GoLite Compiler Milestone 1 Report

Bessie Luo 260708568

Lancer Guo 260728557

Charles Lyu 260681819

February 23, 2020

Abstract

This report tracks all the details of decisions we made during the implementation of a compiler for GoLite language, a variation of GoLang. We went through the phase of scanning, parsing, abstract syntax tree constructing and finally pretty print the code in the complete form of GoLite.

1 Introduction

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.

2 Language Choice

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. Flex and bison help us to produce the .l and .y file easily in our first phases. Second, we used C to do all low-level development sides in COMP206 and C was considered as a good language for doing low-level development like code generation. Also, C is an example language used in class that could ease our work.

3 Scanner

The scanner is implemented in Flex. It tokenizes all the recognized keyword string with regular expression with our predefined string form, such as Hex, Decimal, Rune in regular expressions and existing keyword and operands. 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.

3.1 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.

```
LINECOMMENT \/\/.*\n
BLOCKCOMMENT \/\*([^*]|\*+[^*/])*\*+\/
UNCLOSEDCOMMENT \/\*([^*]|\*+[^*/])*
```

3.2 String type

There are two types of string in GoLite, which are interpreted string and raw string. These two kinds of string each have their unique wrapper. When the scanner is taking the input, it will read everything as char, which makes raw-string trivial to implement. As for interpreted string, we decided to handle this in the scanner, and output the translated to a string variable. Since it already took care of the transformation, we only need to treat it as a normal string in a later stage. To achieve this, we wrote a method in C, traversing through the input string and replace any encountered raw escape sequence with its corresponding code.

3.3 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.

3.4 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.

4 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 package identifier and a list of declarations.

A list of declarations consists of three kinds: Function declarations, Variable declarations or Type declarations.

Variable declarations contain one or multiple variable declarations in the case of distributed variable declaration. We break group declaration into multiple single declarations, and any individual declaration has an id list and an expression list.

Type declarations contain one or multiple type declarations in the case of distributed type declaration. Similar to a variable, we also break any group form into individual ones. A type declaration has one identifier and a list of type.

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. Moreover, a separate non-terminal called simple statement is used in 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 but to treat it as expression. Later on in the weeding section, we will separate expression as primary and other to constraint the syntax.

5 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.

Out AST root is a struct node named PROGRAM. It has a package name and a pointer to declaration.

Declaration node supports multiple nodes using linked list. Each node is either a function, type or variable declaration.

We use a function declaration struct to represent function non-terminal in parser. A linked list to represent function list.

Param struct also has a next pointer if more parameters are needed. It has a Expression type Id list and a type.

Variable declaration is either a type only, expression only or with both. A VarDeclKind enum is used to distinguish between different types.

Type declaration has no inner struct, because it only consist of one type and one identifier. A next pointer is necessary if it is a list.

In type field, we made another struct TYPE represents four different possible types. For array type, we decided to put size type as int instead to automatically reject the invalid expression. As for struct type, it contains a list of fields which are similar to type only variable declarations. For clearer representation we constructed a new struct StructField.

5.1 Statements

In our Statement struct, for each possible statement we declared one struct to keep track of its data.

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.

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.

In the increment and decrement struct, we pass in a boolean variable in parser to indicate if it is a postfix or prefix, which is used for pretty print later.

Our 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.

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.

5.2 Expression

All binary expressions and unary expressions are sotored in two struct field called binary and unary , despite of what each sides are. The parser will recursively expand each side and eventually store the right data.

The literal terms are stored in its own struct. We use integer to represent boolean instead of string becasuse it takes less space and easy to compare. For rune type, both escaped and normal ones are stored in one rune literal struct.

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 type. One special design for array type is, in a declaration for array type, the index field is set to be int type only, but to define array access, we no longer use int but Expression type for index, and handle the index accessing rule in weeder.

6 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 Pretty

Pretty printing is done by traversing our AST and printing accordingly. For the formatting of Pretty printing, We have a tabNum variable to keep track the number of tabs we should add or remove every time and a function called printTab() to print the certain amount of tabs accordingly. For the reusability of code, we define some functions such as prettyIdentifier, prettyEXpression, and prettyStatement which could be used every time we want to print some complex structures.

There are some trick cases and we choose different strategies to solve them. For example, there is one in for loop. There could be three parts in condition of for and these three parts could all be empty in GoLite. Therefore, there could be some meaningless semicolons at the end of each empty part. In sample code, only nonempty expression or statement could be printed inside the condition of for loop. However, we decide to keep all semicolons for now and solve this later. Also, since there is no clear benefit for printing TRUR and FALSE, we decide to stay with 1 and 0 in our pretty printing. The third case would be printing expression. We set the operation variable in expression as a string in our scanner. In this case, it help us to print expressions easily without using switch statement in pretty printing. Another case would be printing decimal, octal and dex numbers. we use strol method to convert input string to int in Scanner and it benefits pretty printing from redundant switch cases as well.

There are some difficulties, like variable and type distribution. To deal with this requirement, we decide to print all variable with an assigned type separately. Basically, instead of printing all variable and type in one line, we print one variable each line and separate them with comma. In this case, the format would be nicer.

8 Work distribution

We hold all decision discussion together and then distributed work equally. We worked on the AST Tree altogether using the CodeCollab tool to make sure everyone agrees on the structure. 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)
- Report

Lancer Guo

- Scanner (revised)
- Parser (first draft)
- Weeder
- Report

Charles Lyu

- Scanner (revised)
- Parser (revised)
- Pretty (first draft)
- Report