Homework 3

Due 22 October

Handout 5
CS242: Autumn 2003
15 October

Reading

1. Read chapters 6 (Types) and 7 (Scope, functions, and storage management) of the text.

Problems

1. .... Time and Space Requirements

This question asks you to compare two functions for finding the middle element of a list. (In the case of an even-length list of $2n$ elements, both functions return the $n+1$st.) The first uses two local recursive functions, `len` and `get`. The `len` function finds the length of a list and `get(n,l)` returns the `n`th element of list `l`. The second middle function uses a subsidiary function `m` that recursively traverses two lists, taking two elements off the first list and one off the second until the first list is empty. When this occurs, the first element of the second list is returned.

```
exception Empty;

fun middle1(l) =
   let fun len(nil) = 0
       |   len(x::l) = 1+len(l)
       and get(n,nil) = raise Empty
       |   get(n,x::l) = if n=1 then x else get(n-1,l)
   in
       get((len(l) div 2)+1, l)
   end;

fun middle2(l) =
   let fun m(x,nil) = raise Empty
       |   m(nil,x::l) = x
       |   m([y],x::l) = x
       |   m(y::(z::l1),x::l2) = m(l1,l2)
   in
       m(l,l)
   end;
```

Assume that both are compiled and executed using a compiler that optimizes use of activation records or "stack frames."

(a) Describe the approximate running time and space requirements of `middle1` for a list of length n. Just count the number of calls to each function and the maximum number of activation records that *must* be placed on the stack at any time during the computation.

(b) Describe the approximate running time and space requirements of `middle2` for a list of length n. Just count the number of calls to `m` and the maximum number of activation records that *must* be placed on the stack at any time during the computation.

(c) Would an iterative algorithm with two pointers, one moving down the list twice as fast as the other, be significantly more or less efficient than `middle2`? Explain briefly in one or two sentences.

2. .... Parameter Passing

Consider the following procedure, written in an Algol/Pascal-like notation:

```
proc power(x, y, z:int)
begin
     z := 1
        while y > 0 do
            z := z*x
            y := y-1
        end
end
```

The code that makes up the body of power is intended to calculate $\mathtt{x}^{\mathtt{y}}$ and place the result in z. However, depending on the actual parameters, power may not behave correctly for certain combinations of parameter-passing methods. For simplicity, we only consider call-by-value and call-by-reference.

(a) Assume a and c are assignable integer variables with distinct L-values. Which parameter-passing methods make $\mathtt{c} = \mathtt{a}^{\mathtt{a}}$ *after* a call $\mathtt{power}(\mathtt{a}, \mathtt{a}, \mathtt{c})$. You may assume that the R-values of a and c are non-negative integers.

(b) Suppose that a and c are formal parameters to some procedure P, and that the expression power(a,a,c) above is evluated inside the body of P. If a and c are passed to P by reference, and become aliases, then what parameter passing method(s) will make $\mathtt{c} = \mathtt{a}^{\mathtt{a}}$ *after* a call $\mathtt{power}(\mathtt{a}, \mathtt{a}, \mathtt{c})$? If, after the call, $\mathtt{c} = \mathtt{a}^{\mathtt{a}}$, does that mean that power actually performed the correct calculation?

3. .... Lambda Calculus and Scope

Consider the following ML expression:

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

In β-reduction on lambda terms, the function argument is substituted for the formal parameter. In this substitution, it is important to rename bound variables to avoid capture. This question asks about the connection between names of bound variables and static scope. Using a variant of substitution that does not rename bound variables, we can investigate dynamic scoping.

(a) The following lambda term is equivalent to the function foo:

$$\lambda x.\ (\ (\lambda f.\lambda x.f(fx))\ (\lambda y.y + x)\ )$$

Use β-reduction to reduce this lambda term to normal form.

(b) Using the example reduction from part (a), explain how renaming bound variables provides static scope. In particular, say which variable above (x, f or y) must be renamed, and how some specific variable reference is therefore resolved statically.

(c) Under normal ML static scoping, what is the value of the expression $\mathtt{foo}(3)(2)$?

(d) In the usual statically scoped lambda calculus, α-conversion (renaming bound variables) does not change the value of an expression. Use the example expression from part (a) to explain why α-conversion may change the value of an expression if variables are dynamically scoped. (This is a general fact about dynamically-scoped languages, not a peculiarity of lambda calculus.)

(e) Give a lambda term in normal form that corresponds to the function that the expression in part (a) would define under dynamic scope. Give a sequence of reductions that gets the expression to the normal form you gave. While doings these reductions you should *not* be renaming bound variables when you perform substitution.

(f) Under dynamic scoping and the reduction order used in the previous part, what is the value of the expression foo(3)(2)?

4. .... 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.

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	

Closures

⟨(   ), • ⟩

⟨(   ), • ⟩

Compiled Code

code for f

code for g
... ...

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

5. .... Function Returns and Memory Management

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

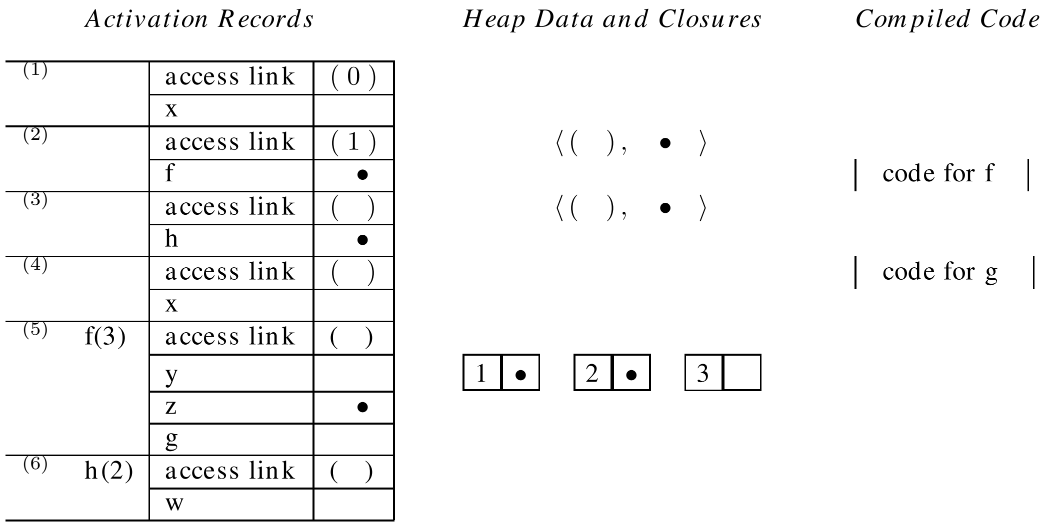
```
val x = 5;
fun f(y) =
    let val z = [1, 2, 3]   (* declare list *)
        fun g(w) = w+x+y    (* declare local function *)
    in
        g                   (* return local function  *)
    end;
val h = let val x=7 in f(3) end;
h(2);
```

(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.

6. .... Closures and Tail Recursion

The function `f` in this declaration of factorial is formally tail recursive: the only calls in the body of `f` are tail calls.

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

This question asks you to consider the problem of applying the ordinary tail-recursion-elimination optimization to this function.

(a) Fill in the missing information in the following outline of the activation records resulting from the unoptimized execution of `f(2, fn x => x)`. You may want to draw closures or other data.

f(2, fn x => x)	access link	
	n	
	g	
f(1, h)	access link	
	n	
	g	
f(0, h)	access link	
	n	
	g	

(b) What makes this function more difficult to optimize than the other examples discussed in this chapter? Explain.

7. .... SML Implementation of Stack Operations

This problem asks you to complete some sample Standard ML code implementing stack storage management for simple blocks and functions, such as these two declarations and call:

```
val x = 3;
fun f(y) = if y=0 then x else  y+f(y-1);
f(5);
```

Our implementation for this code will have the form

```
push_block_AR(!env,3);               (* declare x=3 *)
push_function_AR(!env,show_this,5);  (* create function activ record *)
function_body();                     (* execute function body  *)
pop();                               (* pop activation record  *)
pop();                               (* pop activation record  *)
!show_this;                          (* show function return value  *)
```

where `env` is the environment pointer and `show_this` is a location used to hold the return value of the function call. The functions used above are defined (with a small part missing) below. The first two calls push activation records onto the stack, the first an activation record (AR) for the declaration `x=3` and the second for the call to `f`. For simplicity, we will not create activation records for function declarations. The statement `function_body()` executes instructions associated with the function body of `f`, and the two `pop` operations remove activation records from the run-time stack.

Since we have two kinds of activation records, the datatype of activation records has two different options, plus an option for an empty run-time stack.

```
datatype activation =
   Function_AR of {
      control_link : activation,
      access_link : activation,
      return_result_addr : int ref,
      parameter : int,
      tmp : int ref
} |
   Block_AR of {
      control_link : activation,
      access_link : activation,
      value : int
} |
   NOSTACK;

val env = ref NOSTACK;
```

In words, a function activation record has a control link, an access link, a return-result address, a parameter, and a location for a temporary integer value (local variable or intermediate result). A block activation record has a control link, an access link, and a local value. We begin with the environment pointer pointing to an empty run-time stack.

Here are operations to push and pop activation records on the run-time stack.

```
fun push_block_AR(a,v) =    (* arguments are access link and local value *)
   let val ar = Block_AR{
      control_link=(!env), (* control link always points to previous top *)
      access_link=a,
      value=v}
   in
      env := ar             (* place new activation record on top of stack *)
   end;

fun push_function_AR(a,rv,n) = (* arguments are access link, return-value *)
   let val ar = Function_AR{    (* address and value of function parameter *)
      control_link=(!env),
      access_link=a,
      return_result_addr=rv,
      parameter=
      tmp=ref 0}              (* assignable temporary location initially 0 *)
   in
      env := ar
   end;

fun pop() = case (!env) of
   NOSTACK => () |
   Function_AR(ar)   =>  env := #control_link(ar) |
      Block_AR(ar)   =>  env := #control_link(ar) ;
```

Here are a couple of "helper" functions that may be useful later. They return parts of a block activation record. Since the declarations do not give cases for all possible forms of activation records, the compiler will give you a warning.

```
fun block_value(Block_AR(ar)) = #value(ar);
fun block_access_link(Block_AR(ar)) = #access_link(ar);
```

If you apply `block_value` to a function activation record, SML will raise the `Match` exception, meaning that the match against the pattern `Block_AR(ar)` in the function call failed. However, these functions will work fine if applied to block activation records.

Here is code (with a few parts missing) for the function body of `f`.

```
exception Bad_Activation_Record;

fun function_body() = case (!env) of
   NOSTACK => raise Bad_Activation_Record       |
   Block_AR(ar) => raise Bad_Activation_Record |
   Function_AR{
      control_link= control_link,
      access_link =  access_link,
      return_result_addr=return_result_addr,
      parameter= function_parameter,
      tmp= tmp
    } =>
      if function_parameter = 0 then
            return_result_addr := (*---- missing part -----*)
      else (
            push_function_AR(access_link, tmp, function_parameter-1);
            function_body();
            return_result_addr := function_parameter + !tmp;
            pop()
       );
```

This function looks at the current activation record. If it is a function activation record, then the pattern match in the case expression will bind `control_link, access_link, ...` to these parts of the current activation record on top of the stack. The only missing part is the expression for the function return value with the parameter is 0. When the parameter is not zero, the function body executes the normal calling sequence: push an activation record onto the stack, execute the function body, set the return value, and pop the activation record.

(a) What expression goes in the missing part of the code above? Download the sample code from the Handout section of the CS242 web site, fill in the missing piece, and test your function by executing these instructions:

```
val show_this = ref 0;

push_block_AR(!env,3);              (* declare x=3 *)
push_function_AR(!env,show_this,5); (* create function activ record *)
function_body();                    (* execute function body  *)
pop();                              (* pop activation record  *)
pop();                              (* pop activation record  *)
!show_this;                         (* show function return value  *)
```

(b) We can modify this code to construct and use the run-time stack associated with this sample code:

```
val x = 5;
val z = 3;
fun f(y) = if y=0 then x else  y+f(y-1);
f(5);
```

How would you fill in the missing part of `function_body` for this case? You can test your solution using the sample code from the CS242 web site.

(c) Why is `(*---- missing part -----*)` different for these two situations?

(d) Why is `(*---- missing part -----*)` the only line that changes?

(e) Changing only two lines of the function you wrote for part (a) or part (b), write a version of `function_body` that executes instructions for the function `f` in the following program.

```
val u = 5;
val v = 3;
fun f(y) = if y=0 then u+v else v+f(y-1);
f(5);
```

Turn your definition of this function in using the same submit script as previous homework.