Fall 2019 **Assignment 6(Memory Management)** November 2019

1. For the following Algol-like program, write the number printed by executing the program under each of the listed parameter passing mechanisms.

```
begin
   integer i;
   procedure pass ( x, y )
    begin
      integer x, y; // types of the formal parameters begin
      x := x + 1;
      y := x + 1;
      x := y;
      i := i + 1
    end;

   i := 1;
   pass (i, i);
   print i
end
```

(a) pass-by-value

Solution: **Ans:** 2

Consider this ML code that corresponds to the call for illustration. **Note:** the comments indicate the updated values in the call to pass(i, i)

```
fun pass(x : int, y : int) =
                       let t1 = ref x in
                       let t2 = ref y in
                       (t1 := !t1 + 1;       (* t1 = ref 2 *)
                        t2 := !t1 + 1;       (* t2 = ref 3 *)
                        t1 := !t2;           (* t1 = ref 3 *)
                        i  := !i + 1);       (* i  = ref 2 *)
...
let i = ref 1 in pass(!i, !i)
print !i
```

□

(b) pass-by-reference

Solution: **Ans:** 4

Again, consider this ML code that corresponds to the call for illustration. **Note:** the comments indicate the updated values in the call to pass(i, i)

```
fun pass(x : int ref, y : int ref) =
(x := !x + 1;  (* x = ref 2 *)
 y := !x + 1;  (* y = ref 3 *)
 x := !y;      (* y = ref 3, no need to
                  do this because updating y updates x also,
                  and i for that matter. *)
 i := !i + 1); (* i = ref 4 *)
...
let i = ref 1 in pass(i, i)
print !i
```

□

(c) pass-by-value-result

Solution: **Ans:** 3

You can follow the explanation in the problem description, to understand why this is the right answer. □

Note: in pass-by-value-result, also called call-by-value-result and copy-in/copy-out, parameters are passed by value, with an added twist. More specifically, suppose a function f with a pass- by-value-result parameter u is called with actual parameter v. The activation record for f will contain a location for formal parameter u that is initialized to the R-value of v. Within the body of f, the identifier u is treated as an assignable variable. On return from the call to f, the actual parameter v is assigned the R-value of u.

The following pseudo-Algol code illustrates the main properties of pass-by-value-result.

```
var x : integer;
x := 0;
procedure p(value-result y : integer)
  begin
    y := 1;
    x := 0;
  end;

p(x);
```

With pass-by-value-result, the final value of x will be 1: since y is given a new location distinct from x, the assignment to x does not change the local value of y. When the function returns, the value of y is assigned to the actual parameter x. If the parameter were passed by reference, then x and y would be aliases and the assignment to x would change the value of y to 0. If the parameter were passed by value, the assignment to y in the body of p would not change the global variable x and the final value of x would also be 0.

2.
```
let val x = 5
 in let fun f y = (x+y) -1;
        fun g h = let val x = 7 in h x end
     in
        let val x = 10 in g f end
    end
end ;
```

(a) What is the value of g(f) in the first code example?

Solution: **Ans:** 11 □

(b) The call g(f) in the first code example causes the expression (x+y)-1 to be evaluated. What are the values of x and y that are used to produce the value you gave in part (a)?

Solution:

- x = 5
- y = 7

□

(c) Explain how the value of y is set in the sequence of calls that occur before (x+y)-2 is evaluated.

Solution: f is the actual parameter to g in the call g f, however the formal parameter is h. This means that when we call h x, where x = 7, we are really calling f. And in that case, the formal parameter to f is y, whereas its actual parameter is x, which is bound to 7. Consequently the formal parameter y in the body of f, also gets bound to 7. □

(d) Explain why x has the value you gave in part (b) when (x+y)-1 is evaluated. *Solution:* In a statically scoped language such as SML, every global variable that appears in the body of a function retains the value assigned to it in the first enclosing block of where the function is defined.

In this specific case, in the body of f, $(x + y) - 1$, x will still be 5 because that's its value in the enclosing block. □

3. Consider the following piece of ML code:

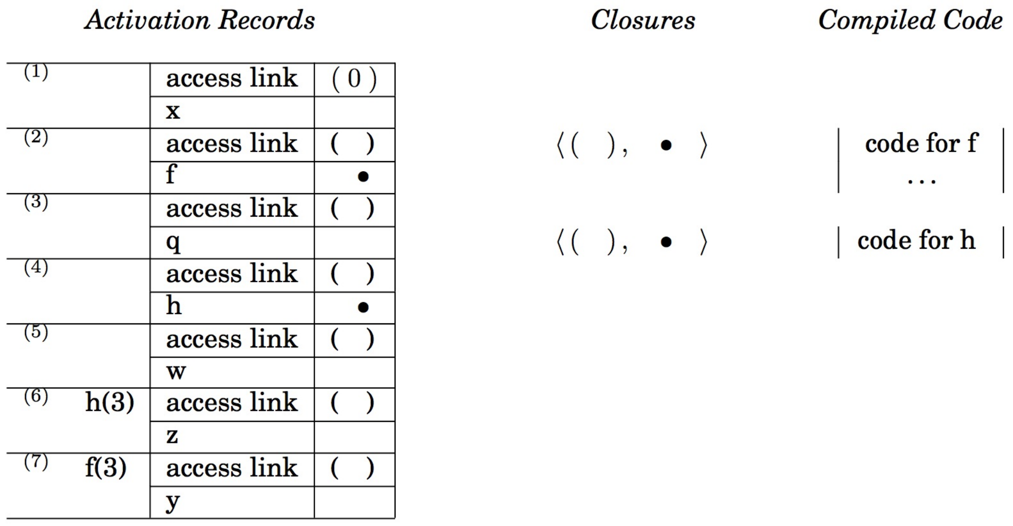
```
let val x = 1
 in let fun f y = y + x;
   in  let val q = 2
        in let fun h z = (f z ) * q;
```

```
                in let val w = h 3 in w
                    end
                end
            end
        end
    end
```

(a) What is the value of w?

Solution: **Ans:** 8 □

(b) Fill in the missing parts in the following diagram of the run-time structures for execution of this code up to the point where the call inside h(3) is about to return. The drawing uses Closures, however, for this exercise a function can point directly to the compiled code. You do not need to make use of Closures. The activation records are numbered 1 − 7, from the top.



Solution: The two closures are

$$c1 = \langle (2), \text{code for f} \rangle$$
$$c2 = \langle (4), \text{code for h} \rangle$$

1	access link x	(0) 1
2	access link f	(1) c1
3	access link q	(2) 2
4	access link h	(3) c2
5	access link w	(4) 8
6 h(3)	access link z	(4) 3
7 f(3)	access link y	(2) 3

□

4. Consider the following ML program:

```
fun foo x =  let fun bar f = fn x =>  f ( f x);
             in   bar (fn y => y + x)
             end ;
```

(a) what is the value of foo(3)(2) according to the standard (statically scoped) semantics of ML?

Solution: **Ans:** 8 □

Now, suppose we try to optimize the above function by first inlining the bar function

```
fun foo x = fn x => (fn y => y + x) ((fn y => y + x) x);
```

beta reducing

```
fun foo x = fn x => (fn y => y + x) (x + x);
```

and beta reducing again

```
fun foo x = fn x => x + x + x;
```

(b) What does foo(3)(2) evaluate to using the "optimized" version of foo?

Solution: **Ans:** 6 □

(c) What was the mistake we made in our attempted optimization?

Solution: The problem is that we captured a variable. This happened when we beta reduced after inlining `bar`. After

```
fun foo x = (fn f => fn x => f (f x)) (fn y => y + x);
```

we should alpha rename to

```
fun foo x = (fn f => fn z => f (f z)) (fn y => y + x)
```

leading to the beta reduced

```
fun foo x = fn z => (fn y => y + x) ((fn y => y + x) z)
```

but instead, we captured the x. NOTE: be lenient on this, so long as they identify the problem as a capture problem give the point □

(d) What happens if using the original, unoptimized, definition of foo we evaluate foo(3)(2) but under dynamic instead of static scoping?

Solution:

Ans: 6

The point of this is to see that substituting wrong is the same as dynamic scoping. □

5. Ex 7.12 from Mitchell

Solution:

$$C1 = \langle (2), \text{code for f} \rangle$$
$$C2 = \langle (3), \text{code for g} \rangle$$

1	access link x	(0) 5
2	access link f	(1) C1
3	access link g	(2) C2
4	access link x	(3) 10
5 g(f)	access link h x	(3) C1 7
6 h(x)	access link y	(2) 7

The expression is $(x + y) - 2$ which we evaluate with the environment of (6). 6 does not define x so we follow the access link to (2) which also does not define x meaning we follow the access link again to (1) where x has the value 5. Thus, we have $(5 + y) - 2$. Next we look up y in (6) where it has the value of 7 this leads to $(5 + 7) - 2$ which reduces to 10. Intuitively, what is going on is that x is a variable from the enclosing scope and this is a statically scoped language scoped and so therefore x has the value it had where this closure was defined. While y is a parameter and so has the value the actual parameter from where the function was called. Even though the actual parameter also had the name x it came from context 5 and so had the value of 7. □

6. Ex 7.13 from Mitchell

   Solution:

   (a)
   ```
   x : int
   f : int -> (int -> int)
   h : int -> int
   ```

   (b)

$$C1 = \langle (2), \text{code for f} \rangle$$
$$C2 = \langle (5), \text{code for g} \rangle$$
$$L = (1, \bullet) \to (2, \bullet) \to (3,)$$

1	access link x	(0) 5
2	access link f	(1) C1
3	access link h	(2) C2
4	access link x	(3) 7
5 f(3)	access link y z g	(2) 3 L C2
6 h(2)	access link w	(5) 2

   (c) **Ans:** $10 = 2 + 5 + 3$ because w, x, and y are accessed from frame 6: w is 2 in frame 6, x is accessed from frame 1 by following access links through frames 6, 5, and 2, and y is in frame 5 (accessed by following control link of frame 6)

□

7. Ex 7.14 from Mitchell *Solution:*

   (a)

$$C1 = \langle (1), \text{code for myop defined in line 1} \rangle$$
$$C2 = \langle (3), \text{code for myop defined in line 6} \rangle$$
$$C3 = \langle (2), \text{code for recurse} \rangle$$

1	access link	(0)
	control link	(0)
	myop	C1
2	access link	(1)
	control link	(1)
	recurse	C3
3	access link	(2)
	control link	(2)
	myop	C2
4 recurse(1)	access link	(2)
	control link	(3)
	n	1
	t_1	
5 myop(1, recurse(0))	access link	(1)
	control link	(4)
	x	1
	y	1
	t_1	
	t_2	
6 recurse(0)	access link	(2)
	control link	(5)
	n	0
	t_2	1

   (b) **Ans: 1**. From frame 6, we follow the access link to obtain the right value of *myop*. That is, we go through frames 2 which points us to frame 1, where we find the closure associated with *myop*. The formal parameters are x, and y and their corresponding actual values are 1 and 1, which we multiply to get 1.

8. **Ans: 2** If dynamic scope was used, we would have no need for the access links, and so we would follow the control links. We follow the links through 5, 4, and finally 3, where we find the definition of *myop*. The formal parameters are x, and y and their corresponding actual values are 1 and 1, which we add to get 2.

□

9. Ex 7.15 from Mitchel *Solution:* All C functions are declared at the top level and so only have access to the global scope. As such, we never need to “close over” anything. □