Second take-home assignment

Edda Steinunn Skúli Arnarsson

Due: February 20th 2018

Problem 1. *Consider the following program:*

```
1  {
2      int x = 1;
3      int p(int z) {
4          x = x * 3;
5          z++;
6          return (z + x);
7      }
8      write( p(x) + p(x)*5 );
9  }
```

What does the program print when addition is evaluated (i) left-to-right, (ii) right-to-left, when the parameter z is (a) call-by-value, (b) call-by-reference? Please answer for all 4 combinations of these features. Explain your answers by describing how the program is executed.

Solution. Assuming left-to-right evaluation, arithmetic expressions on the form $E_1 \bullet E_2$ where $\bullet$ is an operator are evaluated as follows: first E_1 is evaluated as a whole as the leftmost expression, then E_2 is evaluated as a whole, and lastly the operator is applied. Assuming right-to-left evaluation however, E_2 is first evaluated as a whole as the rightmost expression, then E_1 and lastly the operator is applied. In below cases, since the output writes out p(x) + p(x)*5, E_1 is represented as p(x) and E_2 as p(x)*5.

a.i. call-by-value, left-to-right evaluation: as left-to-right expression goes, we first evaluate E_1, namely in this case ***p(x)***. We enter the function p with the global value x as the parameter z which is "copied" for usage within the function since we assume z called-by-value. x is initially 1 when p is entered which is what is copied into the z-parameter. the global x is changed to x=1*3=3 then the copy of x, the z, is incremented by one, meaning z=2. Then 3+2=5 is returned for p(x) in the write statement, which makes up the expression E_1.

Secondly, we evaluate E_2, namely ***p(x)*5***. Now, no specific instructions in the problem are present whether left-to-right or right-to-left evaluation is assumed for multiplication, but that does not matter in this instance since $E_1 * E_2$ is the same as $E_2 * E_1$. For simplicity let's therefore assume left-to-right evaluation and go into evaluating p(x) first; we enter the function p with the global value x as the parameter z which is "copied" for usage within the function since we assume z called-by-value. x is initially 3 when p is entered (because it had been incremented to 3 in our previous evaluation of E_1) which is what is copied into the z-parameter. The global x is changed to x=3*3=9 then the copy of x, the z, is incremented by one, meaning z=4. Then 9+4=13 is returned for p(x) in the write statement. Then we evaluate E_2 as 5 and lastly we apply the operator, which evaluates the expression E_2 as whole to 65.

Third, we apply the operator to the expressions. 5+65=70. **Therefore, the output that the program yields with z called-by-value and assuming left-to-right evaluation is 70** □

a.ii. call-by-value, right-to-left evaluation: as right-to-left expression goes, we first evaluate E_2, namely in this case ***p(x)*5***. Again, no specific instructions in the problem so again for simplicity we assume left-to-right evaluation and go into evaluating p(x) first; we enter the function p with the global value x as the parameter z which is "copied" for usage within the function since we assume z called-by-value. x is initially 1 when p is entered which is what is copied into the z-parameter. The global x is changed to x=1*3=3 then the copy of x, the z, is incremented by one, meaning z=2. Then 3+2=5 is returned for p(x) in the write statement. Then we evaluate E_2 as 5 and lastly we apply the operator, which evaluates the expression E_2 as whole to 25.

Secondly, we evaluate E_1. We enter the function p with the global value x as the parameter z which is "copied" for usage within the function since we assume z called-by-value. x is initially 3 when p is entered (since our evaluation of E_2 previously set our global x to 3) which is what is copied into the z-parameter. the global x is changed to x=3*3=9 then the copy of x, the z, is incremented by one, meaning z=4. Then 9+4=13 is returned for p(x) in the write statement, which makes up the expression E_1.

Third, we apply the operator to the expressions. 13+25=38. **Therefore, the output that the program yields with z called-by-value and assuming right-to-left evaluation is 38** □

b.i. call-by-reference, left-to-right evaluation as left-to-right expression goes, we first evaluate E_1, namely in this case ***p(x)***. We enter the function p with the global value x as the parameter z which is an alias for x within the function since we assume z called-by-reference. x is initially 1. The global x is changed to x=1*3=3 and then via it's alias z it is is incremented by one, meaning z=4 and x=4. Then 4+4=8 is returned for p(x) in the write statement, which makes up the expression E_1.

Secondly, we evaluate E_2, namely ***p(x)*5***. Again, no specific instructions in the problem

are present whether left-to-right or right-to-left evaluation is assumed for multiplication so again for simplicity let's therefore assume left-to-right evaluation and go into evaluating p(x) first; we enter the function p with the global value x as the parameter z which is an alias for x within the function since we assume z called-by-reference. x is initially 4. The global x is changed to x=4*3=12 and then via it's alias z it is is incremented by one, meaning z=13 and x=13. Then 13+13=26 is returned for p(x) in the write statement. Then we apply the multiplication operator, which evaluates the expression E_2 as whole to 26*5=130.

Third, we apply the operator to the expressions. 8+130=138. **Therefore, the output that the program yields with z called-by-reference and assuming left-to-right evaluation is 138** □

b.ii. call-by-reference, right-to-left evaluation as right-to-left expression goes, we first evaluate E_2, namely in this case ***p(x)*5***. Again, no specific instructions in the problem so again for simplicity we assume left-to-right evaluation and go into evaluating p(x) first; we enter the function p with the global value x as the parameter z which is an alias for x within the function since we assume z called-by-reference. x is initially 1. The global x is changed to x=1*3=3 and then via it's alias z it is is incremented by one, meaning z=4 and x=4. Then 4+4=8 is returned for p(x) in the write statement. Then we apply the multiplication operator, which evaluates the expression E_2 as whole to 8*5=40.

Secondly, we evaluate E_1, namely ***p(x)***. We enter the function p with the global value x as the parameter z which is an alias for x within the function since we assume z called-by-reference. x is initially 4 (as it had been incremented in the in our previous evaluation of E_2. The global x is changed to x=4*3=12 and then via it's alias z it is is incremented by one, meaning z=13 and x=13. Then 13+13=26 is returned for p(x) which makes up the expression E_1.

Third, we apply the operator to the expressions. 40+26=66. **Therefore, the output that the program yields with z called-by-reference and assuming right-to-left evaluation is 66** □

Problem 2. *Consider the following program:*

```
1   {
2       int x = 1;
3
4       void p() {
5           x = x * 5;
6           write(x);
7       }
8
9       void q() {
10          int x = 10;
11          x++;
12          write(x);
13      }
14
15      {
16          int x = 8;
17          p();
18          write(x);
19          q();
20          write(x);
21      }
22
23      write(x);
24  }
```

What does it print if the language uses (i) static scoping, (ii) dynamic scoping? Explain your answers by describing how the program is executed.

Solution.
i) Static scoping yields the following output:

5 8 11 8 5

Let's explore this program to better understand it's behaviour. First we explore it's nature; in order of appearance in the program we have have a declaration of a global variable x, two function declarations p and q, then a block of executable code (let's call this block **B** to easily reference this area when explaining), and then a single line of code executed in the global scope (a write statement). First, the code in block B is executed, then the single line of code in the global scope. Let's step through the behaviour:

- First, we observe that the declaration int x = 8 is executed in block B, thus making a local-scoped x variable, visible in the scope of block B. Second, p() is called from block B. p() then executes the code x=x*5. As static scoping goes, once a variable is not found in the local scope, the outer scope is searched. Therefore since there's no variable x in the scope of p(), the x that p() uses is the global scope x which has the value of 1 because when x is not found in p()'s scope, the outer scope of p() is

searched (the global scope) which yields our global x variable. The global x variable is therefore changed to 1*5=5. Then a write statement is executed on our global x which prints **5 as the first value of the output**.

- Once p() has returned, a write statement is executed on variable x. Since static scoping searches local scope first, we use the local scoped x which has the value of 8. Therefore **the second output value is 8**.
- After this, q() is called from block B. q() begins by declaring it's very own local scope variable x with the value 10. Of course with static scoping, the local x is used with the value 10 - this value is incremented by one and printed which explains the **third output value of 11**.
- After q() has returned, a write statement is executed from block B on x. Again we use the local scoped x which still has the value of 8. Therefore **the fourth output value is 8**.
- now block B is exited and we arrive in the global scope of the program. Now a write statement is executed on x. Of course we use our local x which in this case is actually our global x since we're in the global scope. However note that since the p() function we called before in block B changed the value of our global scope x, our global scope x is now 5. This explains **the fifth output value of 5**.□

ii) Dynamic scoping yields the output:

40 40 11 40 1

Let's step through the behaviour to explore the nature of the output values:

- First, we observe that the declaration int x = 8 is executed in block B, thus making a local-scoped x variable, visible in the scope of block B. Second, p() is called from block B. p() then executes the code x=x*5. As dynamic scoping goes, once a variable is not found in the local scope, the caller scope is searched. Therefore since there's no variable x in the scope of p(), the x that p() uses is the x declared in block B which has the value of 8. Block b's x variable is therefore changed to 8*5=40. Then a write statement is executed on our block B x which prints **40 as the first value of the output**.
- Once p() has returned, a write statement is executed on variable x. Since dynamic scoping searches local scope first, we use the local scoped x. Note however that executing p() changed block B's local x variable value from 8 to 40. Therefore **the second output value is 40**.

- After this, q() is called from block B. q() begins by declaring it's very own local scope variable x with the value 10. Of course with dynamic scoping as well as static scoping, the local x is used with the value 10 - this value is incremented by one and printed which explains the **third output value of 11**.
- After q() has returned, a write statement is executed from block B on x. Again we use the local scoped x which still has the value of 40. Therefore **the fourth output value is 40**.
- now block B is exited and we arrive in the global scope of the program. Now a write statement is executed on x. Now we use our local x since one is defined, which in this case is actually our global x since we're in the global scope. The global scope x has not changed in this case during the run time of the program, so our global scope x remains 1. This explains **the fifth output value of 1**.□

Problem 3. *Consider the following program:*

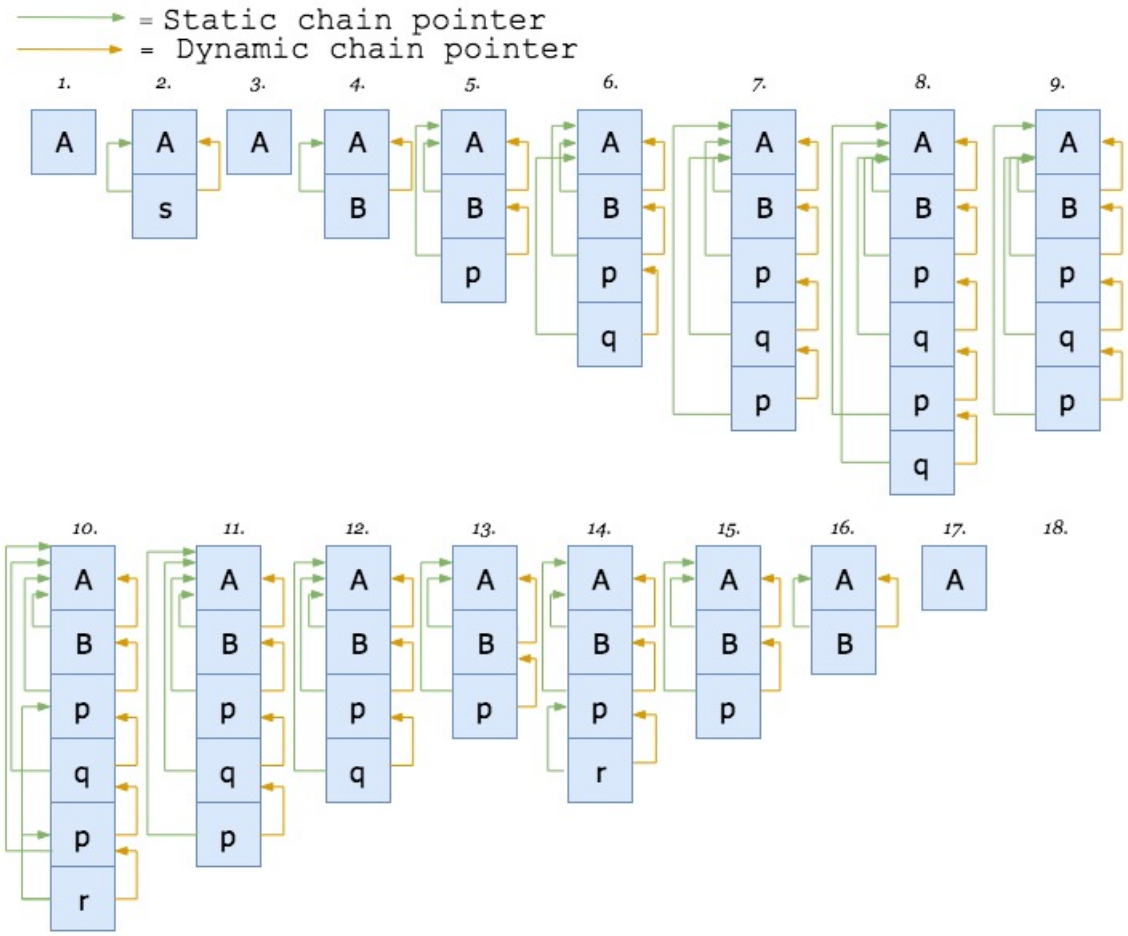
```
1   A : {
2       ...
3
4       void p() {
5           ...
6           void r() {
7               ...
8           }
9           ...
10          q();
11          ...
12          r();
13          ...
14      }
15
16      void q() {
17          ...
18          if (...) then p();
19          ...
20      }
21
22      void s() {
23          ...
24      }
25
26      s();
27
28      ...
29
30      B : {
31          ...
32          p();
33          ...
34      }
35
36      ...
37  }
```

Suppose we enter the blocks in this order:

enter A; enter s; enter B; enter p; enter q; enter p; enter q; enter r; enter r

Complete above sequence with information about when each block is exited. Draw the frame stack state for each of the 18 points of execution time, including the dynamic and static chain pointers. Explain by describing how the program is executed.

Solution. Figure below demonstrates the frame stack state for each of the 18 points of the execution time including the dynamic- and state chain pointers:



Note that in cases where frames in stack that have no dynamic- or state chain pointer drawn from them, these pointers point to null/undefined. Below is a further explanation for the figure by how the program is executed:

1. **A is pushed on stack**: We enter the block A. both pointers are null/undefined
2. **s is pushed on stack**: We enter the function s() from within A.
3. **s is popped off stack**: s() has no other function calls nor blocks to enter so we exit the function s() already.
4. **B is pushed on stack**: After running s(), the block A resumes executing it's code and we enter the block B from within A.
5. **p is pushed on stack**: We enter the function p() from within B.
6. **q is pushed on stack**: We enter the function q() from within the function p()

7. **p is pushed on stack**: We enter the function p() from within the function q() because in this call for q(), the if-statement is true, resulting in a recursive call to p() from within q(), resulting in two instances of the function p() on the stack

8. **q is pushed on stack**: We enter the function q() from within the the second instance of the function p(), another instance of q() is therefore present in the stack frame.

9. **q is popped off stack**: The sequence given of blocks in order implicitly gives us that the if-statement we executed in the second instance of q() was false (that is, by noting that r() is entered right after the second time q() is entered. q() is therefore exited and popped off the stack without any further actions from q(), thus terminating the recursion.

10. **r is pushed on stack**: since q() is finally exited, the second instance of p() on the stack continues after executing q() and goes right into running function r().

11. **r is popped off stack**: r() has no further function calls nor blocks to enter so the r() function is exited.

12. **p is popped off stack**: now once (the second instance of) the function p() has finished running function r(), the function p() is exited

13. **q is popped off stack**: now once the second instance of the function p() has finished running, the first instance of q() has no further calls to functions or blocks and is exited, thus popped off the stack.

14. **r is pushed on stack**: now once (the first instance) function q() finally is exited from within p(), p() goes on to execute the function r(), thus r() is pushed on stack.

15. **r is popped off stack**: r() has no further function calls nor blocks to enter so the r() function is exited.

16. **p is popped off stack**: once function r() is exited, p() has no further function calls nor blocks to enter so p() is popped off.

17. **B is popped off stack**: once function p() is exited, B has no further function calls nor blocks to enter so B is exited, hence B is popped off stack frame.

18. **A is popped off stack**: once block B is exited, A has no further function calls nor blocks to enter so A is exited, hence A is popped off stack frame. Now stack frame is empty indicating that the part of the program in question has finished running □

Problem 4. *Consider the following program:*

```
1   {
2       int z = 2;
3
4       void p(int x) {
5           x = x + 3;
6           z = z + 5;
7           write(x);
8       }
9
10      p(z);
11      write(z);
12  }
```

What does the program print out in case the parameter x of p is (i) call-by-value, (ii) call-by-reference, (iii) call-by-value-result? Explain by describing how the program is executed.

Solution.

i) Passing x as a ***call-by-value*** parameter ***yields the output: 5 7*** . Let's explore how this output is gotten by stepping through the program execution. p() is called and the global variable z is sent in as a parameter. The value of z, which is 2, is copied into the function parameter x because it's called-by-value. x then becomes 2+3=5 (in line 5) then z becomes 2+5=7 (in line 6). Then p() writes out x which is 5, then p() is exited and z is written out which now has the value of 7. □

ii) Passing x as a ***call-by-reference*** parameter ***yields the output: 10 10*** . Let's explore how this output is gotten by stepping through the program execution. p() is called and the global variable z is sent in as a parameter. The value of z, which is 2, is represented by parameter x in the function because it's called-by-reference. x becomes 2+3=5 which means that z becomes 5 as well. Then z becomes 5+5=10. Then p() writes out x which is 10 (the same as z), then p() is exited and z is written out which has the value 10. □

iii) Passing x as a ***call-by-value-result*** parameter ***yields the output: 5 5*** . Let's explore how this output is gotten by stepping through the program execution. p() is called and the global variable z is sent in as a parameter. The value of z, which is 2, is represented by parameter x in the function yet not aliased by it because it's called-by-value-result. x becomes 2+3=5. Then z becomes 2+5=7. Then p() writes out x which is 5, then when p() is exited x is copied into the value of z because of the value-result call, making the previous value change of z within p() redundant. z therefore becomes 5. Then lastly, z is written out as 5. □

Problem 5. *Consider the following program:*

```
1   {
2       int x = 7;
3
4       void p2() {
5           write(x);
6       }
7
8       void p4(void px()) {
9           int x = 4;
10          px();
11      }
12
13      void p3() {
14          int x = 3;
15          p4(p2);
16      }
17
18      {
19          x = 1;
20          p3();
21      }
22  }
```

What does the program print in case of (i) stating scope (and deep binding), (ii) dynamic scope and deep binding, (iii) dynamic scope and shallow binding. Explain by describing how the program is executed.

Solution.

i. static scope (and deep binding) outputs 1. Let's explore this by describing the execution of the program. The executable code in the block of the program (lines 18-21) begins by changing the value of the global x to 1, basically making the value in declaration of x=7 in line 2 redundant. Then p3() is called and p3() creates it's own local x variable with the value 3, then calls p4() with the function p2() as a parameter. When we arrive at p4(), the link between the parameter function px() and the passed-in function p2() is established and since deep binding is used, this is the environment that we use for p2()'s scope. Note however, that this link is established *before* the local x is declared in function p4() (at line 9) which is why that x is not used then in p2(), but rather the global x, as that is the only x visible in the static scope when this link is established (i.e. nothing's in the local scope, but x is in the outer scope of the local scope, i.e. global scope). x of course has been changed to 1 as stated before, which is why p2 outputs 1 □

ii. dynamic scope and deep binding outputs 3. Let's explore this by describing the execution of the program. Again, the executable code in the block of the program (lines 18-21) begins by changing the value of the global x to 1. Then p3() is called and p3() creates it's own local x variable with the value 3, then calls p4() with the function p2()

as a parameter. When we arrive at p4(), the link between the parameter function px() and the passed-in function p2() is established and since deep binding is used, this is the environment that we use for p2()'s scope. Agan we consider that this link is established *before* the local x is declared in function p4() (at line 9) which is why that x is not used then in p2(), but rather the caller's scope x, as dynamic scoping always checks out caller scope when no x is found in the local scope. The caller in this case is p3() which has it's own local x. Therefore, the local x of p3() is used in p2(), namely x=3, which is why p2 outputs 3 □

ii. dynamic scope and static binding outputs 4. Let's explore this by describing the execution of the program. Again, the executable code in the block of the program (lines 18-21) begins by changing the value of the global x to 1. Then p3() is called and p3() creates it's own local x variable with the value 3, then calls p4() with the function p2() as a parameter. p4() first declares a local variable x with the value 4, then calls parameter function px(). Since we're using shallow binding, the environment once parameter function px() ***is called*** is now used as p2()'s scope. Since p4() has it's own local x that has been declared before parameter function px() is called, that local x is in the scope when px() is called. That is why p2() uses that definition of x as it's own, namely x=4. Therefore p2() then outputs 4. □