# GoLite Project Milestone 1

## 1. Introduction

Our chosen language for implementing a GoLite compiler is C. A variety of considerations led to this selection. First, *flex* and *bison* are great tools written in C. These tools greatly reduce the work in creating the first phases of the compiler. Second, the architecture-independent nature of C means that our compiler would be able to be deployed in a wide variety of contexts. Finally, we recognized that C is the example language used in class, which eases our work.

With these decisions in mind, we detail the implementation of the lexer, parser, AST design, weeder and pretty printer.

## 2. Lexer

The lexer was implemented using *flex*. It identifies the GoLang keywords and operators, integers in decimal, octal, and hexadecimal format, floating point numbers with at least one number on either side of the decimal point, standard and raw string literals, and individual characters (*i.e.* runes).

A token is returned for all matches unless the match is a single character operator (*e.g.* +,*,&,!). We opted to convert the integers received in octal and hexadecimal formats into the corresponding decimal value stored inside the token *%union*. This avoids confusion in the later stages of the compiler, particularly in the parser and AST, and fewer grammar rules need to be written as a result and fewer node constructors need to be defined.

Likewise, raw strings are converted to strings by being parsed and converting escape characters to their formatted counterparts (*e.g.* \n becomes \\n). The raw string delimiters ( ` ) are also converted to double apostrophes. We keep the double apostrophes inside the string since it will avoid us the task of generating them back during the codegen phase.

Multi-line comments gave us some troubles. We initially attempted to recognize them using a simple regex expression (“/*”.*”*/”). This does not output unclosed block comment errors properly. Instead, we had to go about declaring explicit DFA states to better recognize the blocks. Upon encountering a “/*” expression, the lexer enters an exclusive “IN_COMMENT” start condition. Note that since flex prioritizes longer matches, if “/*” appears in the context of a string, the start condition is not activated. If it ever is, then it discards all inputs and exits the condition on the first “*/” match. Encountering the end of a file while in it results in an error (unterminated comment).

Due to the design decision of the original Go language, a block comment is transformed into a semicolon by the lexer when the preceding token triggers the semicolon generation rule, as described in the specifications.

Another challenge that was mostly tricky to notice had to do with the ending of a file. If no new-line was present, then no semicolon is generated even if one should be (*e.g.* if the file ends with a closing function curly brace '}'). This is dealt with using start conditions again. An "EOFCHECK" start condition is entered at the very beginning of the file and allows us to match on the end of file pattern. When this occurs, we simply return a semicolon if necessary, while making sure to leave the start condition.

3. Parser

The parser was implemented using *bison*. It consists of a context-free grammar that was constructed to center around certain groups of non-terminals mostly align with the major constructs of an imperative programming language like GoLite.

Expressions are represented in the expression node, which includes most of the operators available in GoLite. To make assigning precedence easier, the expression node expands to a term node, which expands to a feature node. It is not until this feature node that literals, parentheses, and function and struct accesses can be built into the tree.

Declarations of types, top-level variables, and functions are grouped into a top-level program definitions node.

Statements are grouped into simple statements (which can occur before the condition in certain other statements such as "for") and plain statements, which can themselves be simple. Simple statements are assignments, increments/decrements, and expressions (such as function calls). Plain statements also include prints, breaks, continues, returns, ifs, fors, switches, and declarations.

Most of these major groups have block forms which can occur in a variety of contexts. In the case of identifiers and expressions, these blocks are separated by commas. In addition to blocks, declarations and assignments can be constructed to declare/assign many variables in a single line. These sorts of declarations/assignments are passed to the AST in order to be traversed and constructed as a number of declarations/assignments.

4. AST

The AST is made of 6 different types of nodes: program nodes, statement nodes, declaration nodes, short declaration nodes, function nodes, and expression nodes.

Program nodes represent the top level structure of a program and keep track of package declarations, variable and type declarations, and function declarations. Separating function declarations from other types of declarations.

Declaration nodes contain information about the declared type if any, the identifier associated with the declaration and the declared value if any. Declaration nodes can hold any variable, type, struct, and function declarations. The declared value can be either an expression or a body, in the case of structs, consisting of declarations. In the case of functions, a reference to a function body is kept and type declarations also only keep a reference to the actual type (*e.g.* "type num int" would store an int type node).

Declarations can be linked in a list. In some quirky cases, a second pointer was also necessary to avoid the an entire list to be lost except for the head. This would occur inside the distributed declaration construct where multiple declaration node would be assembled into a list. However, the head's *next* field would be changed to the top level declaration in some cases, losing almost the entirety of the list.

Types are stored as a struct containing info about the declared type as a string, the "modifier" of the type (*i.e.* if it is a slice, array, …), and a size attribute that is only used in the case of arrays, but could also be useful for slices eventually. The declared type is stored as a string to more easily deal with user defined types. The "modifier" field will prove to be quite useful in the type checking phase to determine if an operation is valid or not.

Function nodes keep track of the function identifier, the argument count and types, its return type, and a body of statements. The arguments are stored as declaration nodes which will never have a right hand side. Function nodes can be wrapped by expression nodes and declaration nodes depending on the context of the occurrence (function calls in the former case, function declaration in the latter). It is not always possible to specify the name/identifier of the function (*e.g.* "a[3]()" is valid in the reference compiler, without a clear identifier attached to it, at least from the parser's point of view).

Statement nodes store a value and can be linked in a list. Their value can be an expression, a declaration or a statement body (for block statements). It could also be more complex: in the case of conditional statements (if, elif, else, for, while), a struct is maintained to store the condition expression, optional declarations, the statement body, and following statements.

Note: The "following statements" should not be confused with the "next" statement in the linked list. Rather, it will refer to an "else if"/"else" statement in the case of "if" and "else if" statements and to the for loop statement (the one executed after each loop iteration) in the case of a for loop.

Assignment statements are stored as an expression representing an identifier and another expression representing the new value. Multiple assignments are also stored this way and are assembled into a chain of "following" statements. Quick declarations are also stored in the same manner, but are assigned a different flag so that the pretty printer can distinguish both cases. This flag will also be useful during the type checking phase to ensure quick declarations contain at least 1 new variable, while no such condition exists for multiple assignments.

Print statements only need to keep track of the expression to print and whether or not to print a new line at the end.

Switch statements are broken down into 2 components: a switch body and case bodies. The switch body keeps a reference to the condition, any optional declarations and the case bodies. The case bodies keep track of the case condition (a NULL value represents the default case). The body of statements for a case is also stored.

Lastly, expression nodes can either represent a literal, an arithmetic operation, or a function call. The type of arithmetic operation will determine if the node has 1 or 2 children. In the case of a function call, a new function node is constructed and the given arguments will be stored as described above.

5. Weeding

After the construction of the AST, it is traversed in order to be weeded of certain invalid constructs. These are:

- Switch statements with more than one default clause
- All functions with a non-void return type must have at least one return statement in their body
- Return statements only occur in function bodies
- Break and continue statements only occur in the correct context
- Blank identifiers do not occur in expressions or on the right hand side of assignments
- Inline simple statements which occur in if, for, or switch statements cannot be expressions unless the expression is a function call.

The weeder is implemented by a top-down traversal of the AST. Recursive calls carry information about previously seen patterns. The return value from the recursion tells whether a certain pattern was found in an AST subtree overall. The rewinding of the call stack is used to confirm the presence of a return statement in a function.

6. Pretty Printing

Pretty printing is simply done by traversing the tree and printing accordingly as we do so. Some tricky cases involve multiple assignments where we simply iteratively print each expression on the left hand side of the equals sign, and then do the same for the right hand side. To do this in a clean fashion, some helper functions are defined.

Block declarations for variables and types are printed back in a standard format where each variable is listed by itself. This is simpler to do and still maintains the correctness of a program.

7. Division of Labor

The lexer was written by Charles. The parser was written by Gregory. The AST was written by Charles. The pretty printer was written by Charles. The weeder was written by Song. Most of the test programs were written by Song, with small contributions from Gregory. All group members participated in identifying and fixing bugs. Significant revisions to the original parser and AST were made by Charles and Gregory, respectively. This report was written by all group members.

Charles:

- Lexer
- Parser (revised)
- AST (original and revised)
- Pretty printer
- Report

Gregory:

- Parser (original)
- AST (revised)
- Test programs (few)
- Report

Song:

- Test programs
- Bash scripts and Makefile
- Weeder
- GitHub repository organization
- Lexer (few)
- Report