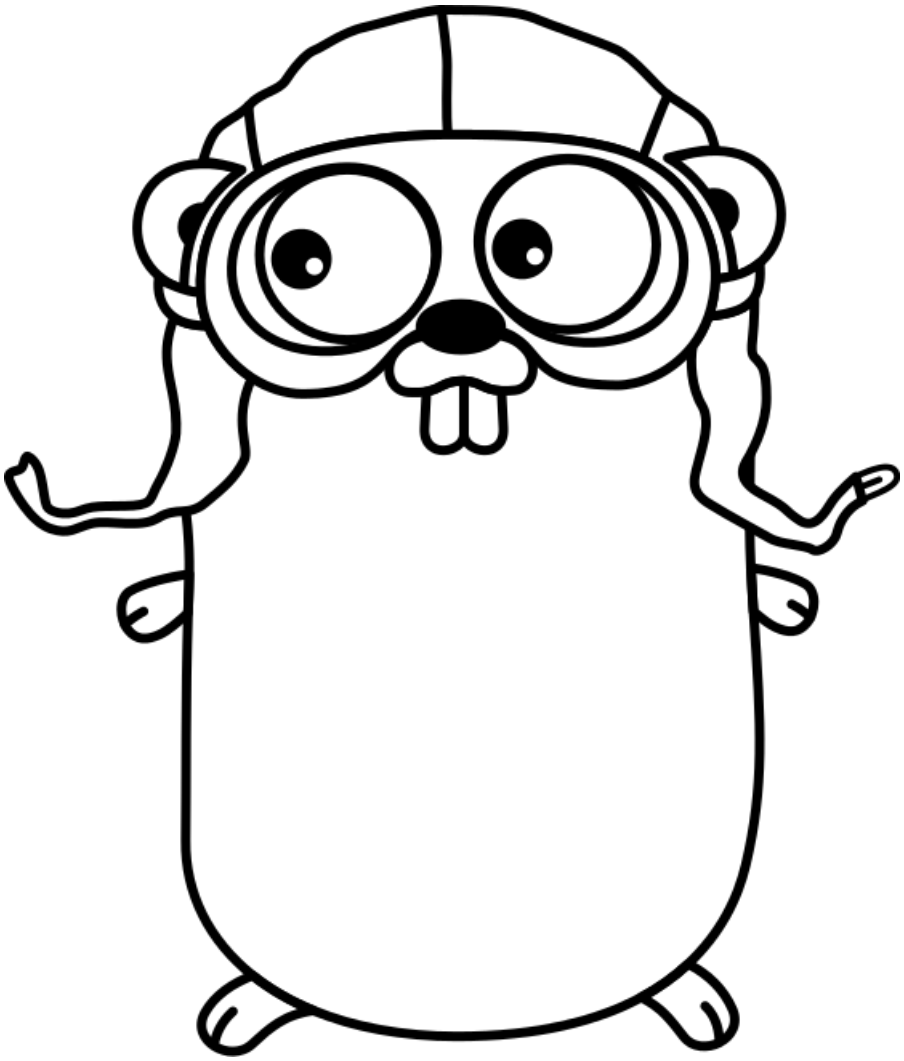
Name: Amir El Bawab
McGill ID: 260645260
E-Mail: amir.elbawab@mail.mcgill.ca

March 28, 2018

Contents

1 Introduction

In this report, we continue discussing the GoLite language, more specifically the design decisions we took for the Code Generation phase of our compiler. Milestone 2 focused on the symbol table and type checking implementation of our GoLite compiler implementation. From the feedback we received in milestone 1 and 2, we fixed the remaining issues in our compiler. We are fairly confident at this point to have solid compiler which correctly parse and type check a given GoLite program as input.



2 General considerations

There are a few considerations we needed to consider before choosing an actual target language for the code generation phase of the GoLite compiler. The choice of the target language for our compiler has been carefully chosen from these considerations. Below is the list of considerations we explored as we chose a target language.

1. First of all, as per the specifications, the GoLite language is a statically typed language. Thus selecting a target language that is also statically typed could potentially be an additional source of confidence that the GoLite semantics and the generated code semantics align.

2. Secondly, the GoLite language would be considered a fairly high-level language. There are a lot of built-in functions that manipulate basic types such as strings, numerical values, arrays and data structures out of the box. Hence having a target language expressive enough and containing a lot of built-in functions for manipulating basic language constructs would be a time-saver for us.

3. Thirdly, the GoLite language allows for a lot of user-defined structures. This means that one may define its own abstractions within his/her program which would make it difficult for a low-level target language to handle. Hence having a target language that is flexible enough in supporting user-defined data structures and types would save us a lot of time.

4. Finally, because we will probably have to test the generated code a lot, we think it is crucial for us to have a target language that is easily debuggable and easy to execute.

3 Target language

As the target language for the code generation phase of our compiler, we chose TypeScript. It is an open-source programming language developped by Microsoft which is a strict syntactical superset of JavaScript. In our case, we run the generated program using NodeJS for program execution.

TypeScript provides several features that will greatly help us for the code generation phase of our GoLite compiler. Below is a short list of TypeScript-specific features:

- Is fully functional with JavaScript.
- Optional static typing (compile-time type checking)
- Classes and modules
- Type inference
- Type alias
- Interfaces
- Generic
- Tuples
- Dynamic Arrays

In terms of our general considerations, we think TypeScript is an excellent choice.

1. First, TypeScript allows for static type annotations. This means that we do not have to rely on the dynamic typing nature of JavaScript and we can have a one-to-one mapping of the types declared within the GoLite language and the generated TypeScript type.

2. Seconly, JavaScript, by nature, is a fairly high-level language and has a lot of built-in functions manipulating basic types and data structures. TypeScript also expands the range of built-in functions available to us, so we do not have to worry for writing trivial operations such as string operations, array manipulations, etc.

3. Thirdly, TypeScript (and JavaScript) allows for user-defined classes, interfaces and types along with the object-oriented programming paradigm. It also bundles the notion of generics. We can define new types explicitely within the TypeScript program and we do not have to rely on a workaround for user-defined types. TypeScript even allows for type aliasing out-of-the-box.

4. Finally, as mentionned earlier, TypeScript is a strict superset language of JavaScript. When compiled, TypeScript generates JavaScript code which can be executed on any JavaScript interpreter available. In our case, we use NodeJS which runs the Google V8 JavaScript engine.

We believe TypeScript is a solid choice as our target compiler. It fullfils every criterias we first considered for a target language. Also, being maintained by Microsoft, it has exhaustive documentation available online and is stable.

The only drawback of using TypeScript in this project, is the performance of the language compared to other ones. TypeScript compiles to JavaScript, which is a relatively slow language.

4 Semantic challenges

This section describes 4 key areas of the semantic of the Go language, and how they were represented in the target language.

4.1 Scoping rules

Scoping rules within JavaScript are especially complex. We had to be creative in order for us to simulate the exact scoping rule exhebited by the GoLite specifications within a JavaScript enviornment.

Consider the following GoLite program:

```
package main
var x int = 3
func main() {
   println(x) // 3
   var x int = x
   println(x) // 3
}
```

The expected behavior is for the variable x within the function scope to be redefined with the value of x within the outer-scope.

In JavaScript, due to its scoping rules, the following program would yield two undefined values. This is because on can use a variable independently of the order of its declaration. Here, because there is a variable declaration for x is within the scope, any reference to x references the variable declared within the scope (and not the outer-scope, which would be the expected behavior of the GoLite program).

```
var x = 3
function main() {
   alert(x) // undefined
   var x = x
   alert(x) // undefined
}
main()
```

To solve this issue, we had to rely on renaming every variable in the generated code. To ensure unicity of the mangled variable names, we use the scope of a given statement to uniquely identify the variable name. For some statements like if, for, switch and block, a unique counter is also added to the name in order to avoid name collision between sibling scopes. Following is an example highlighting the solution for scoping rules in TypeScript.

```
var prog_x = 3
function main() {
   alert(prog_x) // 3
   var prog_func_main_x = prog_x
   alert(prog_func_main_x) // 3
}
main()
```

4.2 Types

4.2.1 Base type

Both GoLite and TypeScript have built-in types recognized by their corresponding compilers. Below is a table showing the one to one mapping of the base types between those two languages.

GoLite	TypeScript
int	number
float64	number
rune	number
bool	boolean
string	string

In this table, it is important to note that rune are treated as numbers in TypeScript because there is a no character type in TypeScript and number is the closest choice as each character is represented by its ASCII code.

4.2.2 Type declaration

In GoLite a user can define type aliases to rename complex types. This concept is identical in TypeScript and has a similar syntax which make it easy to generate code for types declarations. Here are some examples of code written in GoLite and their TypeScript equivalent code.

```
type natural int // GoLite
type natural = number // TypeScript

type numbers []int // GoLite
type numbers = Slice<number> // TypeScript

type ten_numbers [10]int // GoLite
type ten_numbers = Array<number> // TypeScript
```

Slice, Array and Struct will be discussed in the final report.

4.3 For-loops

GoLite for loops in all its possible forms have been converted into while loops in TypeScript. 3-parts for loops are converted into while loop where the initial statements are placed right before the while statements, the post statement just before closing the while body scope and the for condition at the while expression. Here are some examples showing how the different forms of for loops are converted to TypeScript.

```
for {} // GoLite
while(true){} // TypeScript

for expr {} // GoLite
while(expr) {} // TypeScript

for a, b := 1, "Hi"; a < 10; a++ {} // GoLite
// TypeScript
var a : number;
var b : string;
[a, b] = [1, "Hi"];
while(a < 10) {
    a++;
}
```

4.4 Switch statements

Switch statements in GoLite can omit the statement condition, a feature not very common among other programming languages. Because switch statements in all its forms must be handled, GoLite switch statements are converted into semantically equivalent TypeScript while-loop with blocks of if statements representing the switch cases.

```
// GoLite
switch expr {
    case 1, 2:
        // code
    default:
        // code
}

// TypeScript
while(true) {
    if(expr == 1 || expr == 2) {
        // code
        break
    }
    if(true) {
        // code
        break
    }
    break
}
```

5 Code generation status

Go and TypeScript have a lot in common. The only major difference is that Go "stores" objects in variables whereas TypeScript stores reference to objects in variables. Since both languages copy values on assignment, return, passing arguments, etc..., Go would copy the object whereas TypeScript would copy the reference to the object. The consequences of that difference will be discussed in the final report.

Currently, our compiler is capable of generating code for all valid GoLite programs, and the generated TypeScript code simulates an identical execution process as Go for the majority of cases. Features has been incrementally implemented with

no major issues nor workaround since both languages, as mentioned previously, share a lot of similarities. The only known issue, as per the date of writing this report, is representing floating points on the console similarly as the Go language.

To ensure that the generated TypeScript code works the exact same way as Go, further tests programs will be written covering various cases.