

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

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

32 The available modes are :

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

39 **3. Scanner**

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

45 One challenge that we faced in the scanner was optional semicolons. We
 46 had to deal with this by creating a variable called *lastRead*, and updating it
 47 whenever a new token is generated (before it is returned). then, when \n or
 48 **EOF** or block comment including a \n is encountered we check to see with

49 function *insertSemicolon* if there should have been a semicolon there, and
50 we insert the semicolon in that case.

51 Another challenge we faced was handling block comments. Dipanjan ini-
52 tially added a state machine implementation of the handling of block com-
53 ments. However, it was too abstract and none of us knew how exactly it
54 was handling the block comments. So we later decided to switch to regular
55 expressions.

56 4. Parser

57 Defining the production terms was a bit complex. We had a union struc-
58 ture with all the types of nodes we can have in our syntax tree. We then
59 typed the production nodes according to these node types. Thus the syntax
60 tree was created according to the grammar rules in the parser.

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

66 For the error message generation and display, we used bison's error re-
67 porting function *yyerror*.

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

- 70 • = (Assignment operator)
- 71 • || (Logical OR operator)
- 72 • && (Logical AND operator)
- 73 • ==, !=, <, >, <=, >=
- 74 • +, -, |, (XOR operator)
- 75 • *, /, %, <<, >>, &, AND-XOR Operator
- 76 • All unary operators

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

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

83 5. Syntax tree

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

- 91 • *PROGRAM* : Program node, root or beginning of the tree.
- 92 • *PACKAGE* : Package declaration.
- 93 • *TOPELDECL* : Top level declarations. Can be one of the fol-
94 lowing:
 - 95 – *FUNCDCL* : Function declaration
 - 96 – *DCL* : Non-function declaration. They can be :
 - 97 * *VARDCCL* : Top level variable declaration
 - 98 * *TYPEDCL* : Top level type declaration
- 99 • *FUNC_SIGNATURE* : Function signature.
- 100 • *PARAM_LIST* : Parameter list for a function signature.
- 101 • *IDLIST* : List of identifiers for short declarations, assignments and
102 top-level declarations.
- 103 • *TYPE* : Type of a declaration.
- 104 • *STRUCT_TYPE* : Block of the structure.
- 105 • *BLOCK* : Block of code enclosed by braces.

- 106 • *STATEMENTS* : Node acting as a linked list for consecutive state-
107 ments.
- 108 • *STATEMENT* : Program statement.
- 109 • *ELSE_BLOCK* : Special node for *else* part of an *if* block.
- 110 • *SWITCH_CONDITION* : Condition for a switch block.
- 111 • *SWITCH_CASELIST* : Linked-list like node for all the cases and
112 default case for the switch block.
- 113 • *EXPRLIST* : Linked-list like node for expressions (for multiple as-
114 signments in one line).
- 115 • *FOR_CONDITION* : Node for the condition of a for loop.
- 116 • *SIMPLE* : Simple statement.
- 117 • *OTHER_EXPR* : Other statements (including function calls, struct
118 member selector and slice indexing).

119 6. Weeder

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

124 6.1. Single default case

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

131 6.2. Break and continue statements

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

144 6.3. Blank identifier

145 According to Go specification, blank identifiers can be used :

- 146 • as an identifier in a declaration
- 147 • as an operand on the left side of an assignment
- 148 • as an identifier on left side of = assignment only
- 149 • as a parameter name in function declaration

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

- 152 • the weeding of identifier lists of a declaration
- 153 • *LHS* of an assignment statement
- 154 • *LHS* of a short declaration statement

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

159 6.4. Unbalanced assignments

160 If the *LHS* and the *RHS* of an assignment or declaration statement
161 contain unequal number of operands, the compile should quit with appropriate error. This is implemented with two local variables *lhsCount* and
162 *rhsCount*. The *weedIDLIST* and *weedEXPRLIST* functions return the number of operands in the idlist in the *LHS* of an assignment or declaration, or
163 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
164 thrown. Otherwise, the counters are reset to zero for the next statement.
165
166
167

168 6.5. Return statements

169 The language specifies that return statements can only be inside a function body. We use a static global integer variable *insideFunction* that is
170 initially set to zero. When we are weeding the function declaration, we increment the *insideFunction* variable by one and then start weeding of the
171 function block. As soon as weeding of the block is done and control comes back, we decrement *insideFunction* by one. If a return statement is encountered
172 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
173 supported in the language yet, but its a nice provision to have).
174
175
176
177

178 7. Pretty Printer

179 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
180 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
181 printer. This was done by Archit.
182
183

184 Previously, we had a single function split into cases based on the kind of statement/expression inside one function. This time around, we created
185 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
186 printer starts. Semicolons are printed at the end of every statement.
187
188

189 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*.
190 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
191 to print spaces instead of tabs because we have see tabs to create chaotic
192
193

194 indentations in several editors. Thus to keep a standard across all platforms
 195 for pretty printing, we used spaces.

196 8. Symbol Table

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

202 The data structure for the symbol table is as follows:

- 203 • *SymbolTable* : The main symbol table frame structure. It consists of :
 - 204 – *SYMBOL * table[HashSize]* : An array of *SYMBOL* type for
 205 symbols in the symbol table. The *HashSize* is set to be 317.
 - 206 – *SymbolTable * parent* : Pointer to the parent table of the current
 207 symbol table (you can think of it as pointer to the previous element
 208 in stack)
- 209 • *SYMBOL* : Data structure to define the symbol. It consists of :
 - 210 – *name* : String to store the name of the symbol.
 - 211 – *SYMBOL * next* : Pointer to the next symbol in the current
 212 symbol table.
 - 213 – *symTYPE * data* : Type of the symbol and additional information
 214 about the symbol.
- 215 • *symTYPE* : This structure is used to store the type of the symbol.
 216 It is used for printing the symbol table and can furthur be used in
 217 typechecking. It consists of :
 - 218 – *enumSymbolCategory* : The category of the symbol. It can be :
 - 219 * *type_category* : Declared type.
 - 220 * *variable_category* : Declared variable.
 - 221 * *function_category* : Declared function.
 - 222 * *constant_category* : Used specifically for "true" and "false".
 223 It is declared and added to the root symbol table for shadow-
 224 ing in the future.

- 225 – *enumsymbolType* : It is used to identify whether it is a function
226 declaration or not (as they are treated differently).
- 227 – *unionval* : This contains a pointer to the *TYPE* and *FUNC_SIGNATURE*
228 nodes of the AST.

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

233 8.1. *symIndent*

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

237 8.2. *Hash*

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

241 8.3. *initSymbolTable*

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

246 8.4. *scopeSymbolTable*

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

253 8.5. *putSymbol*

254 This function is used to store a symbol in the current scoped *SymbolTable*
255 frame.

- 256 • If the symbol is "_", the symbol is not put in the symbol table and a
257 *NULL* is returned.
- 258 • If the symbol is already defined in the current scoped frame, an error
259 is thrown.

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

263 8.6. *scopeInc* and *scopeDec*

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

267 8.7. *printSymbol*

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

275 8.8. *getSymbol*

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

281 8.9. *defSymbol*

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

286 8.10. *initSymType*

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

291

292

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

295 8.1. *Predeclared symbols*

296 The symbol table contains the following predeclared symbols :

- 297 1. **int**
- 298 2. **float64**
- 299 3. **rune**
- 300 4. **string**
- 301 5. **bool**
- 302 6. **true**
- 303 7. **false**

304 The last two symbols are of constant type and used for shadowing pur-
305 poses.

306 8.2. *Scoping*

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

- 311 • code enclosed in braces
- 312 • function declaration
- 313 • for loop
- 314 • switch block
- 315 • case in a switch block
- 316 • if block

317 • else-if/else block

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

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

324 8.3. Short Declarations

325 Another challenge we faced was to ensure that every short declaration
326 must have at least one undeclared variable. To implement that, we used a lo-
327 cal boolean variable *atLeastOneVarNotDeclared* (initially set to false). We
328 looped through the L.H.S. of the short declaration statement and checked
329 whether a variable is defined in the symbol table or not. If we see that
330 the variable is not defined, we add that variable to the symbol table and
331 change the value of *atLeastOneVarNotDeclared* to true. If, after travers-
332 ing through the identifier list in the L.H.S. of the statement, the value of
333 *atLeastOneVarNotDeclared* is false, we throw an error and exit.