GoLite Compiler - Milestone 2: Symbol Table and Type-checking

Changes Since Milestone 1

Grammar Changes

After running the test suite used to grade our Milestone 1 deliverable, we noticed that the biggest issue in our design was how we dealt with types in the grammar. In our submission, we allowed for struct and array definitions, however when it came to variable declarations we expected types to always be identifiers. This caused a number of problems. The most evident one was the inability to define variables with *untagged* types -- that is types that have not been named through a formal type declaration but which are defined at the same time as the variable itself. A less evident problem was that the grammar rejected parenthesized types in declaration statements, which are allowed in the GoLite specification. To resolve the issue of *untagged* array definitions and parenthesized types we introduced a new `type` nonterminal into the grammar. Then, to allow for *untagged* struct definitions, we combined the `type` and `struct` nonterminals and inserted them as a rule into the definition of the variable declaration nonterminal.

After fixing this issue, we added other smaller improvements to the grammar such as:

- Adding an `identifier_list` nonterminal to replace `expression_list` in some locations, which provides some checking at the parser level as opposed to the weeder level.
- Creating a more detailed type node in the AST directly from the grammar, including information like index number for arrays, and whether the type is a struct.

Tree Restructuring

In light of the feedback received from our Milestone 1 report, we decided that restructuring the tree was a must when moving forward. The main change that we did was to remove the ‘node’ type that basically englobed all the other nodes in the tree (declarations, expression, statement and type). By changing it, we made the top level of our tree composed of declarations only and then all the other nodes branch out from there. It also made more sense to use this type of structure since in Milestone 1 for example, we were storing the left hand side and right hand side of a binary expression as nodes when we know for sure that they will be expressions. Even though this structure made it easier for us to structure our code, it made us have to perform one extra step when traversing the tree which made our tree look more like a CST rather than an AST.

Weeding

Since Milestone 1, we added an extra weeding phase that deals with checking that functions terminate properly. Basically we followed all the cases specified in the Go specifications and that are applicable to GoLite regarding Terminating Statements. We only perform this weeding phase if the function's return type is not void, however, we do not check that the type of the 'return' expression is compatible with function type in this weeding phase as this is done in the type checking phase.

Symbol Table

Symbols:

In our implementation, the symbols are represented by the following:

- The symbol name.
- The symbol type: Defines the type of the symbol, we will discuss it in the following paragraph.
- A linked list list of symbol types used to store information about the arguments types for functions.
- The symbol kind: can either be a function, variable, type or constant.
- A pointer to the next symbol, this is used to store symbols in the hashmap of the symbol table.

In order to implement our symbol types, we decided not to use the 'type' node that was used in the AST since the one linked to the symbol will contain more information and sometimes has a different structure. For example, the AST type stores structs as declarations but in the symbol table we will store them as a linked list of types. The 'symboltype' struct contains the following elements:

- A name: refers to the name of type. For variables, this refers to the name of it's parent type, but for types, this refers to its own name.
- A kind: this can be one of the base types (int, float64, etc), inferred, void, struct or defined.
- A pointer to the symbol of the parent type, this points to the parent when the type is a defined type.
- A pointer to the linked list of symbols (called 'structelements') that represents the elements of structs. These symbols are not pushed to the symbol table.
- Array information: Basically defines whether a type is an array and stores its dimensions. The dimensions are stored as a linked list of integers where an index value of '-1' represents slices.

Whenever a new variable or type is created, if it is based on a defined type, then we link it through the parent pointer and we set the kind of the type to 'defined'. Otherwise, if the type

is a struct, then we process the struct elements and set the kind of the type to ‘struct’. Arrays are handled after this step and they are just copies of the array information that come from the types stored in the symbol table.

Scoping Rules

New scopes can be pushed and popped from the scope stack using the functions ‘_scope_sym_table()’ and ‘_unscope_sym_table()’. In our implementation we follow the following scoping rules:

- In the base scope (the initial scope), we push the base types (int, string....) and the boolean constants (true and false). Then we create a new scope where all our top level declarations are stored. This is done this way since any of the base types and constants can be shadowed and this cannot happen unless the base types and constants reside in a different scope than the rest of the program.
- When a function declaration is encountered, the function is added to the current scope and a new scope is created. In this new scope, the function arguments are first added to the scope before processing the function’s statement. When no more statements are left in the function, this new scope is popped off the stack.
- When an if statement is encountered, a new scope is created where the variables declared inside the ‘init’ part of the if statement are added (we will call it the ‘init scope’), then another scope is created where all the statements residing inside the ‘then block’ are processed. Once that step is done, we pop one scope off the stack. Then we check if there is an ‘else block’ and in that case, we create a new scope and process the statements within the ‘else block’ and then pop the new scope off the stack. Once these steps are done, we pop the ‘init scope’ off the stack. We are handling the scopes this way so that both the statements inside the ‘then block’ and the ‘else block’ have access to the variables declared within the init statement (these variables reside in the ‘init scope’) while each having a separate scope of their own (i.e. the ‘then block’ does not know about the scope of the ‘else block’ and vice versa).
- When a block statement is encountered, a new scope is created and the statements within the block are processed. Once those statements are processed, the new scope is popped.
- When a switch statement is encountered, an ‘init scope’ is created where the variables that are declared inside the init statement of the switch are stored. From that point, with each case statement, a new scope is created and the statements residing within the case are processed and the scope is popped after no more statements are left inside the case. Once no more cases are left, the ‘init scope’ is popped off the stack. This strategy follows a similar scoping rule to what was done with the if statement since we wanted each case statement to have its own separate scope while they can all access the ‘init scope’.

Type Checker

Our typechecker makes use of several helper functions, which will be described, below, in the next few subsections. These helper functions are used extensively throughout the implementation of the type checker.

Helper Functions: Miscellaneous

`_resolve_type( … )`

- Return the resolved type of a given symbol.
- E.g.

```
type A int // _resolve_type( A ) -> int
type B A   // _resolve_type( B ) -> int
type C []A // _resolve_type( C ) -> []A
```

`_is_lvalue( … )`

- Return true if the expression is an "lvalue" (i.e. addressable), and false otherwise.
- Valid lvalues in GoLite include variables (non-constant), slice indexing expressions, array indexing expressions (of an addressable array), and field selection expressions (of an addressable struct).

Helper Functions: Type Classification

`_is_comparable_type( … )`

- Returns true if the type is "comparable", and false otherwise (i.e. can be used in expressions with binary operators "==" and "!=").
- Comparable types include:
  - `int, rune, float64, string, bool`
  - Array types, as long as the elements' type is comparable
  - Struct types, as long as each struct member's type is comparable

`_is_ordered_type( … )`

- Returns true if the type is "ordered", and false otherwise (i.e. can be used in expressions with binary operators "<", "<=", ">", and ">=").
- Ordered types include:
  - `int, rune, float64, string`

`_is_numeric_type( … )`

- Returns true if the type is “numeric”, and false otherwise.
- Numeric types include:
    - `int, rune, float64`

`_is_integer_type( … )`

- Returns true if the type is “numeric”, and false otherwise.
- Numeric types include:
    - `int, rune`

Helper Functions: Type Equality

`_types_are_identical( … )`

- Returns true if two types are identical, and false otherwise.

Type Checking: Expressions

The implementation of expression type-checking is divided into several subroutines -- one subroutine for each type of expression. These subroutines are called from a switch statement in the main `_typecheck_expr( … )` function. Each subroutine determines whether or not the sub-expressions of their main expression are type-correct (as well as the resolved types of those sub-expressions) via mutually recursive calls to `_typecheck_expr( … )` itself on the sub-expressions.

All code external to expression type-checking (e.g. the implementations for declaration and statement type-checking) need only call `_typecheck_expr( … )` to (a) determine if an expression is type-correct, and (b) to get the resolved type of that expression.

`_typecheck_expr( … )`

- Return the type of the expression, or exit with an error code of 1 if the expression is not type-correct.

Type Checking: Declarations

The implementation for declaration type-checking followed a similar approach to expression type-checking. For each type of declaration, a separate subroutine was written to verify its correctness. All these functions were then put together in the main `_typecheck_decl( … )` function, through the use of a switch construct which looks at the type of each particular declaration (package, variable, type or function) to decide with subroutine to call.

`_typecheck_decl( … )`

- Check if the declaration is well typed, if this is not the case, exit with an error code of 1.

Type Checking: Statements

In the same fashion as expression and declaration type-checking, separate subroutines were created to check that all the different statements in the GoLite specification are well typed. These functions were put together under `_typecheck_stmt( … )` using the same switch construct approach mentioned before. Moreover, statement type-checking makes extensive use of `_typecheck_expr( … )` and `_typecheck_decl( … )` to verify the correctness of the parts that make each statement construct.

`_typecheck_stmt( … )`

- Check if the statement is well typed, if this is not the case, exit with an error code of 1.

Type Checking: Top Level

In order to enable type-checking in the compiler project, a single main `typecheck( … )` typecheck function was exposed from the type-checking module. This function takes the root of the parsed AST and forwards it to `_typecheck_decl( … )` since declarations are the only top level constructs allowed in the GoLite specification.

`typecheck( … )`

- Check if the parsed AST is well typed, if this is not the case, exit with an error code of 1.
- The function expects the root of the AST which should be a declaration. This is enforced by the grammar.

Invalid Programs Description

Invalid test programs by Zakaria Essadaoui:

- zak-symbol-func-00.go: Declares the same function twice.
- zak-symbol-func-02.go: Declares the function “main” with a return type. Rule: main and init do not have any arguments or return type.
- zak-symbol-func-04.go: Declares the function “init” with a return type. Rule: main and init do not have any arguments or return type.

- zak-symbol-func-05.go: Tries to call the function “init” from within another function. Rule: “init” does not add any binding and thus cannot be called.
- zak-symbol-type-00.go: Tries to define a type from a non existent type.
- zak-symbol-type-01.go: Defines a type recursively. Ex: ‘type int int’.
- zak-symbol-type-02.go: Defines a type recursively from within a struct.
- zak-symbol-var-00.go: Redeclares a variable.
- zak-symbol-var-01.go: Uses short declaration with two already declared variables. Rule: At least one of the variables in a short declaration has to be undeclared.
- zak-symbol-var-04.go: Uses an undeclared variable.

Invalid test programs by Joshua Inscoe:

- joshua-inscoe-typecheck-binaryexpr-02.go: Attempt to compare 2 structs with same member names / types, but different internal ordering.
- joshua-inscoe-typecheck-builtinappendexpr-02.go: Attempt to append a constant of rune type to a variable of string type.
- joshua-inscoe-typecheck-builtincapexpr-03.go: Attempt to get the capacity of a string constant.
- joshua-inscoe-typecheck-fieldselectionexpr-02.go: Attempt to select a non-existent member of a variable of struct type.
- joshua-inscoe-typecheck-funccallexpr-00.go: Attempt to assign the result of a function that returns nothing.
- joshua-inscoe-typecheck-funccallexpr-05.go: Attempt to call a function with too few arguments.
- joshua-inscoe-typecheck-indexingexpr-01.go: Attempt to index into a variable of slice type using an index of rune type.
- joshua-inscoe-typecheck-indexingexpr-03.go: Attempt to index into a string constant.
- joshua-inscoe-typecheck-typecastexpr-04.go: Attempt to typecast a variable of struct type to another struct type.
- joshua-inscoe-typecheck-unaryexpr-03.go: Attempt to perform unary bitwise complement operation of a variable of float64 type.

Invalid test programs by Angel Ortiz:

- angel-typecheck-switch-00.go: Switch statement with non-comparable condition expression.
- angel-typecheck-switch-03.go: Switch statement with badly-typed init expression.
- angel-typecheck-switch-05.go: Switch statement with incompatible case expression (case expression does not match type of switch condition expression).
- angel-typecheck-return-01.go: Void return on int function.
- angel-typecheck-return-03.go: String return value on int function.
- angel-typecheck-incdec-00.go: Increment non-lvalue.

- angel-typecheck-if-00.go: If statement with non-boolean condition expression.
- angel-typecheck-if-03.go: If statement with badly-typed statement inside else block.
- angel-typecheck-if-04.go: If statement with non-boolean condition expression in else-if construct.
- angel-typecheck-for-03.go: For loop statement with badly-typed post statement.

Team Structure

Roles:

- Zakaria Essadaoui implemented the symbol table, restructured the AST and added an extra weeding phase.
- Joshua Inscoe implemented the expression type-checking.
- Angel Ortiz implemented the declaration and statement type-checking, fixed the grammar from Milestone 1 and added type nodes to the AST.

Consulted Resources

We worked alone.