**CSE 340 Principles of Programming Languages**

**Spring 2014**

**Programming Assignment #3**

**Due Date: Monday, April 14th, 2014**

**OBJECTIVE**

Create a semantic analyzer for the programming language described in class. Incorporating it with the lexical and syntax analyzer (developed in assignments #1 and #2) as a cohesive project.

**INSTRUCTIONS**

1. Assure that your Lexer (assignment #1) and Parser (assignment #2) work properly. If this is not the case, you can ask for help among your classmates and even borrow the Lexer and/or the Parser (assignment #2) of somebody if he/she allows you. You must give proper recognition to him/her.
2. Create a new file to define data structures and functions that support semantic analysis. Call this file **SymbolTable.c or SymbolTable.cpp**. In this file do the following:
   1. Define your symbol table data structure and its functions.

Recommendation 1. Create a class SymbolTable

Recommendation 2. Implement the symbol table container as a std::map<string, vector>

Recommendation 3. Implement at least the following functions: insert, search, and get. Lecture’s slides (March 31st) provide a description of this.

The symbol table for the following code,

int i;

char j; int m;

**void method(int n, char c) {**

int i;

i = j; i = m;

}

Will be something like this:

(“i”, ((“int”, “global”),(“int”, “method-int-char”)))

(“j”, ((“char”, “global”)))

(“m”, ((“int”, “global”)))

(“method-int-char”, ((“void”, “function”)))

(“n”, ((“int”, “method-int-char”)))

(“c”, ((“char”, “method-int-char”)))

* 1. Implement the Cube of Types. The content of the cube is specified in the Appendix A at the end of this document.

Recommendation 4. Implement the Cube using a three-dimensional array. Lecture’s slides (March 31st) provide a description of this.

* 1. Define the stack that you are going to use to calculate types.
  2. Define a global variable to store the “currentScope” (the name of the method being parsed).

1. Create a new file to define the functions needed for semantic analysis. Call this file **Semantic.c or Semantic.cpp.** In this file do the following:
2. Include the definitions that you did in the SymbolTable file and/or create instances (objects) as needed to have access to (1) an instance of the symbol table data structure and methods, (2) the Stack, (3) the Cube, and (4) the variable currentScope.
3. Define a function that reviews declaration and unicity of variables.
4. Define a function that reviews type matching in assignation.
5. Define a function that reviews type matching in the return expression.
6. Define a function that reviews boolean type for “if” and “while” conditions.
7. Define a function that reviews type matching and number of parameters.
8. Define a function, called calculateTypeBinary(string type1, string type2, string op) that uses the Cube and the Stack to calculate the resultant type for binary operators.
9. Define a function, called calculateTypeUnary(string type1, string op) that uses the Cube and the Stack to calculate the resultant type for unary operators.
10. Inside your Parser file:
    1. Create an instance of the Semantic class or include the Semantic file in order to have access to the functions that you defined before.
    2. Add to each of the methods that represent your rules, the needed code (1) to store a variable or method on the SymbolTable, or (2) to review for semantic errors.

Recommendation 5. Lecture’s slides (March 31st) provide a description of this.

1. The semantic analyzer must report the errors and the number of the line (obtained from your current list of Tokens) in which they occur. **Assume that the piece of code being analyzed is free of lexical and syntax errors.** Your semantic analyzer should be able to recognize the following type of errors:
2. **Declaration and Unicity** of variables (considering scope) and methods.
3. **Types** of variables match the values assigned to them.
4. **Conditions**. The conditons have a boolean value.
5. **Return**. The returned value by a method matches the type of the method.
6. **Paramters** in a method match in type and number with the declaration of the method.

The semantic analyzer should have the following messages:

* Duplicated variable <variable> in line <line>
* Duplicated method <method> in line <line>
* Variable <variable> not found
* Method <method> not found
* Type mismatch in line <line>
* Boolean expression expected in line <line>
* Return type mismatch in line <line>
* Type or number of parameters in method <method> are incorrect in line <line>

The output should be in a file, named output.txt. Use the text above, as it is, for each error to write the output file. The words between brackets (< >) should be replaced for the actual value WITHOUT the brackets, i.e., REMOVE the brackets.

1. Read and actively participate in the discussion board. Take advantage of the discussion content to improve your implementation.
2. Create a .zip file for your submission. If you are working in group of two, only one submission is needed. The name of the file should reflect the name(s) of the people submitting. E.g. If you are working alone, your project should be named as Firstname\_Lastname\_Projnumber.zip; if you are working in a group, your project can be named as FirstnameLastname(member1)\_FirstNameLastname(member2)\_Projnumber.zip, first letters of your firstnames and lastnames should be captital such as YuanWang\_JavierGonzalez\_proj3.zip. Your submission should include the following files: SymbolTable, Semantic, Parser, Lexer, Token.
3. Create a **makefile** to control the compilation of your code. The makefile should have at least a default target that builds your project. Google “makefile tutorial” for how to write makefiles. Before submitting your project, don’t forget to try the makefile at general.asu.edu and make sure it is fully working (able to compile and run).

**Remember**

* No credit will be given to programs that do not compile on general.
* No credit will be given to programs that do not execute correctly on general.
* No credit will be given if the output of your program does not match the expected output.

**Appendix A. Cube of Types**

**OP 🡪 {+}**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| OP | int | float | char | string | boolean | void |
| int | int | float | error | string | error | error |
| float | float | float | error | string | error | error |
| char | error | error | error | string | error | error |
| string | string | string | string | string | string | error |
| boolean | error | error | error | string | error | error |
| void | error | error | error | error | error | error |

**OP 🡪 {-, \*, /}**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| OP | int | float | char | string | boolean | void |
| int | int | float | error | error | error | error |
| float | float | float | error | error | error | error |
| char | error | error | error | error | error | error |
| string | error | error | error | error | error | error |
| boolean | error | error | error | error | error | error |
| void | error | error | error | error | error | error |

**OP 🡪 {-} // minus unary**

|  |  |
| --- | --- |
| OP |  |
| int | int |
| float | float |
| char | error |
| string | error |
| boolean | error |
| void | error |

**OP 🡪 {>, <}**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| OP | int | float | char | string | boolean | void |
| int | boolean | boolean | error | error | error | error |
| float | boolean | boolean | error | error | error | error |
| char | error | error | error | error | error | error |
| string | error | error | error | error | error | error |
| boolean | error | error | error | error | error | error |
| void | error | error | error | error | error | error |

**OP 🡪 {!} // no**

|  |  |
| --- | --- |
| OP |  |
| int | error |
| float | error |
| char | error |
| string | error |
| boolean | boolean |
| void | error |

**OP 🡪 {&, |}**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| OP | int | float | char | string | boolean | void |
| int | error | error | error | error | error | error |
| float | error | error | error | error | error | error |
| char | error | error | error | error | error | error |
| string | error | error | error | error | error | error |
| boolean | error | error | error | error | boolean | error |
| void | error | error | error | error | error | error |

Consider the following:

* ERROR operator ERROR is ERROR
* ERROR operator <any type> is ERROR
* <any type> operator ERROR is ERROR
* unary\_operator ERROR is ERROR