COMP 520 MILESTONE 2 REPORT

March 19, 2020

Matthew Lesko-Krleza
260692352
McGill University
School of Computer Science
matthew.lesko-krleza@mail.mcgill.ca

Sophia Hu
260705681
McGill University
School of Computer Science
yu.t.hu@mail.mcgill.ca

1 Contributions

1. Matthew: Worked on the symbol table, some type check, and the report;
2. Sophia: Worked on the symbol table, and was the majority contributor for type check.

2 Symbol Table

The symbol table's purpose is to store any newly declared symbols, whether it be for a variable, type, or function declaration, and their type information into the AST. Secondly, given some use of a symbol, the compiler checks if the symbol exists in the current or in any parent of the current scope. If the symbol doesn't exist, then the compiler throws a relevant error message. Finally, it determines whether a symbol has already been declared when attempting to define a new one of the same identifier and in the same scope. If the symbol has already been declared in the same scope, then the compiler throws a relevant error message. The symbol table is designed in a hierarchical manner, which abstracts the hierarchical nature of scopes. Each child symbol table represents a child scope, symbols are added to their respective symbol table corresponding to the scope they were declared in. 'Boiler-plate' code for creating and querying symbols are similar, if not, identical to the code presented in the slides in the COMP 520 course.

Here we discuss our design choices in such a manner so that the reader could reproduce our table. The first design choice we'd like to discuss is the addition of the `SYMBOL struct` data structure. A code snippet of the data structure is available below:

```c
typedef struct SYMBOL {
  char *identifier;
  SymbolKind kind;
  int shadowNum;
  union {
    TYPE_SPEC* typeSpec;
    VAR_SPEC* varSpec;
    FUNC_SPEC* funcSpec;
  } val;
  struct SYMBOL *next;
} SYMBOL;
```

The symbol is used to store information for type, variable, and function declarations. It additionally stores the identifier of the symbol, the kind of symbol (which can be either the kind of: type, variable, or function), the shadow depth, and a pointer to another symbol. The pointer to the next symbol allows us to implement a linked-list of symbols which can be traversed in a symbol table. Next, we discuss how we traverse the AST and store symbols from its nodes.

2.1 Entry Point

The entry point for the creation of the hierarchy of symbol tables is the root node of the AST: the `PROG` node, and is started by calling the `makeSymbolTable(PROG *root)` function. Afterwards, the rest of the AST is traversed by visiting the first top-level declaration `symTOPLEVELDECL(root->rootTopLevelDecl, global_scope)`.

2.2 Base Type Symbols

At the root AST node, the first symbol table is initialized and abstracts the global scope of the input program. The GoLite base types are added as symbols to the global scope with the `putBaseTypeSymbols(global_scope)` function. This function creates a symbol of kind `k_symbolKindType` for each of the base types: `int`, `float64`, `rune`, `string`, and `bool` and adds them to the root symbol table. It additionally adds symbols for the `bool` constants `true` and `false`. Symbols need to be created for the base types, because they aren't reserved keywords in GoLite, additionally, this allows a programmer to re-declare their types.

2.3 Top-Level Declarations

A top-level declaration can be of three kinds: a function declaration, a type declaration, or a variable declaration. Depending on the kind, the symbols in the top-level declaration are stored either through a `symFUNC`, `symTYPESPEC` or `symVARSPEC` function respectively for function, type, and variable declarations. Additionally, each top-level declaration is called recursively by using the linked-list structure of AST nodes. A code snippet below demonstrates how to traverse nodes in a linked-list and to parse symbols respective to the AST node type:

```
void symTOPLEVELDECL(TOPLEVELDECL *tld, SymbolTable
     *scope) {
    if (tld == NULL) return;
    symTOPLEVELDECL(tld->next, scope);
    switch (tld->kind)
    {
    case k_topLevelDeclFunc:
        symFUNC(tld->val.funcDecl, scope);
        break;
    case k_topLevelDeclType:
        symTYPESPEC(tld->val.typeDecl, scope);
        break;
    case k_topLevelDeclVar:
        symVARSPEC(tld->val.varDecl, scope);
        break;
    }
}
```

This code pattern is used across the majority of other AST node symbol table functions. We use a switch statement to determine how to parse symbols depending on the type of node. We pass the current symbol table to each function call so that symbols can be queried for, created in the scope, and new scopes can be extended from the given scope.

2.4 Function Declarations

If the function declaration has a return type, we query for the symbol that represents the return type and assign the symbol's type information to the return type's type information. Afterwards, we create a symbol for the function name itself and add it to the given symbol table which represents the current scope. Then, a symbol table is created and is extended as a child table to the one given to the function. This acts as an inner scope for the function's statements and input parameters. The function's input parameters (if there are any) are added as symbols with their respective declared types into the given symbol table, then the function's statements are parsed for symbols using the `symSTMT(f->rootStmt->val.blockStmt, innerScope)` function call.

2.5 Statements

Depending on the statement kind, the statement's contents are parsed for symbols appropriately. If a statement is a type or variable declaration, then the `symTYPESPEC()` and `symVARSPEC()` function calls are used respectively once again. Any kind of expressions and block statements that make up the current statement, like in cases for expression statements, for-loop statements, if-statements, switch statements, and return statements, are further parsed for their symbols using the `symEXP()` and `symSTMT()` function calls appropriately. Additionally, for any statement that contains block statements, a symbol table is created and extended on the one given to the function for any kind of initializing statement (part of for-loop, and if statements), then another symbol table is created and extended from the previously created one for any differentiating block statement. So in the case of an if-statement, an inner scope is declared for the initializing statement, then two other scopes are declared that extend from this inner scope, and are used for the true-condition block statement and for the false-condition block statement respectively. Most notably, assignment statements are split between colon-assign and other assignment statements. If an assignment uses a colon-assignment operator, then the statement is treated as a variable declaration, so a new symbol is declared for the identifier. Whereas for other assignment types, the left-hand side and right-hand side expressions of the assignment are simply traversed by the symbol table program using the `symEXP()` function calls.

2.6 Expressions

When an expression is simply an identifier, its symbol is queried from the given symbol table, and is added as a field into the identifier's respective EXP AST node, which will be used later on typecheck for type checking the identifier's type. Furthermore, any sort of binary, unary, field access, array access, or built-in expressions have their respective expressions passed as arguments to the `symEXP` function call so that symbols can be checked for existence.

2.7 Variable Declaration

In a variable declaration, the symbol for the identifier and the variable's type are queried to see if the exist. If the symbol for the variable already exists in the current scope, then an error is thrown, whereas if a type is given, then a symbol for the type identifier is queried to see if it exists. Additionally, identifiers are associated with their type's symbol's type information, so that they can be typechecked in downstream tasks.

2.8 Type Declaration

Type declarations are done similarly to variable declarations, where the newly defined type's identifier is queried as a symbol in the current scope to see if there already exists a type declaration of the same symbol in the same scope. And the type declaration's type is queried to see if it exists. If the uniqueness and existence checks pass, then the newly defined type is assigned a symbol and added to the symbol table. A challenging implementation was for declaring new struct types. For determining whether an identifier in a struct declaration has already been declared, we initialized a separate symbol table only for identifiers in a struct. As the struct's field are parsed and its symbols are added to this separate table, any repeated fields are discovered and an appropriate error message is displayed to the user. Additionally, we determine if each field's type's symbol exists from the current scope of code. So the function's signature looks like this: `void symSTRUCTSPEC(STRUCTSPEC *ss, SymbolTable * scope, SymbolTable *structScope)`. The struct spec is a linked list of a struct's fields and their associated types. The `*scope` is the symbol table used to determine if a field's type's symbol exists, and the `*structScope` is used a separate symbol table to determine uniqueness for each field's identifier in the struct declaration. We designed it in this way, because it allowed us to use less code (can simply use the same code for putting and getting symbols as the other functions) and can retrieve symbols from the separate struct symbol table in O(1) time.

2.9 Printing the Symbol Table

As for printing the symbol table, anytime a new symbol is declared it also gets printed if the `print_sym_table` variable is set to `1`, and their respective scoping is defined by the symbol table they were declared in. Their types aren't typechecked yet, so some symbols have a type `<infer>` that needs to be further typechecked, most notably for short-hand declared variables.

3 Typecheck

The entry point for type checking is again the `PROG *root` AST root node for the input program which has been annotated with symbols from the symbol table portion of the compiler. The type check process of our compiler traverses the annotated AST in a very similar manner to the symbol table. AST nodes of the same kind are traversed using their linked-list property, as shown in the following code snippet example:

```
void typeTOPLEVELDECL(TOPLEVELDECL *tld) {
    if (tld == NULL) return;
    typeTOPLEVELDECL(tld->next);
    ...
}
```

At each AST node, a type check is performed as per the requirements given in the milestone 2 language specifications handout. The main design decisions to discuss for type checking include: type checking expressions, statements, variable declarations, and expression case clauses. Function calls for type checking simply call on function calls that type check function or variable declarations. Type checking function declarations include type checking its return type, and body of statements. We don't need to type check type declarations since those are well-typed as per the symbol table. If a type declaration isn't well-typed it would have been caught by the symbol table of the program (whether a newly defined type's type's symbol exists in the symbol table).

3.1 Statements

1. Empty, Break, Continue statements: For these statements, we didn't need to do any type checking since there isn't any symbol to type check for;
2. Expression statements: We simply type check the expression;
3. Return statements: We type check that a return statement returns a symbol of the same type as that of the function declaration's type;
4. Increment/Decrement statements: Here we need to type check that the expression resolves to a numeric type;
5. Assign statements: These statements were challenging. For colon assignment statements, we need to first type check the right-hand side of the statement, then resolve the left-hand side to an identifier, and assign the right-hand side's type to the symbol's type. This was challenging, because we had to query for the symbol for the left-hand side which could result in segmentation fault errors if it didn't exist. Therefore, we verify that the symbol isn't null before performing any field access with it. As for assignment and operator assignments, we have to make sure the left-hand side is addressable, and that the left-hand side and right-hand side expressions resolve to the same types. We wrote some helper functions to verify these conditions. These functions simply either return true or false depending if a type's resolved base type is addressable (as specified in the language specifications) and whether two types are equal;
6. Print statements: We simply type check that the expression for a print statement resolves to a base type (int, float64, string, rune);
7. Variable declarations: We pass the statement's variable declaration node to the `typeVARSPEC(s->val.varDecl);` function call, more on this later;
8. If-statements: The statements and expressions in an if-statement are all type checked. Additionally, we verify whether the condition expression resolves to a boolean, and throw an error message otherwise;

9. For-loop statements: Similarly, we type check each statement and expression, and verify whether the condition expression resolves to a boolean, and throw an error message otherwise;

10. Switch Statements: We type check the switch statement's statements and expressions. Additionally, we check if the condition expression contains a symbol and assign the switch-statement's expression type to that of the symbol's type, otherwise we set it to a boolean type to avoid ambiguation;

Variable declarations are type checked if they're assigned an expression and specified a type. The right-hand side expression must be type checked by using the `symEXP()` function, and if a type is specified, then the type of the right-hand side and the type specified must match.

3.2 Expression

Buckle up. Similarly to creating the symbols for an expression, depending on the expression AST node kind, we either associate a type to a symbol, or verify if some expression is of an appropriate type. The language specifications details what an expression's type must resolve to if it's a certain expression kind. For brevity reasons, and to avoid redundancy, we urge the reader to read the language specifications handout specifications on expression types, since we followed this document for determining how to type check each kind of expression. However, one of the trickier implementations we'd like to discuss is that for field access expressions. First we need to verify that the expression's object type checks, but the challenging part is determining if getting the field's type. To do so, we iterate over all of the object's type's struct specification data structures. We first validate if the queried field matches with a field declared in a struct spec. Then we resolve the entire expression to the type from this field. This was tricky because accessing this data requires a long list of data structure referencing as noted here: `e->val.fieldAccess.object->type->val.structType`.

3.3 Expression Case Clauses

Type checking expression case clauses is simply type checking a combination of multiple expressions and statements.

4 APPENDIX: INVALID PROGRAM TYPE RULES

1. example/2-1-vardecl.go: A variable is declared and its type is inferred to be of type `rune`. The program is invalid because it attempts to assign a value of type `int` to the variable of type `rune`;
2. example/3.7-assignment-constantLhs.go: The program is invalid because the left-hand side expression is not addressable and is used in an assignment statement;
3. example/3.10-print.go: A variable is declared with a type if `[]int`. The program is invalid because it attempts to print this variable, but its type does not resolve to a base type of either `int`, `float64`, `rune` or `string`;
4. example/3.11-for1.go: The program is invalid because a for-loop statement's condition expression does not resolve to a `bool` type;
5. example/3.12-if1.go: The program is invalid because an if statement's condition expression does not resolve to a `bool` type;
6. example/3.13-switch1.go: The expression in the `case` of type `bool` does not match with the switch's expression type of type `int`;
7. example/3.13-switch2.go: The expression in the `case` of type `rune` does not match with the switch's expression type of type `int`;
8. example/4.1-literal1.go: A variable is declared with an undeclared type in the current scope (it's a base type not supported by GoLite);
9. example/4.3-unaryexp1.go: The program is invalid because it tries to use the `^` operator in a unary expression with a variable of type `bool`, which is illegal. The `^` operator is valid in unary expressions with variables of type `int`.
10. example/4.3-unaryexp2.go: The program is invalid because it tries to use the `+` operator in a unary expression with a variable of type `bool`, which is illegal. The `+` operator is valid in unary expressions with variables that are of a numeric type.
11. example/4.4-binaryexp1.go: The program is invalid because the `&&` operator is used in an expression with values of type `int` and `bool`. The `&&` operator can only be used with variables of type `bool`.

12. example/4.4-binaryexp2.go: The program is invalid because the `||` operator is used in an expression with values of type `string` and `bool`. The `||` operator can only be used with variables of type `bool`.

13. example/4.5-funccall.go: The program is invalid because a function is called with the wrong number of arguments.

14. example/4.6-index-exp1.go: The program is invalid because it attempts to perform an array access on a variable of type `int` that is neither of type `array` nor `slice`.

15. example/4.6-index-exp2.go: The program is invalid because it attempts to perform an array access on a variable of type `float64` that is neither of type `array` nor `slice`.

16. example/4.8.1-append1.go: The program is invalid because it attempts to append a value of type `float64` to a slice type of base type `int`.

17. example/4.8.1-append2.go: The program is invalid because it attempts to assign the result of an `append` function call to a variable of type `string` instead of one of type `slice`.

18. example/4.8.2-capexp1.go: The program is invalid because it attempts to pass a variable of type `struct` to the function `cap()` which expects its arguments to be of types `slice` or `array`.

19. example/4.8.2-lenexp1.go: The program is invalid because it attempts to pass a variable of type `struct` to the function `len()` which expects its arguments to be of types `slice` or `array`.

20. example/4.9-typecast-numeric.go: The program is invalid because it attempts to pass a variable of type `string` to a type cast function `int()` which expects its arguments to be a numeric type.