COMP 141: Basic Semantics — Part 1

Instructions: In this exercise, we are going to static and dynamic scoping in C.

(1) Consider the following program.

```
#include <stdio.h>

int a = 3;                              //p1
int b = 5;                              //p2


int f (int b){                          //p3
        return a + b;
}                                       //p4


int main () {                           //p5
        printf("1.%d\n", b);
        int b = a;                      //p6
        int a = 2 * b;                  //p7
        {
                printf("2.%d\n", b);
                int b = 1;              //p8
                printf("3.%d\n", b);
        }                               //p9
        printf("4.%d\n", b);
        b = f (b);
        printf("5.%d\n", b);
        return 0;
}
```

(a) For each line marked with `//p#`, define the symbol table if **static scoping** is used[1].
For example, in `p1`, `p2` and `p3` the tables are as follows[2].

```
p1.
a : int global

p2.
a : int global
b : int global

p3:
a : int global
b : int local to f :: int global
f : int -> int function
```

(b) According to the static scoping what would be the output of the program (i.e., the prints in standard output)?

[1] In C, scope analysis *is* static

[2] You can use your way of specifying labeled stacks. No need to follow the syntax used here. Note that for each name a stack should be defined, and items within each stack must be delimited from the adjacent ones.

(2) Consider the same program with different line marks. The line marks follow the order in which program is executed.

```
#include <stdio.h>

int a = 3;
int b = 5;


int f (int b){                                  //p6
        int c = a + b;                          //p7
        return c;
}                                               //p8


int main () {                                   //p1
        printf("1.%d\n", b);
        int b = a;                              //p2
        int a = 2 * b;                          //p3
        {
                printf("2.%d\n", b);
                int b = 1;                      //p4
                printf("3.%d\n", b);
        }                                       //p5
        printf("4.%d\n", b);
        b = f (b);
        printf("5.%d\n", b);
        return 0;
}                                               //p9
```

(a) For each line marked with `//p#`, define the symbol table if **dynamic scoping** is used[3].
For example, for `p1` and `p2`,

```
p1:
a :: int = 3 global
b :: int = 5 global
f :: int -> int function
main :: int function


p2:
a :: int = 3 global
b :: int = 3 local to main :: int = 5 global
f :: int -> int function
main :: int function
```

(b) According to the dynamic scoping, what would be the output of the program (i.e., the prints in standard output)?

(3) Consider the following program.

```
(1)   #include <stdio.h>

(2)   int a = 53;
```

[3]We are just *assuming* that the scope analysis is done dynamically for C.

```
(3)   int b = 120;
(4)   int c = 36;


(5)   int g (int a) {
(6)           printf("4.%d\n", b);
(7)           int c = b;
(8)           int b = a + c;
(9)           printf("5.%d\n", b);
(10)          return b;
(11)  }


(12)  int f (int a){
(13)          int b = a + 5;
(14)          printf("3.%d\n", b);
(15)          return g (b);
(16)  }

(17)  int main () {
(18)          printf("1.%d\n", b);
(19)          int b = a;
(20)          printf("2.%d\n", b);
(21)          int c = f (b);
(22)          printf("6.%d\n", c);
(23)          return 0;
(24)  }
```

(a) Define the symbol table if **static scoping** is used, after each of the following lines.
Lines: 2, 3, 4, 5, 7, 8, 11, 12, 13, 16, 17, 19, 21 and 24.

(b) According to the static scoping what would be the output of the program (i.e., the prints in standard output)?

(4) Consider the same program, given in the previous question. Define the symbol table if **dynamic scoping** is used, after each of the following lines.

Lines: 17, 19, 12, 13, 5, 7, 8, 11, 16, 21, and 24.

Note that the sequence of the line above follow the order of program execution.