PA4 -- Semantic Analysis

---

1. Description

For this assignment, you will implement the logic to do the semantic check by "recursively" traversing the whole Abstract Syntax Tree(AST) that you built from PA3.

From PA3, you will have a syntax-correct AST ready to do the semantic check. Your AST is represented by the root node: **Program**. Therefore, the process of semantic check is traversing the whole AST ,starting from the root node, and check whether there is any semantic error or not in each AST node. The way to traverse through every node in the AST is by calling **Check()** on every AST node.(For **Expression** node, it will call **CheckExpr()**).

Therefore, your job is to implement the **Check()** method(For **Expression** node, you will implement **CheckExpr()** method) for most of the AST nodes. After you finish the implementation, the function call: **program->Check()** should recursively call all the **Check()** and **CheckExpr()** methods on all the nodes in AST thus finish the semantic check of the whole program.

Note: When you are implementing the Check() method for ast classes, you don't need to worry about any syntax errors because those errors will be caught during the syntax check.

2. Starter Code

Makefile	used to build the parser
utility.h / .cc	interface / implementation of various utility functions
errors.h / .cc	class that defines error reporting methods
location.h	class that defines type for token location
lexer.h	type definitions and prototype declarations for lexer
lexer.l	flex input file(use your own from PA3)
parser.h	declare parser functions and types
parser.y	implementation of the parser(use your own from PA3)
main.cc	the entry point where the whole program will be started
samples	directory contains test input files
ast.h / .cc	**interface / implemention of AST node classes**
ast_type.h / .cc	**interface / implemention of AST type classes**
ast_decl.h / .cc	**interface / implemention of AST declaration classes**
ast_expr.h / .cc	**interface / implemention of AST expression classes**
ast_stmt.h / .cc	**interface / implemention of AST statement classes**
symtable.h / .cc	**interface / implemention of symbol table**

When you are going through the starter code, please read the comment as well. The comments are really helpful in explaining the structure of the code.

In this assignment you need to implement **Check()** methods(For all of the Expr classes, you will implement **CheckExpr()** instead of **Check()**) for most of the ast classes so you will

need to modify all the ast classes files. You also need to add more methods that you need in symtable.cc.

In order to help you to get started, we have provided some of the implementation for the following three files:

1. symtable.cc

   Take a look at all the methods we provide. Make sure you understand the meaning of each method and how to use them. You will need to use the symbol table for almost all of the **Check()** and **CheckExpr()** methods.

2. ast_decl.cc

   We have provided part of the implementation of **VarDecl** and **FnDecl**. They are not complete implementation, so you need to finish them. The given partial implementation for these two classes should give you a really good idea on how to implement the **Check()** and **CheckExpr()** methods for other classes. Please read them thoroughly.

3. ast_stmt.cc

   In this file we provide the complete implementation of Check() method for **program**. It is the beginning of semantic check. As you can see from the code that it just loops through all of the declarations in the input code and call Check() on each of them.

3. Example

Before you run the executable glc, add program->Check() to the very bottom of the action for Program:DeclList. You can keep program->Print(0) for debug purpose, but it should be deleted for final checkout.

To play with the starter code to see what the given implementation for **VarDecl** and **FnDecl** is doing, you can take a look at the 2 test cases, **test1.java** and **test2.java**, in samples folder. You can see that they are the test cases for **declaration conflict error** since they are trying to use the same name for different declaration in the same scope.

To run it, just do:

```
%  make
% ./glc < samples/test1.java
```

And the output should be:

```
% *** Error line 2.
% boolean a;
          ^
% *** Declaration of 'a' here conflicts with declaration on line 1
```

4. Testing your work

We will provide some simple test cases and the corresponding output which you can find in the **samples** folder. You should definitely come up with your own test cases as well to test that you have covered all the potential semantic errors.

5. Grading

You can go to any teaching staff's lab hours and ask for a check-off when you are done with your assignment. You will be asked to run your assignment against the given test cases, and we will check the output of your assignment. You also need to generally describe how did you approach the assignment to the teaching staff.