McGill University

COMP 520 - Compiler Design

Winter 2018

GoLite

Final Report - 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

April 17, 2018

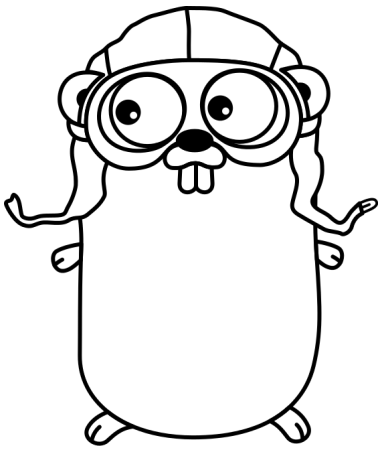
Contents

1 Introduction

The goal of this project was for our team to develop our first fully functional compiler for the GoLite language. GoLite is a programming language that is a subset of Golang. It covers the necessary syntax to build a meaningful program in GoLite syntax while being simplistic enough to be easily understandable by program designers. Our team choose TypeScript as a target language which is an open-source programming language developped by Microsoft which is a strict syntactical superset of JavaScript. In the end, our compiler takes in some GoLite program as input and outputs a TypeScript program, which can be compiled to JavaScript and be executed in a NodeJS environment.

The project was released in several stages, each of which were delivered in separate milestones. The structure of this report is based on the previous milestones and presents the overall finished project of our compiler. Readers eager to know more about the specific milestones is invited to read the reports we published throughout the milestones 1, 2 and 3.

Firstly, the report will go over the considerations for our target language and the tools we used throughout the project. The major components such as the scanner, the parser, the weeder, the symbol table, the typechecker and the code generator of the compiler will be discussed in depth in subsequent sections. To finish of, we will have a brief overview of our first experience designing compiler and we will highlight the different member contributions to this project at the end.



2 Languages

2.1 Go language

Go is a statically typed programming language created by Google. It is recognizably on pair with the C language. It syntax is aimed at improving brevity, simplicity and safety and borrows several language features from the new highly dynamic languages. Some of the novel features include:

- type inference through variable initialization and short variable declarations;
- built-in concurrency features;
- built-in dynamic array support;
- interface system and remote package management;
- fast compilation time;
- generation of native, statically linked binaries;
- and much more.

2.2 GoLite language

As mentionned earlier, GoLite is a subset of the official Go language. It provides enough language constructs to create interesting programs. GoLite was chosen for this project because of its simplicy and relatively easy learning curve. Most importantly, the full documentation of the Go language is exhaustive and the language is fairly popular online. It thus was easy for us to dive into the documentation and look around for support if needed.

Below are the main language features supported by GoLite:

- **Types**: some of the built-in types in Go (int, float64, bool, rune, string);
- **Declarative statements**: variable declarations (with shorthand declarations), function declarations, type declarations (with type references);
- **Imperative statements**: loop blocks (for loops) with break and continue, if-else blocks (with else if), switch blocks;
- **Expressions**: literals, identifiers (with blank identifiers), all of the unary expressions, all of the binary expressions, function calls, casting, array indexing;
- **Slices**: are the dynamic arrays that are built into the Go programming language. Content is accessed using array indices and populated using the append function;
- **Built-in functions**: print(), println(), append();
- **Optional semicolon**: semicolon may or may not terminate statements in Go.

2.3 TypeScript as a 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 greatly helped 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
- Type inference
- Type alias
- Interfaces
- Generic
- Dynamic Arrays

There are a few items we considered for actual target language for the code generation phase of the GoLite compiler. The choice of the target language for our GoLite compiler has been carefully chosen under 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. In TypeScript, static typing is fully optional and is left as a design decision. One may want to specify static types in TypeScript to better estimate the execution behavior. Although not necessary, using static typing could potentially help debugging the semantic used for the generated code.

2. Secondly, the GoLite language is a fairly high-level language. It provides a lot of built-in functions and operations manipulating basic types such as strings, numerical values, arrays and data structures out of the box. It also comes in with basic type conversions as well. We choose TypeScript because it is expressive enough for our needs and contains a lot of built-in functions for manipulating basic language constructs which was a good 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 correctly. The TypeScript language is flexible enough in supporting user-defined data structures: it provides classes which is used to represend user-defined structures, and even comes with built-in type referencing.

4. Finally, because we had to test the generated code a lot, it was crucial for us to have a target language that is easily debuggable and easy to execute. TypeScript was verbose enough and gave us access to various JavaScript-based debbuging tools for us to debug the generated programs more efficiently. For debug purposes, we used the TypeScript Playground and the Chrome JavaScript Debugger.

3 Tools used

Throughout this project, we used several tools which made our life easier.

The compiler was completely written C++. We chose C++ because it provides a good compromise between abstracting complicated data structures through object-oriented design while still being low-level enough to manipulate the complex data structures of our compiler efficiently. It was also a great choice because both members were fairly comfortable using this language.

For code synchronization our primary tool of choice was obviously Github. Every issue encountered was documented through the Github issue tracking system. We also used Github for releases and keeping track of milestones.

For compiling and building the compiler we used CMake, a building toolkit for compiling and linking the binaries. CMake is now the defacto tool choice for new IDEs such as CLion from JetBrains.

There are several open source options available for scanning text input and converting it into tokens. In this project we used Flex for that purpose because it is written in C/C++, the programming language of choice for our team. The flex library uses regular expressions and deterministic finite automata (DFA) to identify sequence of characters and converting them into their corresponding tokens. Both regular expression and DFA, were used throughout our project. For syntactical analysis, our team decided to use GNU Bison as the GoLite language parser as it works in harmony with Flex scanner.

4 Scanner

4.1 Overview

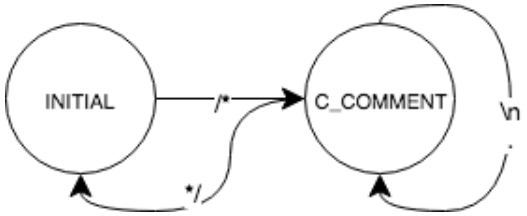
The majority of GoLite syntax is defined using regular expressions. Reserved keywords, operators, literals, white-spaces, new lines, inline comments are all examples of syntax presented to Flex using regular expressions. The only exception where regular expression was not used, but could have been used, are block comments and strings. The latter were chosen to be implemented using a DFA because it is simple to understand and maintain compared their equivalent regular expression representation.

4.2 Major design desicions

Some features were better handled using Deterministic Finite Automatia (DFAs). As mentionned earlier, it is possible to construct a DFA directly in a Flex file. In fact, building DFA manually turned out to be useful in our project for identifying block comments and strings.

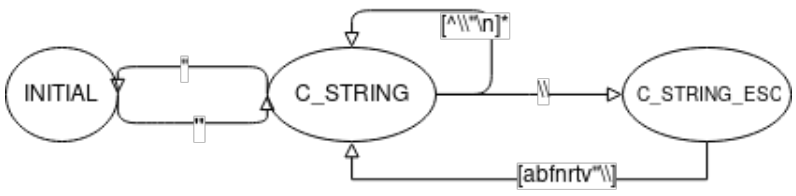
4.2.1 Block Comment DFA

```
"/*"              { BEGIN(C_COMMENT); /*Go to C_COMMENT state*/ }
<C_COMMENT>.      { /* Comment can contain any char */ }
<C_COMMENT>\n     { /* Comment can contains new lines */ }
<C_COMMENT>"*/"   { BEGIN(INITIAL); /* Go back to initial state */ }
```



4.2.2 String DFA

```
<INITIAL>\"  {  /*...*/   BEGIN(C_STRING); }
<C_STRING>[^\\"\n]*  {  /*...*/  }
<C_STRING>\\  {  BEGIN(C_STRING_ESC); }
<C_STRING>\"  {  /*...*/  BEGIN(INITIAL); }
<C_STRING>\n  {  /*...*/  }
<C_STRING_ESC>[abfnrtv"\\]  {  /*...*/  BEGIN(C_STRING); }
<C_STRING_ESC>\n  {/*...*/  }
<C_STRING_ESC>.  {  /*...*/  }
```



4.2.3 Optional semicolon

GoLite allows users to optionally apply semicolons at the end of their statements. Achieving this behavior requires manipulation of the list of tokens generated by Flex where semicolon are auto-injected before new lines and after some predefined set of tokens. The latter are determined by Golang under the first rule of the optional semicolon section: https://golang.org/ref/spec#Semicolons. Several methods can be used for injecting tokens follow a set of rules, in this project optional semicolon concept is implemented by keeping track of the last token before a new line. If the last token is part of the predefined characters, then a semicolon is injected. It is important to note that the grammar has to align with those rules as well. In other words, when the parser analyzes the tokens, it would not accept optional semicolons, in fact they are mandatory.

4.2.4 Integers in different forms

In Golite, integers can be represented in a non-decimal form. A sequence of hex or octal integers are immediately stored into their corresponding decimal form, hence all the successive phases would only carry decimal representation of integers.

4.3 Testing

For testing the scanner, we wrote a test set of both lexically valid and invalid programs. Our testing focused on testing the individual language constructs of the GoLite language with sole goal of testing whether the GoLite syntax is write. Test programs at this stage only test whether the keywords used throughout the program are valid.

5 Parser

5.1 Overview

The role of the parser is to take a stream of tokens and build-up the abstract syntax tree (AST) correctly. The AST is the object representation of the program and is constructed using the GNU Bison parser toolkit. In this section we explore how we designed the main classes and how they model the problem of representing a program in a abstract syntax tree form. We also go over the grammar conflict issue we had to overcome when designing GoLite grammar.

5.2 Major design decisions

5.2.1 Abstract Syntax Tree (AST)

In this section we highlight the main components composing the abstract syntax tree of a parsed GoLite program. We also highlight the main design decisions we made and their impact on the finished compiler. Interested readers are invited to go over the final UML diagram, presented in a later section, describing the relationship between the different classes of the compiler in a graphical manner.

Program class: represent the whole program at a global level. It is designed as a singleton, which makes much sense as GoLite is only concerned with a single program per file. As per the program specification, a program is made of a collection of declarables (no statements are allowed at the program-level). For example, a simple program containing a solely a main function will only contain a declarable for the main function. Program also contains static references to built-in types that are built into the language itself. Those built-in features are declared statically by the compiler itself, hence any reference by the user to those types are the same and lie at the root level of the program.

Declarable and related classes (pure virtual): represent every declarable statement that can be made within the language. In GoLite, one can declare types,

functions and variables. When such declarable is parsed, a *Declarable* object is created and is later referenced throughout the program AST, as needed.

Statement and related classes (pure virtual): represent every general statement within the program that do not require memory allocation to be later referenced in the GoLite program execution. Those constitute everything from variable-expression assignments to imperative statements proper to the language. Each GoLite statement has its own class inheriting from *Statement* to express the *is* relationship between the two classes. For instance, built-in functions such as *println*(...) and *print*(...) are modeled directly as statement classes as they are reserved keywords as per the language specifications.

Expression and related classes (pure virtual): are a special computational statement which are used to return a certain result. In GoLang, expressions are generally either Unary (acting on a single sub-expression), Binary (acting on two sub-expressions, usually a left hand side and a right hand side) or Primary which represent the base case for the recursive definition of both Unary and Binary expressions. Those comprises the basic arithmetic operations such as addition, multiplication and so on. A special case of expressions are what we call "Primary expression". Those are expressions composed of Primary elements. As part of Primary elements, one can find identifiers, function calls (along with its set of passed arguments), array-like structure access (along with the specified index), literals (such as plain integers, strings, etc). Like the *println*(...) and *print*(...) functions, the *append*(...) function is modeled directly as a separate expression sub classes. This is because it is a reserved keyword that cannot be overridden by users but also returns a value.

5.2.2 Conflicts

In Left-to-right Rightmost (LR) parsers, there are two types of conflicts: Shift/Reduce and Reduce/Reduce. Shift/Reduce conflicts were not encountered during the development of the grammar as the language forces the user to use delimiters (e.g. {, (, []), between tokens making it clear for the bottom-up parser what its decision must be. Reduce/Reduce conflicts were more common. The two cases were Reduce/Reduce conflicts appeared were, first when two productions are identical and the parser would be ambiguous to which one it should reduce to, and second when a production represents a particular derivation of a more generic production.

Example of the first type of conflict is presented below.

```
function_args
    : expressions
    | %empty
    ;

expressions_opt
    : expressions
    | %empty
    ;
```

In this example, both productions are identical, but they were initially developed in parallel as their names were descriptive when used in productions. This conflict was solved by removing *function_args* production and using *expressions_opt* instead where applicable.

Following is an example of the second type type of conflict.

```
simple_statement
    : expressions tEQUAL expressions
    | ...
    | identifiers tDECLARATION expressions
    ;

expressions
    : tCOMMA expression
    | expression
    ;

identifiers
    : tCOMMA tIDENTIFIER
    | tIDENTIFIER
    ;
```

The problem in the last example is that *identifiers* production can be generated from the *expressions* production, the former is a particular derivation of the latter. This ambiguity may result in the grammar generating *expressions tDECLARATION expressions* which cannot be reduced to *simple_statement*. The conflict is less clear to detect and even the parser would not detect it at compile time, in fact the runtime behavior would arbitrary select a production (usually the first one to occur). Unlike the first conflict, deleting a production does not completely resolve the issue. In fact, the solution applied in this project allows the input to form an illegal syntax, and later in the code a weeding pass is performed to verify the correct usage of the language specifications. So in the example above, the *simple_statement* production has been modified as follow:

```
simple_statement
    : expressions tEQUAL expressions
    | ...
    | expressions tDECLARATION expressions
    ;
```

5.2.3 Weeding passes

Some parsing decision which could have been implemented in grammar, were instead done in a weeding pass. Weeding pass in parser phase includes checking for syntactically incorrect usage of blank identifier, verifying that the number of elements on left and right side of all assignment forms match and finally checking if *break* and *continue* were used outside a *for* loop or a *switch* statement in case of a *break*.

5.3 Testing

For the scope of the parser we designed GoLite programs that tested the ability of our compiler to parse and scan both valid and invalid programs. GoLite is a flexible language and supports a lot of custom types and structures. Our methodology for designing such tests is mainly based with our exprience of developing the parsing phase of the compiler and the many intricacies of the GoLite language. For both valid and invalid programs, we established about 4-5 programs for each component of the language. About half of those programs are testing basic functionnalities (right syntax, usual use case, etc.) whereas the other half focuses on testing edge cases (unusual syntax, nested structures, etc.). Some of those tests also integrate multiple components toghether and verifies their compatibility as per the official specifications of the GoLite language.

6 Symbol Table

6.1 Overview

Our symbol table implementation is similar to the one we have seen in class for any modern compiler. In a nutshell, a symbol table is a data structure used by the compiler mapping each identifier used within the GoLite source code to information of the node object instance within the syntax tree.

More specifically to our case, it made much sense to have identifiers used throughout the program mapped to an instance of a *Declarable* (used for functions, variables and types) class. As a reminder, a GoLite program, as we designed it, is simply a singleton containing a list of *Declarable*. Thus, in the symbol table implementation, we simply had to map a plain string identifier to a *Declarable* object instance. Other components using an already declared identifier could easily retrieve the node referenced by this identifier using a lookup of the symbol table.

To populate the *SymbolTable* we added a virtual pure (abstract) method *void symbolTablePass(SymbolTable* root)* to all subclasses of the *Declarable* and *Statement* classes. This allowed us to specify the symbol table pass implementation for every components of the GoLite language separately, which is better for future maintainability.

The *Program* singleton object has a reference to the root *SymbolTable* of the program which gets initialized before parsing the program and gets populated by traversing the abstract syntax tree after calling its subsequent *symbolTablePass()* methods on every *Declarable*. In their turns, *Declarable Function*s call *symbolTablePass()* on all its *Statement*s.

The rest of this section is dedicated to the implementation and the challenges we faced as we implemented the symbol table pass within the major components of the by GoLite language. We also explore how a few edge cases are handled.

6.2 Major design decisions

6.2.1 Built-in identifiers

One key aspect of our implementation is that the root *SymbolTable* is initialized with the built-in identifiers. Those built-in object instances remain constant and are declared statically in the code base before being pushed into the root symbol table. Built-in identifiers represent any built-in language constructs such as basic types, boolean values, built-in functions and much more. That is, built-in types and variables provided by the GoLite language are being artificially pushed into the root symbol table of any GoLite program when compiling. Considering our implementation, it is crucial to have those injected at the very root of the symbol table. This means that one can potentially overshadow those values, which is permitted throughout the GoLite language.

6.2.2 Scoping rules

To account for the various scoping rules of the GoLite language, a *SymbolTable* object can also have other *SymbolTable* as children. Upon looking up a symbol, the *SymbolTable* might have to check into its parent (if any) for the definition of any identifier previously declared within the outer scope of the current symbol table. When entering a function for example, we push a newly created blank *SymbolTable* as a children to the root symbol table and use it as the new "root" for objects within the same scope.

6.2.3 Types

GoLite allows for user-defined types. This means that in a given GoLite program, an identifier may be reserved to indicate a custom type that was previously defined by the GoLite program. When a custom type declaration is encountered, we create a symbol entry mapping the identifier and the *Type* object instance in the current *SymbolTable*.

One caveat of custom type declaration is that one cannot redefine a declaration with the same name within the same scope. However, one can shadow a declaration that was defined within an outer-scope (read in a parent symbol table). To allow shadowing of types in the outer scopes, we simply check if the type declaration has been be previously defined only in the current symbol table, ignoring the parent one(s).

A second issue arises with recursive type definitions. In GoLite, it is possible to define a new type that references an already defined type. The newly defined type thus becomes an alias for the old type. For example, one could declare a new type "integer" referencing the built-in type "int". GoLite does not allow recursive type definitions (a type that references itself), except for the case where the recursive definition contains a slice. To solve this, we simply added the necessary checks to verify that the referenced type is not recursive and not a slice.

6.2.4 Functions

Overviously, GoLite allows for user-defined functions. This means that an identifier can reference a user-defined *Function* within the same GoLite program. Upon reaching a function definition in the symbol table pass, we append the current symbol table root with the *Function* declarable object instance and push a new *SymbolTable* instance as a child. This new symbol table contains the functions parameters as new *Variable* definitions. We then call the symbol table pass on the statements contained within the function body with the symbol table as a root. This has the effect of isolating the scope of the function: parameters and variable declarations within the function body are only available within its scope while the identifiers can also reference previously defined *Declarable* within the outer-scope by looking up into the parent symbol table(s).

As per the specification, it is not possible for one to re-define a function at the same scoping level. Our implementation simply checks whether a function exists at the same scoping level and report an error in that case.

Special functions such as *main* and *init* are not allowed to have parameters nor any return type. The latter conditions were implemented in the *symbolTablePass()* by checking if the *Function* name matches any of the two special functions manually. Furthermore, since the *init()* function can be declared several times under the same conditions, a special unique token has been generated for it in the symbol table to avoid name collisions. Consequently, having a special token representing the *init()* function definition in the symbol table, makes it impossible for one to refer to it as the look up would report a miss, which is the behavior expected by the GoLite specifications.

6.2.5 Variables

GoLite allows user-defined variables. This means that an identifier can also reference a user-defined *Variable* declarable. Also, variable declarations can be tricky as they can declare multiple variables at the same time. Luckily, our design manages the variable declarations as a list of identifiers and a list of expressions: we do not have to handle those cases separately. The symbol table pass for variable declaration is fairly straightforward. First, we recursively execute the symbol table pass on the referenced expressions. Secondly, we append the root *SymbolTable* with every identifiers and the *Variable* declarable instance.

The only edge case with variable declaration is that one cannot redefine a declaration within the same scope, but can, in fact, redefine a declaration that was declared within an outer-scope. To fix this, we simply do not check for a identifier within the parent symbol table.

*Declaration*s are essentially a shorthand syntax that bundles a new *Variable* declaration with an assignment together in the same statement. In our implementation, an identifier is mapped to a new *Variable* definition object instantiated by the *Declaration* object at the symbol table pass. This allowed us to re-use the symbol table pass implementation of the *Variable* class.

6.3 Testing

Mainly, we focused our testing effort so as to test every language construct individually. Because GoLite allows a lot of user-defined structure (types, variables, functions) we also made sure to test both user-defined and built-in symbols for every tested component. We also created tests that combined multiple components together and explored many edge cases allowed in the GoLite programming language. Roughly half of the programs are meant to test the usual basic functionalities established in the milestone specifications whereas the other half covers many advanced situations. Some of those tests also verify the compatibility of various constructs as per the official GoLite language specifications.

7 Typechecker

7.1 Overview

In this project, type checking is performed on the program right after the symbol table pass is completed. Making two separate phases also allowed for greater maintainability for future releases of this project.

The GoLite specification provides detailed instructions on what each kind of expression should be evaluated to during the type checking phase. Specifications also provides the semantics for type checking in detail, which eliminates a lot of room for underlying errors. For instance binary expressions require both operand to be checked, then a return type is inferred as a conclusion for the operation. Similarly, unary expression requires specific types which would equally evaluate to a type as a result.

7.2 Major design decisions

7.2.1 Expressions evaluation

Expressions evaluated together are required to share the same type as opposed to resolve to the same type. A key aspect of our type checking mechanism is that compatibility of two types is performed in GoLite by comparing pointers as opposed to comparing names. That is, both types must point to the same element in memory (except for struct). This approach allows us to manage types as some generic class instance which makes for easier debugging and maintainability for future releases.

7.2.2 Built-in types

As mentioned earlier, built-in *Type* object instances are artificially injected in the root symbol table. They essentially have a self reference in the program, which is usually not allowed for user defined types as it would lead to an invalid recursive type declaration. Every user-defined types shall resolve (with possibly slice and/or array) to a built-in type eventually except for the case of the recursive exception discussed previously, and structs.

7.2.3 Struct types

As per the GoLite specification, struct type comparison is performed as follows. The structs fields are compared against each other. The order also has an implication on the compatibility of structural types. If one field does not match these criterion, it must be that types are not compatible with each other.

7.2.4 Weeding passes

Sometimes, an additional weeding pass is required to check some type checking conditions. This weeding pass is executing during the type checking phase of the compiler.

Terminating statement checking is a good example of an operation performed on the additional weeding pass. After evaluating the function body, a weeding pass takes care of making sure that the last statement in the function is a terminating statement, if the function has a return type. This is because the return type of a function can only be inferred after the type checking phase. This means we cannot perform such operation right from the initial weeding pass.

Furthermore, some terminating statement require that the body of the statement does not have a break statement. Such checks are also done using a weeding pass.

Finally, in order to verify that assignment operations does not have constant variables on their left hand side, we have to apply an additional weeding pass to scan the variables and investigate their symbol table entry if they are defined as constants or not.

7.3 Testing

Most test cases were implemented while reading the language specifications about the type checking phase. Those test cases were named according to their corresponding section in the document.

8 Code generator

8.1 Overview

As mentioned earlier, we chose TypeScript to be the target language of our compiler. In this section, we explore the implementation of the code generation mechanism of our compiler.

8.2 Major design decision

8.2.1 Object referencing

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.

To replicate the semantics of GoLite in TypeScript, we generate the appropriate TypeScript code to copy everything that is passed either by function calls and assignments.

To do this, each generated structure class in TypeScript comes with a *clone()* method, allowing us to create a deep copy of a given object. Hence, in a function call for example we simply have to call the *clone()* object method to simulate pass-by-copy.

```
// golite
package main

type teststruct struct {
   age int
}

func test(a teststruct) {
    a.age = 10
}

func main() {
    var aa teststruct
    aa.age = 9
    test(aa)
}
```

```
// generated struct code
class struct_1 {
    age : number = 0;
    clone = () : struct_1 => {
        var obj : struct_1 = new struct_1();
        obj.age = this.age;
        return obj;
    }
    equals = (obj : struct_1) : boolean => {
        return this.age === obj.age;
    }
};
//...
function golite_prog_main() : void {
    var golite_prog_func_main_aa : golite_prog_teststruct = new struct_1();
    golite_prog_func_main_aa.age = 9;
    var expr_1 : void = golite_prog_test(golite_prog_func_main_aa.clone());
    expr_1;
}
```

8.2.2 Index accessing

TypeScript allows for arbitrary access of undefined array indices. This means that the JavaScript interpreter does not enforce the user to specify a static size upon declaring an array. Rather, it allows any accesses to indices that might, or not, have been previously initialized.

For example, the following program to totally valid and does not yeild any out-of-bound error.

```
var a: number[] = []
a[2] = 1 // no error
```

As per the specifications, GoLite enforces the user to access well-defined array indices. Otherwise it throws an out-of-bound error to the end user. To simulate this behavior in TypeScript, we had to extend the prototype of Arrays to include a *check()* method that produces an error if one tries to access an invalid array index. Hence, we simply prefix a function call to *check()* before generating the array access in TypeScript from a GoLite program.

```
// GoLite
package main;

var x [3]int;
func main() {
    x[0] = 1;
    x[1] = 1;
    x[2] = 1;
    x[1 + 2] = 1;
};
// TypeScript
Array.prototype.check = function(index : number) {
    if(index < 0 || index >= this.length) {
        process.stderr.write('Error: Index out of bound' + '\n');
        process.exit(1);
    }
    return this;
}

function golite_prog_main() : void {
    golite_prog_x.check(0)[0] = 1;
    golite_prog_x.check(1)[1] = 1;
    golite_prog_x.check(2)[2] = 1;
    golite_prog_x.check((1 + 2))[(1 + 2)] = 1;
}
golite_prog_main();
```

8.2.3 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.

In JavaScript, due to its scoping rules, the following program would yield two undefined values. This is because one 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()
```

8.2.4 Base types

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.

8.2.5 Type declarations

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
```

8.2.6 Loops

GoLite for loops are converted into TypeScript 3-parts for loops. Loop pre-statement are executed before a for loop statement in TypeScript, thus the pre-statement for TypeScript for loop is always empty. Loop condition and post-statement are promoted into functions which get executed in their corresponding position of the loop statement in TypeScript. Promoting statements into functions is necessary in TypeScript in order to simulate the order of evaluation of expressions of Go. Here is an example of code demonstrating the behavior.

```
// GoLite
for i:=0; i < 10; i++ {}

// TypeScript
var golite_prog_func_main_for_1_o_i : number;
golite_prog_func_main_for_1_o_i = 0;

function for_cond_1() : boolean {
    return (golite_prog_func_main_for_1_o_i < 10);
}

function for_post_1() : void {
    golite_prog_func_main_for_1_o_i++
}

for (; for_cond_1(); for_post_1()) {}
```

8.2.7 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
}
```

8.2.8 Object Comparison

Object comparison in GoLite compares fields of two object of the same type. In TypeScript, variable store object reference, thus == operator would evaluate to true if and only if the two objects refer to the same element in memory. To solve this problem, for each GoLite struct type a TypeScript function called *equals*() is auto-generated to compare non-primitive types.

8.2.9 Struct

GoLite struct types are mix of a class and a type declaration.

```
// GoLite
type Person struct {age int;}

// TypeScript
class struct_1 {
    age : number = 0;
    clone = () : struct_1 => {
        var obj : struct_1 = new struct_1();
        obj.age = this.age;
        return obj;
    }
    equals = (obj : struct_1) : boolean => {
        return this.age === obj.age;
    }
};
type golite_prog_func_main_Person = struct_1;
```

8.2.10 Array

TypeScript has a built-in Array data structure which is augmented by the code generator to support index out of bound detection, clone, initialization and comparison.

```
interface Array<T> {
    check : (index : number) => Array<T>;
    clone : () => Array<T>;
    init : (val : T) => Array<T>;
    equals : (array : Array<T>) => boolean;
}
Array.prototype.check = function(index : number) {
...
}
Array.prototype.clone = function() {
...
}
Array.prototype.init = function (val: any) {
...
}
Array.prototype.equals = function (array: any) {
...
}
```

8.3 Slice

Slice in GoLite has a unique implementation which was simple to replicate in TypeScript using a custom class with an Array component.

```
class Slice<T> {
    size : number = 0;
    capacity : number = 0;
    array : Array<T> = new Array<T>(this.size);
    append = (val : any) : Slice<T> => {
        var slice : Slice<T> = this.clone();
        if(slice.size + 1 > slice.capacity) {
            slice.capacity = (slice.capacity + 1) * 2;
            slice.array = slice.array.clone();
        }
        slice.array[slice.size] = val;
        slice.size++;
        return slice;
    }
    check = (index : number) : Array<T> => {
        if(index < 0 || index >= this.size) {
            process.stderr.write('Error: Index out of bound' + '\n');
            process.exit(1);
        }
        return this.array;
    }
    clone = (): Slice<T> => {
        var slice : Slice<T> = new Slice<T>();
        slice.size = this.size;
        slice.capacity = this.capacity;
        slice.array = this.array;
        return slice;
    }
}
```

8.4 Assignment

GoLite assignment operator performs a copy of the assigned object. Again, by default TypeScript assignment would copy the reference. Thus on each assignment the *clone()* function is executed. Furthermore, assignment in GoLite can be used to swap elements $a, b = b, a$. With a similar syntax, TypeScript achieve the same behavior $[a, b] = [b, a]$.

8.5 Short Declaration

The generated code for short declarations in TypeScript is composed of two steps, first is variable declaration and second is assignment. First step works for both new and existing variables in short declaration left hand side, as variable re-declaration using *var* in TypeScript is allowed.

8.6 Printing

Print statements are converted into equivalent statements that allow our language interpreter to display text into the standard output.

8.7 Expressions

TypeScript code generated preserves the order of expression operation executed as parenthesis enforces the order. Also the order of execution of Go has been simulated by executing part of the expression before the expression statement and storing them into temporary variables.

8.7.1 Raw string

Raw strings in GoLite are manipulated so that each new line is converted into \n and each escape character is escaped to simulate the output of Go raw strings.

8.8 Testing

Testing the code generation phase of our compiler was a tedious process. Our philosophy while designing the test set for the code generation phase of our compiler was to test the semantics of each langauge feature independently as much as possible. Hence, each test script focuses on a particular language construct. Our automated process compiled the GoLite file in actual TypeScript file, executed the generated code in a NodeJS environment and verified that the expected output was correct.

8.8.1 Benchmark programs

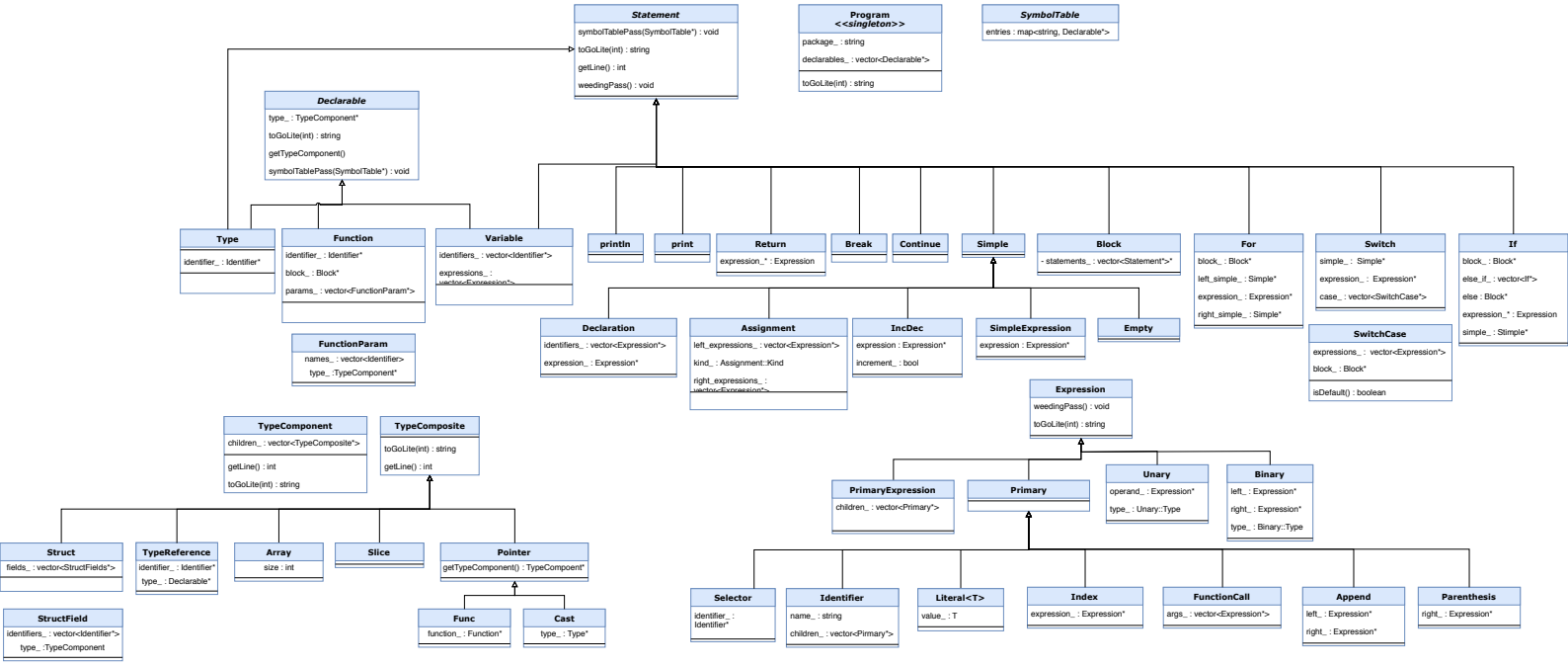
We developed two benchmark programs that test the code generation phase of our compiler thoroughly. The motivation of these programs were to develop program that solved a real-world problem and used every language features of GoLite as much as possible. It was also important that the executing time of those benchmark programs was fairly long, to generate as much code as possible and test the real-life performance of the generated code in the JavaScript interpreter.

The first program runs Dijkstra on Montreal metro stations and display the shortest path for each stations to all other ones.

The second program generates the search space graph for a variant of the 8-puzzle game up to a certain depth.

Each programs were carefully parametrized such that it ran at least 5 seconds in the GoLite playground.

9 Unified Modeling Language



9.1 Conclusion

In conclusion, we think this project, although a lot of work, really helped us gain a hands-on experience in designing every part of a modern compiler. Through the whole experience, our team had to overcome many logistical and technical issues. We believe this project gave us more than just a technical expertise in compiler design, but also a general appreciation for the field of compiler and programming languages as a whole. It was definitely one of the hardest assignment of our university experience, but we are glad we had to experience it. Also, we think this project is a nice addition to the overall course structure. It provides students in class the opportunity to apply the materials learned in class directly making it much more easier to study for.

If we had to re-do the compiler, I think our team would chose another target language. Overall, we think TypeScript was a safe choice for the generated language, but it truly lacks the performance. We think this has much to do with how recursion is implemented in JavaScript. We think the overall implemention in C++ of our compiler truly helped us achieve better performance in this assignment. We were able to use most of the C++ language features at the right level of abstraction such that the codebase was easy to maintain and contribute on.

9.2 Contribution

Both member of our team were very involved in every implementation detail of the compiler. We always discussed thoroughly of the design decisions and the impact they would have on the end product toghether as a team. We also tried to meet as much as possible in person to be as much efficient as possible. Every member of the team contributed in some way or another to every features of the compiler. We also carefully synced up every now and then to make sure we both took part in the design decisions and that we both are proficient in writing actual code.

Amir was mostly in charge of implementing the features and maintaining the overall toolkit used to compile the project. Being more experienced with compiler

design, he also reviewed most of the code changes submitted by Jeremie and made sure we were in track with the schedule.

Jeremie mainly worked on some fundations of the compiler codebase. Although less experienced in C++, he took also part in all implementation decisions with Amir and contributed to the codebase. He also helped the team to come up with multiple test cases for every milestone of the project. He was also in charge of the contents in the reports and researching TypeScript for a potential target language.

Overall, we are satisfied with the work we have done for this project. We are confident that our compiler will do well when evaluated while staying easily maintainable for the subsequent additions of the GoLite project. We plan to release this compiler as an open-source project and hope we win the coveted grand prize: the Gopher doll.