Problem 1

1. (8.1)
   a. twice(pred,1);
      result: val it = 0 : int
      Excpt(0) was raised when executing pred(0), the outter f. Expression is Excpt of int and parttern was also the excpt of int, so it returns x.
   b. Twice(dumb, 1);
      Result: val it = 1 : int
      Excpt(1) was raised when executing the inner f for f(f(x)).
   c. Twice(smart,0);
   d. Result: var it = 1: int
      Excpt(0) was raised when executing pred(0).

2. (8.2)
   hd(nil) raised an uncaught exception Hd, this uncaught exception hd is caught by g's handle Hd, so g(nil) retures nil regardless what tl(l) returns, the evaluation of tl(nil) is skipped.

3. (8.4)
   call f(5)
   call f(3)
   call f(1)
   raise exception Odd
   pop activation record of f(1) off stack
   pop activation record of f(3) off stack without returning control to the f(3)
   handle exception Odd in f(5)
   return -5 from f(5)
   return -5 from f(7)
   return -5 from f(9)
   return –5 from f(11)
   pop
4. (8.5)
   NO, we can't use tail recursion elimination to optimize the whole program, but we can optimize it partially.
   Tail recursion elimination depends on reuse the same activation record for each call to the function, but the exception handing expects to be able to pop activation records off the stack until a handler is found, so tail recursion elimination may not work for functions that establish handlers.
   To solve this problem, we can use un-optimized calls for f(0, count) and f(1, count), but optimized tail recursion calls for other numbers.
   Try to partially optimize:

   ```
   Exception OddNum;
   fun find(x) =
           let fun f(x, count) =
   ```

```
    if x = 0 then count else
        if x = 1 then raise OddNum else f(x-2, count+1)
in f(x, 0) handle OddNum=> -1
end;
```