

COMP 302 W25 Practice Problem Set 1

Problem 1: Type Inference (Higher-Order Functions)

a) Infer the type of:

```
let rec twist lst acc =  
  match lst with  
  | [] -> acc  
  | h :: t -> twist t (h :: acc)
```

`'a list -> 'a list -> 'a list`

b) Infer the type of:

```
let mystery f g x = match x with
  | Some y -> f y
  | None -> g ()
```

```
mystery : ('a -> 'b) -> (unit -> 'b) -> 'a option -> 'b
```

Problem 2: Tracing Expressions with Shadowing

a) Trace:

```
let x = 5 in
let f y = x + y in
let x = 10 in
f x
```

15

b) Trace `sum [1; 2; 3]`:

```
let sum lst =  
  let rec helper acc lst =  
    match lst with  
    | [] -> acc  
    | h :: t -> helper (acc + h) t  
  in helper 0 lst
```

c) Trace:

```
let x = 2 in
let f g = g (x + 1) in
let x = 5 in
f (fun y -> x * y)
```

15

Problem 3: Environment Lookup and Evaluation

a) Implement `lookup : string -> (string * float) list -> float option` that searches an association list for the first occurrence of a variable. Below are a few examples:

- `lookup "x" [("x", 2); ("y", 3)] ⇒ Some 2`
- `lookup "z" [("x", 2)] ⇒ None`
- `lookup "x" [("x", 1); ("x", 2)] ⇒ Some 1`

```
let rec lookup key lst =  
  match lst with  
  | [] -> None  
  | (k, v) :: rest -> if k = key then Some v else lookup key rest
```

- b) Given the types of `expr` and `env` shown below. Implement the function `eval` : `expr -> env -> float option` where the evaluation *fails* if an unknown variable is encountered. (Hint: Use `lookup` you developed in part a)).

```
type expr =  
  | Const of float  
  | Var of string  
  | Plus of expr * expr  
  | Times of expr * expr  
  
type env = (string * float) list
```

```
let rec eval expr env =  
  match expr with  
  | Const c -> Some c (* Constants always return Some value *)  
  | Var v -> lookup v env (* Look up the variable in env *)  
  | Plus (e1, e2) ->  
    (match eval e1 env, eval e2 env with  
     | Some v1, Some v2 -> Some (v1 +. v2) (* Only add if both are valid *)  
     | _ -> None) (* If either operand is None, return None *)  
  | Times (e1, e2) ->  
    (match eval e1 env, eval e2 env with  
     | Some v1, Some v2 -> Some (v1 *. v2) (* Only multiply if both are valid *)  
     | _ -> None) (* If either operand is None, return None *)
```

Problem 4: Structural transformation of Part 3

- a) Rewrite `lookup` to use CPS with two continuations. Observe that since `lookup` was already TR, the continuations in the CPS version of `lookup` don't change between recursive calls.

```
let rec lookup key lst sc fc =  
  match lst with  
  | [] -> fc () (* Key not found, invoke failure continuation *)  
  | (k, v) :: rest ->  
    if k = key then sc v (* Key found, invoke success continuation *)  
    else lookup_cps key rest sc fc (* Continue searching *)
```


b) Rewrite `eval` to use `mystery` from problem 1b instead of explicit pattern-matching.

```
let rec eval expr env =  
  match expr with  
  | Const c -> Some c  
  | Var v -> lookup v env  
  | Plus (e1, e2) ->  
    mystery  
      (fun v1 -> mystery (fun v2 -> Some (v1 +. v2)) (fun () -> None) (eval e2 env))  
      (fun () -> None)  
      (eval e1 env)  
  | Times (e1, e2) ->  
    mystery  
      (fun v1 -> mystery (fun v2 -> Some (v1 *. v2)) (fun () -> None) (eval e2 env))  
      (fun () -> None)  
      (eval e1 env)
```