COMP 520 GoLite Final Report

Yiyang Liu, Yue Lyu, Chunyao Zhao

May 2020

1 Introduction

Go is a programming language with many interesting features. Go doesn't have classes, which means it is not object oriented. Moreover, in Go, the variables, types, and built in constants are all defined as identifiers in syntax. Unlike most of the nowadays popular languages, it doesn't have reserved key words for basic types, which increases the difficulty of designing a compiler for it but will give users extra flexibility for type define and renaming. Go also has blank identifiers and auto semicolon insertion in compile time. The short declaration is supported by Go, which is not common in other popular programming languages. The if statements and switch statements can have init statement, which has an independent, intermediate scope. GoLite is a subset of Go, with simplified features. In GoLite, import statements are not supported. The slice literals, array literals, and struct literals are not supported, so that we could initialize them by appending, indexing, and field selecting.

We implemented a fully worked compiler for GoLite, which was composed of mainly 6 functional parts including scanner, parser and AST tree, weeder, symbol table, typechecker, and c code generator. It checked the syntax and all compile time errors and then generated the output program in C++.

In this report, we will discuss the language and tool choice of our project, summarize in detail the design decisions we made in each component of the compiler and explain why we do so, and finally talk about the flaws of our design or implementation and how we want to change them.

2 Language and Tool Choices

For this project we chose Flex, Bison, and C as our programming tools. The reason why we choose Flex and Bison is because each of us has practiced these tools a lot in class and previous assignment. Moreover, C worked coherently with Flex and Bison. C is a low level programming language, as most of operations that are written in C uses less machine code instruction and thus runs very fast, so implementing our compiler by C would guarantee fast runtime performance. The struct feature in C is object oriented, and it helps us to construct AST more efficiently, which is critical to our project. Meanwhile, In this project, readability of the code is very important to us since we rely heavily on the abstract syntax tree in most of our complier phases. To improve our collaboration, we decided to name every attribute, node type, and nodestructure according variable names in parser.

For target language of out code generator, we chose to finally output our compiled programs in C++, which resembles GoLite a lot and has strong functional libraries that provides higher flexibility. C++ has some desired features like operator overloading, type definition, etc, which helps us to generate target codes efficiently.

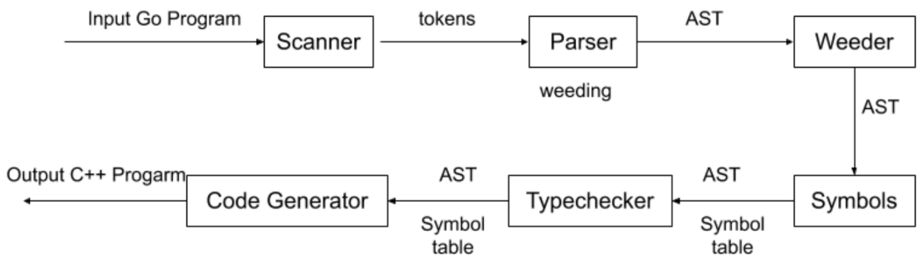

Figure 1: The execution diagram and functional components of our compiler

3 Scanner

3.1 Overview

By virtue of Flex toolchain, we implemented our scanner by listing all the regular expressions for all possible tokens with enforced matching priorities. The reserved keywords had highest priority, followed by operators, literals, identifiers, strings, and comments. The definitions of regular expressions for strings and comments silently solved open quotation marks and nested block comments problems. If none of the above regular expressions were matched, our compiler would report an error with unexpected characters. The comments were ignored by our compiler and wouldn't go into later phases.

3.2 Major design decisions

The functionality of scanner mainly relied on the definition of regular expressions for each kind of token. For the reserved key words, the scanner returned specific key word tokens. For literals, the scanner returned the token according to the literal type (stirng, rune, int, float64) and sotred the values inside. Decimal, Octal, and Hexadecimal numbers were all outputed as int token, which sotred the equivalent decimal value inside. There were no reserved key words for constants, types and variable names, so we output them all as identifier token. In the following section we specified some particular issues that we considered to be harder than others.

3.2.1 comments

For comment we have two rules. The line comment was matched by a double slash "//" then rule out the newline character. Every other character in this line was accepted by the scanner.

The block comment was matched by a "/*" and a "*/" on both ends and in the middle accept every thing except the "/*" and "*/".

3.2.2 string literals

Strings were divided into two rules. The raw string were surrounded two single quotation mark. The content were accepted by ascii characters from "@" to "_" and "a" to " ", without the predefined escape sequences such as newline character. The interpreted string were surrounded by double quotation mark. The content were accepted by unicode values and bytecode values.

3.2.3 semicolons

For semicolon insertion we implemented a method that was able to look back to its last token upon seeing a newline character. If the last token was an identifier, break, or literal, etc, according to GoLite rules, we insert a colon into the token stream.

4 Parser

4.1 Overview

We defined a left recursive context free grammar for GoLite. The terminals were constituted of the set of tokens that could be matched from scanner. We added variables corresponding to different language structures in GoLite. We also defined associativity and precedence of our operators to resolve the conflict caused by ambiguous definitions of our grammar.

4.2 Major design decisions

4.2.1 Associativity and precedence

Our program is left associative. The precedence level from high to low were defined as following:

- unary operations: Uneg, Upos, Unot, Uxor

- binary operations: Mul, Div, Mod, Lsh, Rsh, Bad, Bcl

- binary operations: Pls, Min, Bor, Bxr

- comparison

- and

- or

4.2.2 If statement structure

Our if statement node was consisted of a short declaration, a if condition, if body, and most importantly an else-block. This else-block was empty when there was no else branck; it contained another if block for an else-if statement; or it contains an else block for a normal else. This grammar rule would help us to resolve conflict in parser.

4.3 Specification for particular issues

This part we addressed some peculiar issue arose during our debugging process. The matching rules caused conflicts but wouldn't influence our compiler function.

4.3.1 Primary expressions

This rule matched multiple primary expressions together. However, since identifier was a subrule of primary expression and it can match also to identifiers, this rule would result in a reduce/reduce conflict.

4.3.2 Simple statement

A simple statement can consist of an assignment, an op-assignment, or a short hand declaration. The all have left hand side expression and a right hand side expression so they are parsed together. In particular, the assignment can have a left hand expression as some primary expressions except function call or some identifiers. So we put both rules inside the parser. This would result in a reduce/reduce conflict and a shift/reduce conflict.

5 AST

5.1 Overview

We originally designed the abstract syntax tree as a uniformed tree structure and stored all information as different pointers inside the big tree structure. To clearly separate and group the different tokens, we assigned each node pointer in the AST a name and each node a kind. We searched for the information by their name and kind. However we soon discovered its weakness and switched to separate nodes. We implemented 5 node types in the AST: program, declaration, statement, type, and expression. But the naming system stayed in our code to help us identify different nodes. It had pros and cons. On one hand we can read the code more efficiently and the information were hard to mess up so we didn't need to particularly worry about a method may contaminate the tree node; on the other hand the code is prone to errors and hard to maintain, the information stored in such structure is hard to find.

5.2 Tree structure

The tree conists of five different nodes and each node has several enumerating kinds. For example Exp node had binary expression kinds, unary expression kinds, function calls, identifier kinds, etc.

In the part of our AST that matches multiple sentences resembles the structure of a Huffman tree. For example, each statement node has two children: the right child points to the current statement, and the left child points to the lists of statements comes before it. The same kind of Huffman tree structure can also be seen in switch statement nodes. We implementation the AST this way just to conform to the left recursive structure of our parser. In the expressions part of the tree, there are some nodes named option in the tree. They are set up to improve the readability of the code so that we can easily determine the expected quantity of the expressions since it can cause unexpected behaviour of the parser in certain conditions.

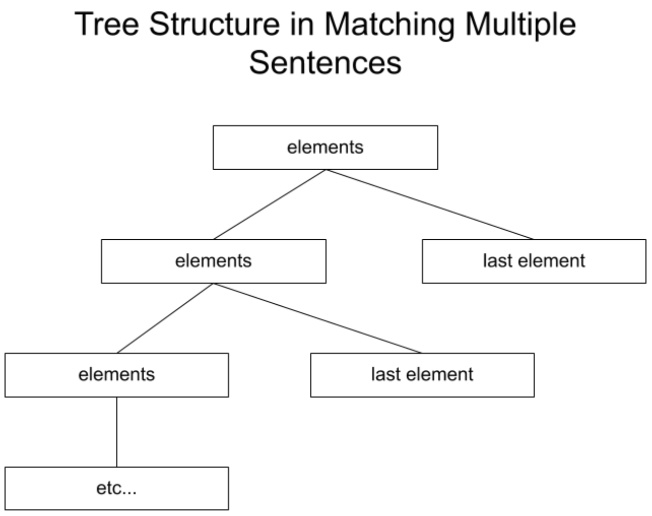

Figure 2: Tree structure in matching multiple elements

5.3 Tree traversal

Traversing the AST was the only way of searching for semantic information, so it was the most essential part of the compiler. Since we separated the nodes, we cannot traverse the tree by a single method starting from the program node. So for each type of node we designed a method to traverse. The methods recursively called on each other to traverse. For example, when encountered a variable declaration, the traversal method of "Decl" node would call on Type node traversal method and expression node traversal method. In later phases, we always traversed AST in dfs mannar, which allowed us to visit nodes corresponding to the definition order of input Golite program.

6 Weeder

Our weeder traversed the AST in and after the parsing phase to weed out the illegal codes that the parser can't distinguish.

6.1 Weed Default

According to GoLite's grammar rule, a switch statement can have only 1 default case. We created an global variable to remember the number of default cases in a switch statement. This global variable is set to zero each time a switch statement is encountered. When traversing the switch statement, the global variable increases by one every time a default case is encountered. After finish traversing the switch statement, if the value of the global variable is greater than

1, then an error message is printed.

6.2 Weed Break

According to GoLite's grammar rule, break statements can only exist in the scope of a for statement or a switch statement. We created a global variable that increases by 1 when a for scope or switch scope is opened, and decreases by 1 when the scope is closed. Every time a break statement is encountered, we check the value of that global variable. If the value is equal to 1, the break statement is in the scope of a for loop or a switch statement. If the value is greater than 1, the break statement is in the scope of nested for loop or switch statements. If the value is less than 1, the break statement is not in the scope of a for loop or switch statement, and an error message is printed.

6.3 Weed Continue

According to GoLite's grammar rule, break statements can only exist in the scope of a for loop. We implemented this in the same way as above, except that the global variable changes only when the scope of a for loop is opened or closed.

6.4 Weed Return

According to GoLite's grammar, if a function has return type, we must guarantee the function would return and there were no statements (unreachable statements) defined after return statement. The returnWeeder took a function declaration as input and recursively traversed its block body. Only for if-else statement that contained a else branch and contained a return statement in each breach would gaurantee a return in the given scope. Otherwise, only a return statement at the end of scope would guarantee a return. We recursively check the block body of the function and the sub-block bodies. We weeded out programs that didn't have a guaranteed return statement in the scope of its block body.

7 Pretty printer

Our latest version of pretty printer was a basic traversal of the AST. The printer printed different tokens corresponding to the node it saw on the AST. The implementation of pretty printer was mainly used for debugging, and it doesn't have specific effect in the work flow of our compiler.

7.1 Curly brackets

We decided to the printing the curly brackets would have been done by the printer only when block statement is invoked. That way we would not receive many undesired brackets when printing.

7.2 parenthesis

Unlike curly brackets that are sufficiently well with only one node printing it, we decided most of the expressions should be surrounded by parenthesis to ensure the correct precedence level in the printed file.

8 Symbol table

8.1 Symbol table structure

For implementation of symbol, we constructed a struct that stored the symbol name, declared type, and symbol kind. The symbol kind could be function kind, variable kind, or type kind, which indicated the symbol was for a function declaration, variable declaration, or type declaration.

The symbol table was structured as a cactus stack, with symbols stored inside using a hash map, and a pointer pointing to its parent, and a pointer array pointing to its children. When looking up a symbol, we would traverse through the symbol tables from the current table and all its ancestors until the root table. If a variable was already declared, an error message would be printed and the program exited with 1.

When printing the symbols, we traversed the symbol tables from the root using dfs. At each symbol table, the scope and all the symbols inside were printed. However, at each scope, there was no guarantee on the ording of symbols since symbols were stored using hash map.

8.2 Type in symbol table and type checker

In the design of our AST, we defined an *identifierType* node that could be matched to any possible types. Since type node is the structure we are extensively using for type checking, We simply stored the type node as a type for each symbol in the symbol table.

When comparing two types, we recursively checked each field pointer and stroed informations. If two structs had exactly the same stored informations, they were evaluated as equal.

We also implemented a method for resolving a given type. Given the specified type, we searched the symbol with the same name from the symbol table of current scope to root. Then we recursively called this method on the underlying type stored in the symbol. If the input type was base type, struct type, array type, or slice type, the type itself was returned as the underlying type. For all defined types, if we can't find the symbol with the same name, an undeclared type error would be printed and the program exited with 1.

8.3 Scoping Rules

To define the built in constants, we created an initial scope of our symbol table, in which we pushed all built in constants and base types. This enabled defining variables in Golite that are aliased with built in constants and base types. Then, we opened a new child scope which contained the symbols for the actual GoLite program, including package declaration, and top-level declarations.

We opened a new child scope every time an opening curly bracket was encountered, and returned to parent scope when the closing curly bracket was encountered. It was important to notice that function header was stored as a symbol in the current scope, while function parameters and function body belonged to the child scope.

We opened new child scope for if, for, and switch statement. The init statements of them belonged to child scope,, and the content inside their block bodies belonged to the grandchild scope

8.4 Detailed Design Decisions for Symbol Table

8.4.1 Declarations

- Variable declarations:

The symbol with variable kind was constructed and added into current symbol table when a variable declaration node was met. The variable name and type were stored into the symbol. If a symbol with the same name had already been in the symbol tables, an redeclaration error message was returned. If the type of the variable was not declared in the declaration, it was automatically inferred by our type checker.

- Type declarations:

Defined types were made as a symbol of type kind, in which the type name and the underlying type were stored. Symbols for base types were created and added in the root symbol table by default before typechecking. If a symbol with the same name had already been in the symbol tables, an redeclaration error message was returned.

- Function declarations

We stored function declarations in a symbol with the function name as symbol name and a struct of input type list and output type as symbol type. We also check for special functions "main" and "init" here and ensure that they have neither input nor output requirements. If a symbol with the same name had already been in the symbol tables, an redeclaration error message was returned.

8.4.2 Statements

- Short declaration:

When a short declaration statement is encountered, we first search in the symbol table to see whether there are undeclared variables in its identifier list. If not, an error message is printed. (see the later half in 4.3.3)

- Block: We scope the symbol table when encountered a new block and unscope when block is finished.

- For, if, and switch statements:

For these statements, they might have a simple declaration (i.e. the init statement) before their conditions. We open a child scope in symbol table just to store the variable declaration that may happened in the init statement. By doing so, we both allowed the init statement to redeclare existing variables, and enabled the variable declared in the init statement to be shadowed by declarations that occurred inside the for/if/switch's block body.

9 Type checker

The type checker was used to infer expressions types and made sure all expressions were well typed. It took a Exp node as input and output the inferred type for the node. It would report an error if the expression didn't typecheck while it was inferring the type for the given expression.

9.1 Type equality

We compared two types by recursively checking each field pointer and stroed informations.

9.1.1 struct type

For struct type, if two structs had exactly field and same stored informations, they were evaluated as equal.

9.1.2 array and slice type

For array and slice type, if two slices had the same underlying type, they were evaluated as equal. For arrays however, they also had to have the same size.

9.1.3 identifier type

For identifier types, this included all types that could be defined by users and the base types. Initially, we only checked for the name of the type for identifier type equality. However this method could not solve the issue of overshadowing types. Specifically, when two types were defined in different scopes, even though their names might have been the same, they cannot be evaluated as same. So we modified the comparison method to support this feature of Go. We need to compare the line number of the type before evaluating them as equal. This way we would have the information as to where this type was initially defined. We also designed a *traceType* method to locate the type definitions in the symbol table. This function would trace a type node and see where it was defined, change the type nodes line number to reflect that, and return the type node.

9.2 Detailed Design Decisions for Type Checker

9.2.1 Declarations

In variable declarations, we have 3 different cases. If the declaration has no expression as value, our compiler put x:T in the symbol table; if the declaration has no declared type but an expression as value, we would infer the type of value; if the declaration has both the declared type and an expression as value, we would infer the type of value, compare it with declared type. For multi-variable declaration we would also be checking the number of variables and number of expressions.

When type checking main and init functions, we made sure that they didn't have parameters or return statements.

9.2.2 Statements

- Expression statement:

When we encounter an expression statement node, we first check that the it's expression child is a function call, then check whether the function is well typed.

- Return statement:

In order to type check return statements ,we create a global variable called curReturnType, which remembers the return type of the current function. Every time a function declaration is encountered, the curReturnType is updated to the return type of the function. When we traversed the function body and arrived at the return statement, we could compare the the type of the its expression curReturnType to type check the return statement.

- Short declaration:

For the undeclared variables, we store them into the symbol with the type of the expression they are assigned. For the declared variables, we first search in the symbol table to find out the types of these variables. Then, we compare the types of the variables with the types of the expressions they are assigned. If all declared variables have the same type as the expressions they are assigned, then the short declaration type checks.

- Assignment:

First, we traverse the identifiers on the left hand side of the assignment statement, search each identifier in the symbol table, and find out their type to make an identifierTypeList. Then, we traverse the expressions on the right hand side of the assignment statement, infer the type of each expression, and make an expressionTypeList. By comparing each entry in the the identifierTypeList with the correspond entry in the expressionTypeList, we type checked the assignment statement.

- For and if statements:

For the condition expression of a for or if statement, we first resolve its base type, and then check whether this base type is boolean.

- Switch statements:

If the switch statement doesn't have an expression, we resolve the expressions of its switch cases and check whether they have the base type of boolean.

- Increment/decrement statements:

We first resolve the type of the operand to find its base type, then check whether the base type is an numeric type.

- Print/println statement:

If the expression child of the print statement is not NULL, we resolve each expression to find its base type, and check whether its base type belongs to int, float64, boolean, rune, or string.

9.2.3 Expressions

We built a function called inferTypeExp and some helper functions to infer and typecheck expressions. The function inferTypeExp would continue traversing the AST in dfs manner, typecheck each expression given the returned type of constituting expressions, and decide a type for the expression following the rules given in the specifications.

- Literals and Identifiers:

The types of literals were inferred by their node kinds in AST. Literals were stored as different kinds corresponding to their types based on our design of AST. The types of identifiers were inferred by querying the symbol table so that the type of first found symbol from current symbol table to root was returned.

- Unary expression:

We typechecked unary expressions by checking if the constituting expression can be resolved to specified types in specifications. If successful, the unary expression would be assigned the same type as the type of constituting expression. Otherwise, it didn't typecheck and an error was returned.

- Binary exression:

We first checked if the two constituting expressions were typechecked and had the same inferred type returned. Then, if the inferred type could be resolved to the specified types, the binary expression was well-typed and assigned the same inferred type or bool in comparison expressions.

- Builtin functions:

When reached builtin expression nodes in AST, we checked if the expression was well-typed and could be resolved to []T or [N]T (or string type for len(expr)). For append builtin, we also checked the second expression was well-typed and resolved to T, then the type of first expression was assigned and returned. For the rest 2 builtins, type int was assinged and returned.

- Field selection and Indexing:

They were typechecked strictly followed the rules in specifications.

- Function call and Type casting:

Type cast would be parsed as a function call in the parser. However, we can separate the two by querying the symbol table, and see matched symbol was a type kind symbol. If so, then proceed to typecheck following the rules for type cast. Otherwise, we typechecked it following the rules for function in specifications.

10 Code generator

10.1 Overview

Our code generator generated compiled GoLite program in C++. C++ is a statically-typed language pretty much similar to GoLite. It is relative high-level to C, so it provides us powerful libraries to manipulate string and arrays without struggling with memory allocation and pointer issues.

In our code generator, we basically traversed the AST tree and symbol table in the meantime, and defined how we should generate equivalent C++ code in our output file based on the kind of node we have met. The symbol table has been created during symbol table and typecheck phase according to our AST. Since the AST reserved the same in later phases of our compiler, we visit the children or parent symbol table at exactly the same places where the symbol table is scoped or unscoped. We also built a helper function to help us traverse down to the correct child symbol table in order. The traverse up function was not necessary since we used recursion to execute dfs on our AST and symbol table. The parent symbol table pointer was memorized and resumed when the child function call returned.

10.2 Detailed design decisions

10.2.1 Execution schema

In Go, top level functions were executed with inits in lexical order first followed by main. To preserved the order of execution order of Golite in C++, we renamed inits in GoLite by tagging a number from 1 to n after the name of init. The number was assigned from low to high based on the declaration order of inits. We also renamed function main in GoLite as main1. After we have translated all the top-level declarations of a given GoLite program in C++, we appended a main function with header *int main();* in our output C++ file. In this main function, all init*i*, where i is a number, were called in their lexical order followed by *main1();*.

10.2.2 Declarations

The same as GoLite, our code generator supported generating function declaration, variable declaration, and type declaration.

- Variable and function declarations:
For variable and function declarations, we translated them directly in C++ syntax.

To avoid declaring a variable with the name that was conflicting with keywords in C++, we renamed all variables (identifiers) by appending a tagged string to their names. For multiple declaration in the same line, we separated the whole declaration into multiple tuples of variable, type, value. We generated a one line standard declaration for each tuple in C++. It's important to notice that for variable declaration without initial values, we automatically assigned the valid default values followed the rules in Go.

- Type declarations:
We implemented type declaration in C++ using typedef.

- Blank identifiers:
We ignored the declaration of blank identifier, _, since they may never be accessed. We wouldn't generate any declaration statement for the blank identifiers. However, for the declaration of blank identifiers that included an assignment of expression, we directly generated the statement of that expression, which was allowed in C++. For blank identifier declared as function parameters, we assigned them different based on their order.

10.2.3 Types

- basic types:
We defined float64 in Go to float in C++. We defined rune in Go to int in C++, since the value of rune in Go is always interpreted as integer. We also defined string in Go to std::string in C++, which provided basic string operations and functions and easied our stress to handle with character pointers. For the other base types, they were defined the same as in Go.

- Composite types:

Arrays:
We utilized std::array in C++ to represent arrays in Go. We we met an indexing expression of an array, we called std::array::at function, which provided bound check for indexing the array. For array equality, == operator would compare the array for their contents, which was desired.

Slices:
We used std::vector in C++ to represent slices in Go, which handled automatic memory reallocation problem when the size of slice was changed. We we met an indexing expression of an slice, we called std::vector::at function, which provided bound check for indexing the vector. For array equality, == operator would compare the vector for their contents, which was desired.

Struct:
We used struct in C++ to represent struct in Go, which has similar syntax. For struct equality, we implemented a checkStruct function that helped to compare each field of two structs according their name, type, and assigned value. We ignored blank identifiers.

10.2.4 Statements

- Assignment:

We enabled multiple assignments in the same line. Our code generator would output multiple sigle line assignments for each one of them. We first stored the values of RHS expressions into temporary variables, and then assigned them back to LHS variables.

We generated an extra statement for expression on the RHS of blank identifiers.

When we met assignment of one array, slice, or struct to another we manually copied all the contents inside.

- Print statement:

We implemented the print statement in C++ using std::cout, which porvided various useful fomating mode. For bool expressions, we cout the expressions in std::boolalpha mode. For float expressions, we cout them in std::showpos and std::scientific mode. For raw strings, we first formatted them using the prefix R before we cout them. For integer, rune, and normal string, they were printed by cout's default formatting. We resumed the default formatting of cout after each print statement to gaurantee the correctness.

- Block Body:

When a block body is encountered, we switch from current scope to its first child scope that hasn't be visited yet. This ensures that the codes inside the block body are generated in the correct context, which means that variables would be generated with the type that they are most recently declared.

- Short Declarations:

We divide each short declaration into two parts. One part is the new variables. This part is generated as variable declarations in C++. The other part is the variables that have already been declared. This part is generated as assignment in C++.

- For, If, Switch statements:

For, if and switch statements have init or post statements. The init/post statement has an independent scope that is between the for/if/switch statement's scope and their block body's scope. To implement this independent, intermediate scope in C++, we decide to generate an if(true) outside the real if/switch statement. In effect, the scope in the block body of the if(true) is the scope for init/post statement, which both allows the init/post statement to redeclare the variables that is declared in the scope the for/if/switch statement belongs to, and enables the block body of the for/if/switch statement to redeclare the variables declared in the init/post statement.

10.2.5 Expressions

To ensure the evaluation of expressions correctness of our output program, we forced to add parentheses enclosing any expression.

- Identifiers:

We printed the tagged identifier name into our output C++ file for all identifiers.

- Binary expressions:

Most of the binary operators in Go were supported by C++. We stated how we handled the exceptions below.

The bit clear operator was implemented as bitwise_and(exp1, (bitwise_not(exp2)), which generated the same output value.

We overloaded the greater than (<) and smaller than (>) operator for string comparison by returning a bool value given by std::string::compare() function.

The concatenation operator (+) for strings was provided in C++.

- Cast expressions:

We implemented cast expression of int to string by first casting the int value to a char and then to a string.

-Append expressions:

Given we defined the slice in std::vector, we used push_back() method for appending new values into the slice, so that we don't need to handle memory allocation by ourselves.

- Length expressions:

We called str.length() to get the length for a string, array.max_size() to get the length for an array, and vector.size() to get the length for slice. The length of slice corresponded to the number of elements we have appended.

- Capacity expressions:

We called std::array::max_size to get the capacity for an array, which was the size of memory allocated to store the elements.

The capacity of a slice was the size of underlying array, so we used std::vector::capacity to get the underlying capacity for slice.

11 Conclusions

We felt our time is very pressed during the process of this project. So we are not able to fix up a lot of the bugs and unexpected behaviour in our implementation especially in code generator. Though some parts of implementation of code generator were not fully worked and well tested, our scanner, parser, weeder, and typechecker fulfills most of the rules and requirements in GoLite and works pretty good. In the future, we would fix the problems in code generator easily since we have captured the basic implementation idea of code generator. We would also implement a more efficient tree structure that makes us easy to store and access the semantic and syntax informations.

12 Contributions

Yue Lyu: Compose report, implement parser, AST, pretty printer, Weeder, implement type checker for declaration and part of symbol table, implement codegen for expression.
Yiyang Liu: Compose report, implement parser, AST, and Weeder, implement type checker for expression and part of symbol table, implement codegen for declaration.
Chunyao Zhao: Compose report, implement parser, AST, and Weeder, implement type checker for statement and part of symbol table, implement codegen for statement.

References

[1] *https://github.com/amirbawab/GoLite*

[2] *https://github.com/EmolLi/MiniLang*