COMP 520 - Milestone 2 Report

Shafin Ahmed - 260782651
Nathaniel Branchaud - 260800960
Brendon Kurl - 260685377

March 31, 2020

1. Weeded Parsing Issues:

To fix some of our parsing issues from milestone 1, we made use of the weeder. In particular, previously, our parser could not handle parentheses around identifiers in assignments. Adding a rule to allow left and right parenthesis before and after the identifier in the assignment would lead to shift/reduce errors. The solution we went with was to simply allow expressions in the assignment instead of just the identifier and weed out any expressions that were not identifiers or identifiers in parentheses. Other parsing issues of a similar nature, e.g. parentheses around types/function names/etc., were also fixed using the weeder in a similar way.

We also use the weeder to check whether functions that have a return type have a terminating statement at the end. This was fairly simple to implement as all we had to do was traverse through the AST to the tail of the statement list in the function and check whether it satisfies the conditions of a terminating statement. If, on traversing to the tail, we find a terminating statement and we still haven't reached the tail, then that means there is unreachable code and we raise an error.

2. Team Organization:

For the sake of simplicity, we distributed the work the same way we had for milestone 1. For milestone 1, Nathaniel covered the declarations, Brendon covered the expressions, and Shafin covered the statements. Therefore, each of us did the symbol table and typechecker implementation for our respective portion. This ensured that we would each be working with the parts of the AST that we had designed ourselves, minimizing confusion related to the design decision taken for milestone 1.

3. Symbol Table Design:

Our symbol table consists of a simple cactus-stack of hash-tables where in addition to a table for the current scope and a parent pointer each stack node has a value representing the return type of the current function (or that the current scope is not in a function). We decided that for any given AST node, typechecking would happen before any changes are made to the symbol table. Therefore, mappings in the symbol table are only added or modified if the associated AST node passes typechecking. In terms of printing the symbol table, we simply print each new mapping as it is encountered. This means that the printed symbol table will be incomplete if an error is encountered, but given that our symbol table is implemented as a cactus-stack it is the simplest solution available to us.

4. Scoping Rules:

The scoping rules we used are fairly straightforward. Each block (if/else, loop, function body, etc.) is treated as its own scope. The init statement in if/else and for loops are treated as their own

inner scope. This allows us to shadow variables in the init statement properly, and the parent pointer of the body block simply points to this scope letting us re-declare variables from the init statement if needed. The parent pointer of the post statement in a for loop also points to the init statement's scope. When checking if an identifier has already been declared, only the immediate scope is checked allowing local variables to shadow global ones. When looking up the value of an identifier however we look through all preceding scopes before deciding the identifier has not been declared. For short declarations, we look at all identifiers in the current scope and keep track of whether there is at least one new identifier being declared. A function's arguments are added to the function's scope before the statements in the function's body are considered.

5. Typechecker Design:

Our typechecker verifies much more than if the type of a variable's specified value matches its declared type. For one, it checks that a struct or set of function arguments do not contain multiple parameters with the same name. This is done by creating a list of the parameter names and associated types and ensuring that the names list contains no duplicates. Since checking membership in a list is very easy in OCaml, the implementation of this is fairly simple.

Another task our typechecker performs is ensuring that the specified type of a variable or type declaration actually refers to a type and not a variable. To do this, we needed to differentiate between declared custom types and variable declared as a custom type. The typechecker then simply needs to check that any specified type is either the former or a built-in type and raise an error if it is the latter.

6. Rules for Invalid Programs:

- `blank_package.go`: The identifier used for the package name cannot be the blank identifier.
- `decl_mismatch.go`: Type 'float64' cannot be assigned to type 'int' without a cast.
- `main_as_var.go`: 'main' can only be declared as a function at the toplevel..
- `main_with_arg.go`: 'main' must be declared with no arguments.
- `non_void_init.go`: The return type of 'init' must be void.
- `param_same_name.go`: A struct cannot be declared with two parameters of the same name.
- `recursive_type.go`: The name of a declared type cannot be the same as its underlying type.
- `redeclaration.go`: A variable cannot be declared twice in the same scope.
- `undeclared.go`: A variable cannot be referenced before it is declared.
- `var_as_type.go`: The specified type in a variable or type declaration must resolve to a type.
- `block_scope.go`: A variable declared in an inner scope cannot be referenced in an outer scope.
- `for_post.go`: The post statement will not type check because bools cannot be incremented.
- `if_not_bool.go`: The expression in an if statement has to resolve to type bool.
- `inc_dec_mismatch.go`: Expressions being incremented/decremented have to be of a numeric type.
- `no_new_vars.go`: Short declarations need to declare at least one new variable.
- `op_assign_mismatch.go`: The operation in the assignment should be between valid types.

- `print_base_type.go`: Print statements cannot print non base types such as structs.
- `return_mismatch.go`: A function's declared type and the type it returns must match.
- `return_void.go`: The return statement of a void function must not return any other type.
- `switch_mismatch.go`: The types of all the case expressions must match the switch expression type.
- `for_not_bool.go`: The middle expression in a for loop should resolve to type bool.
- `non_terminating_func.go`: Functions with return values should end with a terminating statement.
- `string_rune.go`: Cannot subtract a string and a rune.
- `unreachable_code.go`: Functions with return value should not have code after a terminating statement.
- `switch_bool.go`: Case expressions in switch statement without expression should resolve to bool.
- `assign_mismatch.go`: A variable's assigned type and declared type must match.
- `bad_func_call.go`: A function must be called with the same number of arguments it was declared with..
- `binop_mismatch.go`: Both expressions in a binary operation must be of the same type.
- `unop_mismatch.go`: The unary minus operator is only defined for numeric types.
- `invaild_index.go`: Only an int can be used to index a slice or array.

7. Resources Consulted:

We worked alone.