COMP520 - Group 10
Milestone #2

David Herrera	Maurice Zhang	Adam Bognat
Id: 260422001	Id: 260583688	Id: 260550169

March 13, 2017

1 Group Structure

Our team has no hierarchy. We validated each other's design decisions and code implementations and had great collaboration among all team members. The tasks were broken down roughly as follows:

- Adam: produced suite of 60 valid, 50 invalid programs to test type checker, symbol table and weeding phase.
- Maurice: implementing the type checker, implementing dumpsymtab, minor contributions to the symbol table
- David: weeding return statements, symbol table, scoping and declaration errors, pretty printing types, tests, fixing ML 1 mistakes, adding line numbers to nodes.

2 SymbolType vs. Symbol & Type

We made the design decision to separate the type checker and the symbol table in a way that allows each team member to work separately on an implementation. This decision had three consequences. Firstly, two passes of the tree must be done, one for symbol, one for typechecking. Secondly, the symbol tables were attached by the symbol section to nodes in the AST that opened a new scope, symbols were attached to every node referencing another symbol. Lastly, that variables that might not typecheck would be in the symbol table up until the type checker removes them from it in case of an error.

3 Symbol

3.1 Predeclared Types

All the predeclared types are of type alias, as per specification. The type node alias contains a field called actual_type, which reference the actual_node types, this is set by the type phase.

3.2 Shadowing

In order to shadow variables such as predefined types/variables a new scope was defined to intermediate this process. This had the following consequences:

- There was a pre-program symbol table that contained the pre-declarations, the top-level, or program symbol table was a child of that pre-symbol table.
- To handle for, if, and switch statements there were two new symbol tables created, one for the simple/post statements and one for the actual block, in that way, it was possible to achieve the shadowing of variable/type declarations by simple statements and the shadowing of the block statements on the simple statements. This also meant that to print the symbol tables it was necessary to recurse to the next scope symbol table for those particular statements.

3.3 Block Nodes

Separating successfully the type and symbol parts meant that we had to make some modifications to the tree structure, instead of for, if, switch, pointing at statements, they now point to a new node *Block* that contains the statements and the symbol table. (Other nodes were also modified to include a type).

3.4 Struct Type Literals

To keep track of those struct fields, a temporary symbol table was made in order to make sure no field was repeated while checking the types using the upper symbol table.

4 Typechecking

4.1 Comparing Types

4.1.1 Base Types

We defined five constant TYPE nodes that represented the base types:

- `BASE_TYPE_INT`
- `BASE_TYPE_FLOAT`
- `BASE_TYPE_RUNE`
- `BASE_TYPE_BOOL`
- `BASE_TYPE_STRING`

These types were assigned to literals and to the predefined type identifiers (`int`, `float64`, `rune`, `bool`, `string`).

Consider the following case:

```
// valid
var foo int
foo = 3
```

In this case, the expression `3` is attached with the literal base type `BASE_TYPE_INT` and the variable `foo` is attached with the identifier type `int` who's actual type is `BASE_TYPE_INT`. In situations like these, where we compare an identifier type and a literal type, we must determine if the identifier type is an alias to the literal type. Thus, to make the comparison, we would get the actual type of the identifier (which can be another identifier type), check whether or not that actual type is a base type, and then compare between the base types. This sequence correctly invalidates the following situation:

```
// invalid
type int string
var foo int
foo = 3
```

In this situation, the variable `foo` has a type `int`, which is an identifier type who's actual type is `string` (also an identifier type). Thus, the actual type is not a literal base type and the typecheck for the assignment does not pass.

4.1.2 Arrays and Slices

When comparing array and slice types, we check that the inner types are equal. For an array, we also check that the array lengths are the same.

4.1.3 Structs

For structs, we had to check that both struct definitions had the same fields. This was a simple task since the fields had to be in the same order. Thus, we simply iterated through both struct fields and compared the names and types.

4.1.4 Type Aliases

When comparing type aliases, we simply compare the pointers of the `TYPE` nodes since those nodes are only created once per type declaration. This ensures that type aliases with the same name and same actual type are still considered different. For example, the following case is invalid:

```
type num int
var a num
{
  type num int
  a = num(23)
}
```

Although both type aliases have the same name and the same actual type, they are not the same and thus the assignment is invalid.

4.2 Resolving to a Type

While implementing the typechecker, a recurring task was to see if one type resolved to another. For example, when indexing an array, the index expression must resolve to an `int` or `rune`. To help with this, we created a function called `get_root_type(TYPE *type)` which would take a type as input and output it's deepest connected type that wasn't an alias. Consider the following type declarations:

```
type num int
type number num
type number1 number
```

With these type aliases, if we were to pass a `TYPE` node representing the `number1` type alias to `get_root_type()`, it would output the literal base type node representing an `int` (`BASE_TYPE_INT`).

4.3 Type checks

Here is a brief explanation of the implementation for every type check that we made.

4.3.1 Variable Declarations

If there were no expressions on the right hand side of a variable declaration, we simply assigned the type of the variable declaration to all the identifiers on the left hand side (if the identifier wasn't blank).

Otherwise, for each expression, we check that it was well-typed and, that the type corresponded to the type of the variable declaration (if it had one). We then assigned that type to the corresponding identifier on the left hand side (if the identifier wasn't blank).

The corresponding invalid program is `decl7.go` as the expression's type does not correspond to the variable declaration's type.

4.3.2 Function Declarations

For function declarations, we simply checked that every statement in the block was well-typed. To do this, we had a function called `typeBLOCK` that took as input a BLOCK node and type checked every statement. Also, we assigned the function return type to a global variable in order to typecheck return statements (explained in the next subsection).

4.3.3 Return Statements

Since we had the function `typeBLOCK` that was used to typecheck all blocks (and not just function declaration blocks), there was no easy way link a return statement inside a block with its parent function declaration. Therefore, to check that a return statement was well-typed, we kept a global variable that denoted the return type of the last function declared (`NULL` if the function had no return type). There was no issue when overwriting this value since function declarations could only be at the top level. If function declarations could be nested, we could have kept a stack.

If the return statement had a child expression, we would check that it was well-typed and check that its type corresponded to the function return type. If there was no child expression, we would check that the function return type was `NULL`.

Statements after a return statement were still type checked.

The corresponding invalid programs are `func4.go` as the return statement is void whereas the function does have a return type, and `return2.go` as the statement after the return statement does not typecheck.

4.3.4 Short Declaration Statements

Short declaration typechecking was very similar to typechecking variable declarations. The difference was that if an identifier on the left hand side already had a type, then we would check that the corresponding expression on the right hand side had the same type (similarly to variable declarations with a type defined).

The corresponding invalid program is `shortdecl1.go` as there are no new variables on the left side of the short declaration (this is checked in the symbolizing phase).

4.3.5 Assignment Statements

For assignment statements, we checked that both the expressions on the left hand side (if it wasn't blank) and on the right hand side were well-typed. We then checked that the corresponding expressions had corresponding types.

The corresponding invalid program is `decl4.go` as the corresponding expressions do not have corresponding types.

4.3.6 Op Assignment Statements

For op assignments, we created a binary expression node using the expressions on the left hand side and on the right hand side as the two expressions of a binary expression and converting the assignment operator to a binary operator (+= $\rightarrow$ +, -= $\rightarrow$ -, etc.). We then passed that expression node to the function `typeBINARYEXP` and typechecked it as a binary expression (discussed further below).

4.4 Increment/decrement Statements

For these statements, we checked that the child expression was well-typed and checked that the type resolved to a numeric type (this requirement was not mentioned in the specs so we inferred it from the reference compiler).

4.4.1 Expression Statements

For expression statements, we checked that the function call expression was well-typed.

4.4.2 Print Statements

For print statements, we simply iterated through every child expression and checked that it was well-typed and that the type resolved to a base type.

4.4.3 For and If Statements

For both `for` and `if` statements, we checked that the expression was well-typed and resolved to a `bool` type. We also checked that the init statement and post statement (for `for` statements), were well-typed. The typechecking was done in the correct order (init statement, expression, post statement), to ensure that the types were properly attached when encountered. We then checked that the blocks were well-typed (both true and false blocks for `if` statements). For an `if` statement, we then checked that the attached `else if` statement was well-typed if it was present.

The corresponding invalid programs are `for1.go` and `if1.go` as the expressions' types do not resolve to `bool`.

4.4.4 Switch Statements

For switch statements, we first typechecked the simple statement, and then the expression if it was present. We then called the function `typeCASECLAUSE` to typecheck the child case clauses. To that function, we passed the expected type for the case clause expressions. If the switch statement had a child expression, then the expected type would be set to the type of that expression. Otherwise, the expected type would be set to `bool`.

Typechecking the case clauses simply involved typechecking the child expressions, checking that their types corresponded to the expected type, and typechecking the associated block.

The corresponding invalid program is `switch1.go` as the case clause expression's type is not `bool` although the switch statement does not have an expression.

4.4.5 Expressions

To typecheck expressions, we created a function for every different kind of expression, for example `typeBINARYEXP` for binary expressions, `typeFUNCEXP` for function call expressions, etc. These would return the type of the expression if it was well-typed, and throw an error otherwise. We also created a general function `typeEXP` which was used to send the expression to the appropriate function and attach the returned type to the expression node. It would also see if any of the returned types were `NULL` and throw an error if this was the case. This would represent the situation where a function call to a `void` function was used as an expression. Therefore, when typechecking expression statements, where the function call could be to a `void` function, we used `typeFUNCEXP` directly thus avoiding the check to see if an expression was `void`.

4.4.6 Identifier Expression

When typechecking identifiers used as expressions, we get the identifier's symbol to check that the identifier is a variable (and not a type). We also get the identifier's type from the symbol and assign it to the identifier node.

4.4.7 Function Call Expression

For function call expressions, we first check that the expression (not the arguments), is an identifier, or a parenthesized identifier. We then query the symbol table for the symbol representing the identifier and check that the identifier either refers to a type, or a declared function name (otherwise, throw an error). If it refers to a type, we then treat the expression as a typecast (discussed in the next section) by passing it to the function `typeTYPECAST`.

We then verify that the number of arguments is equal to the number of parameters defined in the function declaration. After, we iterate through the expression (arguments) and check that they are well-typed and that the type corresponds to the type of the corresponding parameter in the function declaration.

The corresponding invalid program is `func2.go` as the argument's type does not correspond to the parameter's type in the function definition.

4.4.8 Typecast Expression

For typecasts, we first check that there is only 1 argument. We then check that the type being cast to is castable. After, we check that the argument expression is well-typed and that its type is also castable.

The corresponding invalid program is `cast2.go` as it tries to cast to a string.

4.4.9 Struct Field Selection Expression

For struct field selection expressions, we first make sure that the expression is well-typed and that the type resolves to `struct`. We then check that the identifier is a field in the struct definition. This is done by keeping a hash table in the struct node that has all the fields stored and querying that table.

4.4.10 Array/Slice Indexing Expression

For indexing expressions, we check that both child expressions are well-typed. We then check that the array/slice expression's type resolves to `array` or `slice`, and that the index expression resolves to an integer type (`int` or `rune`).

The corresponding invalid program is `array1.go` as the index expression's type does not correspond to an integer type.

4.4.11 Append Expression

For append expressions, we check that both child expressions are well-typed. We then check that the first expression's type resolves to `slice` and that the second expression's type corresponds to the slice's inner type.

The corresponding invalid program is `append1.go` as the second expression's type does not correspond to the slice's inner type.

4.4.12 Binary/Unary Expression

For unary and binary expressions, we check that all the expressions are well-typed. For binary expressions, we then check that both expressions have the same type. For both unary and binary expressions, we then check that their types are valid with the operator. We must then return the proper type depending on the operator.

The corresponding invalid programs for binary expressions are `binary1.go`, `binary12.go`, and `binary5.go` as the two expressions' types don't match.

The corresponding invalid programs for unary expressions are `unary1.go`, `unary2.go`, `unary5.go`, `unary7.go` as the operator is not valid with the expression.

5 Return Statements

We implemented the return statements like they were originally stated in the Golite specifications, not like GO. Therefore, they do not take into consideration break statements, and have other shortcomings such as not being valid Go programs in some cases.

6 Fixes ML1

There were some fixes to the Milestone one, the highlights are:

- Handling properly escape characters in strings for the scanner
- Adding line numbers to all pertinent nodes and weeding phase
- Fixing two grammar errors, one pertaining simple statements and the other one pertaining empty statements

References