COMP 520 Final Report

Rajveer Gandhi, Archit Agnihotri, Dipanjan Dutta

Abstract

This is the final report for our COMP 520 GoLite project. Here we summarize the project and provide our decisions and contributions to the development of the project.

1 Introduction

Go is an open source programming language from Google, developed by Robert Griesemer, Rob Pike and Ken Thompson. Go inherited mostly from Oberon language and its syntax is from C. Go's OOP is more like Smalltalk but in Go, you can attach methods to any type. And concurrency is mostly from Newsqueak which is a language also from Rob Pike and heavily inspired from Hoare's paper of CSP.

Features of Go programming language :

- **Fast results**: It works like an interpreted language because of the fast compilation.
- **Safe**: Strongly and statically typed and garbage collected.
- **Easy to work with**: It's concise, explicit and easy to read.
- **Modern**: Built-in support in the language itself for multi-core networked distributed applications.

Advantages of Go programming language :

- There is no VM. It compiles directly to the machine code which is fast.
- The programming language design is built for fast compilation.
- Compiles cross-platform to OS X, Linux or Windows.
- Creates only one executable file output after the compilation without any dependencies, so that you can upload it anywhere which Go supports and just run it.

For our project, we use a subset of features available in Go as our source language. This subset language is called GoLite. It has lesser features than Go but holds all the parser specifications of Go. Features not available in GoLite are:

- Goroutines and channels
- Interfaces and methods
- Closures
- The *defer* keyword
- Maps
- Multiple return values
- Module system
- Garbage collection

However, GoLite is an excellent programming language to choose as a source language because it is fast and simple.

This report has the following subsections

- Implementation and Target languages
- Scanner
- Parser
- Weeder
- Symbol Table
- Typechecker
- Code generator
- Conclusion
- Team member contributions
- References

2 Implementation and Target languages

For the implementation of the compiler we decided to use C as our programming language as all of us are comfortable with programming in C. To create the scanner and parser, we used two C-based Unix utilities, *flex* and *bison*. These tools are written in C and their output is C code.

Advantages of using C with Flex and Bison as a implementation language :

- For complex and diverse languages, Flex and Bison capture patterns and parses efficiently.
- Simpler to code and interpret.
- Easy to debug with helpful messages.

As our target language, we agreed on Python for the following reasons:

- All of us knew and were comfortable with it.
- Has some level of abstraction so that we do not have to code in a rudimentary manner.
- Has most of the tools to implement the constructs of GoLite without too much hand-writing.
- Python is dynamically typed, meaning we can leave object types to be handled at runtime.
- Lack of extra characters like semicolons and curly braces.

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.

Tokens were aliased with the addition of t before their natural names and we did not have special tokens for predeclared names as they were not reserved keywords in GoLite.

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). 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, as with our initial regular expression we found while testing that they wouldn't be handled appropriately if they had a new line within them. To handle this, we just added a special case for block comments with newlines in them.

4 Parser

Defining the production terms was a bit complex. We had a union structure with all the types of nodes we planned to have in our syntax tree. We then typed the production nodes according to these node types. Hence the syntax tree was created in accordance to the grammar rules in the parser. Almost all the tokens were of type string except integer and float literals which had their respective types.

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.

Major Design Decisions:

- Assigning the multiple types of nodes in order to have maximum efficiency and manageability of code in later stages.
- Reducing any shift-parse errors
- Treating typecasting as a function call and handling it in later stages instead of treating it as a separate expression type.
- We refactored the entire parser files tree.h and tree.c from a single struct NODE to about 15 different struct types. This significantly simplified the project.

5 Weeder

The weeder is implemented in order to catch any syntactic errors that might not have been directly caught in the parser's grammar. For the purposes of passing the grading scripts, the weeder is split into two phases, once after the parsing phase and once after the typechecking phase. The following checks were implemented in the weeder:

5.1 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 initially 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 correct weed through nested switch cases which are recursive calls. Once the switch block is complete, *hasDefault* is set back to zero.

5.2 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.

5.3 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.

5.4 Unbalanced assignments

If the *LHS* and the *RHS* of an assignment or declaration statement contain unequal number of operands, the compiler 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.

5.5 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).

5.6 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.

6 Symbol Table

The symbol table was created using 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 flag 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.
  - $DataType * data$: Type of the symbol and additional information about the symbol.
- $DataType$: 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.
  - $enumtypeKind$: It is used to identify whether it is a function declaration or not (as they are treated differently).

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.

6.1 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.

6.2 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.

6.3 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.

6.4 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.

6.5 putSymbol

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

- If the symbol is a blank identifier, 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.
- If the symbol is *init* or *main*, an error is thrown if it is not a function declaration type of symbol.

Otherwise, the symbol is added to the current frame, along with its data 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.

6.6 addSymbolType

This function is used to shadow types from parent symbol table frames.

- If the type is a blank identifier, the symbol is not put in the symbol table and a $NULL$ is returned.
- If there is no parent symbol table frame and the type has not been found yet, an error is thrown.

Otherwise, the type of the current type symbol is set as the one with the same name in the current or any parent frame.

6.7 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.

6.8 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 of the symbol is not defined yet, it prints $< infer >$.

6.9 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.

6.10 getSymbol_no_error

This is same as the *getSymbol* function. However the only difference is that when the symbol is not found, instead of throwing an error and quitting, it returns $NULL$.

6.11 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.

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

- 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.
- 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 they 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.
- init functions : *init* functions are not added to the symbol table as function declarations. Their declarations are ignored. However, their body is still scoped and treated like a normal block of code.
- 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). First, we checked that all expressions in R.H.S. are in scope. 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.

- Append expression : For *append* expressions, we first checked that the identifier in the append expression is in the current scope or any of the parent scopes of the symbol table or not. If present, only then we proceed to check that the expression in *append* is in scope.
- Struct access expressions : First, we check that the expression/field in struct access expression is in scope. If it is in scope and it is an identifier, then we retrieve the symbol from the current scope or any of the parent scopes of the symbol table, that is reachable from the current scope. If the symbol is of type struct, then we try to find the identifier name of the struct from the symbol table, if present. Otherwise an error is thrown.

7 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.

7.1 Helper functions

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

- *resolveType* : Resolves the type of a symbol to one of the predeclared base types (*int*, $float64$, *rune*, *string*, *bool*) by string comparison. If it is not one of the base types, it recursive calls upon the type of the symbol until it reaches a base type. It does not resolve type for slices, array or structs.
- *validCast* : It checks whether two given symbols are compatible for casting or not. It checks if the cast and the expression are of same types (basic, array, slice or struct). If so, it checks for basic types :
  - If they have the same underlying types
  - If both of them resolve to *rune*, $float64$ or *int* (numeric types)
  - If an integer/rune is being cast into a string

  If none of them is true, the function throws an error for invalid cast. Array, slice and struct casting remains unimplemented.
- $isBool$, $isNumeric$, $isInteger$, $isNumericOrString$, $isBaseType$, $isFunction$: These six helper functions check whether a given type is the same as one of them 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*.
- $isSliceOrArray$, $isSlice$: These two helper functions check whether a given type is not slice or slice/array. 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 the enclosing type of the slice or array by creating a new $DataType$ variable and returning that variable.
- $isStruct$: Checks whether a given symbol is a struct type or not. If not, then an error is thrown and program exits.
- $mustHaveSameType$: It takes two types ($DataType$ variables), and checks whether they are the same type or not. It first checks if the two types are comparable or not. If comparable, it checks the following :
  - For basic types, it compares if the types are same or not.
  - For slices, it recursively evaluates the equality of the underlying types.
  - For arrays, it additionally checks if they have the same sizes.

  Struct comparison is unimplemented.
- $handleSpecialFunctions$: It checks that *main* and *init* functions do not have any parameters and return types.

7.2 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 check if there is an associated expression list. If not, there is nothing to typecheck as the symbol is added to symbol table in previous script. Then we loop through the *idlist* and *exprlist*. If the variable has a declared type, then we check whether the expression and the variable are of same types. If the variable does not have a declared type in the declaration, we just use the expression's type and add it tot he symbol table as the type of the variable. We ignore blank identifiers. This is done for all the variables in the declaration.
- Type declaration : Since types are already add to the symbol table, there is nothing to typecheck for type declarations. Hence we did not do any typechecking for type declarations.
- Function declaration : We check first that the function name is not *init* or *main*. If so, they are checked to have no parameters or return types. Then the return type of the function is stored in a *DataType* variable and the function body is typechecked.

7.3 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 : We call the same function that we used for top level declarations.
- Blocks : Blocks are typechecked by iterating over the statements in the block. All statements inside a block are typechecked.
- *break* and *continue* statements : These statements are trivially well-typed. Placement of these statements are checked by the prior weeding pass.
- *return* statement : We compare the return type of the function with the type of the expression in the return statement. If the types are not same, an error is thrown. 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 :
  - We first see that the initial statement, if present, is well typed.
  - We check that the conditional expression is well-typed and resolves to *bool*
  - If there is no *else* block, we typecheck the if block.
  - If there is an *else* block, we typecheck the block associated with the *else* block.
  - If there is an $elseif$ block, we typecheck the if statement associated with the *else* block ($elseif$ is treated as a new if statement associated with the *else* block).
- *switch* block : We perform the following typechecks for *switch* statement :
  - We first see that the initial statement, if present, is well typed. (typechecked recursively for statement)
  - We check that the conditional expression is well-typed
  - We evaluate the type of every *case* condition. If there is a *switch* condition, we check that the *case* condition and *switch* condition have same types. If there is no *switch* condition, we check if the *case* condition evaluates to *bool*
  - If there are statements under a *case*, all the statements are recursively typechecked to we well-typed.
  - If there is a $default$ case, and if it has statements, the statements are typechecked.
- for loops : There are 3 kinds of "for" loops :
  - Infinite loop : The loop body or block is typechecked.
  - "$while$" loop : We check if the loop condition expression typechecks and resolves to *bool* type. Then we typecheck the loop body.

- *for* loop : It has three parts :
  - The initial statement, if present, is typechecked.
  - The condition expression, if present, is typechecked and checked if it resolves to *bool*.
  - The post-loop operation statement, if present, is typechecked.

  Then we typecheck the body of the loop.

- *print* statements : We first check that the expression list associated with *print* and *println* statements are well-typed. If so, then we check if every expression resolves to a base type. If it is true, then a *print* or *println* statement is well-typed.
- Empty statements : They are trivially well-typed
- Increment and decrement statements : We check whether the expression is well typed. If it is, we check if the expression resolves to a numeric base type (*int*, *float*64 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) and the lengths of the lists are the same.
  - We ignore all blank identifiers in the L.H.S.
  - If the variable has a type in the symbol table, we check if the expression has the same type as the variable.
  - If the variable does not have a type declared in the symbol table, we update the type of the variable in the symbol table with the expression type.
- Assignment statements : We typecheck L.H.S.and R.H.S. expression lists. We loop through the expression lists (which should be of same size, as weeding pass before this ensures that). For every L.H.S. expression, we check whether its a variable (having a lvalue) or not. If not, an error is thrown. If it is an lvalue, we check whether the L.H.S. expression and corresponding R.H.S. expression are of same type. This is same for assignment and op-assignment statements.

7.4 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 *DataType* 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 *DataType* as the resolved type of the expression, otherwise we throw an error.
- Literals : These are trivially well-typed. We set the type of the literal to the corresponding type. For example, an *int* literal is set an *int* type etc.
- Identifiers : We only had to check for blank identifiers and ignore them. Other invalid cases have been weeded out before already. If it is not a blank identifier, we retrieve the symbol from the symbol table and set the type of the identifier to that found in the symbol table.
- 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.
  - See if the identifier used for function call is a function or not in the symbol table. If it is, then it is a function call. Otherwise, it is a type casting.

  - If it is a function call, we check whether the parameter and argument types match and whether they are equal in number or not. If not, an error is thrown and the program exits.
  - If everything passes, we assign the type of the function call expression as the return type of the called function (by creating a *DataType* variable.
  - If the expression is a type casting, we handled them under this case as well :
    * We check that the type resolves to one of the base types. (by string comparison)
    * We check that there is exactly one argument. No or more than one argument throws an error.
    * We call the *validCast* function to check whether the type casting is valid or not (according to the specification). The arguments to *validCast* are the resolved types of the cast and the expression. This is done by recursive typechecking the expression and the cast.
    * 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 check that the expression for indexing is well-typed. We check that the identifier resolves to an array or slice (helper function) according to the symbol table. We check that the index specified resolves to type *int*. If so, then the expression's *DataType* 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 :
  - We check if x typechecks
  - We check that a resolves to a struct according to the symbol table.
  - We check if the symbol table of the struct contains the variable. If so, it retrieves the symbol and sets the type of the whole expression as the type of the variable/identifier. If not found, we throw an error.

8 Code Generation: Python

For the last phase of our project, we chose to generate Python code for the reasons specified in *Section 2*.

8.1 Identifiers

The names of Go identifiers must start with a letter or an underscore and may be followed by additional letters, digits and underscores. These names are case-sensitive.
Since Go does not allow unused local variables (results in a compilation error), we use blank identifiers as an alternative (denoted by _). These blank identifiers are mostly used when program syntax may require a use but the program logic doesn't.

Go also has 25 keywords which cannot be used as names. These keywords are:

break	default	func	interface	select	case	defer	go	
map	struct	chan	else	goto	package	switch	const	
fallthrough	if	range	type	continue	for	import	return	var

Go also has 36 predeclared names for built-in constants, types and functions. These names are:

Constants:

true	false	iota	nil

Types:

int	int8	int16	int32	int64	uint	uint8	uint16
uint32	uint64	uintptr	float32	float64	complex128	complex64	bool
byte	rune	string	error				

Functions:

make	len	cap	new	append	copy	close
delete	complex	real	imag	panic	recover	

These names are not reserved, so we may use them as identifiers.

Identifiers in Python are defined in the same way as Go, and blank identifiers are a viable way to use unused variables so we may not have to make any changes on this front.
On the other hand, Python has 33 keywords. However, they will not be re-used during codegen because of our mapping strategy. Basically, since we are using a large prefix, it will not conflict with any Python keywords.
Mapping strategy:

1. For all non-blank variable names in Golite, we will keep the base name and prefix it with GOLITE__
2. Blank identifiers: because we're using a dynamic typically language, we don't any special handling of blank identifiers in Golite

8.2 If Statements

If statements in Go may or may not have an else statement. They may include an optional **simple statement** - a short variable declaration, an increment or assignment statement or a function call that can be used to set a value before the conditional is tested and is scoped within this if statement and available for use within all branches.
Note that you don't need parentheses around conditions in Go but braces are required.
Conditionals in If statements must be of the type **boolean**. If the condition is true, the lines of code between the braces is executed.

```
if num := 5; num < 0 {
        fmt.Println("negative")
} else if num < 10 {
        fmt.Println("1 digit")
} else {
        fmt.Println("multiple digits")
}
```

There may also be multiple *else if* statements wherein the condition is evaluated for truth from the top to bottom. Whichever if or else if's condition evaluates to true, the corresponding block of code is executed. If none of the conditions are true, then *else* block is executed. It is important in Go that this else statement is preceded by a closing brace and not on a new line, otherwise a compile time error will occur.

In Python, there may be no optional simple statement before the condition. To map this, we will insert the short declaration immediately above the "if" statement.

```
num = 5;
if num < 0:
        print "negative"
elif num < 10:
        print "1 digit"
else:
        print "multiple digits"
```

8.3 Loops

The *for* loop is the only loop statement in Go. It can have multiple forms. One form is the 3-part loop which is illustrated like:

```
for initialization; condition; post {
        // zero or more statements
}
```

Parentheses are never used around the three-part loop. The braces are mandatory, and the opening brace

must be on the same line as the post statement.
The optional *initialization* statement is executed before the loop starts. If it is present, it must be a simple statement. The *condition* is a boolean expression that is evaluated at the beginning of each iteration of the loop; if it evaluates to true, the statements controlled by the loop are executed. The *post* statement is executed after the body of the loop, then the condition is evaluated again. The loop ends when the condition becomes false.
If there is no *initialization* and no *post*, the semicolons may also be omitted. In this case, the loop acts like a **while** loop.
If the condition is omitted entirely entirely in any of these forms, the loop is **infinite**.
These loops may be terminated by using a **break** or **return** statement.

In Python on the other hand, for loops follow a different syntax to iterate over a list or a string: for *item* in *sequence*:
    statement(s)
Here each object in sequence is assigned to item one at a time and the following statement(s) are executed.

Mapping Strategy:

1. We can redefine an infinite loop to be handled using a while loop:
   while true:
       statement(s)
2. Since Python has while loops, all for loops in Golite without initialization and post may be redefined in our code generator to output:
   while *expression*:
       statement(s)
3. For the threepart Golite for loop, we will have the initialization statement immediately preceding the start of the loop, and the post statement as the last statement inside the while loop:
   initialization
   while *expression*:
       statement(s)
       post

To handle continue statements, we had 4 different cases:

1. If post expression exists, and continue statement is the first statement
2. If post expression exists, and continue statement is not the first statement
3. If post expression exists, and continue statement does no exist
4. If post expression exists and there are no statements in the loop

To handle these cases, we manipulated the AST in order to either insert post expression statement at the end of the block or immediately preceding the first encountered "continue" statement in the block.

8.4 Scoping Rules

In Python, each function and class introduce their own scope. Also, there is no other easy built-in way to introduce a new scope such as with braces in C or C++. Although we were unable to implement scoping rules by the deadline of milestone 4, this is how we plan to implement them for the final compiler: we extend the symbol table to contain the codegen name of a variable, in addition to its given name in the original Golite program. This name would be the original name plus a counter variable and it would work as follows. When we insert a symbol into the symbol table in symbol.c, we search down the stack of its parents to see if that name has already been declared, and if it has, we check that symbol's codegen_name e.g. x_0. We then use the name x_1 for our new symbol that we are inserting. If it is not found then we name it x_0. Another addition we need is that for global variables that are used in functions, we would also have to declare it "global" in Python using the "global" keyword. We can accomplish this by again extending the symbol table to have a "global" boolean for a symbol. This variable is turned on if and only if the symbol appears on a left hand side of an assignment in a scope that is not the first scope (i.e. not the global scope). During codegen, we then have "global <variable>" before we modify this variable.

8.5 Variable Declarations

We implemented variable declarations with ease in Python, since it's dynamically typed and allows assigning multiple variables on a single line. For each basic type we did not do anything unless expressions were specified, in which case it is just a Python assignment. In the case where we had slices or arrays, we declared the variable as we would in Python e.g. slice = [] In the case of structs, we used python classes.

8.6 Types

Since Python is dynamically typed, we only needed to be concerned about the struct type. For structs, we used Python classes. Basically, where ever we encountered a struct, we created a new Python class with the given variables.

8.7 Assign statements

These were trivial in Python since we just used standard Python assignments.

8.8 Short declarations

These were trivial in Python since we just used standard Python assignments.

8.9 Switch statements

Since switch statements do not exist in Python, we used *if/elif/else* statements to handle them. For multiple expressions in case labels, boolean logic (OR) was used to evaluate the expressions left to right until one was true, in which case that body of the case was executed. The *if* statement handled the cases and *else* statement handled the default case. Also, since there is no need to handle *break* statements in such a case, these statements (if existing, since Go doesn't explicitly require them) were removed upon parsing.

8.10 Printing

Since Golite's print() and println() have different semantics than Python's print statement, we used the print_function in the __future__ library to specify the separator and the end character for each appropriate case.

8.11 Functions

The biggest caveat here was that Go is pass by value and Python is pass by reference. So in order to insert copies, we decided to use the deepcopy() method in the "copy" Python library.

9 Conclusion

In conclusion, our compiler still has a lot of bugs that need to be fixed. We have not considered overflow and underflow of variables and order of evaluation in GoLite. We found these constructs very late and we did not have enough time to implement these. We intend to do these modifications after the course for the satisfaction of creating a "near-perfect" compiler.
The experience has be an exhausting one. Archit and Raj worked on the coding part while Dipanjan worked on test programs and report writing. We had ups and downs, and faced a lot of problems on the way. However, in the end, we were able to create a working compiler that was able to compile most of the GoLite programs that we could think of.
GoLite is a simple language yet its execution and implementation is questionable and confusing at times. However, because of its simplicity, we were able to understand and appreciate the nuances of developing a compiler. In the end, we are happy that we completed the project, and have learned a lot from the mistakes we made.

10 Team member contributions

1. **Rajveer Gandhi/Archit Agnihotri**
    - Shared the work to program the scanner, parser, symbol table, type checker, codegen
2. **Dipanjan Dutta**
    - Created the weeder.
    - Created reports for Milestones 1 and 2
    - Created the required Golite programs/benchmarks and test programs for each milestone.