GoLite Milestone I

Pavel Kondratyev	Kaylee Kutschera	Kabilan Sriranjan
pashakondratyev	kkutschera	kabsri
260653115	260608470	260671847

1 Tools and Languages

We chose to implement our compiler in C using flex and bison. The main motivation behind this was that we were already familiar with using these tools from the assignments as well as the lecture content, which let us focus more on creating a successful compiler rather than learning a new tooling.

2 Design

2.1 Scanner

For the most of the features of GoLite the implementation of the scanner was very similar to the minilang compiler, with the following exceptions.

2.1.1 Semicolons

Semicolons were inserted based on the rules given in the specifications-namely keeping tracking of the final token per line. If there is an EOF character we also had to add a semicolon.

2.1.2 Multiline Comments

The multi-line comments were handled by using start statements as described in the flex documentation for C-style commenting. Semicolons were also inserted in this state. While we were in this state, we also had to check if we reached an EOF character and insert a semicolon accordingly.

2.1.3 Strings, Raw Strings, and Runes

For valid plain strings, we scan the entire scanned token and use a function to check if there is an escaped character as per the specifications, and if there is, the full escaped character is inserted as a character instead of the '\' followed by whatever character is. Raw strings are left exactly as scanned. Runes are handled the same way as each character of the plain string, with the difference being the valid escaped characters.

2.2 Parser

The grammar was designed almost entirely off of the official GoLang specifications. There were mild difficulties for constructs that were specified using shorthand in the official documentation that couldn't be directly translated in bison. For example, the official specification defines a Block in the following manner:

$$\begin{aligned} Block &\rightarrow \text{"\{"}\ StatementList\ \text{"\}"} \\ StatementList &\rightarrow \{\ Statement\ \text{";"}\ \} \end{aligned}$$

Since this notation doesn't apply to bison we implemented the rule like so:

$$\begin{aligned} Block &\rightarrow \text{"\{"}\ StatementList\ \text{"\}"} \\ StatementList &\rightarrow Statement\ \text{";"} \\ &\quad | Statement\ \text{";"}\ StatementList \end{aligned}$$

Another situation that required a different type of workaround were for clauses. Officially they were specified as:

$$\begin{aligned} ForClause &\rightarrow [\ InitStmt\ ]\ \text{";"}\ [\ Condition\ ]\ \text{";"}\ [\ PostStmt\ ] \\ InitStmt &\rightarrow SimpleStmt \\ PostStmt &\rightarrow SimpleStmt \end{aligned}$$

On top of this notation not being directly transferable to bison, the constructs InitStmt and PostStmt add unecessary bulk to the grammar. We instead implemented the for clause as:

$$\begin{aligned} ForClause &\rightarrow SimpleStatement\ \text{";"}\ ExpressionOrEmpty\ \text{";"} SimpleStatement \\ ExpressionOrEmpty &\rightarrow \%empty \\ &\quad | Expression \end{aligned}$$

Taking note that a SimpleStatement can already be empty, this was a concise way to represent the official spec in bison.

Essentially, all the tricks used to contruct the grammar in bison were introducing new rules in place of the [] and {} operators and removing extraneous constructs.

2.3 Abstract Syntax Tree

The abstract syntax tree largely followed the structure of the grammar. Constructs like statement lists where there can be one or many components

were handled by using linked lists. Namely, this technique was used for block statements- which are essentially a list of statements- and switch statements- which are essentially a list of case clauses. The switch statement structure was especially complex because on top of including all the relevant information per node such as clause kind and case expression, one of the attributes was itself a statement list. This lead to the switch expression being stored as a somewhat 2D linked list in the AST. This caused some difficulties in later stages like weeding, but it seemed like the most elegant solution to capture the nature of switch expression. In GoLite it is possible to specify multiple variables with a single assignment operator.

Since in GoLite we can only return single values, within the AST we had to check that number of items on each side of the "=" token in a specification are equal. At some points there are multiple declarations or assignments, these are broken down into individual declarations or assignments within the AST.

2.4 Weeder

There were several aspects that weren't handled in the grammar and were handled by a weeding phase. During the short declaration phase, bison was having issues parsing the left hand side as an identifier list, and in order to combat the shift reduce errors the left hand side was transformed into a more general expression list and during weeding phase, each expression in the list was checked.

The grammar accepts programs that have continue statements inside a switch statement or breaks and continues outside of loops. Since in GoLite these cases are not allowed we used a weeder to reject such programs. The way we implemented this is we started weeding from the very beginning of the program and use recursive calls to traverse through the program tree. As soon as a for statement is parsed, all the recursive calls within the for block are passed with arguments signalling that break and continue statements are allowed. If a break or continue statement is parsed with these arguments absent, the weeder will reject the program.

Switch statements can only have at most one default case so we weeded switch statements that had many of them. The default case can go anywhere in the list of clauses. Since all the clauses are stored in a linked list, and the kind of clause- whether it is a default or a case- is stored per clause all that needed to be done was count the number of nodes of the default kind present in the clause list.

All non void functions must return a value. We used the weeder here to traverse all possible paths the control flow can take has a non void return statement. For the switch statements this meant we had to check that each case had a return statement, which was made easier by GoLite not handling fallthrough. In the reverse case, for void functions we weeded for the existence of a return statement with an expression.

2.5 Pretty Printer

The pretty printer was also largely based off the grammar. Almost all the design choices stemmed from choices made when building the grammar. One of the biggest troubles was coming up with a structure such that the indentation and the insertion of newlines had no mistakes. In order to handle this we establish an invariant such that only statement constructs are allowed to print out newline statements (with the exception of distributed declarations). This created a simple and elegant codebase without having to propagate any information about whether to insert a newline. Tabulations were handled by recursively passing the current tab level.

Because of the way we scanned in the valid strings and runes, we had to undo the process for escaped characters and print escaped backslashes so for example a newline escaped character would be displayed as "print("\t")" and not "print("	")".

During the parsing/abstract syntax tree phase short declarations are unrolled into their long forms so each declaration will have its own line when printed out. If a distributed variable or type declaration only has one entry in it is unrolled into a single statement, otherwise the original distributed format is preserved.

3 Contributions

Pavel did the scanner and pretty printer as well as helped with the parser, weeder, and adding test programs. Kaylee did the AST and most of the test programs. Kabilan did most of the parser and helped with the weeder and adding test programs. We all equally contributed to the report.