GoLite Milestone II

Pavel Kondratyev
pashakondratyev
260653115

Kaylee Kutschera
kkutschera
260608470

Kabilan Sriranjan
kabsri
260671847

1 Design

1.1 Symbol Table

The symbol table was implemented using a similar skeleton to what was covered in class. A symbol is a struct which stores the identifier, the declaration kind (function, variable, short, or type), as well as the relevant information depending on the kind of declaration. For a variable declaration we need store the type the variable resolves to. For a function declaration we store the return type and parameter list. For a type declaration we store the type and the type it resolves to. The symbol table itself is a hashmap using the following hash function, given in class.

```
while (*str) hash = (hash << 1) + *str++;
```

Each declaration and statement is read in order to open and close scopes. New scopes are only opened on blocks as well as switch cases. Special note should be brought to if/for statements as they may contain a declaration so a scope is opened around both the if and else statement.

1.1.1 Type Mapping

The way the parser was constructed, every time a type was read in a new type node was created in the abstract syntax tree. This makes sense as during the parser stage we do not have the symbol table. But when we read

```
type int string
var test string = 4
```

we are not able to lookup if string is a reference type, a base type, or a compound type (such as arrays, slices, or structs). We originally tried to mark everything as a reference type and then resolve it but then we couldn't distinguish real reference types from other types without having a convoluted type checker. The streamlined solution was to mark all types in the parser using an enum so that they can be linked in the abstract syntax tree using the symbol table. This way

types are pointing to the same type nodes from the symbol table and can be compared directly by reference, resulting in simpler code and easier debugging. This also saves space as each type has only one node in the AST that can be referenced many times. The implementation for this was to recursively descend through the type (such as in the case of compound types) and resolve each of these fields to the proper type and return the properly mapped one in the end. Note that even though we are using the word resolve we are not resolving the type to the base type or resolving a type to its reference.
When we put types into the symbol table, we also keep track of the type it resolves to in a pointer. This will make it easier later to check if a type resolves to a base type without having to do a traversal of the symbol table each time.

1.1.2 Scopes

In the symbol table scopes are added as per the rules given to us. Namely in block statements such as those found in for loops, if/else statements, switch cases, as well as standalone block statements. Additionally scopes must be opened up for initialization statements that are found in such statements which are separate, so that if a variable is declared it can also be shadowed in the body of the statements. The scopes are attached to the node of the body in the abstract syntax tree, a new field was added in the relevant nodes to hold scope.

1.1.3 Short Variable Declarations

For short variable declarations, for the syntax to be correct, the left hand side must contain at least one new variable in the scope. In order to implement this, each declaration in the left hand side is checked to be in the current scope in the symbol table. If the current scope (not including any of the parent scopes) contains the symbol an error is returned. Otherwise all the new symbols are added to the scope.

1.1.4 Unmapped Identifiers

In golang and golite, the identifier "_" does not get mapped into the symbol table. The init function also does not get mapped and there can be multiple init functions in the top level declarations. For type, variable, and short declarations this just means that if the identifier is "_", that it should not be put into the symbol table. However for functions, the inner scopes must still all be put into a symbol table and type checked later on so a scope is created for the body of the function.

1.2 Type Checker

The typechecking rules were all directly implemented from the spec. There were a few challenges along the way and the solutions are outlined below.

1.2.1 Inferred Types

During the symbol table phase, certain declarations do not have types which are known immediately such as the type of test below

```
var test = 4
```

We would need to check this declaration during the typechecker phase and fill in the blank in the symbol table, so that we know for later usages of test.

1.2.2 Type Comparisons

For assignment statements and comparison statements it was important to know if the two types are the same (without resolutions). Because of the way that we mapped the types, comparing two non compound types was easy as they will point to the same space in memory. When we are dealing with arrays and slices we check the inner types as well. For structs we check that all the fields appear in the same order, have the same names, and have the same types (in other words be identical). These are all requirements based on the gospec.

1.2.3 Assignments

One step we had to take when typechecking assignment statements on top of checking that the types on each side matched was to also make sure the the left hand side of every assignment was an Lvalue. This applied to variable declarations as well as op-assignments. Since these can be nested within arbitrary amounts of parentheses and struct fields we had to implement a recursive method to verify whether an identifier was an Lvalue or not.

1.2.4 Type Resolution

For type resolution, as mentioned in the symbol table section, for each type definition we keep track of the type it resolves to. This makes checking if two types resolve to the same type very easy, as we can just retrieve the symbols and directly check if the types are equal using the method above. In all situations we only needed to check if two types resolved to the same underlying type so there was no need to implement traversals through the symbol table to compare types.

1.2.5 Return Statements

When typechecking return statements there are a few cases to consider.

1. If the function has a void return type, are there return statements which try to return a value
2. If the function has a non void return type is it guaranteed to return, i.e do all branches of the function return
3. If a function has a non void return type do all the return statements return the correct type

We were first unsure of whether we should write a separate traversal to check these conditions(as 1 and 2 seem like something we could have handled in the weeding stage) or attempt to add it onto our function which typechecks a statement. The solution we went for was to return whether a statement is guaranteed to return in the function for typechecking statements. In the case of infinite for loops we check if there is a break. For statement lists, such as blocks, we only need to check that one of the statements is guaranteed to return. This way we can recursively build up if all possible paths return. We have some special cases which we address.

- If Statements - For if statements, the if statement and all the possible else/if and else statement must return so we make sure to check that.
- Switch Cases - For switch cases, in order to consider the block as "returning" each case must return, since we do not have fallthrough statements and break statements are implicit.
- For Loops - For infinite for loops which do not break, we can consider as a valid typechecking statement regardless of what the return type of the function is. This makes sense as if a function can never return then it can never return an incorrect type. Hence we say a for loop returns either if it has no condition and contains no break statements or if all possible branches inside the for loop return. In all other cases it is not guaranteed to return and does not typecheck.

1.3 Invalid Programs

Below are the 30 submitted invalid programs and a brief description of how they break a type rule. The type rule each program breaks is outlined in the appendix.

3.4-return-wrongType.go: Breaks a typing rule since the function has return type *int* whereas the returned expression is *bool*.

3.4-return-extra.go: Breaks a typing rule since the function has no return type whereas there is a returned expression of type *int*.

3.5-shortDecl-declared1.go: Breaks a typing rule since all variables in the short declaration have already been declared.

3.5-shortDecl-typeChange.go: Breaks a typing rule since the type of the first variable is orginally declared as a *string*, but they new expression set to x is of type *int*.

3.7-assignment-struct.go: Breaks a typing rule since the variable is of type point and the expression is of type int.

3.7-assignment-types1.go: Breaks a typing rule since a is of type string whereas the expression assigned to it is of type int.

3.10-print-baseType1.go: Breaks a typing rule since the expression in the print statement is a slice and not a base type.

3.10-println-baseType1.go: Breaks a typing rule since the expression in the println statement is a slice and not a base type.

3.11-forLoop-exprBool1.go: Breaks a typing rule since the conditional expression is a string and not a boolean.

3.12-ifStmt-exprBool1.go: Breaks a typing rule since the conditional expression is a rune and not a boolean.

3.14-dec-invalidType1.go: Breaks a typing rule since the expression is a string and $string \notin \{int, float64, rune\}$.

3.14-incr-invalidType2.go: Breaks a typing rule since the expression is an array and $[5]int \notin \{int, float64, rune\}$.

4.3-unaryExp-!1.go: Breaks a typing rule since the expression is of type int $int \neq bool$.

4.3-unaryExp-ˆ1.go: Breaks a typing rule since the expression is a float and $float64 \notin \{int, rune\}$.

4.3-unaryExp-+1.go: Breaks a typing rule since the expression is a boolean and $bool \notin \{int, float64, rune\}$.

4.3-unaryExp–1.go: Breaks a typing rule since the expression is a boolean and $bool \notin \{int, float64, rune\}$.

4.4-binaryExp-||4.go: Breaks a typing rule since the first expression is of type int, the second expression is of type boolean, but $int \neq bool$.

4.4-binaryExp–6.go: Breaks a typing rule since the first expression is of type int, the second expression is of type float, but $int \neq float64$.

4.4-binaryExp-<<5.go: Breaks a typing rule since the first expression is of type float, the second expression is of type int, but $float64 \neq int$.

4.5-funcCall-funcType.go: Breaks a typing rule since the parameter is an array, but the function is declared to take no parameters.

4.5-funcCall-params1.go: Breaks a typing rule since the function foo has type $int \times float64 \rightarrow int$, but the first parameter is of type $float64 \neq int$.

4.5-funcCall-params2.go: Breaks a typing rule since the function foo is of type $int \times int \rightarrow int$, but the second parameter is never specified in the function call.

4.6-index-index1.go: Breaks a typing rule since the index is of type $float64 \neq int$.

4.6-index-exp1.go: Breaks a typing rule since the variable used in the index expression is an integer and not a slice or array.

4.7-fieldSelect-exp1.go: Breaks a typing rule since the variable used in the field select expression is an integer and not a struct.

4.7-fieldSelect-id1.go: Breaks a typing rule since val is not a field in the struct.

4.8.1-append-elem1.go: Breaks a typing rule since we are trying to append a float to an integer slice.

4.8.1-append-slice1.go: Breaks a typing rule since we are trying to append to an array and not a slice.

4.8.2-cap-exp1.go: Breaks a typing rule since the expression in the function call is a struct and not an array or slice.

4.8.3-len-exp2.go: Breaks a typing rule since the expression in the function call is an int and not a string, array, or slice.

2 Contributions

Kaylee wrote the vast majority of the test programs as well as detailed all the invalid programs in the report. Pavel created the symbol table and fixed errors from Milestone 1. Pavel and Kabilan worked equally on the typechecker as well as created tests as necessary.

3 Appendix

Below are the type rules for all the invalid programs specified above.

3.0.1 3.4-return-wrongType.go

$$\frac{V,T,F \vdash e : \tau \qquad F = \tau \qquad V,T,F \vdash rest}{V,T,F \vdash return\ e; rest}$$

3.0.2 3.4-return-extra.go

$$\frac{V,T,F \vdash e : \tau \qquad F = \tau \qquad V,T,F \vdash rest}{V,T,F \vdash return\ e; rest}$$

3.0.3 3.5-shortDecl-declared1.go

$$\frac{\begin{array}{c} V,T,F \vdash e_1 : \tau_1 ... V,T,F \vdash e_k : \tau_k \\ \exists i \in \{1..k\} : x_i \notin V \\ \forall i \in \{1..k\} : x_i \in V \implies V(x_i) = \tau_i \\ V \cup \{x_1 : \tau_1, ..., x_k : \tau_k\}, T, F \vdash rest \end{array}}{V,T,F \vdash x_1, ..., x_k := e_1, ..., e_k; rest}$$

3.0.4 3.5-shortDecl-typeChange.go

$$\frac{\begin{array}{c} V,T,F \vdash e_1 : \tau_1 ... V,T,F \vdash e_k : \tau_k \\ \exists i \in \{1..k\} : x_i \notin V \\ \forall i \in \{1..k\} : x_i \in V \implies V(x_i) = \tau_i \\ V \cup \{x_1 : \tau_1, ..., x_k : \tau_k\}, T, F \vdash rest \end{array}}{V,T,F \vdash x_1, ..., x_k := e_1, ..., e_k; rest}$$

3.0.5 3.7-assignment-struct.go

$$\frac{\begin{array}{c} V,T,F \vdash v_1 : \tau_1 ... V,T,F \vdash v_k : \tau_k \\ V,T,F \vdash e_1 : \tau_1 ... V,T,F \vdash e_k : \tau_k \\ V,T,F \vdash rest \end{array}}{V,T,F \vdash v_1, ..., v_k = e_1, ..., e_k; rest}$$

3.0.6 3.7-assignment-types1.go

$$\frac{\begin{array}{c} V,T,F \vdash v_1 : \tau_1 ... V,T,F \vdash v_k : \tau_k \\ V,T,F \vdash e_1 : \tau_1 ... V,T,F \vdash e_k : \tau_k \\ V,T,F \vdash rest \end{array}}{V,T,F \vdash v_1, ..., v_k = e_1, ..., e_k; rest}$$

3.0.7 3.10-print-baseType1.go

$$\frac{\begin{array}{c} \forall i \in \{1..k\} : V,T,F \vdash e_i : \tau_i \\ RT(T,\tau_i) \in \{int, float64, rune, string, bool\} \\ V,T,F \vdash rest \end{array}}{V,T,F \vdash print(e_1, .., e_k); rest}$$

3.0.8 3.10-println-baseType1.go

$$\frac{\begin{array}{c} \forall i \in \{1..k\} : V,T,F \vdash e_i : \tau_i \\ RT(T,\tau_i) \in \{int, float64, rune, string, bool\} \\ V,T,F \vdash rest \end{array}}{V,T,F \vdash println(e_1, .., e_k); rest}$$

3.0.9 3.11-forLoop-exprBool1.go

$$\frac{V,T,F \vdash e : \tau \qquad RT(T,\tau) = bool \qquad \forall i \in \{1..k\} : V,T,F \vdash stmt_i}{V,T,F \vdash for\ e\ \{stmt_1; ...; stmt_k\}; rest}$$

3.0.10 3.12-ifStmt-exprBool1.go

$$\frac{\begin{array}{c} V,T,F \vdash init \\ V,T,F \vdash expr : \tau \\ RT(T,\tau) = bool \\ V,T,F \vdash stmts \end{array}}{V,T,F \vdash if\ init; expr\ \{stmts\}}$$

3.0.11 3.14-dec-invalidType1.go

$$\frac{V,T,F \vdash e : \tau \qquad RT(T,\tau) \in \{int, float64, rune\} \qquad V,T,F \vdash rest}{V,T,F \vdash e - -; rest}$$

3.0.12 3.14-incr-invalidType2.go

$$\frac{V,T,F \vdash e : \tau \qquad RT(T,\tau) \in \{int, float64, rune\} \qquad V,T,F \vdash rest}{V,T,F \vdash e + +; rest}$$

3.0.13 4.3-unaryExp-!1.go

$$\frac{V,T,F \vdash e : \tau \qquad V,T,F \vdash RT(T,\tau) = bool}{V,T,F \vdash !e : \tau}$$

3.0.14 4.3-unaryExp-ˆ1.go

$$\frac{V,T,F \vdash e : \tau \qquad V,T,F \vdash RT(T,\tau) \in \{int, rune\}}{V,T,F \vdash \hat{e} : \tau}$$

3.0.15 4.3-unaryExp-+1.go

$$\frac{V,T,F \vdash e : \tau \qquad V,T,F \vdash RT(T,\tau) \in \{int, float64, rune\}}{V,T,F \vdash +e : \tau}$$

3.0.16 4.3-unaryExp–1.go

$$\frac{V,T,F \vdash e : \tau \qquad V,T,F \vdash RT(T,\tau) \in \{int, float64, rune\}}{V,T,F \vdash -e : \tau}$$

3.0.17 4.4-binaryExp-||4.go

$$\frac{V,T,F \vdash e_1 : \tau_1 \qquad V,T,F \vdash e_2 : \tau_2 \qquad \tau_1 = \tau_2 \qquad RT(T,\tau_1) = bool}{V,T,F \vdash e_1 || e_2 : \tau_1}$$

3.0.18 4.4-binaryExp–6.go

$$\frac{V,T,F \vdash e_1 : \tau_1 \qquad V,T,F \vdash e_2 : \tau_2 \qquad \tau_1 = \tau_2 \qquad RT(T,\tau_1)\ is\ numeric}{V,T,F \vdash e_1 - e_2 : \tau_1}$$

3.0.19 4.4-binaryExp-<<5.go

$$\frac{V,T,F \vdash e_1 : \tau_1 \qquad V,T,F \vdash e_2 : \tau_2 \qquad \tau_1 = \tau_2 \qquad RT(T,\tau_1)\ is\ integer}{V,T,F \vdash e_1 << e_2 : \tau_1}$$

3.0.20 4.5-funcCall-funcType.go

$$\frac{\begin{array}{c} V,T,F \vdash e_1 : \tau_1 ... V,T,F \vdash e_k : \tau_k \\ V,T,F \vdash e : \tau_1 \times ... \times \tau_k \to \tau_r \end{array}}{V,T,F \vdash e(e_1, ..., e_k) : \tau_r}$$

3.0.21 4.5-funcCall-params1.go

$$\frac{\begin{array}{c}V,T,F \vdash e_1 : \tau_1 ... V,T,F \vdash e_k : \tau_k \\ V,T,F \vdash e : \tau_1 \times ... \times \tau_k \to \tau_r\end{array}}{V,T,F \vdash e(e_1, ..., e_k) : \tau_r}$$

3.0.22 4.5-funcCall-params2.go

$$\frac{\begin{array}{c}V,T,F \vdash e_1 : \tau_1 ... V,T,F \vdash e_k : \tau_k \\ V,T,F \vdash e : \tau_1 \times ... \times \tau_k \to \tau_r\end{array}}{V,T,F \vdash e(e_1, ..., e_k) : \tau_r}$$

3.0.23 4.6-index-index1.go

$$\frac{\begin{array}{c}V,T,F \vdash e : \tau_1 \quad RT(T, \tau_1) \in \{[]\tau, [N]\tau\} \\ V,T,F \vdash i : \tau_2 \quad RT(T, \tau_2) = int\end{array}}{V,T,F \vdash e[i] : \tau}$$

3.0.24 4.6-index-exp1.go

$$\frac{\begin{array}{c}V,T,F \vdash e : \tau_1 \quad RT(T, \tau_1) \in \{[]\tau, [N]\tau\} \\ V,T,F \vdash i : \tau_2 \quad RT(T, \tau_2) = int\end{array}}{V,T,F \vdash e[i] : \tau}$$

3.0.25 4.7-fieldSelect-exp1.go

$$\frac{V,T,F \vdash e : S \qquad RT(T, S) = struct\{..., id : \tau, ...\}}{V,T,F \vdash e.id : \tau}$$

3.0.26 4.7-fieldSelect-id1.go

$$\frac{V, T, F \vdash e : S \qquad RT(T, S) = struct\{..., id : \tau, ...\}}{V, T, F \vdash e.id : \tau}$$

3.0.27 4.8.1-append-elem1.go

$$\frac{V, T, F \vdash e_1 : S \qquad RT(T, S) = []\tau \qquad V, T, F \vdash e_2 : \tau}{V, T, F \vdash append(e_1, e_2) : S}$$

3.0.28 4.8.1-append-slice1.go

$$\frac{V, T, F \vdash e_1 : S \qquad RT(T, S) = []\tau \qquad V, T, F \vdash e_2 : \tau}{V, T, F \vdash append(e_1, e_2) : S}$$

3.0.29 4.8.2-cap-exp1.go

$$\frac{V, T, F \vdash expr : S \qquad RT(T, S) \in \{[]\tau, [N]\tau\}}{V, T, F \vdash cap(expr) : int}$$

3.0.30 4.8.3-len-exp2.go

$$\frac{V, T, F \vdash expr : S \qquad RT(T, S) \in \{string, []\tau, [N]\tau\}}{V, T, F \vdash len(expr) : int}$$