Laurence Stolzenberg (260657154)

Maxence Hull (260706895)

Milestone 2

Due March 12th 2017

Compiler Design (COMP520)
Presented to Alexand**er** Krolik

McGill University, School of Computer Science

Decisions and details of our project:

Type checking:

This milestone was significantly more difficult than the previous milestone. Here is a list of a few of our design decisions made during the course of this milestone:

- Type checking for function calls: Basically, we created a separate hash table for function declarations, where we used the function name as the key, and the object was a structure containing a return type (or null) and the types of each variable in the parameter list.
    - Any time that function was called, we checked that each parameter in the function call had the **SAME** as what was in the function declaration.
    - This decision is based on the act that we have explicit casts and I do not think we mentioned implicit casts, meaning if we have a basic type alias and we use it instead of the basic type used when declaring the function, it would probably be valid, but in our compiler, it will throw an error. Please keep this in mind.

- For our return statement, we simply check if the parent node is a function declaration, if so we get its name (and ensure it is not a cast statement), then we check in our function declaration table and compare types to ensure validity.

  - Once again, an alias cannot be substituted for the basic type here by design, this is simply how we interpreted the assignment. If this is not the case, please tell us!

- If ever we should have supported implicit casts, it is very simply to do so in our code, it is simply necessary to uncomment 3 lines and comment the first line in the method “safeTypeCompare” at the beginning of the typechecker.java file.

Scope:

- For scope, our scope contains all variables and type of parent scopes. Our scope is actually a stack of hash tables, where each hash table is a scope containing a list of variables and types. When we exit a scope, we pop the top hash table in the stack and the information from the child scope disappears. This part is somewhat standard.
- Hashtables are composed of a special data structure. This structure contains an identifier (a string), a category (a variable, a type …), a type (represented by a string) and a boolean (which indicates if this entry can be shadowed or not).

- We choose to represent types as strings because they are very easy to manipulate. As an example an array of integers is represented as “[5]int”, a slice of type point as “[]point” and a structure point (composed by two integers x and y) as “int,int”. Regular expressions are used to determine if a type is an array, a slice or a structure.
- Every time we pop a table, we check if the symtab or symtaball flags are set, and if either is in fact set, we print the appropriate information.

SableCC:

- We extensively used SableCC DepthFirstAdapter to browse the AST. This functionality has been particularity helpful: it significantly reduces size of our code.
- We took advantage of SableCC nodes hierarchy. As an example, a class Statement has children EmptyStatement, ExpressionStatement, DeclarationStatement… This native functionality of java (an object-oriented language) makes some difficult tasks easier (recursions is one of this case).

Problems:

- We changed our grammar using some new knowledge of how to make an AST with SableCC. This made our code and grammar much simpler, but pretty much destroyed our pretty printer. We restarted coding it, but since it was not mentioned in the grading for this assignment, it will have to wait until next milestone.
- SableCC has a few drawbacks: nodes names can become very long. Moreover, class names are tightly coupled to the SableCC file (basically, changing the name of a high-level production will completely destroy the class hierarchy).

- SableCC error messages concerning the AST construction are very obscure in some cases (resulting in a lot time wasting trying to fix a minor mistake).

Team member contributions:

Maxence:

- Majority of the basic type checking code.
- Implemented the scope / stack code.
- Created most of the type tests.
- Improved weeding phase (concerning return statements)

Laurence:

- Implemented the flags and information printing.
    - Implemented dumpsymtable and dumpsymtableall.
- Began recoding the pretty printer.
- Function declaration type checking
- Return statement type checking
- Function call type checking
- Cast statement type checking
- Created a few type tests.

Tests:

- My_inv1.go: //Using an incorrect type in a function call parameter list.
- My_inv2.go: //Assigning function call that results in an int to a string variable.

  Invalid

- Unknown_identifier.go //Use an undeclared variable
- Switch_1.go //Use a float as a case expression instead of an integer
- Access_2.go //Access an array with a float as index
- Assignement_1.go// Assign a float to an int
- Comparable_1.go //Compare (==) two arrays
- Dec.go // Decrement a string
- For_stmt_1.go // Use an integer as an expression (instead of a boolean)
- Short_dcl_2.go //Lack of new variable in a short declaration
- cast1: cast from string type (invalid)
- cast2: cast to string type (invalid)
- cast3: cast from struct type (invalid)
- cast4: cast to struct type (invalid)