McGill University

COMP 520 - Compiler Design

Winter 2018

GoLite

Milestone 2 - Group 02

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

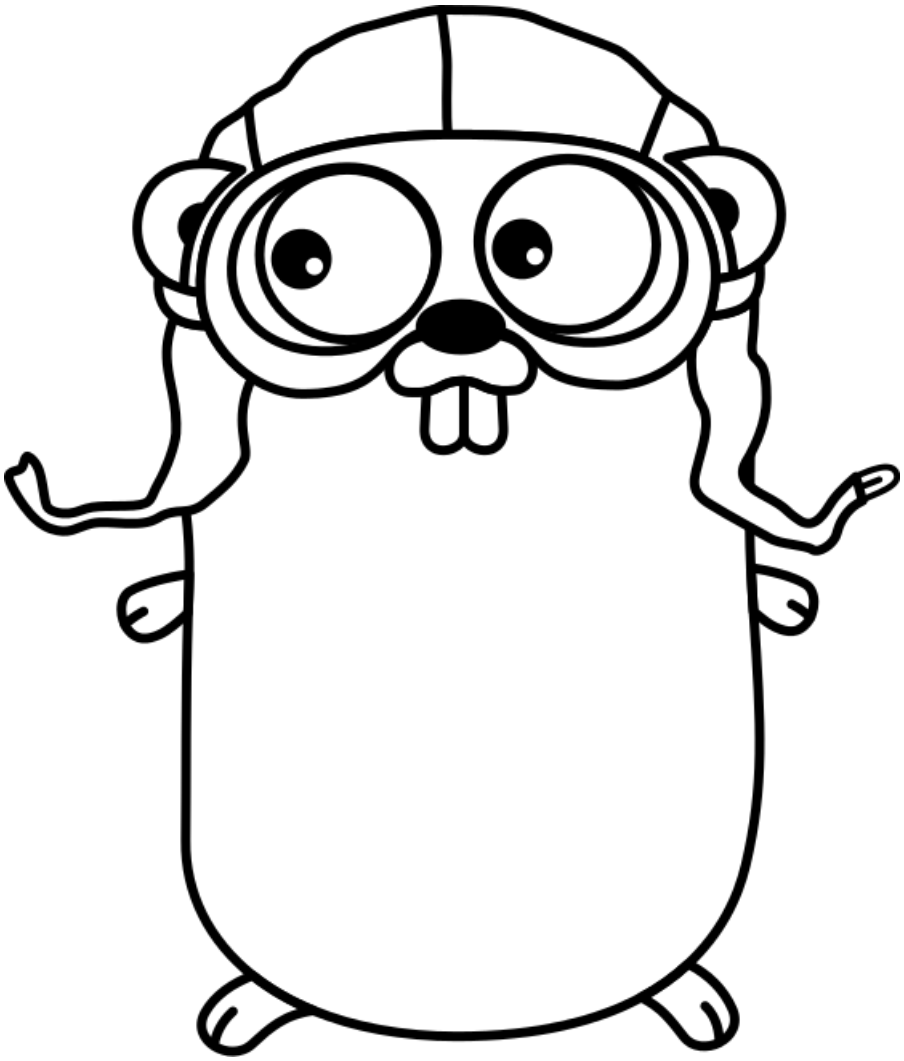
Name: Amir El Bawab
McGill ID: 260645260
E-Mail: amir.elbawab@mail.mcgill.ca

March 14, 2018

Contents

1 Introduction

In this project, we continue discussing the GoLite language, more specifically the Semantic Analysis phase including building the Symbol table and performing Type checking. In Milestone 1, we focused on Lexical and Syntax Analysis. After the evaluation of the deliverable, minor bugs have been discovered by automated test scripts. All bugs and issues related to the first two phases of the compiler design have been fixed before working on Milestone 2, as a strategy to facilitate code debugging.



2 Symbol Table

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. 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 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.

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. 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.

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.

2.1 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*.

The first edge case with 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).

```
package main
func main() {
    type GoLite struct {}
    var GoLite = "GoLite is awesome!" // GoLite redeclared in this block
    println(test)
}
```

The second edge case has to do with recursive type definitions. In GoLite, it is possible to define a new type that references an already defined type. The newly defined type 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.

```
package main
func main() {
    type B []B; // valid recursive type B
    type A A; // invalid recursive type A
}
```

To solve this, we simply added the necessary checks to verify that the referenced type is not recursive and not a slice.

2.2 Functions

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).

The first edge case with function declarations is that one cannot redefine a declaration within the same scope.

```
package main
func myFunction() {}
func myFunction() {} // myFunction redeclared in the same block
```

The second edge case is the implementation of the conditions for the special functions *main* and *init*. The special functions are not allowed to have parameters nor 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.

The third edge case is reporting an error when the variable and type declarations have the same names matching a special function identifier. To check for such case, when adding an entry in the top level symbol table, if the declaration is not a *Function* declaration and matches a special function name, an error is displayed to the user.

2.3 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.

2.3.1 Declaration

One tricky situation we encountered while designing the symbol table pass had to do with declarations. *Declaration*s are essentially a shorthand syntax that bundles a new *Variable* declaration with an assignment together in the same statement.

During the symbol table pass, we had the choice of either mapping the identifier to a *Declaration* object instance directly through the symbol table (a trivial implementation) or we could implement it such that it maps the identifier to a new *Variable* definition object instantiated by the *Declaration* object at the symbol table pass.

Clearly, the first approach had the advantage of being simplistic. It would have been fairly easy to implement and maintain. However, it would have made it much more difficult to for other components referencing variables to check for an existing variable with an identifier through the symbol table. Indeed, each time we would have to handle two cases, one for a typical variable declaration and one for a shorthand declaration, which is clearly not ideal.

The second approach was much more flexible and allowed us to re-use the symbol table pass implementation of the *Variable* class.

However, this approach also raised concern with the following example. Consider the case where declaration act as an assignment for an existing variable.

```
package main
func main() {
    var a int = 1
    a, b := 0, 1
    print(a,b)
}
```

As per the official GoLang specification:

> Unlike regular variable declarations, a short variable declaration may redeclare variables provided they were originally declared earlier in the same block (or the parameter lists if the block is the function body) with the same type, and at least one of the non-blank variables is new. As a consequence, redeclaration can only appear in a multi-variable short declaration. Redeclaration does not introduce a new variable; it just assigns a new value to the original.

This means we had to be careful not to trivially try to redefine a new variable upon each declaration. We first had to check whether a variable was already declared using this name.

3 Type Checking

In this project, type checking is performed on the program right after the symbol table pass is completed. An alternative design would be performing type checking while building the symbol table, however such design would make it more difficult for more than one person to collaborate at the same time. Making two separate phases also allowed for greater maintainability for future releases of this project.

3.1 Evaluate expression types

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.

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.

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.

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.

3.2 Additional 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.

4 Testing

In this milestone, we developed an extensive test set which thoroughly tests the ability of our compiler to detect underlying issues related to the symbol table and the type checking passes respectively. 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.

Out test set comprises about 231 invalid programs and 43 valid ones for the symbol table and types checking phase. Most test cases were implemented while reading the language specifications. Those test cases were named according to their corresponding section in the document. Another set of test cases with a prefix "999-More-" were created to test random edge cases of the language and serve as an extra layer of testing.

5 Team

We both started off by reading the GoLite specification carefully as a team to first establish the inter-dependencies within the components of this milestone. Because reading week was coming up, we knew would have to synchronize remotely during the following week. It was essential to establish the different working units of this milestone clearly. After meeting up, we established 4 different units that we could work on independently: the symbol table, the type checker, the integration of both components and the report. We decided that Amir would work on the type checking and its integration with the symbol table while Jeremie would work on the initial implementation of the symbol table and the report. Concerning the

test cases, we learned from milestone 1 that it is much more effective to have each team member design his own test set for his part. This makes it much more efficient and the test cases quality does not suffer from it.

Knowing that the type checking phase was dependent upon the initial implementation of the symbol table, we decided to create the minimal data-structure allowing to have a mock symbol table as soon as possible. This allowed Amir to start working on type checking right away using a fake "mocked" symbol table. It also allowed our team to work efficiently in parallel (which was especially useful while working remotely) and eliminated the need for us to integrate the symbol table with the type checking from time to time. Once Jeremie had a working version of the symbol table implementation, Amir simply had to integrated it with the type checking phase for good. Meanwhile, Jeremie started working on the report. Both members worked on their report section respectively and the final report was reviewed by Jeremie.

The challenges we faced during this milestone mainly had to do with working remotely and syncing up as a team. We agreed that it would be better to meet-up in person more frequently in the future to better keep track of our progress. Overall, we are satisfied with the work we have done for this milestone. We are confident that our compiler will do well when evaluated while staying easily maintainable for the subsequent milestones of the GoLite project.

6 Design

6.1 Changes from Milestone 1

The overall structure of the program did not change from Milestone 1 except for the *Expression* class. Initially, the project design had *Expression* inherits from *Statement*. However, after analyzing the relationship between the two classes, we decided to modify the relationship from *is* to *has*. In other words, we introduced a new class *SimpleExpression* which inherits from *Statement* and has an instance

of *Expression*. A *SimpleExpression* is simply a container for an expression which must evaluate to a function call.

Furthermore, a *SymbolTable* class has been added to serve as a hierarchical look up table by various components in the program.

6.2 Unified Modeling Language