GoLite Compiler Final Report

Lancer Guo, Bessie Luo, Charles Lyu

1 Introduction

This report gives an overview of a compiler implementation for a subset of the Go language, GoLite. Specifically, it summarizes the major design decisions for the 7 main stages of a compiler: Scanner, Parser, Weeder, AST, Symbol Table, Typechecker and Code Generator.

Go is a programming language designed by Google and it is mainly used in multicore network servers and distributed systems. GoLite adopts most of Go's syntax but on a smaller pool of base types and build-in functions. Most noticeably, GoLite only supports the ASCII charset, unlike Go which is UTF-8 encoded. Also, GoLite disallows slice/array/struct literals. Nevertheless, GoLite supports all of Go's operators, control-flow structures and the type declaration feature.

2 Language & Tool Choice

We chose flex/bison model in C as our primary tool for the implementation. This structure is easy to learn and apply for the early stages. It has also provided us great flexibility and efficiency in previous practices of our assignments.

The language that we choose to implement the GoLite Compiler is C. There are several reasons. First, flex and bison are both written in C and we've used flex and bison in our first two assignments, thus we are pretty familiar with these two tools. Secondly, all group members are familiar with C and C is considered as a good language for doing low-level development like code generation. Also, C is an example language used in class and thus we could gain insights from the slides and adopt in our own implementation.

3 Target Language

We choose Java as the target language of our compiler firstly because all of our team members are familiar with it. Secondly, Java is a high-level language, unlike C, which handles string concatenation, printing and structs (essentially a Java class) beautifully and doesn't require functions to be declared before its use since Java uses a two-pass compiler. In Java, every line of code has to reside in a class unlike GoLite. But this is an easy fix; we created a *public class GoLite*, put the code in there and renamed the generated executable to *Golite.java*. However, we do encounter some difficulties. For example, Java's built-in array doesn't quite match with GoLite's array behaviours so we have to make our own array class. In addition, Java employs pass-by-reference for non-primitive types which is not the desired behaviour for GoLite. Another major problem is designing the slice data structure in Java since it has so many complications (more to be explained later).

4 Scanner

The scanner is implemented in Flex. It tokenizes all the recognized keywords and operators with regular expression, integers in decimal/octal/hex, floating points, rune, interpreated and raw strings. Essentially, it returns a corresponding enum type to the parser. The scanner will eliminate any lexical error. Any unmatched sequence will be reported as error and print to the console.

4.1 Comment

It supports both one-line comment and block comment. For one-line comment, it will capture anything started by // and end with a new line.

For the block comment, it will capture everything enclosed by /* and */. If the comment block is never closed, it fails to capture the string and will result in an error. To fix the issue, we have a second expression to capture unclosed comment, anything started with /* but no */ found, and print an error message when a match is found.

However, it is still not enough to handle all the cases, 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.

We also checked for EOF tokens at the end of the file, so that we don't have to end the file with a newline all the time.

4.2 String

There are two types of string in GoLite, interpreted string and raw string. Each string has their unique wrapper. When the scanner is taking the input, it will read everything as char, such as '\n' will be read as '\' and 'n' . Before passing the string to the parser, to make the later stage easier, we process the string data so that Java can treat the string literal as it is. We wrote a C method in the scanner, processing the character one by one. Whenever we see a character '\', we replace this character as well as the one after it with the correct escaped sequence. Now we can translate pure character string into interpreted string. For the rawstring, if we can use the operator 'rawstring', we can just output the read string without any modification. However, on the other side, Java doesn't support rawstring, a direct output of '\' 'n' would be printed as '\n' in Java and be interpreted. So we cannot just wrap around the data with '...'. To fix this, we then updated the scanner to construct a new string that adds extra \before any escaped character, and use the new string in the Java context. (example follows)

```
‘hello world\n‘  (Golite)
"hello world\\n" (Java)
```

4.3 Rune

Rune type also involves recognizing an escaped sequence. As we read any string that needs to be escaped, we switch on the read character and output the ASCII code instead.

4.4 Escape sequence

Although interpreted string and rune both supports escape sequences, however, they each have subtle differences. Interpreted strings supports all escape sequences like rune, except the single-quote, which is replace by the double-quote. Rawstring prints everything as is, so there is no escape sequence thus doesn't support the back-quote escape.

One complication about escape sequence is that Java doesn't support `\a` and `\v`. So declaring a string with these two escape sequences will make Java throw a compile error. To solve this, although not a perfect solution, we replaced the \ with ~. Then the codegen phase will recognize these two special escape sequences and handle them accordingly.

```
var s string = "\a \v"  (Golite)
String s = "~a ~v";     (Java)
```

4.5 Auto Semicolon

In GoLite, when a line of code terminates, a semicolon will be auto inserted if none is presented. We first have a global variable that represents the last token type, which will be updated consistently. We defined a semicolon check function that will be invoked when a new line is detected. Based on the last token, a semicolon will be inserted properly.

5 Parser

The parser was implemented in bison. It contains the summarized context-free grammar for the language. Starting from the package and centered around the top-declaration, all the non-terminals are expanded programmatically and logically.

Program is in the form of a package identifier and a list of top-level declarations.

A list of declarations consists of three kinds: Function declarations, Variable declarations or Type declarations. We implemented the list using left recursion due to its low stack usage than right recursion. But this would mean that the list is stored backwards and all of our methods in the later stages have to always use recursion for traversal.

Variable declarations contain one or multiple variable declarations in the case of distributed variable declaration. Thus the variable declaration is also parsed as a reversed list. Since we can initialize multiple variables in one declaration, the identifiers are also implemented as a list, as well as the expressions on the RHS.

Type declarations contain one or multiple type declarations in the case of distributed type declaration. A type declaration has one identifier as the type name and its underlying type. The underlying type is a recursive structure. For example, a `[][]int` is parsed as `[]type -> [][]type -> [][]int`

A Function declaration contains a signature, which can be short or long form, an optional type, and a body formed by statements.

Statements are categorized by its type. Variable and type declarations are special ones that also exist in statements. An expression can also be a statement, but only if it is a function call. We will eliminate other expressions in the weeder. Moreover, a separate non-terminal called simple statement is used in if/else, for and switch statements. We also list break and continue along with other types, and later use the weeder to handle its special rules.

Expressions consist of both expressions and literals. It includes most of available operators in GoLite, such as binary, unary and builtin types. We think it is not necessary to make another non-terminal for terms like identifier, or other literals, so we will treat them also as expressions. Later on in the weeding section, we will separate expression to primary expressions and other expressions to constraint the syntax.

To eliminate ambiguity in the parser, precedence levels are enforced and left associativity is used. Logical operators have the lowest precedence level. Then comes comparison and bit-wise operators. Unary operations have the second highest precedence. We assigned a dummy directive for Unary and used %prec Unary to override the precedence level of `+, -, !, ^`. The dot (struct field access), square bracket(indexing) and parenthesis have the highest precedence level.

6 AST Tree

In our parser, there are many cases where a non-terminal is a list of other non-terminals. For example, a function declaration may contains many parameters, which are grouped as a list, the function body is a collection of statements, which are also listed in a order. To handle several cases like this, our solution is to have a next pointer point to a object that has a same type as the current node. In later stage, we will traverse the linked list and operate on each of them.

6.1 Top Level Declaration

Out AST root node contains a package name and a pointer to the top-level declaration list. The root node also contains a list of struct symbols with its assigned java class name. This list keeps tracks of the unique java "classes" we have seen so far and will be populated in the symbol stage.

Declaration node supports multiple nodes using linked list. Each node is either a function, type or variable declaration. Variable declaration consists of three different types: type only, expr only or both type and expr. In the tree, we have a different struct node to hold each of the different kinds. But later on we realized that we can combine these three into a single one and populate the fields with NULL as needed.

Function declarations consist of its name, a list of parameters (composed of id list and type),a return type and a pointer to the first statement inside its body.

Type declaration is made up of an identifier for the type's name and a TYPE node which embodies the underlying type. The TYPE struct is a recursive node to account for the case of multidimensional array and slice types. The struct type contains a list of fields inside the struct. These struct fields node contains an list of identifiers and its corresponding TYPE node.

6.2 Statement

In our Statement structure, for each possible statement type we declared one AST node to keep track of its data and a next pointer to the next statement.

The assignment statement consists of a kind variable that specify the operands of the assignments. Instead of creating enum for each operator, we directly store the operator as string (e.g. "&=") and compare it with our expected string in later stage. In this way, we thought that it could help to reduce the size of code since we don't match to match each enum to its corresponding operator in the codegen phase.

The expression statement is made of only one pointer to a Expression object. We rely on the weeder to ensure that expression statement can only be a function call. Short declaration has two pointers to Expression objects, id list and expression list.

If statement is divided into three parts. If statement, else-if statement and else statement. In the parser, if the statement will be linked to any following else if statement. For else if statement it will have a link to any found else statement. Later on we realized that we don't need an else-if statement since it is essentially a combination of else & if.

For the for statement, we created two types, while and three parts. To identify the infinite loop, we assign the condition field NULL. Same for the three part, checking if the second condition is NULL helps us to distinguish which kind of for loop is there.

The switch statement requires an optional condition statement, a main condition expression and many case clauses. A case can either be a default expression or a case expression. A default case only has a Statement body while a case clause can have a expression list as the condition.

6.3 Expression

Expression nodes will represent literals, identifiers, binary operations, unary operations, function calls, indexing, field access and build-in fuctions in GoLite.

Other than the regular type of expressions, builtin operations such as function call, array access, field access, append, len and cap each have its own kind. It is easy for us to weed specific rules for them by querying the kind of the expression.

7 Weeder

- Blank Identifier

  Blank identifier is handled by weeder as it can only appear in certain expressions. We defined a function weedBlank() and call it whenever a variable checkBlank is set to 1.

- Break and Continue

  Break statement can only appear in switch cases, and in each case there cannot be more than one break statement. Continue can only be in the for loop, and only one is allowed as well. We used variables inFor, inSwitch as the flag , indicating when should we check for these conditions. When we first encounter a kind of switch or for, we set the variable to 1, then set back to 0 after weeding the statement. The benefit of using next pointer in Tree node earlier is that we can just traverse the statements in case object and count number of break statements to determine the validity.

- #LHS = #RHS

  This constraint is satisfied by counting number of statements of each side. We traverse the list using pointers and terminate if numbers are not matched.

- 1 Default case in switch

  As we are weeding the statements in switch, we increment the count variable as long as we see a default kind.

- Expression statement can only be a function call

This is trivial since we can just compare expression kind to a function call kind.

- LHS of short declaration

  In the tree, the LHS of short declaration is a list of Expressions, and we need to ensure that only identifers are allowed. To deal this case, we compare the kind of each expression and reject if any non-identifier expression is found.

7.1 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.

8 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
- rename (unique name used in codegen)
- isContant (boolean to denote a constant)

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).

Originally, we want to have Symbol store only the essential information needed to process the typechecking, and this symbol will be associated with a AST node. Inside symbol structure, we had a pointer point to the Type, however, the information contained inside Symbol has a lot of similarity with the Type struct, and instead of shrinking down the Symbol, we decided not to use Type but restore the necessary data into Symbol structure. Later on, we had to come back to Symbol structure and add more fields to satisfy our need, eventually the Symbol structure has become larger than we ever expected, making the code less readable and legible and we regret our design choice now.

8.1 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. Since there can be shadowing of the base types, these constant symbols are flagged with its isConstant field set to true.

8.2 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.

For every struct we encountered, either tagged or untagged, we will compare with the struct class list stored in the root node and check if the current struct symbol is "equal" to any of the struct symbols stored in the list. Equality of structs implies the fields in the struct has to match exactly, ie, the field's name, type and order have to match. If the current struct doesn't match any of the structs we've seen so far, we add it to the class list and assign it a unique class name for the codegen stage. If the current struct "equals" one of the structs in the class list, we then assign its class name to be the same of that other struct.

Finally, 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.

8.2.1 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.

8.3 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.

8.4 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.

8.5 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.

8.6 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.

8.7 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.

8.8 Expression

Since expressions don't introduce new bindings, we would simply recurse every expression until the resulting expression is of type identifier (literals are ignored). Since all identifiers used in a sound expression should have been declared before, we would get the corresponding symbol from the symbol table and store it in the identifier node in the AST for later typeching and codegen stages. If no symbol of the same name can be found, then an error is raised.

9 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.

9.1 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.

9.2 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

9.3 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 % =, ˆ=, <<= 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.

10 Code Generation

10.1 Declaration

Naming conflicts for a variable or function name can occur when the identifier is legal in GoLite but is a reserved keyword in Java. To solve this, we added a prefix, *__golite_*, to every identifier's name. To resolve the same variable declaration in different scopes, we first concatenated the depth of the scope in which the symbol was created (ie, how deep is the symbol table in the cactus stack) in the name of the identifier. But this created an issue issue since all the switch cases have the same scope depth and declaring a variable with the same name in the switch cases would cause redeclaration error to occur in the Java code(more about how we designed switch later). So instead of appending the scope depth, we created a global counter and its value will be appended to the name of the identifier and the counter will be incremented every time a new variable/function is declared. In this way, every identifier name is guaranteed to be unique. We added a new field, *rename*, in the symbol to store this new name binding and we will use the *rename* instead of its original name in code generation. Note that the examples shown below don't have the identifier's name renamed according to the scheme proposed for readability purpose.

10.1.1 Variable declaration

Top-level variable declarations in GoLite are translated to *public static* variables in the Java class so they act as global variables in the Java class. Blank identifiers will still be generated because the expr on the RHS may have side-effect so it should still be executed. Like the naming scheme for identifiers, blank identifiers will all start with the prefix "blank" and will be appended with a counter value to avoid redeclaration error.

Implicit initialization in Java aligns with the init values in GoLite except for Strings. In java, String values are implicitly initialized to null, not an empty string like in GoLite. So instead, for every var declaration with a type but does not have a value assigned, we will assign its default init value according to the type.

```
 var s string       (GoLite)
----------------------------
 String s = "";     (Java)
```

Since Java is statically typed, we need to determine the type for variable declarations that doesn't have a type based on the value it is assigned. Luckily, the symbol that this identifier AST node stores will contain the necessary type information after typechecking (we updated the type of symbol with the type of the expr). We always loop through the LHS id list and the RHS expr list to check for multiple declarations.

```
 var x, y = 5, "hi"  (GoLite)
------------------------------
 int x = 5;          (Java)
 String y = "hi";
```

For declarations that contain both the type and expr, we will simply print out the type and expr.

```
 var x int = 5       (GoLite)
-----------------------------
 int x = 5;          (Java)
```

10.1.2 Type declaration

Type declarations are ignored in Java becuase Java doesn't allow for user-defined types. But because GoLite has this feature, every time when a type info needs to be emitted, we have a function, *varTypeName()*, that will find the type's underlying type and convert to the corresponding type in Java. Ideally, we should have stored the underlying base type to the symbol during the typechecking phase. But we didn't leave enough time for ourselves to refactor our code, so we just left it like that (definitely something we can improve on!).

Base Type

Base types in GoLite are converted to Java as follows:

```
 GoLite             Java
----------------------------
 int/rune           int
 float64            double
 bool               boolean
 inter/raw string   String
```

Although rune can be converted to char in Java, we thought it is easier to convert it to int instead, since GoLite prints out rune literals in their ASCII code and int can be used with char interchangeably.

Struct

As mentioned before, structs are represented by Java classes. We take the struct class list stored in the root AST node and construct Java class for each one of them. Every field in the struct (blank fields ignored) is a public class field and implicitly initialized to its default value. The name of the field is prefixed with *__golite_* to avoid keyword restrictions in Java.

Every class has two methods that override the default methods provided by the Object class. One of them is the *equals(T o)* method which returns a boolean true if every field of the current object is equal to the compared object, and false otherwise. If the field is of primitive type, '==' operator is used. If the field is non-primitive, like String or Object, 'equals' is used to compare. Note that if the class doesn't have any fields, *equals()* automatically returns true. The *equals(T o)* method is used for the binary comparison '==' and '!=' for two struct types. To dynamically fill in the equals function that will compare all the fields of the struct, we first traverse through all the fields and count the number of fields. Then on the second traverse, we connect all the field comparison expression with operator and return the result.

The other method, *clone()*, is used to deep clone the current object and return a copy of itself. The returned copy is a new instance of the current class and we populate its fields with the values of the current object. If the field is of non-primitive type, then we call the field's clone method. The *clone()* method is used to deep clone function arguments and return values (to be explained later).

Array and Slice

We initially used Java's build-it array to implement GoLite array structure. But we soon realized there are two major differences between the two. In GoLite, every element in a string array is initialized to empty strings but in Java it is filled with null values. In addition, GoLite arrays are deep cloned in assignments, function args and return values, which is very different from Java's shallow copy notion. Besides, when it comes to handle mixed type of slice and array, we have to deal with array of object, or objects that hold array of objects, which will easily cause confusion and increase the amount of work.

With all the problem in hand, we come up with the question: Is there a universal solution for both Array and Slice? In the end, we build our own customized array and slice data structure in java. First, we observed that there are many similarities between Array and Slice, such that they all have operations to update, access element, get size and capacity, using clone and equal functionality etc. Essentially, they are just lists with different design. So we can construct an abstract class called Golite_List to encapsulate all the functionality needed. The actual class Golite_Array and Golite_Slice extends the list and have their own behaviour implemented. Now, all the operations are formulated and it also provides a way to implement mixed type by treating both types as lists.

The two concrete classes have some similar setting. Each class instance has two ArrayList, one to store sub-list as a intermediate node and one for base type at the bottom layer. For intermediate node, its base-type list will be NULL but sub-list list is used, and vice versa.

As for the construction, it has the following required arguments

$Golite\_List < T > (depth, dim, sizes, baseType, root)$

For the starter, depth has the same value as dim, which is the dimension of the list we want to construct. Then we pass in a int array that has the same size as the dimension, each of the slot represents the actual size for each level. For example, a array [5][2][3]int would have size of 5, 2, 3, and a slice [][][]int would have size of 0, 0, 0. In the constructor, until the depth is 1, we fill the ArrayList with as many Golite_List as the size of current depth, if the size is 0, we fill in with a new Slice instead of Array, and each of the Golite_List will construct itself with depth - 1, same dimension and size array. With the correct depth value, it takes no effort to find the size for each sub-level. Besides, each Golite_List also has a reference to its base type class for instantiating base type in generic setting, as well as a reference to the top most level. When depth is 1, we are at the bottom layer, if base type class is of Integer or Float, we initialize the element to 0, or to empty string if the base class is String.

Both Array and Slice have the same function to set an element and get an element, for intermediate nodes we can set a list or get a list to modify the ArrayList. What is different is that for Array it doesn't make sense to add an element, thus no actual behaviour is implemented. On the other hand, Slice involves more complicated implementation for adding a list because we need to expand the capacity and clone the element. Essentially this operation is equivalent to the built-in Append operation.

The last part of the class is the clone and equal function. Both are implemented in a recursive way, by calling the clone or equal function of the object stored in the ArrayList. In the clone function, in the newly constructed object, we fill the ArrayList with each element's clone, and synchronize all the necessary variable such as size and capacity. In the equal function, we compare two object's size and capacity, as well as calling the sub-level's object's equal function repeatedly.

After a series of design and implementation, we successfully created our own representation of Array and Slice. Now we can convert all the Array and Slice related expression into Java code. Due to the lack of time, at first we only tested around base type like int or string, and because we didn't do enough testing, some edge cases are not fixed until the solution cases are released.

10.1.3 Function declaration

If the function's name is a blank identifier, then we simply ignore it in our emitted Java code since it can never be called.

The main function in GoLite is directly translated to the `public static void main(String[] args)` function in Java. GoLite allows multiple init functions which will be executed in the order they are declared before the main function is executed. Since Java requires unique function names, we append its name with a counter. Inside the Java's main class, we will call the init functions one by one in the same order before executing any statements inside the main function.

Blank parameters in functions are not ignored since function calls will include the arguments. Since there can be multiple blank parameters to a function, we will use the same blank renaming scheme described earlier. Function return type is already stored in the symbol, so we can print out the type easily. If the function doesn't have a return type, Java requires the function signature to denote "void" as the return type.

10.2 Statement

10.2.1 Print

A print statement has two kinds: println and print. When we process the statements, the generator will go through all the expressions that needs to print, and for each of them, perform operations based on the type of the expression. For example, if the variable is of type float64, then we format to 6 floating point in scientific notation. Print statements are converted into "System.out.print()" in Java. Integer values in different bases are already convert to base 10 in the scanner phase so nothing is done in the codegen phase. As for println statements, all expressions are separated by a space and a newline is added at the end. To do this, we convert println statement to multiple instances of print statement, for all the expressions in between, a space is followed, and we print a new line once we reach the end of the list.

Printing interpreted strings involves a bit more work since there are two escape sequences, `\a, \v` that are not supported in Java. To print the bell and the vertical tab, we would print their corresponding ASCII values and convert it to char. We coded a method in Java, *printEscapeChar()*, to print interpreted strings and to handle this conversion of escape sequences.

```
 print("hi \a")                   (GoLite)
---------------------------------------
 System.out.print("hi ");         (Java)
 System.out.print((char)7);
```

10.2.2 Switch

In GoLite, there are several kinds of switch statement. At first, there could be an optional simple statement. If there is a simple statement inside switch condition part, we put this simple statement right before our switch in Java.

For switch condition, there are three cases. The first case is "no condition". If there is no switch condition, it turns out that missing switch expression means "true". Then we convert switch statement into if statement. For each case, we create a corresponding if statement using the original case expression.

```
   switch {        (GoLite)
       case a:
       case b:
   }
----------------------------
   if (case a)     (Java)
   if (case b)
```

In GoLite, each switch case is auto-break, which means each case must have a break. Plus, we are allowed to have some statements after break statement which is not legal in Java. Therefore, we decide to use try catch block and exception to replace the break statement in GoLite. It should be noted that there will be an unreachable error if we throw new exception directly. To solve it, we put our exception statement into one if statement.

```
    switch {        (GoLite)
        case a:
            stmt 1;
            break;
            stmt 2;
        case b:
    }
----------------------------------
    try {            (Java)
    if (case a)  {
        stmt 1;
        if (true) {
            throw new Exception();
        }
        stmt 2;
        if (true) {
            throw new Exception();
        }
    }
    if (case b) {
        if (true) {
            throw new Exception();
        }
    }
    } catch (Exception e) {
    }
```

The second is bool type condition. If the condition is in bool type, we also implement if statement to replace the switch statement. However, the switch condition could not be boolean in Java. We need to change the original case expression. In this case, we also use try catch and exception to replace break.

```
    switch (true|false) {        (GoLite)
        case a:
        case b:
    }
-------------------------------------
    if (case a == true | false) (Java)
    if (case b == true | false)
```

Each switch could have a default case. Once there is no case match found, the default case will be executed automatically. Therefore, we set a global variable called caseFound. The default value of caseFound is 0. If we find one right case, we set caseFound to 1 and we will skip the default part. In the example below, stmt 3 and stmt 4 will be executed. It should be noted that even though switch has auto break in GoLite, we decide to not add auto break to default part. It is not necessary to have a break in default case, since it is the last part to be executed anyways.

```
    a := 1;
    switch (a) {        (GoLite)
        case 2:
            stmt 1;
            stmt 2;
        default:
            stmt 3;
            stmt 4;
    }
-------------------------------------
    int a = 1;          (Java)
    switch (a) {
        case 2:
            stmt 1;
            stmt 2;
            if (true) {
                    throw new Exception();
                }

        default:
            stmt 3;
            stmt 4;
    }
```

The third case is expression condition except bool. In this case, we decide to stay with the original switch statement. There is no need to change it into if statement.

```
    switch (exp) {        (GoLite)
        case a:
        case b:
    }
    ------------------------------
    switch (exp) {        (Java)
        case a:
        case b:
    }
```

There is an alternative method to implement switch break into Java. We still convert switch to if statement. We could use a while loop with only one iteration and put our if statement inside. Then, we do not need to use try catch block and exception, instead, we could use break in Java directly. Since the running time of exception handling in Java is pretty long, this method could reduce the processing time.

10.2.3 If & Else

The implementation of if and else statement is relatively easy. The format of if and else statement in GoLite is pretty much the same as that in Java. The main difference is the optional simple statement and its scope. There could be a simple statement in the expression of if and else if statement in GoLite. We decide to transform else if statement into an else statement with a nested if statement. Therefore, the optional simple statement could have the same scope as the if statement.

10.2.4 Increment & Decrement

Increments and decrements of a single value identifier is easy. However, if the expr is an index access to array or slice, eg. `a[0]`, since our arrays and slices are implemented as lists, we cannot just simply do `a.setList(0)++`, since this syntax is not allowed. We need to first store the previous value into a tmp variable, perform inc/dec on the tmp variable, and store back the value of the tmp variable back to the element. Another complication is that the index can be a function call which has side effects. So we need to store the index into a tmp variable as well.

```
    a[func()]++                   (GoLite)
------------------------------------------
    int index = func();           (Java)
    int tmp = a.getElement(index);
    tmp++;
    a.setElement(tmp, index);
```

10.2.5 Assignment

In GoLite, we are allowed to put multiple assign statements in one assignment. To generate code in Java, we need to separate it into several single assignments.

```
    x, y = 5, "hi"  (GoLite)
----------------------------
    x = 5;          (Java)
    y = "hi";
```

However, `x, y = y, x` is allowed in GoLite so to perform the switch correctly, we create tmpX and tmpY and we store the original x, y values into these new tmp variables. We decide to apply it on every assignment. Therefore, we do not need to distinguish if it is swap or not, which saves a lot of work. Note that assignment to a blank variable is not ignored since the RHS expression may produce side-effects. But since blank variable cannot be referenced, after storing the RHS value into a tmp variable, we don't need to store that value back to the blank variable.

```
  x, y, _ = y, x, 5     (GoLite)
----------------------------
  int tmpX = x;         (Java)
  int tmpY = y;
  int blank0 = 5;
  y = tmpX;
  x = tmpY;
```

If the variable being assigned to is of type struct, array or slice, we need to call *clone()* since GoLite's assignment for non-primitive type performs deep cloning.

Another complication is that we have two methods for array/slice indexing, a setter and a getter. If the LValue is on the RHS, a getter is used and if the LValue is being assigned to, a setter is used.

There can be different assignment kinds such as `+=, -=` and etc. Most of the assignments can be directly translated into Java, except one, `&^=` (AND NOT) since NOT in Java is denoted by `~`.

```
  x &^= 5          (GoLite)
----------------------------
  x = x &(~5);     (Java)
```

10.2.6 Short declaration

Short declaration is essentially a combination of assignment operation (if the variable is already declared before), or a var declaration (if it is a new variable). To distinguish if the variable is already declared or not, we use the *declaredBefore* boolean field in the AST node's symbol.

```
  var x int = 0
  x, y := 5, "hi"  (GoLite)
---------------------------
  int x = 0;       (Java)
  int tmpX = 5;
  String tmpY = "hi"; //declaration
  x = tmpX;           //assignment
  String y = tmpY;
```

10.2.7 While & For Loop

In GoLite, a while loop without condition means infinite loop, so we directly convert to a while-true statement. When a while loop has condition, its condition will be converted into the condition for java while loop, as well as the body. The tricky part comes with the for loop as in multiple pre and post conditions are allowed.

We decide to convert for loop into while loop. For the pre-condition, we declare all the variable before the loop as temporary variables with unique names. The second part of the for loop condition will become the actual condition of the while loop. Inside the loop, we first use a try-catch-finally block that catch any exception occurs. In the try section, we will execute the body of the loop. If any exception is caught we will do nothing and finally proceed the post-conditions that exist in the finally section. By this mean, no matter what, the post-conditions will be executed and we may also have multiple variable defined for our loop.

10.3 Expression

10.3.1 True and False

To allow redeclarations of true and false values in Java, as described in the symbol stage, the constant true/false symbols are flagged with `isConstant = true`. If we encounter an identifier AST node with name of `true/false`, we check if the symbol it contains is a constant or not. If it is, then we simply emit `true/false`, else we need to use the symbol's *renamed* name in the codegen phase since it shadows the true/false values.

```
  var a bool = true      (GoLite)
  var true = false
  var b = true
---------------------------------
  //this true is a constant value
  boolean a = true;
  //this true's value is actually false
  boolean __golite_true0 = false;
  boolean b = __golite_true0;
```

10.3.2 Binary and Unary operation

Most of the GoLite operators can be directly translated to Java. However, since the operators in Java are for primitive types, we need to use special methods on non-primitive types. For `==, !=`, if the compared objects are of type string, struct, array or slice, we need to invoke *equals()* for equality checks. `<, <=, >= , >` on string type is converted to Java's String *compareTo()* method. The `&^` operation is translated to `&~`.

Unary operations is easy to implement since they are all supported in Java. Java uses a different character for NOT, so we just need to change GoLite's `^` to Java's `~`.

10.3.3 Array and Slice access

When accessing a element in an Array or Slice object, there are two possible scenario: writing or reading. Normally, in goLang, being written or read make no difference in terms of the syntax, however, in Java, our design doesn't support general syntax like "x[][]" because we are using class and object. We have to use function call on the object to get data and assign data. We wrote two helper functions to handle writing and reading separately. When processing any short declaration or assignment statement, we check if the left hand side expression is of type slice/array accessing. If so, we process the left hand side by calling the writing helper function, it will generate the code to update the data in the slice/array. The same concept apply to the right hand side as we call the reading helper function to generate the right code.

How does the helper function work exactly? By passing the expression, we have to process it, recursively traversing towards the base type, and recording the current depth of dimension. After reaching the base type, we return the difference between the whole dimension and current depth. If the difference is 0, it means we are at the last depth, and we should output "setElement(...)" or "getElement(...)" for writing and reading respectively.

```
var x [5]int

x[3] = 5            (GoLite)
----------------------------
x.setElement(3,5)     (Java)

y = x[3]            (GoLite)
----------------------------
y = x.getElement(3)   (Java)
```

As the example demonstrates, variable x has 1 dimension, and when we read x[3], we count the number of [..] as the current depth. The difference between two dimension is 0 in this case, so we know it is setting or getting a element instead of a list.

If the index passed is more complicated, like a struct's field or return value of a function, the generator will recursively process the expression and convert the index expression to its Java version.

Earlier, we did not include the struct and class case because we were using strcat and our function to process expression directly print to the file using fprintf function. So either we write the same code in two places or we need to change the function implementation to print to the file directly. In the end, we fixed this by directly print all the generated code to the output file.

10.3.4 Struct field access

All the fields in the java class are made public (no getter and setter involved). Thus to access a field of a certain object, it is simply using the `.` operator plus the field name(renamed version).

10.3.5 Type Cast

Typecasting in Java is easy for primitive types since it is essentially calling `(type)value`.

For string type casting, we need to invoke String class's *String.valueOf()* method so we can convert any primitive types to a string. One exception in String casting is to cast a integer or rune to string. We need to convert the integer value into a character first using the keyword *char* before calling *String.valueOf()*.

```
var s = string(65)  //s = 'A' in Golite
-------------------------------
String s = String.valueOf((char)65);
```

10.3.6 Function call: Return-by-value and Pass-by-value

On of the major difference between GoLite and Java is that Java uses pass-by-reference and return-by-reference for all non-primitive types. This would mean that for every non-primitive type argument passed to the function, we would have to first clone it so any operations on them inside the function is on the cloned copy and thus would not impact the value of the original arguments. The same concept goes for the return value of the function. For all non-primitive type return values (array, slice, struct), we would return their clones by invoking their *clone()* function.

10.3.7 Built-in functions

There are three built-in operations:

Len(): In goLang, Len() targets data with type String, Slice or Array. For string, Java has its own API to query the string length, so we convert len() into Java's syntax of string length. As for Slice and Array, since we designed our own class structure for these, we also implemented functions in the class that return the length. Since the base type of our Slice and Array class is ArrayList in Java, then we can simply return ArrayList's size as the length of the current array / slice.

Cap(): The concept of Cap is not common in Java environment. For Array object, its capacity essentially means its length of array, so we just return the same data as above. For Slice object, the capacity starts at 0, when the first item is added to the slice, its capacity becomes 2, and expend itself by 2 every time the limit is exceeded. Whenever we have this situation, we need to create a new object with the same content but twice bigger the capacity. To do this, we use clone method to clone a exact copy and return its reference. Because Slice is very similar to ArrayList in Java, we do not need to worry about how much space we need to create for the new Slice, instead, we just need to keep read all the copies and track the right capacity, then store it to a variable in the class.

The reason we need to do a clone is because the old object after expanding the capacity might still be referred by other variable. However, changing the new object's content should not affect the old object. Therefore, we need to do a deep copy instead of simply linking the address.

Append(): Append operation is also encapsulated in the Slice class. Based on our own implementation, the added element can vary based on the index. In our own Slice class, adding a base type element would be easy with ArrayList. However, if we want to add another Slice or Array object, we have to add to a different ArrayList that doesn't hold base type object but List. So based on the type of the appending element, we need to call its corresponding function.

```
var x [][]int
//append a base type element(int)
append(x[2], 3)
var y []int
//append a list
append(x, y)
----- convert to Java -----
...
//append a element
x.getList(2).addElement(3);
...
//append a list
x.addList(y);
```

In order to find the correct function to call, we have to process the first argument of the append function. First we find its current dimension, and compare to the variable's original dimension, if they are the same, then we are accessing the very bottom of the object, and we want to add an element to the slice. If less than the base dimension, meaning we are appending a list.

11 Testing

The testing strategy we use is testing while coding. We first come up with our own implementation. After it can be fully compiled, we write our own test cases. Then, we use the given test cases from lecture slides to test if it works. For each case, we compare our result and the result from reference compiler. For scanner, parse, symbol and type checking, the results are exactly the same. For code generation, even though the target languages are different, we still could see the logic behind and check if our implementation works fully or not.

In our early stages, we could finish our implementation and come up with more test cases, especially some edge cases. In code generation, it is more time intensive. Therefore, we do not have sufficient time to do lots of tests. Instead, we only have time to do all sample cases from lecture slides.

12 Conclusion

This compiler project for GoLite is one of the largest projects we've ever accomplished at McGill. Throughout the course, we've learned the 7 main stages of a compiler and we are guided well to write our own compiler. We all learned a lot in this project. In this project, no required language is given which means we could choose our own implementation language and target language. In our case, we choose C as our implementation language and Java as our target language. Because of this project, we've done a lot of programming in C which makes us be more familiar with C. Also, We all have deeper understanding of the syntax of both Go language and Java language now. Moreover, except the sample test cases in slides, there is rarely test cases provided for us. In another word, we need to brainstorm a lot and think carefully to make some development decisions at every stage of developments. In our case, we've changed our design a lot during the process. This project turns out to be very challenging and time consuming but also very rewarding. For our future changes, we currently use symbol to store our type information which could be simplified by storing the type information directly. Also, in the code generation of switch statement, we use try catch block and exception to replace break statements which are not optimized in Java and will slow down the running time of our emitted Java code. In the future, instead, we would like to use while loop and break to emit more efficient code.

Another thing is that we learned that commenting our code is very important in collaborative projects. Due to the long time span of this project, sometimes when we look back at the code we wrote for the early milestones, we would forgot what a block of code does or another teammate would tweak the code a bit for bug fixes and you wouldn't understand the code afterwards. This caused quite an issue because eventually we are reluctant to change or optimize our previous code because we don't understand the code completely and afraid it would cause more bugs later.

Overall, we've done a lot and we've learned a lot in this project. It is all worthy.

13 Work distribution

We hold all decision discussion together and then distributed work equally. We worked on AST together and use the CodeCollab tool to make sure everyone agrees on the design. For the other parts, we have one person work on it and two other people check afterwards. Such work flow provides the lowest error tolerances and great balance between working together and individually.

Bessie Luo

- Scanner (first draft)
- Parser (revised)
- Pretty (revised)
- Fix errors from Milestone1
- Symbol (first draft)
- Type Check (revised)
- Codegen(Scoping, Struct declaration, binary operation, Var declaration, Function declaration, inc/dec)
- Test
- Report

Lancer Guo

- Scanner (revised)
- Parser (first draft)

- Weeder
- Fix errors from Milestone1
- Symbol (revised)
- Type Check (first draft)
- Codegen(while loop, slice/array structure, built-in functions, array/slice/field access, print statement)
- Test
- Report

Charles Lyu

- Scanner (revised)
- Parser (revised)
- Pretty (first draft)
- Fix errors from Milestone1
- Symbol (first draft)
- Type Check (revised)
- Codegen(Switch statement, If statement, Assignment, Short declaration)
- Test
- Report

14 References

Some of the code for the symbol table is directly taken from the course slides. Also we referenced the JOOS example in the github to take inspirations and the C++ code generated by the reference compiler to give us a rough idea of how to implement the code generation.