Assignment 3
Semantic analyzer (CPSC 411)
Due date November 30

1. Description:

In this assignment, your task is to create a semantic analyzer for the C-programming language building on the scanner and parser that you developed in assignments 1 and 2. The semantic analyzer should take the output from the parser and determine if the semantics are valid. In the case that the input is semantically valid your program should print the AST, with some additional information on the nodes, including types for expressions, and unique identifiers for IDs. Otherwise your compiler should indicate failure and halt. Create a set of complete tests to exercise all of the tricky corner cases.

2. Requirements:

You should pay attention to the following requirements while implementing your parser:

I. You can use any language for this assignment as long as it can be executed on the undergraduate Linux servers.
II. The assignment submission must include:
    a. Instructions on how to run the project and how the output can be interpreted.
    b. Explanation of what the test cases are and what they are testing for.
III. Your assignment should build off of the parser implementation that you made for Assignment 2.
IV. When printing the AST do so in a manner that makes it clear the children of any given AST node, and their order.
V. Print the type of all expression, and declaration AST nodes, and when encountering an ID, make it clear what variable/function it refers to.

3. Semantic Checks to Perform:

Checks on names (functions, and variables):
Ensure that names are always declared before their use. When matching a name to its declaration, start looking from the closest scope, going out until the global scope is reached. Names should not be redeclared in the same scope, however they can be redeclared in separate scopes. Parameters of a function have the same scope as the compound statement of that function. The last declaration in the global scope should be a function named "main" of type void, with void parameters.

Declaration before use:
```
void foo ( void ) {
     x = 1;
}
int x;
void main ( void ) { }
```
output:
```
Error: unknown name "x" at or near line 2
```

Redeclaration in different scopes:
```
void foo ( void ) { }
void main ( void ) {
     int foo;
     foo = 0;
}
```
output:
This should produce a valid ast, there should be indication that the foo in "`foo = 0;`" refers to the foo inside the scope of main's compound statement, rather than the foo in the global scope, which is a function.

Redeclaration in same scope:
```
void main ( void ) {
     int x;
     int x;
}
```
output:
```
Error: "x" redefined at or near line 3
```

Parameter Scope:

```
void foo ( int x ) {
     x = 1;
}
void main ( void ) { }
```

output:

This should produce a valid ast, it should be clear that the x in the assignment statement refers to the same x in the parameters of foo.

Parameter Scope Redeclaration:

```
void foo ( int x ) {
     int x;
}
void main ( void ) { }
```

output:

```
Error: “x” redefined at or near line 2
```

You could also produce a redefinition error by having two parameters with the same name.

Main Function Error:

```
void main ( void ) {
}
int x;
```

output:

```
Error: last global declaration is not main function
```

Type Checking:

There are several different types that you will need to take into account, namely: int, void, and int array. Each declared name will have an associated type. The type of a variable will be either int or int array. The type for a function will be either void, or int. For functions, you will also need to keep track of the types of its parameters as a list.

All binary expressions are of type int, you will need to check that their left and right child are also of type int. Assignment statements are unique in that their left child should specifically be a name corresponding to an int variable, or a name corresponding to an int array variable which is indexed into and the right child of an assignment should be of type int. Type checks should be done on the arguments of a function call to ensure that they match the function’s declaration.

The usage of names must match their type. Thus the function call syntax can only be used on function names, and the resulting type is based on the function being called (Is either int or void), the array indexing syntax can only be used on array variable names and results in type int.

Type Checking Expressions:

```
void main ( void ) {
     main() + 1;
}
```

output:

```
Error: Operand mismatch for ‘+’ at or near line 2
```

Here we get an error because the plus expects both its children to be of type int, but the call to main is of type void

Incorrect use of function call syntax:

```
void main ( void ) {
     int x;
     x();
}
```

output:

```
Error: Calling something that isn’t a function at or near line 3
```

Incorrect use of array index syntax:

```
void main ( void ) {
     int x;
     x[10];
}
```

output:

```
Error: Indexing something that isn’t an array at or near line 3
```

Incorrect function call parameters:

```
void foo ( int bar[] ) { }
void main ( void ) {
     int x;
     foo(x);
}
```

output:

```
Error: call of ‘foo’ does not match type/number of arguments
from function declaration at or near line 4
```

Return Statement:

The type of return statement used in a function must match the function's type. In void functions return statements must not have an expression, while return statements in int functions must have an expression.

Return with expression in void function:
```
void main ( void ) { return 1; }
```
output:
```
Error: returning value in void function at or near line 1
```

Empty return in int function:
```
int foo ( void ) { return; }
void main ( void ) { }
```
output:
```
Error: empty return statement in int function at or near line 1
```