**Sp2015 Q:Value or error**

A:let ans =

let x = 0 in

let a1 = let x = 1 in

fun y z -> [x;y;z]

in

let a2 = let x = 100 in

in a2 x

ans = [1;100;0]

Explanation:a1 takes two parameters y,z and makes a list of [1;y;z] because x was defined to be 1 in a1. Then in a2, x is defined to be 100 and passed into y for a1. This makes a2  a function that takes one parameter z and makes a list of [1;100;z].

then a2 x is called. x=0 was called in the first line, and all the subsequent x=something calls were in function definitions so those were only part of the specific function closures. So a2 0 is called resulting in **[1;100;0].**

**Q:Type ‘a set = Set of ‘a list**

**Implement val empty: `a set**

A:let empty = Set []

**Q:val member: `a->`a set->bool**

**Write funct. s.t. member x return true if x is in set corresponding to s and false otherwise.**

A:let member x s = match s with   
 | Set [] -> false  
 | Set (h::t) -> List.exists (fun y -> y = x) (h::t)

**OR** List.mem x (h::t) (\* non-rec \*)

**OR** if h = x then true else (member x (Set t));;

**Q:val add:`a->`a set->`a set**

**Write funct. s.t. add x s return a set w/ all elements of s and also element x.**

A:let add x s = match s with  
 | Set ls -> Set (x::ls)

**Q:val del:`a->`a set->`a set**

**Write funct. s.t. contain all elements of s except the element x. Hint use List.filter**

let del x s = match s with  
 | Set xs -> Set (List.filter (fun xs' -> x <>xs' ) xs)

**Q:Write funct. s.t. free e returns the set of free variables in an expression e.**

val free : expr -> string set

let rec free e = match e with  
| Var x      -> add x empty  
| Const n   -> empty  
| Bin (e1,op,e2)->union(free e1)(free e2)

(\* free variables in a binop is the union of free vars in the LR operands \*)   
| App(e1,e2)->union (free e1)(free e2)

(\* free vars in an app are the free vars in the lambda free vars in the argument. Params of Lambda is not free \*)  
 | Let (x,e1,e2)-> del x (union (free e1) (free e2))  
  (\* free var in let exp is whatever is free in e1 + free in e2,(sans refs to e1 in e2 because that is bound to x)\*)  
 | Fun (x,e1) -> delete x (free e1)  
  (\* free vars in a lambda is whatever is free in the body, sans the param \*)  
**Q:Complete the implementation.**

let isWellFormed e = ((free e) = empty)

**SP14-------------------------------------------------------**

1b)let rec chain fs = match fs with

| [] -> fun x -> x

| f::fs' -> fun x-> f (chain fs' x)

**A:** (‘a -> ‘a) list -> ‘a -> ‘a = <fun>

**So fs as seen in “match fs with” is a list of something. so lets called it ‘a list.**

**Then from this line “ fun x -> x”, we know that fs must be a list function**

**that takes input of type whatever x is, and output the same type. hence, it is**

**(‘a -> ‘a) list -> ‘a -> ‘a-- first (‘a - > ‘a) list describes the parameter fs-- first ‘a describes the parameter taken by fs.**

**Q1c)** let ans = chain [(fun x->x\*x);

(fun x->16\*x);

(fun x->x+1)] 1

**A:**It is 1024 because eventually u input 1 as x in the last function so fun 1 -> 1 + 1 = 2 now plug in middle function so fun 2 -> 16 \* 2 = 32 now plug in first function fun 32 -> 32\*32 = 1024.

**Q1g)**let rec glub f t = match t with

| leaf -> leaf

| Node(x,l,r)->Node(f x,glub f l, glub f r)

**A:**type is: (‘a -> ‘b) -> ‘a tree -> ‘b tree

2a) tail recursive version of sum:

let sumTR n =

let rec helper acc n = match n with

| 0 -> acc

| n -> helper (**acc+n**) (n-1)

in in helper 0 n;;

2b) tail recursive version of fac:

let facTR n =

let rec helper acc n = match n with

| 0 -> acc

| n -> helper (**acc\*n**)(n-1)

in in helper 1 n;;

2c)

let foldn f b n =

let rec helper acc n = match n with

| 0 -> acc

| n -> helper (f acc n)(n-1)

in in helper b n

2d)

let sum = foldn (fun a b -> a+b) 0

let fac = foldn (fun a b -> a\*b) 1

3b)

let rec lookup k kvs = match kvs with

 | []         -> None

 | x::xs’   -> if k = (fst x)

then Some ( snd x)

      else lookup k xs’;;

3c)

let lift1 f xo = match xo with

 | Some a -> Some (f a)

 | \_            -> None;;

3d)

let lift2 f xo yo = match xo, yo with

 | Some a, Some b -> Some(f a b)

 | \_                          -> None;;

3e)

let rec eval env e = match e with

 | Var x -> lookup x env

 | Con i -> Some i

| Neg e’ -> lift1 (fun x -> 0 - x) (eval env e’)

 | Plus (e1,e2) -> lift2 (+) (eval env e1) (eval env e2)

**SP12-----------------------------------------**

val ( <.> ) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

10

let giftList l =

let rec helper acc l = match l with

| [] -> acc

| g::l’ -> g^”and”^helper(acc, l’)

  in helper “that’s what I want for Christmas!” l;;

**Q2:**let rec getEven xs = match xs with

| [] -> None

| x::xs' -> if(x mod 2 = 0)

then Some x

else getEven xs'

**A: type is:**  int list -> int option

**Qb**

let rec find\_first f xs = match xs with

| [] -> None

| x::xs’ -> if (f x)

then Some x

else find\_first f xs’

**Qd**

let rec post\_fold f b t = match t with

| Leaf -> b

| Node (x,l,r)-> f x post\_fold(f “” l) post\_fold(f “” r)

**Qe**

let rec in\_fold f b t = match t with

| Leaf -> b

| Node(x.l.r) -> infold (f (f (in\_fold f b l) x) r)

**Q3 type (`a, `b) either = Left of `a | Right of `b**

**3a** Write expr that has (int, string) either and

(int, bool) either

**A:**Left 0

**3b** Write expr that has (int, string) either, but not

(int, bool) either

**A:** Right “”

**3c Write a funct assoc: of type**

**val assoc: `k->(`k\*`v) list->(`k,`v) either**

let rec assoc key kvs = match kvs with

| [] -> Left key

| (k,v)::kvs’ -> if key = k then Right v

   else assoc key kvs’

**3d Write a funct map: of type**

**val map: (`b->`c)->(`a,`b) either-> (`a,`c) either**

let map f e = match e with

| Left x -> Left x

| Right y -> Right (f y)

**3e** **Write a funct map2: of type**

**val map: (`b->`c->`d)->(`a,`b) either-> (`a,`c) either-> (`a,`d) either**

let map2 f e1 e2 = match e1 with

| Left x1 -> Left x1

| Right y1 -> match e2 with

| Left x2 -> Left x2

| Right y2 -> Right (f x2 y2)

**Q4**

**Qa Write a funct lookup: of type**

**val lookup: string -> (string \* int) list -> (error, int) either**

let lookup x env = let y = assoc x env in match y with

| Left a -> Left UnboundVariable a

| Right b -> Right b

**Qb Write a Funct of type:**

**val safeDiv: int -> int -> (error, int) either**

let safeDiv n m = match m with

          | 0 -> Left DivideByZero

| \_ -> Right (n/m)

**Qc Write a funct with type:**

**val eval: (string \* int) list -> expr -> (error, int) either**

let rec eval env e = match e with

| Const i -> Right i

| Var v -> lookup v env

| Bin(e1, Plus, e2) -> map2 (+) (eval env e1) (eval env e2)

| Bin(e1, Div, e2) -> match (eval env e1, eval env e2) with