Exercise 1

Exercise 8.1 in Mitchell's book

```
exception Excpt of int;
fun twice(f,x) = f(f(x)) handle Excpt(x) => x;
fun pred(x) = if x=0 then raise Excpt(x) else x-1;
fun dumb(x) = raise Excpt(x);
fun smart(x) = 1+pred(x) handle Excpt(x)=>1;
```

What is the result of evaluating each of the following expressions?

a) twice(pred,1);

val it = 0 : int   Second call to pred raises exception which is handled in twice

b) twice(dumb,1);

val it = 1 : int   First call to dumb raises exception, once again handled in twice

c) twice(smart,0);

val it = 1 : int   First call of pred raises exception, but the exception is handled in smart
Therefore the call is able to continue executing. Smart indeed

Exercise 2

Exercise 8.3 in Mitchell's book

```
datatype 'a tree =
      Leaf of 'a
       | Nd of ('a tree)*('a tree);

fun closest(x,Leaf(y)) = y:int
  | closest(x,Nd(y,z)) = let val lf = closest(x,y)
                             and rt = closest(x,z)
                         in
                            if abs(x-lf) < abs(x-rt) then
                               lf
                            else
                               rt
                         end;


fun closest(x,t) =
 let exception Found
     fun cls(x,Leaf(y)) = if x=y then raise Found else y:int
       | cls(x,Nd(y,z)) = let val lf=cls(x,y)
                              and rt=cls(x,z)
                          in
                             if abs(x-lf) < abs(x-rt) then lf else rt
                          end
 in
    cls(x,t) handle Found => x
 end;
```

```
Test examples:

- closest (5, Nd(Leaf(4), Nd(Leaf(1),
Nd(Leaf(5),Nd(Leaf(6),Leaf(7))))));
val it = 5 : int

- closest (5, Nd(Leaf(4), Nd(Leaf(1),
Nd(Leaf(50),Nd(Leaf(6),Leaf(7))))));
val it = 6 : int
```

a)

Q: "Explain why both give the same answer"

Same algorithm, the exception is only raised
if x == y. If not, y will be returned. In short,
a value equal to y is always returned

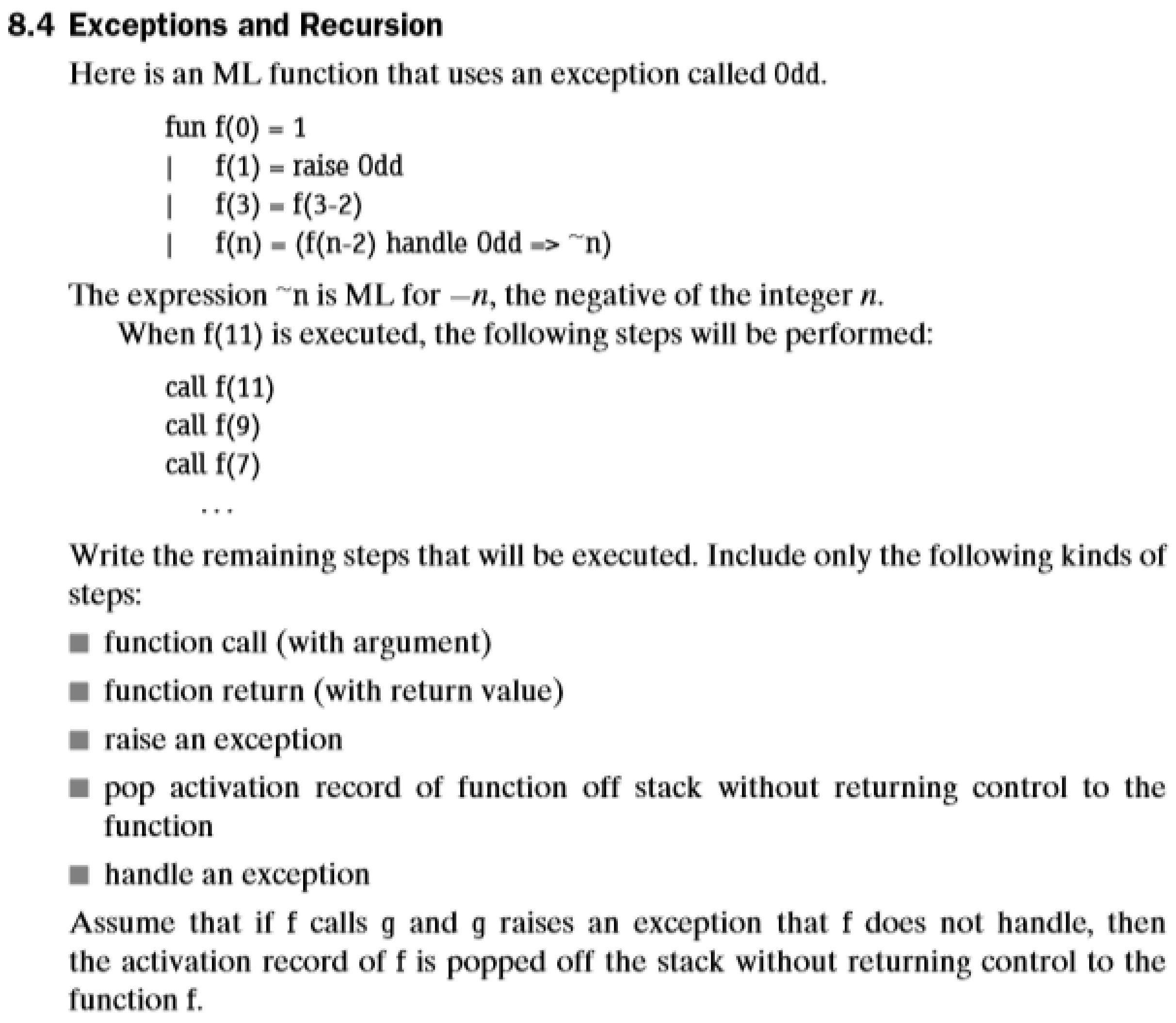
b)

Q: "Explain why the second version may be more efficient"

Two reasons:

1. If equal value is found, the recursion stops
It will never find a value closer to x

2. If equal value is found, an exception avoids
the normal backtracking through the run-time stack



8.4 Exceptions and Recursion

Here is an ML function that uses an exception called Odd.

```
fun f(0) = 1
|   f(1) = raise Odd
|   f(3) = f(3-2)
|   f(n) = (f(n-2) handle Odd => ~n)
```

The expression ~n is ML for $-n$, the negative of the integer n.

When f(11) is executed, the following steps will be performed:

```
call f(11)
call f(9)
call f(7)
   . . .
```

Write the remaining steps that will be executed. Include only the following kinds of steps:

- function call (with argument)
- function return (with return value)
- raise an exception
- pop activation record of function off stack without returning control to the function
- handle an exception

Assume that if f calls g and g raises an exception that f does not handle, then the activation record of f is popped off the stack without returning control to the function f.

f(11)	The 3 first recursions receive ~5 and are all John Snow
f(9)	
f(7)	
f(5)	Handles with negating of n
f(3)	Does not handle
f(1)	Does not handle
Exception raised	

Exercise 4

Exercise 8.7 in Mitchell's book

Q: An *exception* aborts part of a computation and transfers control to a handler that was established at some earlier point int he computation. A *memory leak* occurs when memory allocated by a program is no longer reachable, and the memory will not be deallocated. (The term "memory leak" is used only in connection with languages that are not garbage collected, such as C.) Explain why exceptions can lead to memory leaks in a language that is not garbage collected.

Call stack between raising and handling popped

May contain pointers to memory areas in the heap with no other pointers pointing to them

No way to reach said memory area and, as a consequence, no way to free the memory

But memory still in use and cannot be used for anything else

Exercise 5

(Taken from Paulson's book "ML for the working programmer")

Given a certain amount of money and a list of coin values, we would like to receive change using the largest coins possible. This is easy if the coins values are supplied in decreasing order. The following naive algorithm implements it:

```
fun change (coinvals, 0)          = []
  | change (c::coinvals, amount)
      = if amount < c then
           change(coinvals, amount)
        else
           c :: change(c::coinvals, amount-c);
```

The first argument is the list of valid coins and the second one is the given amount to be changed.

The algorithm returns the first way of making change: if the target amount is zero, no coins are required; if the largest coin value c is too large, discard it; otherwise use it and make change for the amount less c.

Notice that the algorithm is NOT EXHAUSTIVE
(you will get the following message: "Warning: match nonexhaustive").

The algorithm is less trivial than it seems. For the following cases

```
change([], 12);

change([5,2], 16);

change([5,2], 18);
```

it will give as an answer: "uncaught exception Match [nonexhaustive match failure]".
However, it works well in many other cases like the following:

```
- change([5,2], 12);
val it = [5,5,2] : int list

- change([5,2], 14);
val it = [5,5,2,2]: int list
```

```
exception Backtrack;
fun changeBack (coinvals, 0)          = []
  | changeBack (nil,_)                = raise Backtrack
  | changeBack (c::coinvals, amount)
      = if amount < 0 then
           raise Backtrack
        else
           c :: changeBack(c::coinvals, amount-c)
           handle Backtrack => changeBack(coinvals, amount);
```

```
changeBack([5,2], 16);

[5,2],16 -> 5::change([5,2],16-5)
[5,2],11 -> 5::change([5,2],11-5)
[5,2], 6 -> 5::change([5,2],6-5)
  [5,2], 1 -> change([2],1)
  [2]  , 1 -> change([],1)
  []   , 1 -> [1]
[2], 6   -> 2::change([2],6-2)
etc.
```

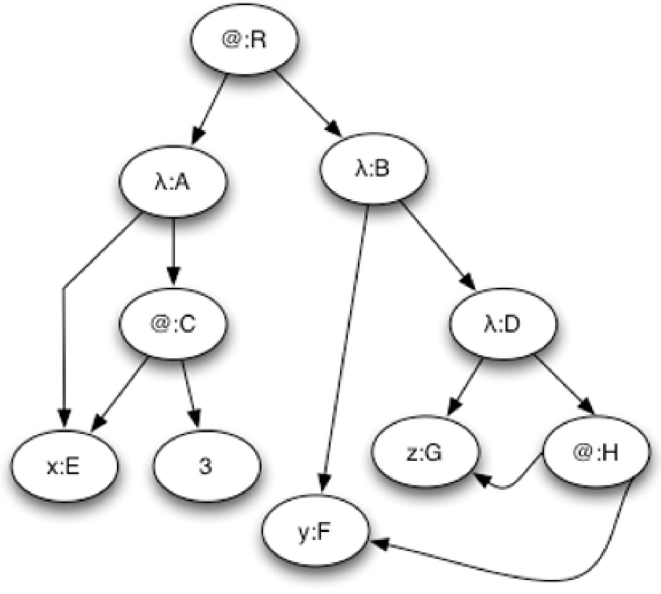
From the 2013 exam

2g

Use the following graph to apply the type inference algorithm to the expression, and give the type of the expression, or in the case of a type error, explain how you detect this.

```
(fn x => x 3) (fn y => fn z => z y)
```

Use the given node-labels (for better readability given in upper-case) and assume that the literal `3` has type `int` to derive the equations, and describe your steps.



Let’s do it together! :D

Just in case I mess up, the answer is R = (int-> H) -> H

From the 2012 exam

2 (b+c+d+e)

Given the two following datatypes, one for expression trees, and one for a stack-based language:

```
datatype exp =
    Const of int
  | Neg of exp
  | Add of (exp*exp);
```

```
datatype stacklang =
    Push of int
  | Op of (int list -> int list);
```

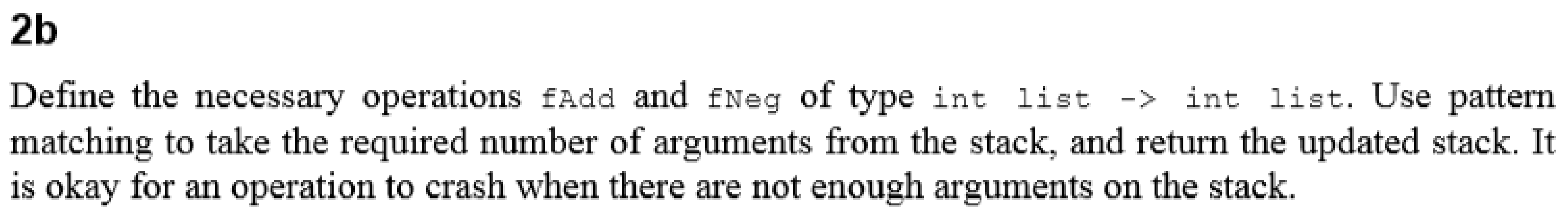
We can then for example write an expression resembling the term "~(3+5)":

```
val t = Neg (Add ((Const 3), (Const 5)) : exp
```

Instead of evaluating this term directly, we would like to convert it into its stack language-representation first, a list of stack-operations: all arguments to arithmetic operations must first be pushed onto the stack (here, a list of integers). An operation will remove as many elements from the top of the stack as it needs, and put the result back. For example, the above expression can be converted into:

```
val st = [Push 3, Push 5, Op fAdd, Op fNeg] : stacklang list
```

From the 2012 exam



2b

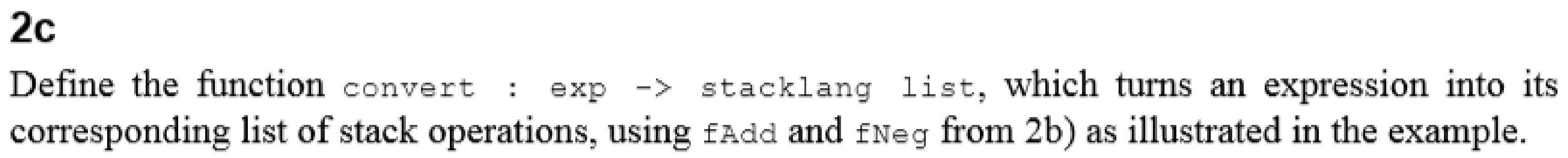
Define the necessary operations `fAdd` and `fNeg` of type `int list -> int list`. Use pattern matching to take the required number of arguments from the stack, and return the updated stack. It is okay for an operation to crash when there are not enough arguments on the stack.

So what exactly is fAdd supposed to do?

Just take the first two elements from a list, add them and put the new value back in front

How about fNeg?

Just take the first element from a list, negate it, and put the negated value back in front



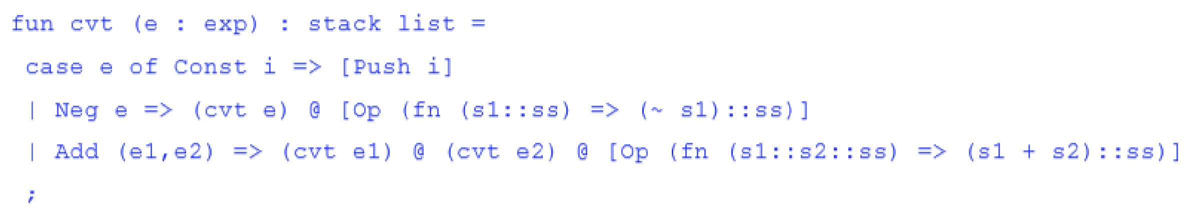
2c

Define the function `convert : exp -> stacklang list`, which turns an expression into its corresponding list of stack operations, using `fAdd` and `fNeg` from 2b) as illustrated in the example.

So, as the text says, we need to make a function that takes AN exp and returns the corresponding stacklang list.

Divide the problem into many small, but manageable pieces.

Code given as solution (including 2b):



```
fun cvt (e : exp) : stack list =
 case e of Const i => [Push i]
 | Neg e => (cvt e) @ [Op (fn (s1::ss) => (~ s1)::ss)]
 | Add (e1,e2) => (cvt e1) @ (cvt e2) @ [Op (fn (s1::s2::ss) => (s1 + s2)::ss)]
 ;
```

Personally I prefer pattern matching, so something like this:

```
fun convert(Const(i)) = [Push(i)] |
    convert(Neg(e)) = convert(e) @ [Op(fNeg)] |
    convert(Add(e1, e2)) = convert(e1) @ convert(e2) @ [Op(fAdd)];
```