Handout 5
CS242: Autumn 2004
20 October

Study Questions

Reading

1. Read Chapter 7 on scope and Chapter 8 on control in sequential languages.

Problems

1. .... Parameter passing comparison

For the following Algol-like program, write the number printed by running the program under each of the listed parameter passing mechanisms. Pass-by-value/result, also sometimes called copy-in/copy-out, is explained in exercise 6 of Chapter 7 of the text.

```
begin
   integer i;

   procedure pass ( x, y );
      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

(b) pass-by-reference

(c) pass-by-value/result

2. .... Static and Dynamic Scope

Consider the following program fragment, written both in ML and in pseudo-C:

```
1  let x = 2 in                      { int x = 2; {
2    let val fun f(y) = x + y in         int f (int y) { return x + y; } {
3      let val x = 7 in                    int x = 7; {
4        x +                                 x +
5            f(x)                                f(x);
6      end                                 }
7    end                                 }
8  end;                              }
```

The C version would be legal in a version of C with nested functions.

(a) Under static scoping, what is the value of x + f(x) in this code? During the execution of this code, the value of x is needed three different times (on lines 2, 4, and 5). For each line where x is used, state what numeric value is used when the value of x is requested and explain why these are the appropriate values under static scoping.

(b) Under dynamic scoping, what is the value of x + f(x) in this code? For each line where x is used, state which value is used for x and explain why these are the appropriate values under dynamic scoping.

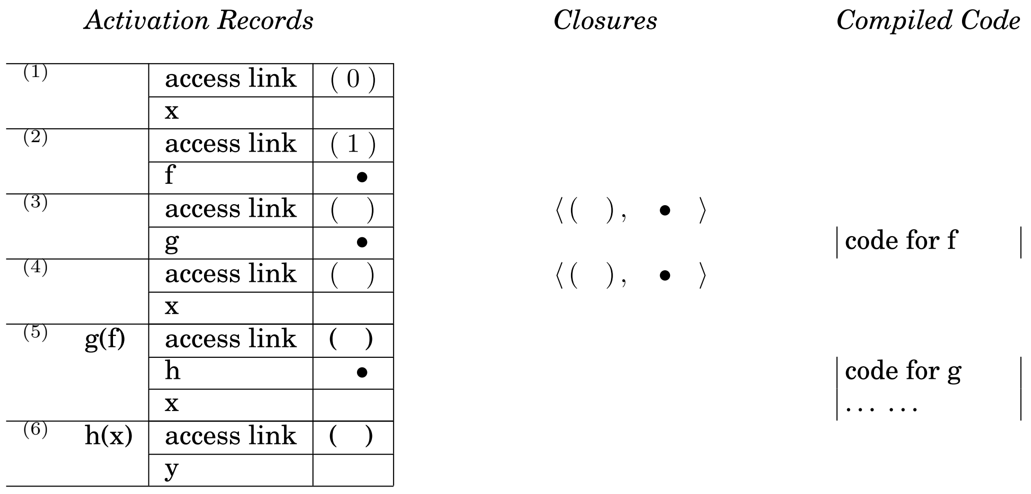
3. .... Function Calls and Memory Management

This question asks about memory management in the evaluation of the following statically-scoped ML expression.

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

(a) Fill in the missing information in the following depiction of the run-time stack after the call to h inside the body of g. Remember that function values are represented by closures, and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code.

In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code. Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link." The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.



(1)	access link	(0)
	x	
(2)	access link	(1)
	f	•
(3)	access link	( )
	g	•
(4)	access link	( )
	x	
(5) g(f)	access link	( )
	h	•
	x	
(6) h(x)	access link	( )
	y	

(b) What is the value of this expression? Why?

4. .... Function Returns and Memory Management

This question asks about memory management in the evaluation of the following statically-scoped ML expression.

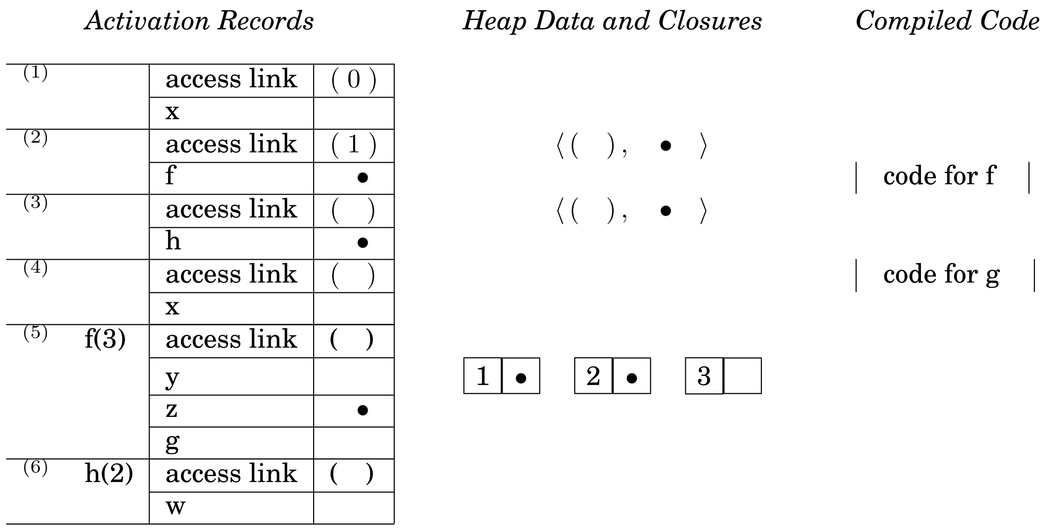
```
val x = 5;
fun f(y) =
    let val z = [1, 2, 3]  (* declare list *)
        fun g(w) = w+x+y   (* declare local function *)
    in
        g                  (* return local function  *)
    end;
val h = let val x=7 in f(3) end;
h(2);
```

(a) Write the type of each of the declared identifiers (x, f, and h).

(b) Since this code involves a function that returns a function, activation records cannot be deallocated in a LIFO stack-like manner. Instead, let us just assume that activation records will be garbage collected at some point. Under this assumption, the activation record for the call f in the expression for h will still be available when the call h(2) is executed.

Fill in the missing information in the following depiction of the run-time stack after the call to h at the end of this code fragment. Remember that function values are represented by closures, and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code.

In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure, compiled code or list cell. Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link." The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.



(c) What is the value of this expression? Explain which numbers are added together and why.

(d) If there is another call to h in this program, then the activation record for this closure cannot be garbage collected. Using on the definition of garbage given in the Lisp chapter, explain why, as long as h is reachable, mark-and-sweep will fail to collect some garbage that will never be accessed by the program.

5. .... Exceptions and Recursion

Here is an ML function that uses an exception called Odd.

```
fun f(0) = 1
|   f(1) = raise Odd
|   f(3) = f(3-2)
|   f(n) = (f(n-2) handle Odd => ~n)
```

The expression `~n` is ML for $-n$, the negative of the integer n.

When `f(11)` is executed, the following steps will be performed:

```
call f(11)
call f(9)
call f(7)
  ...
```

Write down the remaining steps that will be executed. Include only the following kinds of steps:

- function call (with argument)
- function return (with return value)
- raise an exception
- pop activation record of function off stack without returning control to the function
- handle an exception

Assume that if `f` calls `g` and `g` raises an exception that `f` does not handle, then the activation record of `f` is popped off the stack without returning control to the function `f`.

6. .... Tail Recursion and Continuations

(a) Explain why a tail recursive function, as in

```
fun fact(n) =
    let fun f(n,a) = if n=0 then a
                     else f(n-1, a*n)
    in  f(n,1) end;
```

can be compiled so that the amount of space required to compute `fact(n)` is independent of `n`.

(b) The function `f` used in the following definition of factorial is "formally" tail recursive: the only recursive call to `f` is a call that need not return.

```
fun fact(n) =
    let fun f(n,g) = if n=0 then g(1)
                     else f(n-1, fn x=>g(x)*n)
    in  f(n, fn x => x) end;
```

How much space is required to compute `fact(n)`, measured as a function of argument `n`? Explain how this space is allocated during recursive calls to `f` and when the space may be freed.