COMP 520 Milestone 2:

Symbol Table and Type Checking for GoLite

Members:
Ralph Bou Samra 260503614
Jonathan Lucuix-Andre 260632816
Yan Qing Zhang 260520930

Introduction:

The goal of milestone 2 is to write the infrastructure code that allows for type checking of the grammar implemented in milestone 1. In the first milestone, we completed the scanner, the parser, and the AST for the GoLite language. Given the feedback from the first assignment, we reiterated over the work we did to refine our code to address almost all comments. In the second milestone, we implemented the next three stages: weeding, semantic analysis, and type-checking.

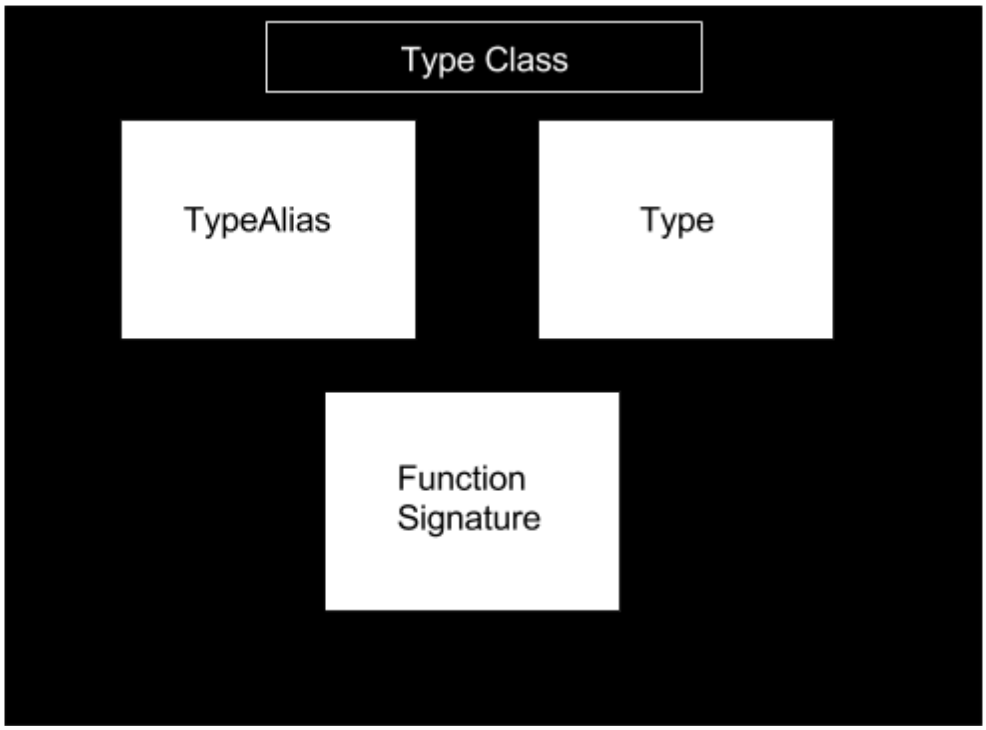
Design Decisions:

For weeding, the pass checks for break statements such that they only appear within a loop or a switch statement. It also checks that continues are only found within loops. The reason we do this in weeding is because the statements are passed in as a list and we are unsure which specific item in the list during parsing is the continue or break statement. It is very hard to do this in the grammar as we would have to literally check every possible position in the production. It is only after we parse the entire block that we can look through it and see if there are mistakes with the placement of the break or continue statements. The weeding also checks for blank identifiers as it would generate too many additional grammar productions if we were to check in the parser.

For our type checking logic, we chose to do two passes. The first pass is done in SemanticAnalyzer.java. This pass collects all variable declarations that are typed as well as function declarations and type declarations. During this pass, these ids and types are filled in the symbol table. In the second pass, which is done in TypeChecker.java, the declared ids and their associated types are then checked versus the evaluated expressions and statements. For inferentially typed variables, the first pass added the ids and an empty symbol, and the second pass evaluates the right hand expressions and populated the symbols with the correct type.

The reason for two passes is that it is easier to separate the populating logic and the type checking logic and allowed for cleaner code as well as task separation. It would be possible to combine the logic but that is far more intertwined and would need a person to have a greater knowledge of other sections in order to implement his own task.

Certain things that weren't properly implemented due to issues with shift reduce were struct declarations with slices and arrays, such as "type x []struct {}" as well as selectors with array indexing. These issues will be fixed in the next milestone, but required a complete rewrite of the current expression grammar as well as a fundamental change to how types are described within the AST, and thus were not fixed for this implementation.



TypeClass:

This class is a wrapper for all information regarding an AST node's type. It contains information about type aliasing, array dimensions and function signatures. Using a centralized wrapper for type checking made type checking easier, since the wrapper has all the information needed for comparing two types whilst type checking.

Furthermore, instead of storing all this information within the Symbol class, we encapsulated it within the TypeClass. In turn, a type could be easily cloned, without needing to clone the entire symbol. Consequently, short declaration statements benefitted by being able to clone the right-hand side's expression type, and copying that type to the identifier on the left-hand side.

Type:

The type enum denotes all the basic literal types. It also contains a function that converts a literal name to the corresponding enum type. This allows us to quickly compare the base types of two different variables or expressions.

TypeAlias:

TypeAlias is a class that keeps track of all type aliasing done on a variable's type.

For example, take the following source code:

```
type int1 int;
type int2 int1;
var x int2;
```

Within a TypeClass, the the following list is stored in the type class belonging to variable x:
TypeClass.typeAliases = [(int2 -> int1), (int1 -> int)]

This mapping allows us to easily check that two expressions evaluate to the same type in a binary operation.

FunctionSignature:

Assuming a function call an identifier refers to a function call, the *FunctionSignature* class stores the signature of the function. This allows the compiler to quickly validate the argument types being passed to the function. It also enables simple access to a function's return type. In turn, the return type can be matched with types on the left or right-hand side of the function call.

For example:

```
func f(x int) int {

}

func test() {
        var x = f(2)
}
```

In this example, *f's* symbol will store a function signature of the form (int) -> int. This is done in the semantic analyzer, thereby giving the type checker all the information it needs to type checking function calls.

TypeChecker:

The main logic of type checking resides in the TypeChecker file.
By and large, most of the nodes generated in the AST are traversed via a post-order traversal. However, there are certain scenarios where this rule did not apply. For instance, when the function type in a function signature is used to validate the actual type of the value returned, a top-bottom approach is used. That is because the node created for a function signature is different from the node created for the return expr. The return type in the function signature node is saved in a global variable that is used to validate the actual return type within the return expr node.

Another exception was applied when dealing with switch-case statements. Since the AST has 2 different nodes for a case and a switch statement. Instead of checking the type of the expression in the switch statement and then check if the case expressions have the same type, the case expressions were checked first, the type was saved in a global variable and then compared with the expression type in the function signature. If the case expressions did not

have the same types, the type checker aborts and throws an error without checking the type of the switch statement expression.

Scoping Rules:

A SymbolTable class is used to wrap a HashMap<String, Symbol> variable, which maps identifier names to their corresponding symbols. By wrapping the symbol table in a class, it was convenient to store a pointer to the next scope, thereby allowing us to push and pop symbol tables from our stack.

The scoping rules were implemented by pushing a new symbol table on the stack when a new scope is entered. When the scope is left, the symbol table is popped from the symbol table stack. This allows the scoping and shadowing rules to be correctly implemented. For instance, when an identifier is encountered, it is looked up in the symbol table stack, and the matching symbol in the outermost scope is used. As a result, type aliases and variable declarations are correctly shadowed.

When entering a switch, for, or if statement, two new scopes are created: one for the init statements, and one for the statement blocks. This allows the init statement to shadow any previously-declared variables, and enables the declarations in the block to shadow variables declared in the init statement. As such, the compiler performs expected behaviour on these types of statements.

NOTE: We also utilized a Symbol class to wrap information about a symbol, such as its kind. If a node's symbol is of kind SymbolKind.FUNCTION, the TypeChecker knows that the corresponding identifier denote the start of a function call, which contains the appropriate number of arguments. If the symbol is of kind SymbolKind.STRUCT, the type checker knows it must check the structNode field for the corresponding struct type. Struct comparisons work by comparing struct nodes. If they are equal, the comparison is valid. Otherwise, the comparison is invalid. Thus, by storing a reference to the struct declaration node, type-checking struct comparisons is quick and efficient

Type Inference:

For type inference, a list called *symbolsToInheritType* was maintained in the Symbol class. This is a list of symbols that refer to the variable. For instance, consider this code segment:

```
var x = 3
print(x)
```

In this example, the symbol table will initially map x to an empty TypeClass, since the semantic analyzer doesn't yet know x's type. When *x* is next encountered in *print(x)*, the identifier "x" will be searched in the symbol table, and the symbol created for x's declaration will be returned. The semantic analyzer then determines that this symbol belongs to a variable whose type must be inferred. A new symbol will be created for the *x* identifier in the

print statement. This new symbol will then be inserted into the *symbolsToInheritType* list of the symbol belonging to the declaration of x.

Subsequently, when the type checker infers that 3 is of type int, and that x has an empty TypeClass, it will call Symbol.setType() on x's symbol. This function populates the symbol's type, along with the type of each symbol in the *symbolsToInheritType* list. As a consequence, the type checker will know that x is of type int, and further type checking will succeed.

This list made it easy to infer the type of a variable, and propagate that type to each subsequent use of the variable. Although we could have made an additional pass through the AST for type inference, we decided that maintaining a list was an easier and more efficient alternative.

Type Checks:

Expressions:

- Unary operations on expressions -> programs/invalid/types/rbs_3.go
- Binary operations on expressions -> programs/invalid/types/j_2.go, j_3.go, j_8.go
- Function call argument type checking -> programs/invalid/types/j_5.go, j_6.go
- Type casting type checking -> programs/invalid/types/j_4.go
- Append type checking -> programs/invalid/types/j_7.go,

Statements:

- Base literals type-checking(INT, FLOAT64, BOOL, RUNE, STRING)
- Expression types within for loops -> programs/invalid/types/rbs_1.go
- Expression types within if statements -> programs/invalid/types/rbs_4.go
- Expression types within switch statements -> programs/invalid/types/rbs_6.go
- Expression types within Print/Println -> programs/invalid/types/rbs_2.go
- Expression types after return statements(also no return statement) -> programs/invalid/types/rbs_5.go, j_1.go
- Short Declarations -> programs/invalid/types/m_1.go, m_6.go
- Assignment -> programs/invalid/types/m_2.go
- Op-Assignment

Declarations:

- Variable declaration (with type, no expressions) -> programs/invalid/types/m_3.go
- Variable declaration (with type and expressions) -> programs/invalid/types/m_4.go
- Variable declaration (with expressions, no type) -> programs/invalid/types/m_2.go
- Type declarations -> programs/invalid/types/m_3.go
- Struct declarations -> programs/invalid/types/m_5.go
- Function declarations -> programs/invalid/types/m_7.go

Contributions:

Code:

Jonathan:

Symbol table building

Semantic analyzer
Function signatures
Type alias symbols
Primitive type casting
Short declaration type inference
Variable declaration type inference
Identifier type checking

Ralph:
Base literals type-checking
Unary operators Expression type-checking
Return statement type-checking
For loops type-checking
If statements type-checking
Switch statements type-checking
Print/Println statements type-checking
Block statements type-checking
Return Weeding

Yan Qing:
Binary Expression type-checking
Function call type checking
Type casting type checking
Append call type checking
Aliasing type checking
List assignment type checking
Inferring variable types

Report and Test Cases:
Everybody contributed equally to the report as well as the test cases.