# GoLite Compiler Milestone 2 Report

Lancer Guo, Bessie Luo, Charles Lyu

## 1 Modifications of Milestone 1

### 1.1 Block and Line comment

Originally, our scanner ignores all the stuff in a block comment, regardless of the existence of newline. This causes a problem when a block comment is between a statement. For example:

```
var x /* block comment
      */ int
```

This will throw an error message in the scanning phase, because block comment contains a newline, and before the newline there is identifier which satisfy the condition of semicolon insertion. Thus a semicolon will be inserted.

```
var x int = /* block comment
            */ 5
```

This example is a valid case because "=" is not a valid token for insertion, so the statement is valid.

To fix the problem, we define two regex, with newline and without, and only check the semicolon insertion when a newline is detected.

For line comment, our scanner didn't support any inline comment after statements because we included newline symbol as part of the comment regex. By removing it allows us to insert line comment anywhere.

Another fix is to check EOF tokens at the end of the file, so that we don't have to end the file with a newline all the time.

### 1.2 Terminating Statements

To check if functions with a return type do end in a terminating statement, we added a method in the weeder that checks for all the cases of terminating statements using recursion. This method returns a boolean to indicate if the last statement in the function is indeed a terminating statement or not.

## 2 Symbol

Symbol is a construct we used as the basic element in our symbol table. A symbol contains the name of the value it represents, its symbol kind, a next pointer for the next symbol in the symbol table, and a reference to another symbol which encapsulates one of the four following:

- variable type (for variable declaration with type)
- parent type (for type declaration)
- a list containing fields in the struct
- parameters and return type for function

The symbol kind can be one of the 8 following: variable symbol, function symbol, type symbol(whose underlying type is a base type), struct symbol, slice symbol, and array symbol, infer symbol (the type of the variable is inferred during typechecking stage), and null symbol (used to populate the empty spots in symbol table)

### 2.1 Symbol Table

The symbol tables stores every symbol in a scope and contains a pointer that references its parent scope's symbol table. Since there are three kinds of declarations: type, variable and function, the symbol table has three separate arrays to store these different types of symbol. Even though it increases the memory usage, we thought it would make our code cleaner and easier to keep track all kinds of symbol. But during the late stages of the symbol table implementation, we realized that keeping the different symbols in one table instead of separate arrays will make redeclaration checks much easier since it doesn't require us to traverse all three separate arrays in the current symbol table (redeclaration check is performed every time when a new symbol is added to the table). Nevertheless, changing the symbol table's structure might introduce new bugs so we kept our initial design.

At the top level, we initialized five base types to put into the global symbol table: int, float64, string, rune, and bool. Also, we define two variables true and false as the constant value of bool.

### 2.2 Type Declaration

For a type declaration, `type T1 T2`, we first check if `T1` is already defined in the current scope or not. If it is, a redeclaration error is raised. We are not checking in the parent symbol table to allow shadowing of types. If we are in the top-level scope, `T1`'s name cannot be main or init. Then we will retrieve the symbol with name `T2` (if `T2` is not an array, slice nor struct) and create a new symbol with `T1`'s name and assign the parent type field to `T2`.

Thus in the case of a long chain of type definitions (eg. `type a int, type b a`), the parent type is essentially a list of all the underlying types, `b -> a -> int` that allows us to traverse all the way to its base type, in this case, int.

if `T2` is of type array or slice, then we will create intermediate symbols that represent the number of dimensions of the array or slice and assign the parent type of `T1` to these symbols. For example, in `type a [5][]int`, `a`'s parent type is `"[5]"`, `"[5]"`'s parent type is `"[]"`, and `"[]"`'s parent type is `"int"`. Note that these intermediate symbols used to denote the dimension are not stored in the symbol

table.

if T2 is of type struct, then instead of assigning a parent type to T1, T1 symbols's structField field will be populated by the fields in the struct, which are themselves symbols with a name and a type. So essentially a struct opens a new "scope" for these fields but since these fields can only be accessed through the struct, we are not storing them into the symbol table.

Symbol T1 is stored into the type array in the symbol table with its symbol kind set to the corresponding type kind of T2. In the special case of `type _ T2`, we do not introduce new mapping in the symbol table, however, we must check that T2 is already declared before.

#### 2.2.1 Recursive Type

In GoLite, recursive type definition (a type that references itself) is not permitted unless the type definition contains a slice. For example, `type X X` and `type X [5]X` are not allowed, but `type X []X` and `type X [5][]X` are allowed. To check for invalid recursive types, if the type definition doesn't contain a slice, we compare the names of T1 and T2 and if they are equal, then this recursive type definition is not allowed. If T2 is of type struct, then we have to check the type of every field in the struct and make sure they have valid recursive type definition.

### 2.3 Variable Declaration

**Case 1.** `var x T`

We first check for the naming constraint (cannot be main/init) if this is a top-level declaration then perform redeclaration check on `x`. `T` should already be stored in the symbol table, if not, then an error is raised. We finally assign `x` symbol's variable type field to symbol `T` and store it in the current scope's symbol table.

If the variable is a blank identifier, the only difference is that no mapping would be added to the symbol table. However, we should still check for the validity of `T`.

**Case 2.** `var x = expr`

Like case 1, the only difference is that symbol `x`'s variable type would be set to `"infer"` which means we will infer the type of `x` in typechecking phase. We raise an error if any variable in the `expr` is not declared before. If x is a blank identifier, then the expr would still be evaluated but the variable would be discarded.

**Case 3.** `var x T = expr`

Like case 1 to assign the variable type, however, we will leave to the typechecker to check the equality of Type(`T`) and Type(`expr`). Like case 2, we would still evaluate `expr` even in the case of blank identifiers.

Not only does the variable array in the symbol table store the variable symbol, we also store it back to its corresponding AST node for the later typechecking stage.

### 2.4 Function Declaration

Every function adds a new symbol with its corresponding function name into the function array of the symbol table. Each function introduces a new scope for the types and variables declared in its body whose parent scope is the scope that the function was declared in. Every formal parameter of the function is added to the new scope as well. Since the typechecking phase requires the return type and parameters for function calls, we also store them as fields in the function symbol. Because there can be multiple parameters and the next field in the symbol is already used to link the symbols in the symbol table, we wrapped the parameter symbols stored in the func symbol using a `Node` struct that has a symbol field and a next pointer to the next parameter.

Init functions are not added to the symbol table, however, we will still perform the relevant checks on its parameters, return type and body.

Although blank functions are not stored in the symbol table, but we still need to perform typechecking on the return statement with the return type, thus a function symbol is stored in the AST node for the typechecking stage.

### 2.5 Short Declaration

To ensure that at least one variable on the left-hand side is not declared in the current scope, we keep a counter to keep track of any variables that are not stored in the current symbol table as we iterate through the variables in the left-hand side (blank identifiers are ignored). To ensure there is no repeated declaration of new variables, instead of putting the symbol immediately into the table (which will cause a problem because we cannot use our usual redeclaration check in short declaration), we will store all new symbols in a list. Every time when we create a new symbol, we check with the list to find any redeclaration error. At the very end, all symbols in the list are stored back to table.

### 2.6 Assignment

The symbol table has to make sure that all variables in the left-hand side and right-hand side are declared before, ie. they are already stored in the symbol table. Otherwise an error is raised. Literals in the right-hand side are ignored since they don't introduce mapping between an identifier and value.

### 2.7 For, If, Else, Switch, Block

`init` in `for`, `if` and `switch` statements opens a new scope for the newly initialized variable in which the rest of the statements will be evaluated in. Every pair of curly braces opens up a new scope, so the body of `for`, `if`, `else`, `switch` and `block` creates a symbol table whose parent is the current scope's symbol table. In addition, every case in `switch`, including the default case, also opens a new scope for the statements followed.

# 3 Type Check

A significant design we have for type checking is, like the example in the class, to assign symbol to its corresponding AST node. We add new field in AST struct called sym. This structure allows no involvement of symbol table and everything is set when we need to compare the type. Because symbol and type checking are two different but consecutive phase, it also helps us to divide the work.

## 3.1 Type checking function

We defined a set of functions to help us satisfy the constraints. They are repeatedly used over the whole process.

- *compareType()*: compare if two type list are the same, no parent type comparison.
- *checkParentType()*: given two type symbol, check if one type is the other type's parent, used for typecast. When traversing from child to parent type, it stops at any struct, slice or array type.
- *checkLValues()*: check if a type symbol is a kind of Lvalue.
- *checkIndex()*: check if expression is of base type int.
- *checkBOOL()*: check if expression is of base type Boolean.
- *checkArithmetic()*: check if expression is of base type int, rune, or float64.
- *checkArithmeticOrString()*: check if expression is of base type int, rune, float64, or string.
- *checkBitWise()*: check if expression is of base type int, rune.
- *checkOrderedBinary()*: check if binary expression can be ordered.
- *checkBaseType()*: check if expression is of base type int, bool, rune, float64, or string, used for typecast.

## 3.2 Deal with expression list

At the earlier stage, our idea of representing the type was by using a type struct declared in the AST. A symbol object has a pointer to its own type, and we access the type from symbol in the type checking phase. We quickly realized that this approach doesn't work well because a type declaration is also stored as a symbol. For example, a type symbol with name `int` will have to store another type struct with the same name.

As the result, we found that the symbol itself is sufficient to provide the information we need for type checking. We stopped using type struct and use symbol as the representation of type. All the type comparisons are comparisons between symbol.

However, another problem was quickly raised. During the type check, for example, the left hand side of assignment might contain many expressions. We need a list of type corresponding each expressions, and compare with the type list on the right. Because our type is actually symbol, and our symbol didn't have any pointer available to construct the list. So we added a wrapper to the symbol by defining a new struct Node for linked list. This structure supports:
**A single type**: a Node with a symbol and a null next pointer
**type list**: multiple single nodes linked by their next pointer

## 3.3 Overall structure

Similar to the previous phases, we start from the root and traverse the nodes in a recursive fashion. As for the top declaration:

- **type declaration**: nothing needs to be done, because we don't need to type check any type declaration
- **function declaration**: parameters and return type of the function have been stored in the symbol, so we just need to make sure the body type checks
- **variable declaration**: most work has been done in symbol stage, here we need to update inferred symbol and type check declaration if a type is given.

  First we type check both sides, and it will give us a list of type on both sides. Then we check if a type is given in the declaration, if not then it might be a inferred kind. If so, we update the node's symbol on the left hand side to the right hand side's symbol. If a type is given, we compare the type on both sides and throw error if they are not the same.

After declaration, we switch on statement's kind and make sure all the statements type check. We will focus on some complex statement and explain in further details.

#### Return statement

A return statement is closely associated with its function declaration. In weeder, we first link each return statement with its belonged function. In symbol phase, we store function's symbol in its function AST node, so in type check, we have access to function's return type. The next step is type check return statement's expression, acquire its type and compare with defined return type.

#### Assignment

Assignment is separated as normal assignment and special operation. If a blank identifier _ is presented, we skip it because a blank identifier has no symbol, thus has no pre-defined type. As for the regular assignment, we use compareType() function to compare two lists of type. When it comes to special operation, we check if right hand side is arithmetic for +=, -=, *=, /=, and check bitwise for % =, ^=, <<= and >>=. We also need to ensure that special operation only support one expression on both sides.

If we encounter expressions, we type check expression by its kind, and return a list of type. Eventually, all the expressions will be resolved as literals or identifiers. When it is a

identifier, if it is a variable, we add its variable type(symbol) to the list. If it is a function, struct or other type, then we simply add its current stored symbol. As for literals, they were not stored in the symbol table so we make a matched symbol based on its literal kind, and add it to the list.

#### Binary and Unary

Binary and unary expressions require type check respect to the kind of operation. Similar to assignment, we check different condition on different operator. A special case is preventing any division by 0. When we meet a literal expression, we store the value into the symbol, and compare its value when any division expression detected.

#### Built-in function

Type checking for built-in expressions is straightforward because they mainly target slice and array type, as well as string type for len(). As mentioned in symbol phase, a slice or array type symbol has its kind assigned to slice or array enum. Thus by checking its kind we can conclude if the current expression is valid. As for the result, we return a int type for len() and cap() and void type for append().

#### Function call and Typecast

Function call and type cast have the same format. In the parser, both are recognized as function call because we don't know if the identifier is a type or a function until type checking.

```
var x = f(5)
```

Follow the example above, once we detect a function call expression, we identify if it is a function call by type check **f**. It will give us a symbol **s** with either type kind or function kind, telling us which route to take.

If **f** is a type kind (type cast), then

- determine if **f** and **5** have compatible type using checkParentType() function.
- if fails, determine if **f** has type "string".
- if fails, check if both **f** and **5** have either type of int, rune or float64.

If all cases have passed, then the resulting type is type **f**.

If **f** is a function kind (function call). By type checking **f** we are able to access all the parameters symbols stored in the function symbol. By specification, the expressions fed into the function have to match all the parameter type. Once all the parameters are checked, we can return a type symbol same as function's return type. This type symbol will be used to update variable **x** because **x**'s type was set to `"infer"` in the symbol stage.

#### Indexing

```
var x [][]int          (1)
x[0][3] = 4            (2)
var y []int = x[10]    (3)
```

The challenge in indexing expression is that we do not know the dimension of any given slice or array. So with naive implementation, a expression like x[0][3][4] would be valid and x[0] would not be resolves as a type []int.

As for the example above, when (1) is executed, a variable symbol x is put into symbol table and it has type of slice. When (2) is executed, it is a assignment, so we typecheck the left and the right, then compare their types. In the parser, x[0][3] will be wrapped into structure like

```
x[0][3] = temp[3]
temp = x[0]
```

so we treat multi-dimension slice as one dimension slice in a recursive manner. The typechecker will recurse on the very left part until x is reached, and it detects that x is of type slice, so a slice symbol will be returned. The time of recursion in this case will determine how many dimension it is accessing. We defined a global flag and count variable **c** to solve this problem. When we enter a indexing expression, set the flag, and every time we recurse to this part we increment the count **c** until it stops. Besides, it is easy to determine the dimension of its type [][]int since we can simply search its parent type and record times until a single type, in this case, int is reached. Now we have

```
x[0] => []int
x[0][3] => [][]int
x[0][3][1] => exceed limit (error)
```

At last, for the the index, function checkIndex() is used so that only int typed variable is allowed.

#### Field access

```
var x = a.b.c
```

When we want to access a field of struct, we first need to get the pointer pointing to its symbol. Start from the dot, the left part is checked first and expect a symbol of struct. Then we traverse all the fields contained in the struct, and compare its variable name with the right hand side. If we find a match, return the field variable's symbol type, which then is used for either nested struct access or direct comparison with the left part of the assignment.

## 4 Testing

Below are the 30 invalid cases corresponding to different errors and rules.

- assign1.go: assignment of a non-declared variable.

  Rule: can only assign a variable which is declared already.

- assign2.go: left-hand side of short declaration is not a variable type.

  Rule: left-hand side of short declaration need to be a variable.

- assign3.go: incompatible type string to rune.

  Rule: rune is not assignment compatible with string in variable declaration

- binary.go: assigning bool type to user-defined type inheriting from bool.

  Rule: assignment types need to be strictly equal, no resolving.

- cap1.go: cap() returns an int, cannot be assigned to string

  Rule: cap() could only return an int which is not compatible with string in declaration

- cap2.go: only array or slice can call cap()

  Rule: cap() expects slice or array type as argument

- dec.go: decrement on struct type

  Rule: decrement expects int, rune, and float64

- div.go: invalid div assign

  Rule: div expects int, rune, and float64

- field-select.go: invalid field selection

  Rule: can only access defined field in struct

- for.go: expr in for loop is type int

  Rule: expr in for loop condition must resolve to type bool

- func1.go: declaring both `x` as parameter and in the body of the func

  Rule: cannot redeclare a variable as parameter and inside the function

- func2.go: assign a variable as a type to another variable

  Rule: cannot make a variable be the type of another variable

- func3.go: func with return type but without terminating statement

  Rule: func with return type needs to have a terminating statement

- func4.go: main func has parameter and return type

  Rule: main function must have no parameters and no return value

- func5.go: init func has parameter and return type

  Rule: init function must have no parameter and no return type

- if.go: Incompatible type in if condition

  Rule: expr in if statement must have type bool

- inc.go: invalid increment on string type

  Rule: increment expects int, rune, and float64

- len.go: len() cannot be called on int type

  Rule: only string, array or slice can call len()

- recursive1.go: invalid recursive type a

  Rule: Recursive type (expect slice) cannot have both id and type with same name

- shortdecl1.go: short declaration contains no new variables

  Rule: all variables declared in the same scope for short declaration

- shortdecl2.go: repeated identifier on lhs of short declaration

  Rule: cannot have repeated identifier on lhs of short declaration

- shortdecl3.go: "a" is not a variable type

  Rule: cannot assign value to a type

- switch1.go: empty switch condition, non-bool case statement

  Rule: if switch condition is missing, then all the case should evaluate to bool

- switch2.go: provided expressions have different types: int and bool

  Rule: switch's condition should have the same type as the case type

- type1.go: "b" is not declared.

  Rule: cannot use a type that is not declared before

- typecast1.go: non-base type typecasting

  Rule: invalid expression for type cast, the expression type must resolve to a base type

- typecast2.go: typecasting slice to string

  Rule: cannot have conversion between incompatible types

- typecast3.go: int is not assignment compatible with string in variable declaration

  Rule: int is not assignment compatible with string in variable declaration

- var1.go: naming var as main

  Rule: var cannot be named as main in global scope

- var2.go: naming var as main

  Rule: var cannot be named as init in global scope

# 5 Work distribution

We hold all decision discussion together and then distributed work equally. We first use the given test solution to fix our errors from milestone1 and work on the Symbol Table and Type Checker together using the CodeCollab tool to make sure everyone agrees on the structure. For the test parts, each of us wrote over 10 valid and invalid cases. Then, we have one person work on all tests and two other people check afterwards and fix errors. Such work flow provides the lowest error tolerances and great balance between working together and individually.

**Bessie Luo**: Fix errors from Milestone1, Symbol (first draft), Type Check (revised), Report, Test

**Lancer Guo**: Fix errors from Milestone1, Symbol (revised), Type Check (first draft), Report, Test

**Charles Lyu**: Fix errors from Milestone1, Symbol (first draft), Type Check (revised), Report, Test