COMP 520 - Milestone 2

Alexandre St-Louis Fortier (260720127)
Stefan Knudsen (260678259)
Cheuk Chuen Siow (260660584)

March 24, 2016

Design Decisions

Symbol table

Initially, the symbol table consisted of a list of maps (as in dictionaries). We briefly considered the alpha-renaming approach, but since hash tables are typically used for symbol table because of its performance, we ended up changing the symbol table to utilize hash tables instead.

The way we design the symbol table is to implement a cactus stack of hash tables, as had been presented in class. The root represents the global scope with a hash table which stores the information for top-level declarations. Whenever a block is entered, a new frame is generated. This frame (`F`) has a new hash table associated with the new scope, and also has a pointer to the parent frame. This allows for the lookup of a declaration starting from the innermost scope that `F` resides, and recursively traverse towards the root node through the parent frame, but not the neighboring nodes as the scopes of these nodes and `F` are disjoint.

We went a little of what was suggested to handle some of the primitives. This can be seen with our extended type system which includes `TKind` types. We use `TKind` as the type of variables. So all base types have type `TKind(TSimp("#"))` where `"#"` is the type at the top of the type hierarchy. The use of `TKind` allowed us to handle type checking as the function call of a `TKind` identifier. Accordingly, `type` statements add a symbol of type `TKind` to the symbol table.

Since our ast doesn't encode the last position of a node, it was difficult to print the last line of a scope. Instead we provide an alternative view of the symbol table with the `-smartsymtab` which gives a better view of how some symbols shadow others.

Type checker

Type-checking closely followed the `typechecker.pdf`. New scopes are added at the beginning of every block, and at the end of every block, the top hash table is popped. At variable, type, and function declarations, a check is made to see if the `id` has been declared in the current scope. A type error is raised if it's already been declared.

Because of the initial idea of using an immutable map rather than a mutable hash table, type checking consisted of "threading" the context through lists, that is, mapping while passing each context onto the next element of the list. Since we decided not to use the map, this is no longer needed, (simply mapping suffices).

Because type declaration is allowed in `GoLite`, the approach of matching the types and operators that we had seen for the `minilang` compiler wouldn't work. We took the approach of pattern matching on the operator and check if two types were unifiable. If their upper bound were belonged to the class of types allowable for the operator, then the upper bound was the type returned.

Weeder

The weeder checks for invalid programs that we deferred from milestone 1. In particular, we check for

- Invalid use of basic types (`int`, `float64`, `bool`, `rune`, `string`) as an identifier, since we didn't specify those basic types as keywords.
- Invalid use of underscore.
- Negative index for arrays.
- Invalid use of expression as expression statement or left value.
- Multiple `default` in a switch statement.
- Invalid use of `break` and `continue` outside of loops.

Pretty printer

We decided that we needed to modify the AST to include annotations. This required making changes to the pretty printer from the first deliverable. Due to the nature of the AST, only minor modifications were then needed to allow type printing of the typed AST. Types are printed as block comment right after the expression it is associated with.

Testing

1. `programs/invalid/types/1_functionDecl.go` checks for the redeclaration of a variable in a function body, which has the same name as the parameter of that function.
2. `programs/invalid/types/2_shortDeclaration.go` checks for short declaration that has at least one variable on the left-hand side that is not declared.
3. `programs/invalid/types/3_opAssignment.go` checks for the types of left value and right-hand side expression and whether both types are compatible for the operator.
4. `programs/invalid/types/4_block.go` checks for variable that is declared in another scope.
5. `programs/invalid/types/5_forLoop.go` type checks a three-part for loop. In this case the middle expression of the for loop is not a `bool` type.
6. `programs/invalid/types/6_switchStmt.go` type checks a switch statement. In this case the `print` statement in the case clause fails the type check because of invalid use of arithmetic negation on a string.
7. `programs/invalid/types/7_functionCall.go` checks for the number of parameters of a function and the number of arguments used in a function call matches.
8. `programs/invalid/types/8_fieldSelection.go` checks for undefined field access of structs.
9. `programs/invalid/types/9_append.go` checks whether the variable used in the first argument of `append` is of slice type.
10. `programs/invalid/types/10_typeCast.go` checks for equivalent types between the type of the left value and the type cast.
11. `programs/invalid/types/11_typeRedeclaration.go` checks for binary operations between two variables that share the same type alias, but then one of the variable had a type alias redeclared to another type in an inner scope.

One thing we noticed about the reference GoLite compiler is that the `init` statements for the `if` and `switch` statements can shadow variables declared in the same scope, but the `init` statement for the three-part `for` loop does not. For our compiler, the `init` statements for `if`, `switch`, and `for` do shadow variables declared in the same scope. This is in accordance with the Golang compiler and the Golang spec that mentions that `if`, `for`, and `switch` are enclosed in an implicit `block` statement.

* The remaining valid and invalid test programs for type checking are located in `programs/type/`

Contributions

Main

Cheuk Chuen made modifications to main.ml that allowed for the -dumpsymtab and -pptype flags.

AST

Alexandre modified the AST so as to add annotations.

Symbol table

Cheuk Chuen wrote the symbol table, and Alexandre made changes according to the modified type checker.

Type checker

Stefan laid the ground work for the type checker, and Alexandre made changes according to the modified AST.

Weeder

Alexandre wrote the weeder.

Pretty printer

Stefan modified the pretty printer to work for the new AST and to print the types of typed expressions.

Testing

Cheuk Chuen wrote the majority of the tests.