Homework 5

1. Consider the following pseudocode:

```
1. procedure main()
2.    x : real := 5.0
3.    y : real := 3.2

4.        procedure middle()
5.            y: real := x

6.            procedure inner()
7.                print x,y

8.            x : real := 11.7

9.            --- body of middle
10:          inner()
11.          print x,y

12.  --- body of main
13.  print x, y
14.  middle()
15.  print x,y
```

   Suppose this was code for a language with the declaration order rules of C, but with nested subroutines allowed: that is, names must be declared before use, and the scope of a name extends from its declaration through the end of the block. At east print statement, indicate which declarations of a and b are in the referencing environment. What does the program print, or does the compiler identify static semantic errors? Repeat the exercise for the declaration order rules of C#, where names must be declared before use but the scope of a name is the entire block in which it is declared, and for Modula-3's rules, where names can be declared in any order, and their scope is the entire block in which they are declared.

2. Consider the following pseudocode, assuming nested subroutines and static scope:

```
procedure main()
  g : integer

  procedure B(a : integer)
    x : integer

    procedure A(n : integer)
      g := n

    procedure R(m : integer)
      print x
      x := x / 2
      if x > 1
        R(m+1)
      else
        A(m)

    -- body of B
    x := a * a
    R(1)

  --body of main
  B(2)
  print g
```

   (a) What does the program print?

   (b) Show the frames on the stack when A has just ben called. For each frame, show static and dynamic links.

   (c) Explain how A will find g.

3. Consider the following pseudocode. Assume that print works like the python function, so that `print a, b` will output the two integers separated by a space, with a newline at the end (so that each print command will go on a different line.

```
a : integer
b : integer

procedure first(n : integer)
   b : integer
   a := n
   b := n + 2

procedure second(n : integer)
  a : integer
  a := n
  b := n - 2

procedure big_function
  a : integer
  a := 0
  first(1)
  print a, b
  second(17)
  print a,b

a := 0
b := 0
first(11)
print a,b
second(6)
print a,b
big_function()
print a,b
```

   (a) What does this program print if the language uses static scoping?

   (b) What about dynamic scoping?

4. Consider the following pseudocode:

```
x : integer := 1
y : integer := 2

procedure mult()
  x := x * y

procedure second(P : procedure)
  x : integer := 3
  P()

procedure first()
  y : integer := 4
  second(mult)

first()
print x
```

   (a) What does the program print if the language has static scoping?

   (b) What does it print if the language uses dynamic scoping with deep binding?

   (c) What does it print if the language uses dynamic scope with shallow binding?

5. Consider the expression in C: `(x/y > 0)&&(y/x > 0)`. What is the result when x is 0? What about when y is 0? Would it make sense to design a language where this is guaranteed to be false when either x or y was 0, or do you think this behavior in C is acceptable?

6. Consider the following snippet of code:

```
type X = array [1..10] of integer
       Y = X
A : X
B : Y
C : array [1..10] of integer
```

   Which of the variables A, B, and C will the compiler consider to have compatible types under structural equivalence, strict name equivalence, and loose name equivalence?

7. Consider the following C++ code fragment:

```
#include <list>
using std::list;

class fizz { //code here}
class buzz : public fizz { //code here }

static void print_all(list<fizz*> &L) { //code here}

int main() {
  list<fizz*> fizzlist;
  list<buzz*> buzzlist;
  //code to add things to lists
  print_all(fizzlist); //works find
  print_all(buzzlist); //static semantic error
```

Why won't the compiler allow the second call? Give an example where this could lead to bad behavior, to explain why it is not allowed.