Milestone1

Yiyang Liu, Yue Lyu, Chunyao Zhao

February 2020

1 Introduction

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 the previous assignment and that C worked coherently with Flex and Bison. Furthermore, C is a very bottom language. 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 will rely heavily on the parser and the abstract syntax tree in our future work. To improve our collaboration and future modifications, we decided to name every attribute, node type, and node structure according variable names in parser.

2 Design Methodology

2.1 Scanner

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. If none of the above regular expressions were matched, our compiler would report an error with unexpected characters. It silently solved open quotation marks and nested block comments problems. We also implemented a global variable which could memorize the previous token matched. Then, we could follow the GoLite rules to append additional semicolons when we saw a new line token was matched. Our defined regular expressions and the first longest match enforced by Flex worked well with GO syntax rules.

2.2 Parser

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 definiton of our grammar.

2.3 Abstract syntax tree

2.3.1 Uniform node

We decided to make only one kind of node struct for all kinds of language constructs and distinguish between different language constructs by checking the NodeKind attribute, an enumerate type that specifies the language construct that the node is representing. We also defined an attribute struct for all the type nodes matched by the parser. So we were able to access the underlying information by accessing the corresponding struct attribute. This uniform node decision makes it possible to traverse the whole AST using just one recursive method. Since the pretty printer, the symbol table, and the type checker all requires the traversal of the AST, this easy-to-traverse feature provides great convenience to our later work.

2.3.2 tree structure

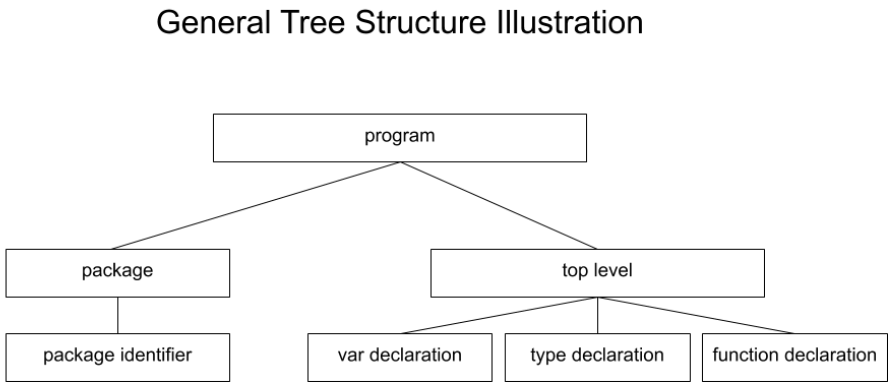

Figure 1: Top Level structure

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.

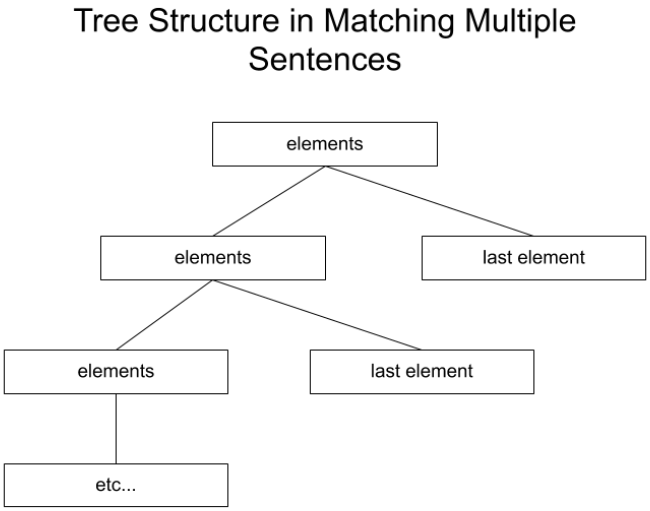

Figure 2: Tree structure in matching multiple elements

2.3.3 tree structure

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.

2.4 Weeder

In this project, we included weeding process to check the number of default cases in switch cases statement and the use of "break" and "continue" statements.

Since our parser parsed the GO program left to right and bottom up, we could simply check the subtree at the root of each switch statement node when parsing the program to weed out invalid programs that contained more than one default cases. When traversing each subtree of switch statement node, we increased a counter for default statements each time we visited a default statement node. When our counter was greater than 1, our compile would report the error and exit on 1.

To weed out the improper use of break and continue statement, we traversed the subtree from each function declaration node, since invalid break and continue statements would only appear inside function declarations. We implemented a depth first search algorithm to traverse the function declaration node. We increased our for_counter by one when we entered a for loop and and decreased our for_counter by one when we left a for loop. The same applied for switch_counter. If we encountered a break or continue statement with the corresponding

counter equal to 0, our compiler would report an error and exit on 1.

Our implementation of weeder is runtime efficient. The total time of all weeding processes was bounded by O(n), where n is the size of AST.

2.5 Pretty printer

Our pretty printer was essentially a depth first search traversal of AST. The implementation of the pretty printer was decided by the features of our AST. Since all nodes in the AST had the same type, the pretty printer could visit all the nodes and subtrees by recursively calling itself. Also, since each node had a NodeKind attribute that specified which language construct it was, we made switch cases based on NodeKind so that each kind of node could be handled within its own switch case.

3 Team work

Yue Lyu: Scanner implementation, parser implementation, AST implementation, Weeder implementation, report writeup
Yiyang Liu: Parser implementation, AST implementation, pretty printer implementation, report writeup
Chunyao Zhao: Parser implementation, AST implementation, pretty printer implementation, report writeup

4 Related work

We consulted the compiler design of GoLite wirtten in C++[1] and tree structure design of Minilang [2] by other compiler engineers.

References

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

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