COMP 520 Milestone 2

Rajveer Gandhi, Archit Agnihotri, Dipanjan Dutta

Abstract

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

1 Introduction

For the purposes of our project, we use a subset language of *Go*, called *GoLite* and implement the specifications according to those given in class. In this milestone, we were asked to design and implement the symbol table and the typechecker. We were given the design specifications and we implemented the symbol table and the typechecker according to these specifications.

In this report, we aimed for lucidity and reproducibility and covered most of the details of our design.

2 Design Decisions

2.1 Syntax tree

The syntax tree created in the first milestone had a lot of bugs so we decided to recreate the syntax tree altogether. Previously, we had one single convoluted NODE with a single type. This time around, the node is split up to support multiple types of nodes. This allowed us to easily manage the coding structure and help traverse the syntax tree with lesser difficulty. 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).

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 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.

2.2.5 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 the type is not given, we set the type as $NULL$. If *g_tokens* is 1, the symbol is printed as well. It also returns the symbol.

2.2.6 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.

2.2.7 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. If the type is not known or to be derived from R.H.S. of an expression, "<infer>" is printed as an indication that it has to be inferred during typechecking phase.

2.2.8 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.

2.2.9 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.

2.2.10 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.

2.2.11 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.

2.2.12 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 :

- struct body
- 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.

2.2.13 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.

2.3 Typechecker

The typechecker was created in a fashion very similar to what was discussed in the lecture slides. The typechecker has a recursive approach where a program component is well-typed if all its sub-components are well typed and it follows its own typechecking rule. In order to implement typing comparisons and set types of expressions, we used the *symTYPE* struct from **symbol.h**. A new variable of this structure is created after evaluating an expression and the return type of that expression is stored in this struct. We decided to use this struct because it helps brings a standard in the typing of expressions in our code. We added a *symTYPE* variable to the *EXPR* and *OTHER_EXPR* nodes.

2.3.1 Scoping rules

The scoping rules are consistent with the Milestone 2 language specifications.

- Blocks define a scope. Blocks can be :
  - struct body
  - code enclosed in braces
  - function declaration
  - for loop
  - switch block
  - case in a switch block
  - if block
  - else-if/else block

- A variable/type/function declared in a scope can be accessed in that scope
- A variable/type/function declared in an outer scope can be redeclared in an inner scope.
- A variable/type/function declared in current scope cannot be redeclared in current scope.
- A variable/type/function cannot be accessed before declaration.

2.3.2 Helper functions

In order to implement typechecking, we created a few helper functions to perform some tasks need for type-checking.

- *initTypes* : Creates the types for the predeclared symbols in the symbol table. (*int*, $float64$, *rune*, *string*, *bool*, "*true*" and "*false*"). "*true*" and "*false*" are set as boolean types for shadowing.
- *hasSameType* : It checks whether two given $symTYPE$ variables have same base type or not. If they do, the function returns *true*. Otherwise an error is thrown.
- *isBool*, *isNumeric*, *isInteger*, *isNumericOrString* : These four helper functions check whether a given type is the same as the other one or not. If it is not, it gives an error message saying that it is an incorrect type and exits with status code 1. Otherwise, it returns *true*.

2.3.3 Top-level declarations

A top level declaration can be of three types. Each has its own typechecking rule.

- Variable declaration : For every variable declaration in the top-level, we go through the L.H.S. of the declaration (*idlist*) and we see if there is an expression on the R.H.S., it should be well-typed. Then we check for previous declaration of the variable. If undeclared, we add a mapping of the variable to the type of the R.H.S. expression. If it is declared in an outer scope, we create the new mapping as a shadow of the outer mapping. If it is declared in the current scope, we throw an error.
- Type declaration : Similar to variable declaration. We check if it is declared previously. If undeclared, we add a mapping of the type to the underlying type. If it is declared in an outer scope, we create the new mapping as a shadow of the outer mapping. If it is declared in the current scope, we throw an error.
- Function declaration : We check the function signature first. If the function of that name is already declared, we throw an error. Otherwise we typecheck the block of the function. We have two special functions that needed to be typechecked with special conditions :
  - *init* : The *init* function cannot have any parameters or return types. It can, however, be redeclared.
  - *main* : The *main* function cannot have any parameters or return types.

  After this we recursive typecheck the block of the function.

2.3.4 Statements

Typechecking on statements is done using a linked-list like node $STATEMENTS$. For every statement, typechecking is done recursively for every component of the statement. Note that statements themselves do not have any a type of their own.

- Variable and type declarations : Similar to top-level declarations.
- Blocks : Blocks are typechecked recursively. It is well typed if all the statements in a block are well typed.
- *break* and *continue* statements : These statements are trivially well-typed. Placement of these statements are checked by the prior weeding pass.
- *return* statement : *return* statements are well-typed if :
  - The *expr* component is well-typed and the resolved type of *expr* is the same as the enclosing function return type.
  - There is no *expr* and the enclosing function has no return type.

Also, one thing we missed initially, but caught late on, is that we should typecheck all statements after the *return* statement, as it can be within a conditional or iterative block inside the function.

- if block : An if block is well-typed if :
  - Its initial statement is well typed (typechecked recursively for statement)
  - The conditional expression is well-typed and resolves to *bool*
  - The statements in the blocks typecheck (done by recursively typechecking the block)
- *switch* block : A *switch* block is well-typed if :
  - Its initial statement is well typed (typechecked recursively for statement)
  - The conditional expression is well-typed
  - The *case* expressions are well-typed and they resolve to the same type as the *switch* condition, otherwise an error is thrown.
  - All the statements under every *case* are recursively typechecked to we well-typed
  - If there is no *switch* condition (the $symTYPE$ of the condition is NULL), then all *case* expressions must resolve to a *bool*.
- for loops : There are 3 kinds of "for" loops :
  - Infinite loop : It is well-typed if its body block is well-typed.
  - "*while*" loop : It is well typed if its conditional expression is well-typed and resolves to *bool* and its body block is well-typed. If the condition does not resolve to *bool*, an error is thrown.
  - for loop : It has three parts :
    * The initial statement is recursively checked to be well-typed.
    * The condition typechecks and resolves to *bool*, otherwise an error is thrown.
    * The post-loop operation statement typechecks.

    If the above three conditions are valid, we typecheck the body of the loop to be well-typed.
- *print* statements : *print* and *println* statements are well-typed if the expressions enclosed in it are all well-typed and resolves to a base type (*int*, $float64$, *rune*, *string*, *bool*). Declared types, even if derived from base types, are not allowed.
- Empty statements : They are trivially well-typed
- Increment and decrement statements : These are well-typed if the L.H.S. expression is well-typed and resolves to a numeric base type (*int*, $float64$ or *rune*)
- Short declarations : This was one of the trickiest typechecks. We checked that :
  - All expressions on R.H.S. and L.H.S. are well-typed (by recursion)
  - There is one variable in the L.H.S. that is not declared in the current scope (done by using a flag). If all variables are declared in current scope, an error is thrown.
  - The expressions in the R.H.S. must resolve to same type as the corresponding variables in L.H.S. that are declared in current scope (present in the current frame of symbol table).

  This was one of the hardest typechecking. We performed several tests on the reference compiler and the Golang compiler and cross-referenced with the given specifications to get it right as much as possible.
- Assignment statements : For all assignment statements, both L.H.S. and R.H.S. are checked to be well-typed for every corresponding pair, by iterating over the *idlist* on L.H.S. and recursivey typechecking both L.H.S. and R.H.S. Every variable in the L.H.S. is checked to be already declared. Then for every pair, it is checked that the type of R.H.S. expression is the same as the type of L.H.S. variable as stored in the symbol table. If there is a type mismatch, an error is thrown.
- Op-assignment statements : Similar to assignment statements. For $vop = expr$, The operator acts as a function that has 2 operand of types $typeof(v)$ and $typeof(expr)$. Unlike assignment statements, the op-assignment returns a value of type same as $typeof(v)$.

2.3.5 Expressions

Expressions are typechecked in a way similar to statements. Iterating over the linked-list like node $EXPRLIST$, we typecheck each expression recursively. The type of expression is selected using a switch-case block and typechecking is done according to whichever case is matched with the kind of expression.

- Expressions with binary operators : They are recursively typechecked for both L.H.S. and R.H.S. If they are well-typed, we check whether they resolve to same types and also, whether these types are accepted by the operator. For this, we used the specification given as reference and mimicked the behavior of the operators as specified. Finally, we set the type of the expression (by declaring a $symTYPE$ variable) and set type to the resolved type of the expression, according to the specification table given.
- Unary operations : These are typechecked similar to binary operations. We check if the R.H.S. expression is well-typed. If it is, we look at the operator and see if the R.H.S. type is compatible with the operator (as mentioned in the specification). If so, we set $symTYPE$ as the resolved type of the expression, otherwise we throw an error.
- Literals : These are trivially well-typed. We only check that a literal node has the corresponding type correct. For example, an *int* literal should have an *int* type etc.
- Identifiers : We check the identifier name in the symbol table. If present, it is well-typed and $symTYPE$ is set as the type of the identifier in the symbol table. If it is not found in the symbol table, an error is thrown.
- Function calls : This was another difficult typechecking that we had to handle. Our approach was :
  - Recursively check all arguments are well-typed by iterating over the argument list.
  - The function is defined and has the parameter types same as that of the argument list (done by symbol table lookup).
  - The function name is not "*init*" (it cannot be called)
  - Since type casting is also handled as a function call, we handled them under this expression as well :
    - The type resolves to one of the base types (by string comparison)
    - The expression that is to be cast must be one of the types that are allowed to be cast to the target type. This is done by recursive typechecking the expression. (according to the specification)
    - By recursively accessing the symbol table, we find out the underlying type of the type to be cast to and match it to the type of the expression, or both of them are numeric types, or it behaves like a conversion to string, where *type* can be *string* and *expr* can be an integer (*int* or *rune*). If not, an error is thrown.
- Array or slice indexing : We checked that the name of the array or slice is well-typed and present in the symbol table. Then we checked that the index specified is of type *int*. If so, then the expression's $symTYPE$ has a type same as that specified in the symbol table. We do not check for out-of-bounds access.
- Struct member access : For expression like $a.x$, we :
  - Checked the type of a in the symbol table and if it is not a struct, we threw an error.
  - Checked if the nested symbol table frame of that struct has a field declared called x. If not, an error is thrown.

  If both of these are valid, the expression returns the type of the variable$x$, as defined in the struct.

2.4 Weeder

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

2.4.1 Single default case

The language specifies that there can be at most 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.

2.4.2 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.

2.4.3 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.

2.4.4 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.

2.4.5 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).

2.4.6 Terminating statements

The language specifies that for functions that has a return type, must have a terminating statement as the last statement. We implemented it by passing through the function body after weeding the block and finding the last statement of the body. We pass that statement as parameter to the helper function $isTerminating$ in order to check if its a terminating statement or not.

- $return$ statement : If the last statement is a $return$ statement, the function returns true.
- block : If the last statement is a block, we pass through the statements of the block and recursively call $isTerminating$ with the last statement of that block and returned the value returned by the recursive call.

- if statements : We declare two local *bool* variables : $ifTerminating$ and $elseTerminating$, both initially set to false.
  - If the if statement does not have an *else* part, the function returns zero.
  - If it has an empty if block, zero is returned, otherwise we recursively call $isTerminating$ on the last statement of the block and store the value returned in $ifReturning$.
  - If it has an empty *else* block, zero is returned, otherwise we recursively call $isTerminating$ on the last statement of the block and store the value returned in $elseReturning$.
  - If it has an empty $else - if$ block, zero is returned, otherwise we recursively call $isTerminating$ on the $if - else$ block and store the value returned in $elseReturning$.

  If, in the end, we return the value of $ifReturningANDelseReturnung$.
- for loop : According to the specification, a for loop is terminating if there are no $break$ statements in the body and there is no loop condition. If these conditions are satisfied, 1 is returned otherwise 0 is returned.
- *switch* block : A *switch* block can be a terminating statement if there is no $break$ in it (checked by traversing through all statements under ever *case*), it has a $default$ case (checked using a local *bool* flag $isDefault$) and all the cases end in terminating statements (checked by recursive call on the last statement of every *case* in the case-list). If all the conditions are satisfied, the function returns 1 otherwise it returns 0.
- For all other statement types, the function returns zero.

If the returned value if $false$ or zero, we throw an error and exit with status code 1.
Additionally :

- If a function with non-void return type has a *return* without an expression, an error is thrown.
- If a function with no return type has a *return* statement with an expression, an error is thrown.

2.5 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. Here, 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.

3 Team member contribution

- **Archit Agnihotri**
  - Created the symbol table
  - Created the typechecker
- **Rajveer Gandhi**
  - Created the symbol table
  - Created the typechecker
- **Dipanjan Dutta**
  - Created the weeder
  - Created the invalid programs
  - Managed the report
- **All of us**
  - Tested and debugged the programs