CIS 425 : Final Exam – Fall 2017

Your Name : _____

[8 Points] What is the value of the following expression

```
let val x = 2 in
  let fun f y = x in
    let val x = 5 in
      let fun g y = f y in
        let fun f z = f z in
           g (f x) end end end end end ;
```

under

(a) static scoping and call-by-value evaluation?

(b) dynamic scoping and call-by-value evaluation?

(c) static scoping and call-by-name evaluation?

(d) dynamic scoping and call-by-name evaluation?

[**12 Points**] This question asks about memory management in the evaluation of the following statically-scoped ML expression.

```
val x = 9;
fun f(y) =  let val z = [1, 2, 3]
                fun g(w) = w * x * y
            in g end;
val h = let val x= 3 in f(x) end;
val z = h(x);
```

1. Draw the run-time structures (with static link, dynamic link, values of local variables, etc.) as they appear before returning from `h(x)`.

2. What is the value of z?

[6 Points] Give the ML types of the following function and explain how you derive them from the set of constraints. In other words, you cannot just give the final type, you have to show how ML obtains the final type.

```
fun g (h,x) = fn y => (h x) + (y * 2);
```

[**4 points**] The comparison operator in ML is overloaded, that is, it can have different types. For example, one can write `2 < 3` or `2.1 < 3.2`. If not enough information is given, ML assumes by default that the elements to be compared have type `int`. What are the types ML infers for the following two functions:

[**2 points**] `fun f x y = x < y`

[**2 points**]
```
fun sort nil = nil
  |   sort (x :: nil ) = [x]
  |   sort (x1 :: x2 :: xs) = let val (y :: res)  = sort (x2 :: xs)
                                 in if x1 < y then (x1 :: y :: res)
                                              else y :: (sort (x1 :: res))
                              end
```

[4 points] What are the types Haskell would infer for the following two functions:

[2 points]
```
f x y = x < y
```

[2 points]
```
sort nil = nil
sort (x : nil ) = [x]
sort (x1 : x2 : xs) = let (y : res)  = sort (x2 : xs)
                      in if x1 < y then (x1 : y : res)
                                   else y : (sort (x1 : res))
```

[**13 points**] Complete the call-by-value, statically-scoped interpreter below, by filling in the blanks with the appropriate code.

```
datatype E = Variable of string | Literal of int | Lambda of string * E
  | Lett of ((string * E) list) * E | App of E * E ;

datatype env = Env of (string * ____) list                                  <-----
and   result = Value of int | Closure of E * env | Error of int ;

fun extend_env (Env(oldenv), id, value)             =   Env( (id, value):: oldenv);
fun extend_env_all (Env(oldenv), id_value_list) =  Env(id_value_list @ oldenv);

fun  lookup_env (Env(nil), id) =
                      (print("Free Var!! "^id); raise not_found)
    |lookup_env (Env((id1,value1)::b), id) =
                            if (id1 = id) then value1
                       else lookup_env(Env(b), id) ;

fun interp1(exp,env) =
    case  exp of
        Variable(id)   =>  _____                                             <-----
      | Literal(x)   =>  ???                                               <-----
      | Lambda(id,exp) =>  ???                                               <-----
      | App(E1, E2)    => let val V1 = ???                                   <-----
                             val V2 = ???                                   <-----
                         in
                             case V1 of
                                 Closure(Lambda(id,exp),???) =>             <-----
                                    let val new_env = extend_env(???,       <-----
                                       id, ???)                             <-----
                                    in
                                       interp1(exp, ???)                    <-----
                                    end
                               | _ => raise err(2)
                          end
      | Lett(id_exp_list, E) =>
                  let val id_result_list =
                     map (fn (id,exp) => (id, ???)) id_exp_list             <-----
                  in
                     let val new_env =
                                    extend_env_all(???, id_result_list)     <-----
                     in interp1(E, ???)                                     <-----
                     end
                  end
```

[**5 Points**] With just a single change, this interpreter would implement dynamic scoping rules instead of static. Indicate where this change would be made in the interpreter above, and write the code that would be used instead.

[**14 Points**] Complete the call-by-name, statically-scoped interpreter below, by replacing the `???` with the appropriate code.

```
datatype E = Variable of string | Literal of int | Lambda of string * E
  | Lett of ((string * E) list) * E | App of E * E ;

datatype env = Env of (string * ??? ) list ;                              <-----

datatype   result = Value of int | Closure of E*env | Error of int ;

fun extend_env (Env(oldenv), id, exp,env) = Env( (id, (exp,env)):: oldenv);
fun extend_env_all (Env(oldenv), id_exp_list) = Env(id_exp_list @ oldenv);

fun  lookup_env (Env(nil), id) = (print("Free Var!! "^id); raise not_found)
    |lookup_env (Env((id1,(exp,env))::b), id) =
                            if (id1 = id) then (exp,env)
                     else lookup_env(Env(b), id) ;

fun interp_name_1(exp,env) =
    case  exp of
        Variable(id)   => let val ??? = ????                              <-----
                           in ???  end                                    <-----
      | Literal(x)   =>  ???                                             <-----
      | Lambda(id,exp) =>  ???                                            <-----
      | App(E1, E2)    => let val V1 = ???                                <-----
                          in
                              case V1 of
                                  Closure(Lambda(id,exp),???) =>          <-----
                                     let val new_env = extend_env(???,    <-----
                                             id, ???)                     <-----
                                     in
                                        interp_name_1(exp, ???)           <-----
                                     end
                                | _ => raise err(2)
                           end
      | Lett(id_exp_list, E) =>
                 let val id_exp1_list =
                           map (fn (id,exp) => (id, ??? )) id_exp_list    <-----
                 in
                    let val new_env = extend_env_all(???, id_exp1_list)   <-----
                    in
                       interp_name_1(E, ???)                              <-----
                    end
                 end
```

[5 points] Using exceptions write a program in ML that produces 0 under call-by-value and 1 under call-by-name.

[**6 points**] Convert the function preorder to CPS. Do not convert the functions :: (cons) and @ (append) to CPS; just use them as they are in the non CPS version of preorder (e.g. if l1 and l2 are lists, it is ok to write 'i :: (l1 @ l2)'). If you do not remember what CPS is, your job is to translate the preorder to a tail recursive function.

```
datatype tree =
    Empty
  | Bin of int * tree * tree

fun preorder Empty = []
  | preorder (Bin(i,t1,t2)) = i :: ((preorder t1) @ (preorder t2))
```

[6 points]

- Write a Haskell function `intList` n that will create a list of integers from n to infinity: n, n+1, n+2, $\cdots$ (You may not use the special built-in list syntax for this; build the list using only the cons operator (:).)

- Write a Haskell function `takeN` that returns the first n elements from a list. (Do not use any standard functions for this.) For example,

```
takeN 4 (intList 10)
```

  should evaluate to

```
[10, 11, 12, 13]
```

[**9 points**] Write the same functions in SML.

- Define a SML data type `Seq` that represents infinite sequences of integers.

- Write a SML function `intList` n that will create a sequence of integers from n to infinity: n, n+1, n+2, $\cdots$ The type of intList is Int $\rightarrow$ Seq.

- Write a SML function `takeN` that returns the first n elements from a sequence. The type of `takeN` is `int` $\rightarrow$ `Seq` $\rightarrow$ `int list`.
`takeN 4 (intList 10)` should evaluate to `[10, 11, 12, 13]`.

[**6 points**] Consider the following Haskell interpreter for a very simple language of arithmetic:

```
data E = Literal Int | Add (E,E) | Div (E,E)

interp :: E -> Int
interp (Literal x) = x
interp (Add (e1,e2)) = (interp e1) * (interp e2)
interp (Div (e1,e2)) = (interp e1) `div`  (interp e2)
```

Suppose you want to raise an exception when you try to evaluate an expression of the form `Div (Literal 1, Literal 0)`. However, you might recall that Haskell does not provide the way to raise an exception. Rewrite the above interpreter using the representation of an exception in terms of the `Maybe` data type. In case of a division by zero you want to return `Error "Division by zero"`.

```
data Maybe a = Ok a | Error String
interp ::  E ->  Maybe Int
interp (Literal x) = ?
interp (Add (e1,e2)) =




interp (Div (e1,e2)) =
```

[**10 points**] Recall the definition of the `Monad` type class:

```
class Monad m where
 return :: a ->  m a
 (>>=)  :: m a -> (a -> m b) ->  m b
```

Monads capture the essential plumbing required to implements side-effects in a functional setting. Show that the type constructor `Maybe` is a monad by providing the implementations of the `return` and `bind` operations.

- ```
  return :: a --> Maybe a
  return x =

  (>>=) ::  Maybe a --> (a --> Maybe b) --> Maybe b
  (>>=) m k =
  ```

- Complete the definition of the interpreter using the `Maybe` monad. In other words, instead of threading the error in the code we use the `return` and `bind` operations of the `Maybe` monad.

  ```
  data E = Literal Int | Add (E,E) | Div (E,E)

  interp :: E ->  Maybe Int
  interp (Literal x) = return x
  interp (Add (e1,e2)) = do {x1 <- ...........
                                          x2 <- ..........
                                          return ..........
                                          }
  interp (Div (e1,e2)) = do { x1 <- ..............
                                          x2 <- ...............
                                          if ........ then Error "division by zero"
                                          else return ............
                                          }
  ```
  ```