COMP 520 Milestone 2

Rajveer Gandhi, Archit Agnihotri, Dipanjan Dutta

Abstract

This is a design and implementation report of the Milestone 2 of the GoLite project.

1. Design Decisions

Since we got less marks in the Milestone 1 report, we decided to make this report more detailed and descriptive. We decided to include the ideas we missed regarding the scanner, parser and the syntax tree in the first milestone report.

In this milestone Dipanjan created the invalid programs and was responsible for the report. Dipanjan was also responsible for creating the type-checker and the weeder that we could not implement properly in the first milestone. Archit and Raj did the script for symbol table generation and fixed the AST and Pretty printer from Milestone 1.

2. Flex and Bison

In the last report, we mentioned that we chose flex and bison because all of us are comfortable with programming in C. Flex and Bison are frameworks in C that can generate code for C. Our program mainly consists of the following files :

- **src\golite.l** : Scanner
- **src\golite.y** : Parser
- **src\tree.h** : Header file for syntax tree
- **src\tree.c** : Syntax tree generator
- **src\weed.h** : Header file for the weeder

- **src\weed.c** : Weeder
- **src\pretty.h** : Header file for the pretty printer
- **src\pretty.c** : Pretty printer
- **src\main.c** : Main file which controls execution
- **src\Makefile** : Script to run flex and bison to create the lexer and parser
- **build.sh** : Shell script to clear older versions and run the latest version of **Makefile**
- **run.sh** : Shell script to run **main.c** on a program file with a specified mode.
- **test.sh** : Shell script to test programs (used of internal testing only)

The available modes are :

- "*scan*" : Scans the program and gives **OK** if it scans successfully, otherwise throws the error.
- "*tokens*" : Prints the tokens encountered in order inside the program.
- "*parse*" : Parses the program, creates the syntax tree and prints **OK** if successful, otherwise exits after printing the parsing error.
- "*pretty*" : Pretty prints the program from traversing the syntax tree.

3. Scanner

The scanner was developed with a very standard setup. We had macros for decimal, octal and hexadecimal representations, and also for escape sequences for runes and strings. Tokens were captured with the help of regular expressions. Every operator has its individual token. We decided not to group operators in order to keep the code more transparent and interpretable.

One challenge that we faced in the scanner was optional semicolons. We had to deal with this by creating a variable called *lastRead*, and updating it whenever a new token is generated (before it is returned). then, when $\backslash n$ or **EOF** or block comment including a $\backslash n$ is encountered we check to see with function *insertSemicolon* if there should have been a semicolon there, and we insert the semicolon in that case.

Another challenge we faced was handling block comments. Dipanjan initially added a state machine implementation of the handling of block comments. However, it was too abstract and none of us knew how exactly it was handling the block comments. So we later decided to switch to regular expressions.

4. Parser

Defining the production terms was a bit complex. We had a union structure with all the types of nodes we can have in out syntax tree. We then typed the production nodes according to these node types. Thus the syntax tree was created according to the grammar rules in the parser.

Tokens were aliased with the addition of t before their natural names. Almost all the tokens were of type string except integer and float literals which had their respective types. We did not have special tokens for data types and boolean *true* and *false* as they were not reserved keywords in GoLite.

For the error message generation and display, we used bison's error reporting function *yyerror*.

Associativity and precedence was also defined in the parser. The rule of precedence chosen was (from lower to higher):

- = (Assignment operator)
- || (Logical OR operator)
- && (Logical AND operator)
- ==, ! =, <, > <=, >=
- +, −, |, (XOR operator)
- ∗, /, %, <<, >>, &, AND-XOR Operator
- All unary operators

All operators are associated left-to-right, except assignment operator, which is associated right-to-left.

The production rules are created by a joint agreement between all three of us. One thing Dipanjan missed was that typecasting was actually treated as a function call and not a separate expression type. Raj caught that and changed it.

5. Syntax tree

The syntax tree that was in the first milestone had a lot of bugs so Archit decided to create the syntax tree altogether. Previously, we had one single convoluted NODE with different structs for different types. This time around, the node is split up into 23 different types of nodes. It might seem too many nodes but it helps bring clarity to the coding structure and helps traversing the syntax tree intuitive. The different types of nodes that are defined are as follows:

- $PROGRAM$: Program node, root or beginning of the tree.
- $PACKAGE$: Package declaration.
- $TOPLEVELDECL$: Top level declarations. Can be one of the following:
  - $FUNCDCL$: Function declaration
  - DCL : Non-function declaration. They can be :
    - $VARDCL$: Top level variable declaration
    - $TYPEDCL$: Top level type declaration
- $FUNC\_SIGNATURE$: Function signature.
- $PARAM\_LIST$: Parameter list for a function signature.
- $IDLIST$: List of identifiers for short declarations, assignments and top-level declarations.
- $TYPE$: Type of a declaration.
- $STRUCT\_TYPE$: Block of the structure.
- $BLOCK$: Block of code enclosed by braces.

- *STATEMENTS* : Node acting as a linked list for consecutive statements.
- *STATEMENT* : Program statement.
- *ELSE_BLOCK* : Special node for *else* part of an *if* block.
- *SWITCH_CONDITION* : Condition for a switch block.
- *SWITCH_CASELIST* : Linked-list like node for all the cases and default case for the switch block.
- *EXPRLIST* : Linked-list like node for expressions (for multiple assignments in one line).
- *FOR_CONDITION* : Node for the condition of a for loop.
- *SIMPLE* : Simple statement.
- *OTHER_EXPR* : Other statements (including function calls, struct member selector and slice indexing).

6. Weeder

The weeder is implemented in order to catch any syntactic errors that might not have been directly caught in the parser's grammar. Dipanjan created the weeder on his own. The following checks were implemented in the weeder:

6.1. Single default case

The language specifies that there can only be one default case inside a single switch block. We used a local variable *hasDefault* that is initailly set to zero. If a default case is encountered, it is set to one. If another default case is found while *hasDefault* is 1, we throw an error. We used a local variable to correclt weed through nested switch cases which are recursive calls. Once the switch block is complete, *hasDefault* is set back to zero.

6.2. Break and continue statements

According to the language specification, *continue* statements can only appear inside a for loop and *break* statements can only appear inside a for loop or switch block. We had two global static integer variables *insideFor* and *insideSwitch* that stores the level of depth inside a for loop and switch block, respectively (Initial values of both are zero). When a for loop is encountered, *insideFor* is incremented by 1 and control enters the for loop. When control comes back after finishing the for loop, *insideFor* is decremented by 1. The behaviour is similar for *insideSwitch* as well. If a *continue* statement is encountered when *insideFor* is zero, an error is thrown. If a *break* statement is encountered when both *insideFor* and *insideSwitch* are zero, an error is thrown. Making *insideFor* and *insideSwitch* global and integer helps weed nested loops and nested blocks in a program.

6.3. Blank identifier

According to Go specification, blank identifiers can be used :

- as an identifier in a declaration
- as an operand on the left side of an assignment
- as an identifier on left side of = assignment only
- as a parameter name in function declaration

We used a global boolean variable *isBlankIdValid* that is initially set to false. It is set to true before :

- the weeding of identifier lists of a declaration
- *LHS* of an assignment statement
- *LHS* of a short declaration statement

Whenever the recursive weeding of the above are done, the *isBlankIdValid* variable is set to false. If during any expression or statement evaluation, we encounter an ”_” we check if *isBlankIdValid* is true or false. If it is false, we throw an error.

6.4. Unbalanced assignments

If the *LHS* and the *RHS* of an assignment or declaration statement contain unequal number of operands, the compile should quit with appropriate error. This is implemented with two local variables *lhsCount* and *rhsCount*. The *weedIDLIST* and *weedEXPRLIST* functions return the number of operands in the idlist in the *LHS* of an assignment or declaration, or the number of expressions in the *RHS* of the same. The counters are stored in *lhsCount* and *rhsCount* respectively. If they do not match, an error is thrown. Otherwise, the counters are reset to zero for the next statement.

6.5. Return statements

The language specifies that return statements can only be inside a function body. We use a static global integer variable *insideFunction* that is initially set to zero. When we are weeding the function declaration, we increment the *insideFunction* variable by one and then start weeding of the function block. As soon as weeding of the block is done and control comes back, we decrement *insideFuction* by one. If a return statement is encountered and the value of *insideFunction* is zero, an error is thrown. Use of an integer variable allows for weeding of nested functions (which are not supported in the language yet, but its a nice provision to have).

7. Pretty Printer

The pretty printer is almost similar in coding structure to the syntax tree. Besides the fact that we implemented a different node structure for syntax tree, we encountered multiple segmentation faults in Milestone 1 and several other issues. For this reason, we decided to a complete overhaul of the pretty printer. This was done by Archit.

Previously, we had a single function split into cases based on the kind of statement/expression inside one function. This time around, we created separate functions corresponding to each type of node in the syntax tree for the pretty printer. The *PROGRAM* is the root node from where the pretty printer starts. Semicolons are printed at the end of every statement.

We also have a separate function, *prettyIndent*, that prints 4 spaces per level of indentation level. Indentation level is stored in a variable *g_indent*. Every time we enter a block for pretty printing, the *g_indent* variable increments by one. On exiting the block, *g_indent* is decremented. We decided to print spaces instead of tabs because we have see tabs to create chaotic indentations in several editors. Thus to keep a standard across all platforms for pretty printing, we used spaces.