Weekly exercises in INF3110 - ML IV
week 42 13-17.10.2008

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**

pred is applied twice giving 0 the first time, raising an exception the second time, which is handled in the declaration of the function twice, giving 0 as a result.

b) twice(dumb,1);

val it = 1 : **int**

The call to dumb just throw the exception handled in the declaration of twice (giving 1, which is the argument to dumb).

c) twice(smart,0);

val it = 1 : **int**

The call will produce smart(smart(0)); the inner call will produce 1 + pred(0) which will raise an exception. The hanlder used is the last defined, in the run-time stack, i.e. the one defined in the smart function. Hence, the result will be 1.

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"

A: The "Found" handler is only raised if x=y, and then it returns the first argument, which is equal to y. If x=/=y, then it returns y. So it will allways return a value equal to y.

b)
Q: "Explain why the second version may be more efficient"
A: The second function is more efficient since throwing an exception when a Leaf with equal value to the first (integer) argument is found empties the run-time stack with all the pending calls. In the first definition of closest a backtracking has to be performed.

Exercise 3

Exercise 8.4 in Mitchell's book

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

```
f(11)
f(9)
f(7)
f(5), handle Odd => ~5
f(3)
f(1)
Odd is raised
pop activation record of call f(1)
pop activation record of call f(3)
handle the exception with value n=5
~5 is returned
```

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.

A Short:
Because of pointers to allocated memory which dissapears when the exception is called.

A Long:
Exception may lead to memory leaks when no garbage collection is used because of the dynamic nature of the exception handling mechanism. All the references (to the heap) in the call stack in between the raising and the handling of an exception are popped out, so the memory previously pointed by these references is not reachable anymore. Since the memory is not garbage collected, such space cannot be reused: the memory manager "believes" the memory is still being used.

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
```

The reason is that the algorithm is greedy, trying to express the
given amount with the largest coin first and then trying with the
rest of the coins (for instance, in "change([5,2], 16)", it tries
5,5,5 and then 1, which is not a valid coin).

In order to design a better algorithm we need "backtracking":
responding to failure by undoing the most recent choice and trying again.
One way of implementing backtracking is by using exceptions.

Write a function `"changeBack : int list * int -> int list"` in
ML which implements a backtracking algorithm for solving the above
problem using exceptions (define "exception Change;").

These are some examples of expected results:

```
- changeBack([5,2], 17);
val it = [5,5,5,2] : int list

- changeBack([], 12);
uncaught exception Change

- changeBack([5,2], 16);
val it = [5,5,2,2,2] : int list
```

<u>v1</u>

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

<u>v2</u>

```
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);
```

<u>What happens for v2:</u>

```
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.
```