CSE 340 Spring 2018

HOMEWORK 6 - Solution

Problem 1. Consider the following code in C syntax:

```
int b;
int temp = 3;

int set_b(int i, int a)
{
    i = i + 1;
    b = a;
}

int p(int x, int S)
{
    for (x = 0; x < 3; x++) {
        temp = temp + S;
        S = temp;
    }
    return temp + b;
}

main()
{
1:     int a[3][3];
2:     int b;
3:     int r;
4:     int temp = 4;
5:     int i = 1;
6:     set_b(i, a[i][i]);
7:     a[0][0] = 100; a[0][1] = 10;  a[0][2] = 10;
8:     a[1][0] = 1;   a[1][1] = 10;  a[1][2] = 100;
9:     a[2][0] = 100; a[2][1] = 100; a[2][2] = 100;
10:    b = 0;
11:    r = p(i, a[i][b]);
12:    printf("%d %d %d %d\n", i, temp, b, r);
13:    printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);
14:    printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);
15:    printf("%d %d %d\n", a[2][0], a[2][1], a[2][2]);
}
```

- What is the output of the program if parameters are *passed by value*?

Answer:

Line 1-5: set local temp to 4 and i to 1, the rest of the variables are not initialized. Note that global temp is initialized to 3.

Line 6: Calling set_b(i, a[i][i]) will call set_b(1,0), where 0 is the initial value of a[1][1]. In the call the global b is set to 0. Incrementing i in the body of the function does not affect the argument i.

Lines 7-9: initializes the array a.

Line 11: calls p(i,a[i][b]) = p(1,a[1][0]) = p(1,1). In the call, S is initialized to 1 (second argument) and temp to 3. Then we execute temp = 3+ 1 = 4 and S = 4. Then temp = 4 + 4 = 8 and S = 8. Then temp = 8+8 = 16 and S = 16. The value returned is temp + b where b is the global b whose value is 0 (see above). So, 16 + 0 = 16 is returned. r is finally assigned the value 16.

Lines 12-15. First line prints	1	4	0	16
The three following lines print:	100	10	10	
	1	10	100	
	100	100	100	

- What is the output of the program if parameters are *passed by reference*?

Answer: If parameters are passed by reference, the execution proceeds as follows:

Lines 1-5: as before.

Line 6: Calling set_b(i, a[i][i]) will call set_b(i,a[1][1]). In the call i is incremented by 1. This changes the value in the location associated with the variable i in main from 1 to 2. The global b is set to 0, and the initial value of a[1][1] is 0.

Lines 7-9: initializes the array a.

Line 11: calls p(i,a[i][b]) = p(i,a[2][0]). This will associate the memory of i with x and the memory of a[2][0] with S. In the call, the initial value of S is the value in the memory of a[2][0] at the time of the call, namely 100. The initial value of temp is 3 (global temp). Then we execute temp = 3+ 100 = 103 and S = 103. Then temp = 103 + 103 = 206 and S = 206. Then temp = 206+206 = 412 and S = 412. The value returned is temp + b where b is the global b whose value is 0 (see above). So, 412 is returned. r is finally assigned the value 412. When the function returns, the value of x is 3 (last increment occurs when x = 2 so x gets the value 3 and the loop body is not executed). So, at the exit of the function, the value of i is 3 and the value of a[2][0] is 412.

Lines 12-15 produce the following output:

```
3    4    0   412
100  10   10
1    10   100
412  100  100
```

- What is the output of the program if parameters are *passed by name*?

Answer: To simplify the call by name, let us change all variable names so every name is unique. This will not affect the result, but it makes call by name easier.

```
int b_global;
int temp_global = 3;

int set_b(int i, int a)
{
    i = i + 1;
    b_global = a;
}

int p(int x, int S)
{
    for (x = 0; x < 3; x++) {
        temp_global = temp_global + S;
        S = temp_global;
    }
    return temp_global + b_global;
}

main()
{
    int a[3][3];
    int b_main;
    int r;
    int temp_main = 4;
    int i_main = 1;
    set_b(i_main, a[i_main][ i_main]);
    a[0][0] = 100; a[0][1] = 10;  a[0][2] = 10;
    a[1][0] = 1;   a[1][1] = 10;  a[1][2] = 100;
    a[2][0] = 100; a[2][1] = 100; a[2][2] = 100;
    b_main = 0;
    r = p(i_main, a[i_main][ b_main]);
    printf("%d %d %d %d\n", i_main, temp_main, b_main, r);
    printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);
    printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);
    printf("%d %d %d\n", a[2][0], a[2][1], a[2][2]);
}
```

Note that I gave each function a color and I colored the local names of each function with the color of the function. I colored the global names in red.

Now, it is easy to do the textual replacement for call by name.

```
int b_global;
int temp_global = 3;

int set_b(int i, int a)
{
    i = i + 1;
    b_global = a;
}

int p(int x, int S)
{
    for (x = 0; x < 3; x++) {
        temp_global = temp_global + S;
        S = temp_global;
    }
    return temp_global + b_global;
}

main()
{
    int a[3][3];
    int b_main;
    int r;
    int temp_main = 4;
    int i_main = 1;

    // ----- First call ---------
    // set_b(i_main, a[i_main][ i_main]);
    // i -> i_main
    // a -> a[i_main][ i_main];
    // in
    //         i = i + 1;
    //         b_global = a;
    //---------------------------
             i_main = i_main + 1;
             b_global = a[i_main][ i_main];

    a[0][0] = 100; a[0][1] = 10;  a[0][2] = 10;
    a[1][0] = 1;   a[1][1] = 10;  a[1][2] = 100;
    a[2][0] = 100; a[2][1] = 100; a[2][2] = 100;
    b_main = 0;

    // continued on next page!
```

```
    // ----- Second call --------
    // r = p(i_main, a[i_main][b_main]);
    // x -> i_main
    // S -> a[i_main][b_main]
    // in
    //         for (x = 0; x < 3; x++) {
    //                temp_global = temp_global + S;
    //                S = temp_global;
    //         }
    //         return temp_global + b_global;
    //----------------------------
            for (i_main = 0; i_main < 3; i_main ++) {
                   temp_global = temp_global + a[i_main][b_main];
                   a[i_main][b_main] = temp_global;
            }
            r =  temp_global + b_global;

    printf("%d %d %d %d\n", i_main, temp_main, b_main, r);
    printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);
    printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);
    printf("%d %d %d\n", a[2][0], a[2][1], a[2][2]);
}
```

With the code modified, the output is straightforward to compute. We get:

```
3    4   0  204
103  10  10
104  10  100
204  100 100
```

Problem 2. Consider the following program written in Ada syntax with the execution stack shown on the side. The line numbers are used to refer to the code and are not part of the code.

```
01    procedure env is
02        x_env: integer;
03        y_env: integer;
04            procedure a is
05                x_a: integer;
06                    procedure b is
07                        x_b: integer;
08                            procedure c is
09                                x_c: integer;
10                            begin
11                                x_env = x_b + x_a;
12                                d;
13                            end c;
14                    begin
15                        x_b = x_a;
16                        c;
17                    end b;
18
19                    procedure d is
20                    begin
21                        env;
22                    end d;
23            begin
24                b;
25            end a;
26    begin
27      a;
28    end env;
```

```
          ________
env  |________| 1
 a   |________| 2
 b   |________| 3
 c   |________| 4
 d   |________| 5
env  |________| 6
 a   |________| 7
 b   |________| 8
 c   |________| 9
 d   |________| 10
env  |________| 11
 a   |________| 12
```

- Give the code that should be executed to set the access link for activation record 2

  Answer: the access link of activation record 2 points to activation record 1, therefore:

  `mem[sp + AL`$_{offset}$`] = fp;`

- Give the code that should be executed to set the access link for activation record 3

  Answer: the access link of activation record 3 points to activation record 2, therefore:

  `mem[sp + AL`$_{offset}$`] = fp;`

- Give the code that should be executed to set the access link for activation record 4

  Answer: the access link of activation record 4 points to activation record 3, therefore:

  ```
  mem[sp + AL_offset] = fp;
  ```

- Give the code that should be executed to set the access link for activation record 5

  Answer: the access link of activation record 5 points to activation record 2, therefore:

  ```
  address = fp;                          // fp of frame 4
  address = mem[address + AL_offset];    // fp of frame 3
  address = mem[address + AL_offset];    // fp of frame 2
  mem[sp + AL_offset] = address;         // AL of frame 5 points to
                                         // frame 2
  ```

- Give the low-level code (in terms of mem[ ]) for `x_b = x_a;` on line 15

  Answer: `x_b` is a local variable, therefore its address is

  ```
  x_b address = fp + x_b_offset
  ```

  `x_a` is defined in scope of procedure a, therefore its address is calculated as follows

  ```
  address = mem[fp + AL_offset]          // fp of a
  x_a address = address + x_a_offset     // offset x_a_offset within
                                         // frame of a
  ```

  The low-level code for `x_b = x_a;` is therefore the following

  ```
  x_b address = fp + x_b_offset
  address = mem[fp + AL_offset]
  x_a address = address + x_a_offset
  mem[x_b address] = mem[x_a address];
  ```

- Give the low-level code (in terms of mem[ ]) for `x_env = x_b + y_env;` on line 11

  Answer: There is a discrepancy between the question and the code in line 11. The code on line 11 is `x_env = x_b + x_a;` and the question asks for `x_env = x_b + y_env;`

  We give both answers

1. **Answer for** `x_env = x_b + x_a;`

```
x_b      is one defining environment higher than c
x_a      is two defining environments higher than c
x_env    is three defining environments higher than c
```

```
address        = fp                           // fp of c
address        = mem[address+AL_offset]       // fp of b
x_b address    = mem[address + x_b_offset]    // offset x_b_offset within
                                              // frame of b
address        = mem[address+AL_offset]       // fp of a
x_a address    = mem[address + x_a_offset]    // offset x_a_offset within
                                              // frame of a
address        = mem[address+AL_offset]       // fp of env
x_env address  = mem[address + x_env_offset]  // offset x_env_offset
                                              // within frame of env

mem[x_env address] = mem[x_b address] + mem[x_a address]
```

2. **Answer for** `x_env = x_b + y_env;`

```
x_b      is one defining environment higher than c
x_env    is three defining environments higher than c
y_env    is three defining environments higher than c
```

```
address        = fp                           // fp of c
address        = mem[address+AL_offset]       // fp of b
x_b address    = mem[address + x_b_offset]    // offset x_b_offset within
                                              // frame of b
address        = mem[address+AL_offset]       // fp of a
address        = mem[address+AL_offset]       // fp of env
x_env address  = mem[address + x_env_offset]  // offset x_env_offset
                                              // within frame of env
y_env address  = mem[address + y_env_offset]  // offset y_env_offset
                                              // within frame of env

mem[x_env address] = mem[x_b address] + mem[y_env address]
```