- use half-point increments
- scheme: don't worry about parenthesis matching
- ocaml: don't worry about ending in ;;
- #tailrecursive: -1pt for non-tail-recursive answers

Question 1. [2+2]

```
let hd list = match list with
  | [] -> raise (Failure "hd")
  | x::_ -> x

let tl list = match list with
  | [] -> raise (Failure "tl")
  | _::xs -> xs
```

Allow anything that looks like a reasonable guess at raising an error.

Question 3. [3] #tailrecursive

```
let length list =
  let rec len lst acc = match lst with
    | [] -> acc
    | _::xs -> len xs (acc + 1)
  in len list 0
```

Question 4. [3] #tailrecursive

```
(define (length list)
  (define (len lst acc)
    (if (null? lst) acc
      (len (cdr lst) (+ acc 1))))
  (len list 0))
```

Question 6. [5] #tailrecursive

```
(define (maximum list)
  (define (max lst mx)
    (cond ((null? lst) mx)
          ((< mx (car lst)) (max (cdr lst) (car lst)))
          (else (max (cdr lst) mx))))
  (if (null? list) #f
    (max (cdr list) (car list))))
```

only the inner function needs to be tail recursive

Question 7. [5] #tailrecursive

```
let maximum list =
  let rec max lst mx = match lst with
    | []-> mx
    | x::xs -> max xs (if mx < x then x else mx)
  in match list with
    | [] -> None
    | x::xs -> Some (max xs x)
```

Be generous on None/Some.

Question 8. [3] #tailrecursive

```
let reverse list =
  let rec rev lst out = match lst with
    | [] -> out
    | x::xs -> rev xs (x::out)
  in rev list []
```

Question 9. [3] #tailrecursive

```
(define (reverse list)
  (define (rev lst out)
    (if (null? lst) out
        (rev (cdr lst) (cons (car lst) out))))
  (rev list '()))
```

Question 10. [4]

```
Object subclass: Counter [
  |count|
  Counter class >> new [ ^ super new init ]
  init [ count := 0 ]
  add: amt [ count := count + amt ]
  value [ ^ count ]
]
```

Question 11. [(1+1)×4]

- parametric / template / generic
  - `'template <typename T> class stack {...}`
  - `'a list -> 'b list -> 'c list`
  - `class stack<T> {...}`
- inclusion / inheritance / virtual functions
  - `Object subclass: Expr [...]`
  - `class foo: public bar {...}`
- conversion / coercion
  - `void f(double); ... f(3);`
  - `void f(const string&); ... f("abc");`
- overloading
  - `void f(double); void f(int) // C++`
  - `3+4; ... 3.0+4.0 // C`
  - Java examples are possible
  - no overloading is possible in Scheme, Ocaml, or Smalltalk.

Question 12. [3]

```
let rec zip list1 list2 = match list1, list2 with
    | [], _ -> []
    | _, [] -> []
    | x::xs, y::ys -> (x,y) :: zip xs ys
```