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.

8. Symbol Table

The symbol table script was designed by Raj and Archit. The symbol table was created using a an array of the $SYMBOL$ struct. In a logical sense, the symbol table is designed like a stack of frames, where each frame is associated with its corresponding AST node. The check for printing the symbol table is stored in a external integer variable g_tokens.

The data structure for the symbol table is as follows:

- $SymbolTable$: The main symbol table frame structure. It consists of :
  - $SYMBOL * table[HashSize]$: An array of $SYMBOL$ type for symbols in the symbol table. The $HashSize$ is set to be 317.
  - $SymbolTable * parent$: Pointer to the parent table of the current symbol table (you can think of it as pointer to the previous element in stack)
- $SYMBOL$: Data structure to define the symbol. It consists of :
  - $name$: String to store the name of the symbol.
  - $SYMBOL * next$: Pointer to the next symbol in the current symbol table.
  - $symTYPE*data$: Type of the symbol and additional information about the symbol.
- $symTYPE$: This structure is used to store the type of the symbol. It is used for printing the symbol table and can furthur be used in typechecking. It consists of :
  - $enumSymbolCategory$: The category of the symbol. It can be :
    * $type\_category$: Declared type.
    * $variable\_category$: Declared variable.
    * $function\_category$: Declared function.
    * $constant\_category$: Used specifically for "true" and "false". It is declared and added to the root symbol table for shadowing in the future.

- *enumsymbolType* : It is used to identify whether it is a function declaration or not (as they are treated differently).
- *unionval* : This contains a pointer to the $TYPE$ and $FUNC\_SIGNATURE$ nodes of the AST.

In order to implement the symbol table, we defined some helper functions as well. These functions have some specific purpose, to help with printing the symbol table, to get a symbol from symbol table etc. The functions are described in the following subsections.

8.1. symIndent

The *symIndent* function is used to print proper indentation for the symbol table printing. For every indentation level (tracked by global integer variable *g_symIndent*) 4 spaces are printed.

8.2. Hash

This function is used to create the corresponding hash for a symbol, in order to map and store it in the symbol table. Hashing is done by Division-remainder method.

8.3. initSymbolTable

In this function, a node of type *SymbolTable* is created and initialized. The *table* array is initialized and all elements are set to null in order to avoid segmentation faults due to bad dereferencing. The parent of the symbol table is also set to null. The newly created symbol table is then returned.

8.4. scopeSymbolTable

This function is called when a new sub table (or sub-block or stack element) is to be created with new scope, inside the current symbol table. *initSymbolTable* is called to create a new *SymbolTable* frame. The parent of the *SymbolTable* frame is set to be the symbol table frame in which the scope was at the time of this function call. The function returns the new *SymbolTable* node.

8.5. putSymbol

This function is used to store a symbol in the current scoped $SymbolTable$ frame.

- If the symbol is "_", the symbol is not put in the symbol table and a $NULL$ is returned.
- If the symbol is already defined in the current scoped frame, an error is thrown.

Otherwise, the symbol is added to the current frame, along with its type and category. If *g_tokens* is 1, the symbol is printed as well. It also returns the symbol.

8.6. scopeInc and scopeDec

It increments and decrements the *g_symIndent* variable for printing the symbol table, respectively. It also prints the "{" and "}" for the opening and closing of symbol table frame.

8.7. printSymbol

This function pretty prints a symbol from the symbol table. First it prints the correct indentation (calls *g_symIndent*). Then it finds out the category of the symbol and stores it in *sym_cat*. It then prints according to the type of the symbol. If it is a basic, array, slice or struct type, it calls the $prettyTYPE$ function from the pretty printer. Otherwise, if it is a function declaration, it calls a special pretty printer function $symPrettyFUNC_SIGNATURE$ that prints the function symbol in proper format.

8.8. getSymbol

This function is used to fetch a symbol from the symbol table. It uses a recursive approach to traverse through the stack of frames. If, on reaching the root frame, the symbol is not found, an error is thrown stating that the symbol is not declared. If, however, the symbol is found in a frame, it immediately returns the symbol.

8.9. defSymbol

This is similar to *getSymbol*, however, it returns a boolean value. It returns true if the symbol is found, false otherwise. It is used in checks while traversing the AST to create the symbol table. The purpose of creating function is to not handle $NULL$ pointers in any check.

8.10. initSymType

This function sets the type and category of a symbol and returns a $symTYPE$ variable. It takes a void pointer p that contains the type of the symbol, and the enum value of the symbol type. p is casted appropriately and stored in the val union of the $symTYPE$ node.

The symbol table generation is very much similar to the pretty printer or the syntax tree generation. There are, however, a few additional features.

8.1. Predeclared symbols

The symbol table contains the following predeclared symbols :

1. **int**
2. **float64**
3. **rune**
4. **string**
5. **bool**
6. **true**
7. **false**

The last two symbols are of constant type and used for shadowing purposes.

8.2. Scoping

The first challenge we faced while generating the symbol table was creating nested scopes/frames. We came up with a plan of creating a new frame (by calling $scopeSymbolTable$) every time we encounter a block of code or branching from the flow of the program. A new frame is created for every :

- code enclosed in braces
- function declaration
- for loop
- switch block
- case in a switch block
- if block

- else-if/else block

All variables in parent frames can be redeclared in these frames but a variable declared in this frame cannot be redeclared in the same frame.

Infinite loops and while loops also have separate frames for symbol table, but thy are essentially the same frame as created for a block of code. For loop can have declaration in the first part of the three-part, thus, we decided to create a separate condition for frame before the recursion.

8.3. Short Declarations

Another challenge we faced was to ensure that every short declaration must have at least one undeclared variable. To implement that, we used a local boolean variable $atLeastOneVarNotDeclared$ (initially set to false). We looped through the L.H.S. of the short declaration statement and checked whether a variable is defined in the symbol table or not. If we see that the variable is not defined, we add that variable to the symbol table and change the value of $atLeastOneVarNotDeclared$ to true. If, after traversing through the identifier list in the L.H.S. of the statement, the value of $atLeastOneVarNotDeclared$ is false, we throw an error and exit.

9. Invalid programs and their typing rule violation

1. *incompatible_type_append_with_slices.go* : *append* is well-typed if the first term is a slice and the second term is of the same type as first type. Here, they have different type aliases (even though they derive the same base type).

2. *incorrect_number_of_function_call_arguments.go* : A function call is well-typed if all of its arguments are well-typed and it has the same number of arguments and the types of arguments are same as corresponding types of parameters. Her, the number of arguments do not match the number of parameters.

3. *incorrect_assignment_of_function_return_value.go* : An assignment statement is well-typed if both L.H.S. and R.H.S. are well typed and type of every pair of corresponding *lvalue* and expression is the same. In this program, the function returns a $float64$ but the variable it is stored into is of type *int*.

4. $invalid\_adding\_bool\_and\_string.go$: A + operation requires both its operands to be either numeric or string. In this program, the L.H.S. operand is of type boolean, which is a violation.

5. $invalid\_function\_return\_value\_diff\_type\_alias\_arrays.go$: The return statement is well typed if :

    - It has no expression and its enclosing function has no return type
    - The type of the expression of return statement is the same as the function's return type.

    In this program, the return type of the function is $[5]num2$ but the function returns $[5]num1$ (even though both $num1$ and $num2$ have same base types).

6. $invalid\_assignment\_return\_value\_array\_size\_mismatch.go$: Same rule as no. 3. Here the function returns $[5]int$ but the value is stored in a variable of type $[6]int$

7. $invalid\_function\_return\_statement.go$: Same typing rule as no. 5. Here the return type of the function is int but the function does not return anything (returns a void or null, to say).

8. $invalid\_function\_return\_value\_diff\_type\_alias.go$: Same typing rule as no. 5. In this program, the return type of the function is $num2$ but the function returns $num1$ (even though both $num1$ and $num2$ have same base types).

9. $invalid\_int\_condition\_in\_if.go$: An if statement type checks if:

    - Initial declaration, if present, type checks
    - Condition expression is well-typesd and resolves to type *bool*
    - Statements in the if block typechecks
    - Statements in $else - if/else$ blocks typecheck

    In this program, the expression is well-typed but resolves to an *int* type instead of a *bool* type.

10. $invalid\_return\_int\_in\_void\_function.go$: Same typing rule as no. 5. Here, the function has no return type, but inside the function body, it return an *int* value.

11. *invalid_return_statement_in_function.go* : Same typing rule as 5. Here, the function has *int* return type, but inside the function body, it return a *string* value.

12. *invalid_short_dec_type_mismatch_declared_var.go* : A short declaration is well-typed if :

    - All the expressions in R.H.S. are well typed
    - At least one variable in L.H.S. is undeclared
    - Declared variables in L.H.S. and corresponding expressions in R.H.S. must be of the same type.

    In this program, the third clause is violated. Variable a, already declared as *int*, is assigned a *string* value.

13. *invalid_string_decrement.go* : Increment/decrement statements are well-typed if their expressions are well-typed and resolve to a numeric base type (*int*, *float64* or *rune*). Here, decrement operation is done on a *string* expression.

14. *invalid_struct_member_assignment.go* : A field selection *expr.id* is well-typed if the expression is of a type that resolves to a struct and that struct contains the *id*. Here, p is of type a that has an *int* variable x, but it is being assigned a *string* value.

15. *invalid_type_assignment.go* : Same typing rule as no. 3. Here, *p.y* is of type a but the R.H.S. is of type c (even though the structs a and c have identical structure).

16. *invalid_type_comparison.go* : The binary expression == is well typed if both the operands are comparable (both of same types). Here, the operands are of type *num1* and *num2* respectively, hence they are in violation (even though the base types of *num1* and *num2* are *int*).

17. *invalid_type_comparison_2.go* : Same typing rule as no. 16. Here the variables are of types *num1* and *num2*, but the *num2* type is of type *num1*. However, they are still different types, so they are in conflict with the typing rule.

18. *invalid_type_comparison_3.go* : Same typing rule as no. 16. Here the two variables compared have two different struct types.

19. *invalid_typecasting_from_string_to_int.go* : A type cast *type*(*expr*) is well typed if :

    - *type* is well typed and resolves to a base type
    - *expr* is well typed and satisfies one of the three conditions :
        (a) *type* and *expr* resolve to underlying same types
        (b) *type* and *expr* resolve to numeric types
        (c) *type* resolves to *string* and *expr* resolves to *rune* or *int*

    Here, integer type casting is done in a *string* variable, which is in violation with the above rule.

20. *invalid_unary_not_op_on_int.go* : The unary logical *NOT* expression is well typed if the R.H.S. expression resolves to a *bool* value. In this program, *NOT* is done on an *int* variable.

21. *invalid_unary_operation_on_string.go* : The unary minus expression is well typed if the R.H.S. expression is well typed and resolves to a numeric type (*int*, *float*64 or *rune*). Here, the unary minus is implemented on a *string* variable.

22. $invalid_usage_of_variable_before_declaration.go$: In GoLite, identifiers must be declared before they are used. In this program, *a* is initialized with the value of *b*, but *b* is declared after *a*.

23. *invalid_use_of_or_operator.go* : The binary operation || (logical *OR*) is well typed if both its operands resolve to *bool* types. In this program, both the operands for the || operation are *int* values, which are not supported in GoLite.

24. *invalid_variable_array_size_declaration.go* : In GoLite, arrays declared must have a constant size and the size cannot be a variable. This program is in direct conflict with that rule.

25. *invalid_while_loop_int_condition.go* : The condition for *while* loop must resolve to a *bool* value. However, in this program, the condition for the *while* loop has an *int* type.

26. *mismatched_type_in_op_assignment.go* : In Go, the "+ =" operation requires both its operands to be of same type. In this program, L.H.S operand is of type *float*64 and R.H.S. operand is of type *int*.

27. *redeclaration_of_variable_in_same_scope.go* : This is actually in violation of a scoping rule. According to the scoping rules, a variable declared in a higher scope can be redeclared in a lower scope, but a variable (re)declared in a certain scope cannot be redeclared in the same scope (which is what is being implemented in this program).

28. *switch_case_expression_type_mismatch.go* : In a switch statement, the case expressions must have the same type as the switch condition. In this program, the switch condition expression is of type *int* but the case expressions are of type *rune*.

29. *type_redeclaration.go* : This program violates the same scoping rule as no. 27, which is applicable for types as well.

30. *undeclared_variable.go* : A variable cannot be used or accessed without declaring it, either by *var* or ":=" declarations.

31. *invalid_modulo_op_rune_float.go* : A modulo (%) operation is well typed if both its operands are integer values (*int* or *rune*). In this program, the % operation is done on a *rune* and a *float*64 operand.