Lab 3 : A Semantic Analyzer for `smallC`

1 Objectives

The objective of this assignment is to build a semantic analyzer for a simple programming language like `smallC`. In the process, you will learn how to build an abstract syntax tree (AST), build a symbol table, and perform actions using the AST and the symbol table to check for semantic errors. You will utilize ANTLR4 and the grammar for `smallC`.

2 Reading and Preparation

In preparation for the assignment, please read the following online documents:

- ANTLR4 documentation on actions and attributes, available here.
- The *updated* "`smallC` Programming Language" handout, released with this assignment.
- The C++ Standard Template Library (STL), particularly the `vector` and `map` containers. A description of STL containers can be found online here.
- The use of C++ template classes. A brief overview is available here.

3 Problem Statement

This assignment consists of three parts:

1. In part 1 of the assignment, you will write a set of C++ classes that describe nodes of the AST of a `smallC` program. You will then extend the ANTLR4 grammar of `smallC` with *actions* that build the AST of a program as it is parsed.
2. In part 2 of the assignment, you will build a class to represent a symbol table and then further extend the AST classes you built in part 1 above to connect symbol tables to nodes, as appropriate.
3. In part 3 of the assignment, you will write a semantic checker that utilizes the AST and the symbol tables to perform a set of basic semantic checks.

3.1 The AST

The example AST shown in Figure 1 illustrates the structure of the AST for a `smallC` program. The root of the tree is a program node, which has scalar, array, and function declarations (prototypes and otherwise) as children.

A scalar declaration node stores pointers to the declared identifier node and the primitive type node of the declaration. Similarly, an array declaration node has a pointer to the declared identifier node and a pointer to an array type node. The identifier node in each case stores the name of the variable. The primitive type node simply stores the type of the variable while the array type node also stores the size of the array.

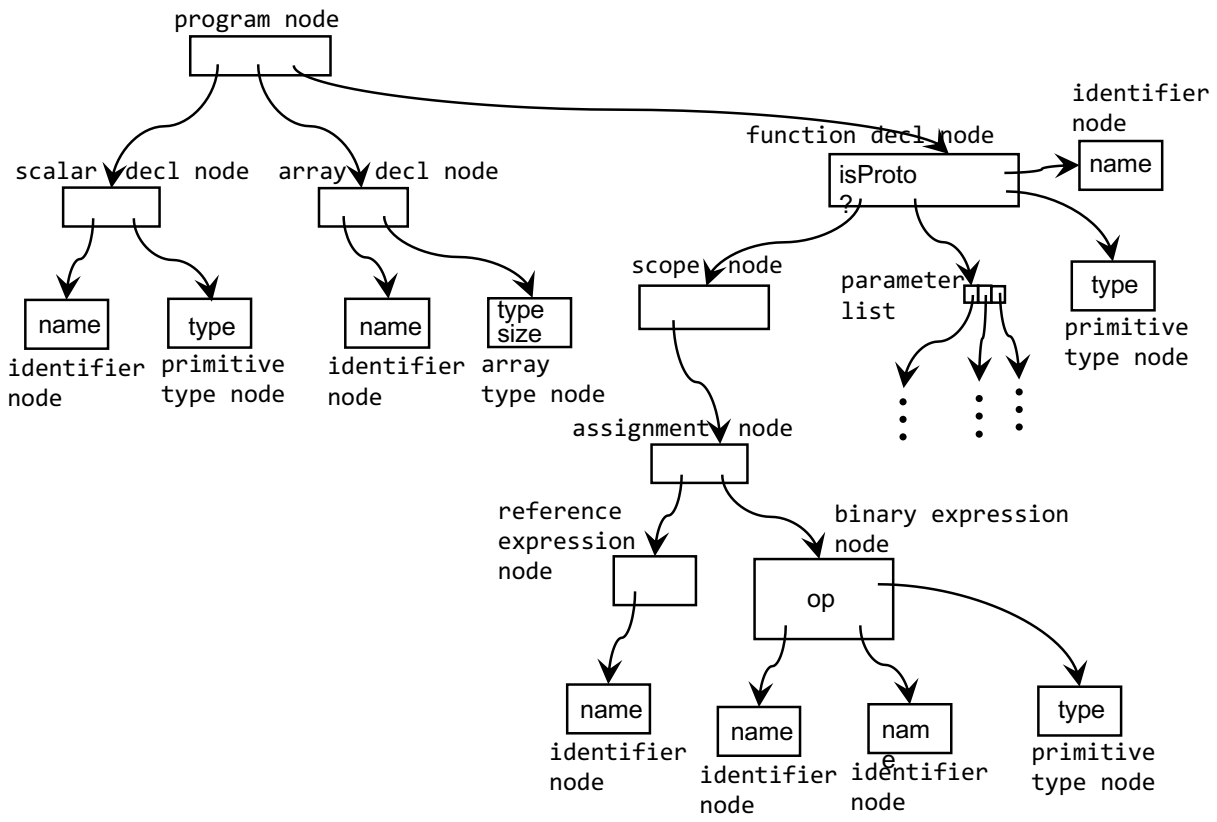

Figure 1: An example AST.

A function declaration node contains pointers to the function name, its return type and a list of function parameter and their types (as identifier nodes and type nodes). It stores a boolean flag to indicate if the corresponding function declaration is a prototype. It also has a pointer to its scope node that represents the function's body (when the declaration is not a prototype).

A scope node has a pointer to its own scalar and array declarations. The figure does not show such declarations due to space constraints. The scope node also has as children has the statements in its body. The figure shows only one child, which is an assignment statement.

The assignment statement has a pointer to its left-hand-side (lhs) reference node as well as the right-hand-side (rhs) expression node, which in the example is a binary expression node. This expression node stores the operation and a pointer to the type of the expression and has pointers to the lhs and rhs nodes of the binary expression, shown as identifier nodes in the figure.

The AST is implemented with a set of classes, each representing a specific type of node in the AST. The definitions of these classes are given in the file `ASTNodes.h`. The class hierarchy of the classes is shown in Figure 2. All AST nodes inherit from `ASTNode`. There also exists a number of classes that are base classes for related groups of nodes. For example, the `StmtNode` class is used as the base class for all concrete statement nodes, such as `ifStmtNode`, `whileStmtNode`, etc. Note that you may change the definition of these classes as you wish, but it is recommended that you use (and thus implement) the proposed definitions.

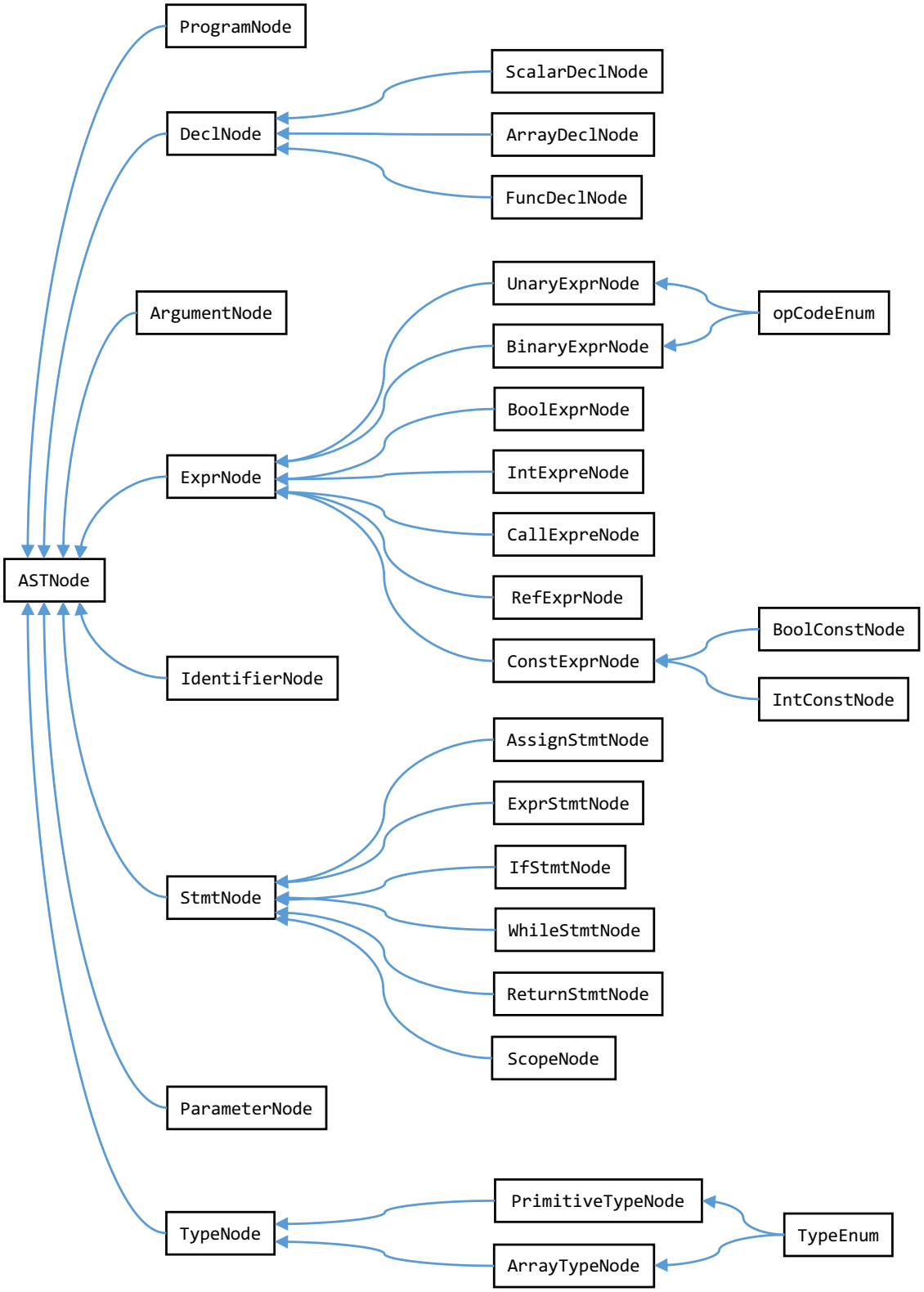

Figure 2: The AST class hierarchy.

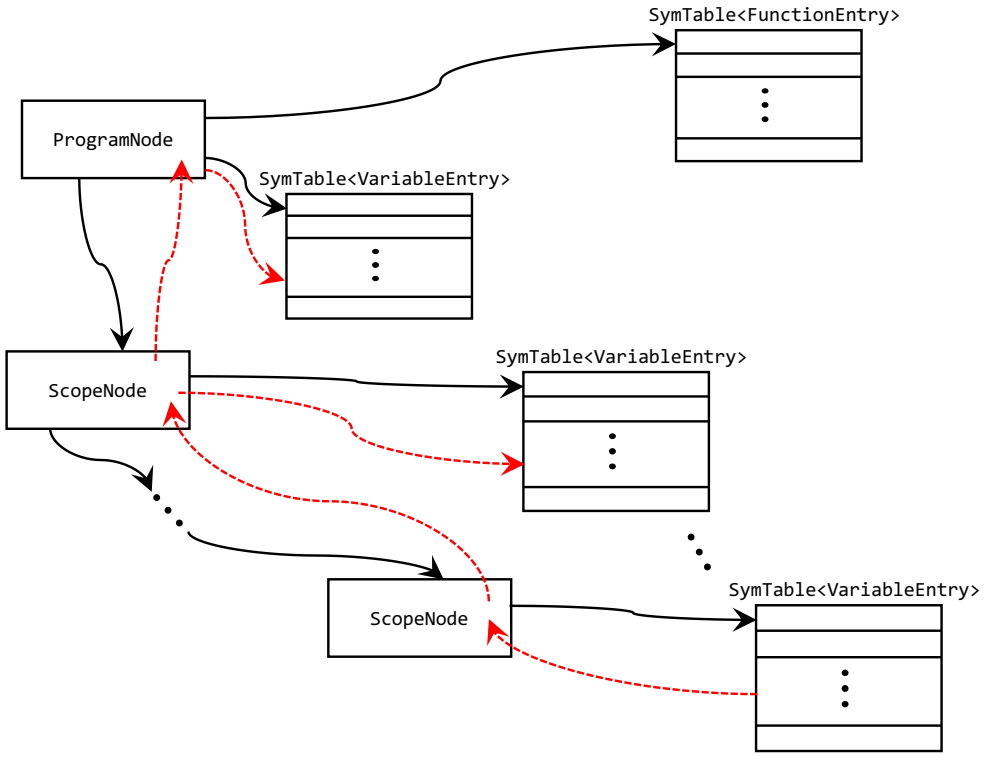

Figure 3: The symbol table structure. The red dotted arrows show the implicit chaining of nested scope tables.

3.2 Symbol Table

The symbol table consists of a number of individual tables, one for each scope, as shown in Figure 3. The individual tables are implicitly chained together through their respective `ScopeNode`s. The `ProgramNode` has one table for global variables and another for functions defined in the program. The chaining allows one to search in a table of a scope, and, if necessary, search in other *parent* scopes in which the scope in nested in, ultimately ending with the globals table.

For simplicity, each symbol table is defined by an STL `map`, where the key is the name of the symbol and the value is a symbol table `entry`. An entry is one of two type: a *variable* entry or a *function* entry. A variable entry stores the type of a variable and whether it is any array or a scalar. In contrast, a function entry stores a function's return type, and list of its parameters and their types. It also indicates of the entry is for a prototype or a definition.

The `SymTable.h` gives the `SymTable` class declaration. Note that `SymTable` is a template class and this instances of it must be instantiated and used with the appropriate template type, e.g., `SymTable<VariableEntry>` or `SymTable<FunctionEntry>`. You will implement this class in the file `SymTable.cpp`.

3.3 Semantic Analysis

Semantic analysis is performed by traversing the AST, processing children before parents. When a node is visited semantic checks can be performed.

In order to facilitate visiting AST nodes, a base visitor class called `ASTVisitorBase` is provided, both the definition and the implementation. Thus, the semantic analyzer can be built as a class

```
int z;

int main() {
   int x;
   bool y;

   if (x) return -1;
   if (y) return 1;
   return 0;
}
```

(a) Example input `smallC` program.

```
Program [useIO=0](1,0)
    Scalar Declaration    Int(1,0)
        Identifier[name:z](1,4)
    Function[isProto=0]    Int(3,0)
        Identifier[name:main](3,4)
        Scope(3,11)
            Scalar Declaration                Int(4,4)
                Identifier[name:x](4,8)
            Scalar Declaration                Bool(5,4)
                Identifier[name:y](5,9)
            IfStmt[hasElse=0](7,4)
                Int Expression(7,8)
                    Reference(7,8)
                        Identifier[name:x](7,8)
                ReturnStmt(7,11)
                    Int Expression(7,18)
                        IntConstant[val=-1](7,18)
            IfStmt[hasElse=0](8,4)
                Int Expression(8,8)
                    Reference(8,8)
                        Identifier[name:y](8,8)
                ReturnStmt(8,11)
                    Int Expression(8,18)
                        IntConstant[val=1](8,18)
            ReturnStmt(9,4)
                Int Expression(9,11)
                    IntConstant[val=0](9,11)
```

(b) The corresponding `ASTPrinter` output.

Figure 4: An example of `ASTPrinter` output for a simple program.

that inherits from this base visitor and extends it with the necessary semantic checks.

An example visitor class called `ASTPrinter` is provided with the assignment release. This class inherits from and extend the base visitor class `ASTVisitorBase` to print out the names of the AST nodes, along with some key information about each node. The names of the nodes are indented to reflect parent-child relationships. An example program and the output produced by invoking the AST printer is shown in Figure 4. The `A3Sema.cpp` file shows how to construct and invoke this visitor. You can follow the example of this visitor to implement your semantic analyzer.

3.3.1 Semantic Checks

The set of semantic checks that must be performed by the semantic analyzer are:

1. No two variables may have the same name in the same scope. A failure of this check results in error message #0. In this case `ident` refers to the variable name.

2. A variable or a function must not be used before it is first declared in a visible scope. A failure of this check results in error message #1. In this case, `ident` refers to the variable/function

Error #	Error Message
0	sema: `ln#:cl#` redefinition of `ident`
1	sema: `ln#:cl#` use of undefined identifier `ident`
2	sema: `ln#:cl#` no matching definition for `ident`
3	sema: `ln#:cl#` mismatched return statement
4	sema: `ln#:cl#` definition inconsistent with earlier definition of `ident`
5	sema: `ln#:cl#` invalid condition in `stmt` statement
6	sema: `ln#:cl#` type mismatch
7	sema: `ln#:cl#` size cannot be negative for array `ident`
8	sema: `ln#:cl#` invalid use of identifier `ident`

Table 1: Semantic error messages. `ln#` is the line number, `cl#` is the column number, `ident` is the identifier name, and `stmt` is either `if` or `while`.

name.

3. No two functions may be defined with the same name, return value and signature (i.e., parameters and their types). A failure of this check results in error message #0. In this case, `ident` refers to the function name.

4. Operands of arithmetic operators (i.e., +, -, * and /) must of the integer type. A failure of this check results in error message #6.

5. Operands of logical operators (e.g., `==`, etc.) must of the same type (either integer or boolean). A failure of this check results in error message #6.

6. The type of the left hand side variable of an assignment statement must be of the same as the type of the expression of its right hand side. A failure of this check results in error message #6.

7. The type of the condition of an `if` or a `while` statement must be of the boolean type. A failure of this check results in error message #5. In this case, `stmt` is either `if` for an `if` statement or `while` for a `while` statement.

8. A function definition must match its prototype (if one is provided). That is, the return types and the number/types of arguments must match. The same applies to multiple prototypes declarations of the same function. Note that multiple identical prototype declarations of a function are allowed. A failure of this check results in error message #4. In this case, `ident` refers to the function name.

9. A function's return statement expression must match the return type of the function. Further, a function with return type `void` must not have an expression in its return statement. Similarly, a non-`void` function must have an expression in its return statement. Note that it is acceptable for a `void` function to have no return statement. However, you can ignore the case in which a non-`void` function has no return statement. A failure of these checks results in error message #3.

10. A function call must have the same number/types of arguments as its definition. A failure of this check results in error message #2. In this case, `ident` refers to the function name.

11. Array sizes in array declarations must be positive integers. A failure of these checks results in error message #7. In this case, `ident` refers to the array name.

12. Scalar variables must not be used as arrays and vice versa, i.e., array variables must not be used as scalars. A failure of these checks results in error message #8. In this case, `ident` refers to the scalar or array name.

Please note that the errors must be allowed to *chain*. For example, in an assignment `z = x + y`, if `x` is not declared prior to its use, the type of the expression `x + y` must be set to `void` to allow the for a type mismatch error to be generated for the assignment statement.

The above are not all the semantic errors that need to be checked in a `smallC` program. For example, there are no checks for unary expressions nor for passing array variables to functions. However, the above set of checks will suffice for the purpose of the this lab assignment.

3.4 Example

The is an example input file (line number are not part of the code, they are only for your reference):

```
 1 #include "scio.h"
 2 int x;
 3 int x; // re-definition
 4
 5 void helper1(int x, int y);
 6 void helper2(int x, bool y);
 7
 8 int main() {
 9    bool w;
10    bool w; // re-definition
11
12    int a[10];
13    int a[11]; // re-definition
14
15    int x;
16    int k;
17    bool t;
18    bool y;
19    bool z;
20
21    int b[-1]; // invalid array size
22
23    ab = -3; // undefined variable
24    w = c; // undefined variable
25
26    fcn2(); // undefined function
27
28    // Operand mismatch
29    x + t; // operand mismatch
30    t < x; // operand mismatch
31    t == true;
```

```
32    t = false;
33    t = 5; // operand mismatch
34
35    // Call mismatch
36    helper1(w);
37    helper2(w,z);
38
39    if (x) return -1; // If condition
40
41    while (x) {} // while condition
42    while (y) {}
43    while (1) {} // while condition
44    while (y!=z) {}
45
46    k = a; // invalid use of a
47    x = k[4]; // Invalid use of k
48
49    return z; // return mismatch
50 }
```

The semantic analyzer produces the following output for this example input:

```
sema: 3:0 : redefinition of x
sema: 10:3 : redefinition of w
sema: 13:3 : redefinition of a
sema: 21:3 : size cannot be negative for array b
sema: 23:3 : use of undefined identifier ab
sema: 24:7 : use of undefined identifier c
sema: 24:3 : type mismatch
sema: 26:3 : use of undefined identifier fcn2
sema: 29:3 : type mismatch
sema: 30:3 : type mismatch
sema: 33:3 : type mismatch
sema: 36:3 : no matching definition for helper1
sema: 37:3 : no matching definition for helper2
sema: 39:3 : invalid condition in if statement
sema: 41:3 : invalid condition in while statement
sema: 43:3 : invalid condition in while statement
sema: 46:7 : invalid use of identifier a
sema: 47:7 : invalid use of identifier k
sema: 49:3 : mismatched return statement
```

4 Procedure

Create directory called `lab3` in your `ece467` directory. Make sure that the permissions of this new directory are such that it is readable by none other than you. Download the `zip` file containing the assignment release files and place it in this `lab2` directory. Unzip the file, which will create the assignment files in the directory. You can move the zip file out of the directory or remove it after this step.

Also ensure that you are running a bash shell (see Lab 1 handout for details on how to run the bash shell).

The following files are included in the release. Please note that you are allowed to change the contents of some of these files, as indicated below. However, **you must not change the names** of any of the files. Further, you are **not allowed to add new files**. Any code that you write must be in one of the files listed below and that you may change,

- `smallC.g4`. This is the **updated** grammar file for `smallC`, which you will extend to generate the AST and symbol tables. You must use this file, and not use your solution for the previous lab. The `smallC` language has been slightly extended to use `false` and `true` for boolean constants.

- `A3Sema.cpp`. This file contains the `main` function of the semantic analyzer. You will need to modify this file to build the AST, and then invoke your semantic analyzer function.

- `ASTNode.h` and `ASTNodes.cpp`. The first of these two files contains the definition of the AST classes. The second is where you write the implementation of these classes. Note that you may change the definitions (and thus implementations) of these classes as you wish.

- `ASTVisitorBase.h` and `ASTVisitorBase.cpp`. These files contain the definition/implementation of base classes of AST visitors. There is no need for you to modify these class, although you may. You must not modify these files.

- `ASTPrinter.h` and `ASTPrinter.cpp`. These files contain the definition/implementation of a visitor that "prints" the nodes of the AST, as described earlier. They serve as both and example of how to extend the `ASTVisitorBase` class and as utility for printing the AST. They are not needed for the assignment solution. Thus, you must not modify these files.

- `SymTable.h` and `SymTable.cpp`. These files contains respectively the definition and implementation of the symbol table classes. You will write the implementation of these classes in the `SymTable.cpp` file.

- `SemanticAnalyzer.h` and `SemanticAnalyzer.cpp`. These files contains respectively the definition and implementation of the semantic analyzer. You will write the implementation of the analyzer in the `SemanticAnalyzer.cpp` file.

- `Makefile`. This file is used by the make utility to build the ANTLR4-generated code, compile and link this code with the remaining files. You must not modify this file . To make the executable, simply type: `make` (followed by `make depend` the first time you run `make`). The resulting executable is called `A3Sema` and you will run this executable to test your semantic analyzer. You must not modify this file.

The `exercise` command is helpful in testing your parser. Run exercise as follows:

```
~ece467i/public/exercise 3 A3Sema
```

The `exercise` command will let you know if your code has errors by providing it with several test cases. Please note that some of the `exercise` test cases will be used by the autotester during marking of your assignment. However, not all the autotester test cases are used by `exercise`. Thus, you should create additional test cases yourself.

There is also a reference executable `A3Sema-reference`, the location of which is released with the assignment. This executable will run only on the UG machines and you should use it when in doubt about any of the specifications of the assignment. Please note that the reference executable does not check for extreme corner cases of input and that such cases will not be used in marking the assignment.

5 Deliverables

The autotester will be used to check the error messages produced by the semantic analyzer. to do so, it will copy all the files listed above and build your solution before testing it. Thus, to submit your files, *make sure you are in the directory that contains the files, i.e., your* `ece467/lab3` *directory*, and type the command:

```
~ece467i/public/submit 3
```

The above command will submit the following files: `ASTNodes.h`, `ASTNodes.cpp`, `SymTable.h`, `SymTabke.cpp`, `A3Sema.cpp`, tt smallC.g4, `SemanticAnalyzer.h`, and `SemanticAnalyzer.cpp`. No other files will be submitted.