Milestone 2 Report

Group comp520-2017-01

1. Team Work

Yi Qiao

- Weeding phase for returns
- Symbol table
- Typechecker
- Tests

Thomas

- Typechecker
- Fixed errors in Parser
- Tests

Charlie

- Typechecker
- Typed pretty printing
- CLI code
- Test suite code
- Test programs

2. Design Notes

A. Weeder

-- As part of the weeding required to verify that functions are properly returned, we have implemented the "Terminating Statements" section of the GoLang spec (https://golang.org/ref/spec#Terminating_statements) . We initially believed that this was the expected implementation for returns, thus, even though it was not required, we have included this implementation. This implementation is recursive in nature, and enumerates all the different cases outlined in the spec (https://golang.org/ref/spec#Terminating_statements) . For example, in the case of `switch` statements, a `switch` statement is considered to be terminating if it has a `default` case, no breaks, and if all of its `case` paths are terminating. To check that all of the `case` paths are terminating, we would run the same function that ran on the case statement, on the block of statements in each `case`, returning the result to the parent.

B. SymbolTable

-- The symbol table itself was implemented as a stack of frames (denote this stack as the frame stack), where each frame was a hash map that represented the mappings available in the current scope. Bundled along with the stack of frames, were two other stacks. These two other stacks were used to implement the features associated with the `--dumpsymtab` and `--pptype` flags.

The first of these two other stacks, denoted as the history stack, was used to store the state of the symbol table at each scope. This was used to implement the `--pptype` flag. Due to Haskell's immutability, it was decided that this was the simplest and easiest way, for us, to pass along type information to the pretty printer. This approach has its pros and cons. An advantage is that it required no need to rebuild a typed version of the AST. This, in turn, redueces the amount of code that needed to be written and tested. Futhermore, this approach was also simple to implement. All that is required, is to store a copy of the entire symbol table on every scope exit. However, the approach we took also introduces unnecessary computation; to pretty print a file with type annotations, we type the file twice.

The second stack was used to store the history of all popped symbol table frames. Every time a frame is popped, we push it to this second stack. At the termination of the typechecking, either on success or failure, print this stack in reverse order. Again, this decision was made due to the constraints that Haskell imposes.

C. TypeChecker

-- The typechecker was done in a recursive nature, similar to what we covered in class. For example, statements, while they have no type themselves, are correctly typed if they satisify the rules outlined in the spec. To satisfy the rules, we might need to type sub-components of the statement, and these are handled by the same recursive function that is typing the statement. Likewise, expressions are well typed if their arguments are well typed, their arguments are well typed if the sub arguments are well typed. This propagates down to the base cases of the type system, things are trivially well typed.

Scoping rules were implemented to be consistent with the milestone 2 spec. In particular, an identifier is in scope if it is declared in the current scope, or any parent scope.

To make sure Binary operations were well typed, first we checked to make sure both sides of the operation had the same type. Then we made sure that the type was compatible with the operation by finding out which built-in type the type correlated with if it was an alias.

The type checks we made were identical to those specified in the milestone 2 spec. In the examples below, we demonstrate 20 typecheck errors stemming from:

- Binary operations
- Append statements
- Assignment statements
- Casting
- Blocks (scoping blocks)
- For loops
- If statements

- Function declarations
- Variable declarations
- Short varaible declarations

The exact reason for an error and the enumerated type check rules leading to the error are given below.

3. Invalid Programs

-- Below, we give explainations as to why each program in types (https://github.com/Sable/comp520-2017-01/tree/master/programs/invalid/types) is incorrectly typed.

alias1.go

-- For the function main to be correctly typed, its body must be correctly typed. To correctly type its body, we must type the last variable declaration `var a bool = p1 == p2`. To type this declaration, the right hand side of the declaration must be correctly typed, and its type must equal `bool`. To type the right hand side, we must type the boolean equals operator. This is correctly typed if both arguments are of the same type. However this is not the case as p1 is of type point1 and p2 is of type point2.

append1.go

-- For the foo function to be correctly typed, the assignment `x = append(a,b)` must be correctly typed. This is correctly typed if the first argument is of type `array` or `slice`. and the second argument is of the same type as the underlying type of the first argument. In this case, the first argument is an int.

append2.go

-- For the foo function to be correctly typed, the assignment `x = append(a,b)` must be correctly typed. This is correctly typed if the first argument is of type `array` or `slice`. and the second argument is of the same type as the underlying type of the first argument. In this case, the second argument is not of the same type as the underlying type of the first argument. The first argument is an array with underlying type int, whereas the second argument is of type float64.

append3.go

-- For the foo function to be correctly typed, the assignment `x = append(a,b)` must be correctly typed. This is correctly typed if the first argument is of type `array` or `slice`. and the second argument is of the same type as the underlying type of the first argument. In this case, the second argument is not of the same type as the underlying type of the first argument. The first argument is slice with underlying type int, whereas the second argument is of type float64.

assign1.go

-- For the foo function to be correctly typed, the assignment `x, y, z, a = 1,2,3,4` must be correctly typed. for this to be correctly typed, the left hand side must be correctly typed. The identifier a is undeclared.

assign2.go

-- For the foo function to be correctly typed, the assignment `x, y, z = 1,2,p` must be correctly typed. for this to be correctly typed, the right hand side must be correctly typed. The identifier p is undeclared.

assign3.go

-- For the foo function to be correctly typed, the assignment `x, y, z = 1,2,1.234` must be correctly typed. for this to be correctly typed, the ith identifier must have the same type as the ith expression. The identifier is declared as type int in this scope, but it is assigend to `1.234`, which is of type float64.

block1.go

-- For the foo function to be correctly typed, the statement `return x` must be correctly typed. For this return statement to be correctly typed, its argument must be correctly typed. The identifier x is not declared in the scope of the return statement.

cast1.go

-- For the foo function to be correctly typed, the assignment `y = string(x)` must be correctly typed. For this assignment to be correctly typed, the type of `y` must equal the type of `string(x)`. The right hand side is correctly typed if the cast `string(x)` is correctly typed. This is not, since we can't cast `int` to `string`.

cast2.go

-- For the foo function to be correctly typed, the assignment `x = int(y)` must be correctly typed. For this assignment to be correctly typed, the type of `x` must equal the type of `int(y)`. The right hand side is correctly typed if the cast `int(y)` is correctly typed. This is not, since we can't cast `string` to `int`.

for1.go

-- For the foo function to be correctly typed, the for loop must be correctly typed. For the for loop to be correctly typed, its body must be correctly typed. For the body to be correctly typed, we need to type the assignment `x=y`. This is incorrectly typed, since the right hand side is not well typed as `y` is undeclared in the scope.

for2.go

-- For the foo function to be correctly typed, the for loop must be correctly typed. For the for loop to be correctly typed, its expression must be correctly typed and resolve to type `bool`. This is not the case since in the given scope, `expr` is typed as an int.

for4.go

-- For the foo function to be correctly typed, the for loop must be correctly typed. For the for loop to be correctly typed, its initialization statement must be correctly typed. For the init statement to be correctly typed, the assignment `x=1.5` must be correctly typed. This is not the case as x is already an integer in the current scope.

funcdec1.go

-- For the foo function to be correctly typed, it must return the same type it specifies as its return type. This is not the case as the return type is defined to be an `int` but `1.234`, a float64, is returned.

funcdec2.go

-- For the foo function to be correctly typed, the return type must type check. For the return type to type check, it must be defined. This is not the case since `bar` is undefined in the scope.

if1.go

-- For the function foo to be correctly typed, the if statement must type check. For the if statement to typecheck, its init statment must type check. For the init statement to type check, the assignment `x=y` must typecheck. This is not the case since the right hand side, `y` is undeclared in the given scope.

print1.go

-- For the function foo to be correctly typed, the statement `print(a)` must be correctly typed. For a print statement to be correctly typed, its argument can't be of some array type, it must belong to `bool, int, float64, string`.

expr20.go

-- For the function foo to be correctly typed, the assignment `z = x > y` must be correctly typed. For this to be correctly typed, the right hand side must be correctly typed. For the right hand side to be correctly typed, the binary operation `x > y` must be correctly typed. For this to be correctly typed, both arguments must be of the same, comparable, type. This is not the case since x is of type `int` and y is of type `float64`.

shortdec2.go

-- For the function foo to be correctly typed, the short variable declaration `x, y, z := 100, 200, 300` must be correctly typed. For this short declaration to be correctly typed, at least one identifier on the left hand side must be undefined in the current scope. This is not the case since all are aready declared in the scope.

vardclr1.go

-- For this file to be correctly typed, the second variable declaration `var x int` must be correctly typed. For this declaration to be correctly typed, the identifier `x` must not be already declared in the given scope. This is not the case since x is already declared in the given scope.