Homework 5

1. Consider the following pseudocode:

```
procedure main()
  x : int := 3
  y : int := 2

    procedure middle()
      x: int := y + 2
      print x,y

      procedure inner()
        print x,y
        y : int := 9

      --- body of middle
     inner()
     print x,y

 --- body of main
 print x, y
 middle()
 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 each print statement, indicate which declarations of x and y 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. (Note that Modula will reorder if necessary, so it will use the correct value even if initially declared later in the scope.)

2. Consider the following pseudocode, assuming nested subroutines are allowed and static scope rules are in place:

```
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 + 1
    R(10)

  --body of main
  B(4)
  print g
```

   (a) What does the program print?

   (b) Show the frames on the stack when A has just been called. For each frame, show both the static and dynamic links (and please clearly indicate which is which!).

   (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 -1

procedure second(n : integer)
  a : integer
  a := n
  b := n + 1

procedure big_function
  b : integer
  b := 0
  first(3)
  print a, b
  second(15)
  print a,b

a := 0
b := 0
first(12)
print a,b
second(7)
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 := 3
y : integer := 4

procedure compute()
  x := x * x - y

procedure second(P : procedure)
  y : integer := 5
  P()

procedure first()
  y : integer := 6
  second(compute)

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 following snippet of code:

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

   Which of the variables X, Y, and Z will the compiler consider to have compatible types under structural equivalence, strict name equivalence, and loose name equivalence?

6. As discussed in Section 6.4.2, languages vary in how they handle the situation in which the controlling expression of a case statement does not appear among the labels of the arms. C and Fortran say the statement has no effect. Pascal and Modula say that it results in a dynamic semantic error. Ada says the label MUST cover all possible values for the type of the expression, and so the question of a missing value cannot come up at run time. What are the tradeoffs among these alternatives, and which do you prefer (and why)?

7. Consider the following C++ code fragment:

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

class foo { //code here}
class bar : public foo { //code here }

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

int main() {
  list<foo*> foolist;
  list<bar*> barlist;
  //code to add things to lists
  print_all(foolist); //works find
  print_all(barlist); //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.