Homework 2
COSE212, Fall 2016

Hakjoo Oh

Due: 10/14, 24:00

Problem 1 Write two functions

```
max: int list -> int
min: int list -> int
```

that find maximum and minimum elements of a given list, respectively. For example `max [1;3;5;2]` should evaluate to 5 and `min [1;3;2]` should be 1. Assume that the input list is non-empty. (Hint: Use `fold`.)

Problem 2 Write the function `filter`

```
filter : ('a -> bool) -> 'a list -> 'a list
```

Given a predicate `p` and a list `l`, `filter p l` returns all the elements of the list l that satisfy the predicate p. The order of the elements in the input list is preserved. For example,

```
# filter (fun x -> x mod 2 = 0) [1;2;3;4;5];;
- : int list = [2; 4]
# filter (fun x -> x > 0) [5;-1;0;2;-9];;
- : int list = [5; 2]
# filter (fun x -> x * x > 25) [1;2;3;4;5;6;7;8];;
- : int list = [6; 7; 8]
```

Problem 3 Write a function

```
double: ('a -> 'a) -> 'a -> 'a
```

that takes a function of one argument as argument and returns a function that applies the original function twice. For example,

```
# let inc x = x + 1;;
val inc : int -> int = <fun>
# let mul x = x * 2;;
val mul : int -> int = <fun>
# (double inc) 1;;
```

```
- : int = 3
# ((double double) inc) 0;;
- : int = 4
# ((double (double double)) inc) 5;;
- : int = 21
# (double mul) 1;;
- : int = 4
# (double double) mul 2;;
- : int = 32
```

Problem 4 Binary trees can be defined as follows:

```
type btree =
  Empty
 |Node of int * btree * btree
```

For example, the following `t1` and `t2`

```
let t1 = Node (1, Empty, Empty)
let t2 = Node (1, Node (2, Empty, Empty), Node (3, Empty, Empty))
```

are binary trees. Write the function

```
mem: int -> btree -> bool
```

that checks whether a given integer is in the tree or not. For example,

```
mem 1 t1
```

evaluates to *true*, and

```
mem 4 t2
```

evaluates to *false*.

Problem 5 Natural numbers can be defined as follows:

```
type nat = ZERO | SUCC of nat
```

For instance, `SUCC ZERO` denotes 1 and `SUCC (SUCC ZERO)` denotes 2. Write two functions that add and multiply natural numbers:

```
natadd : nat -> nat -> nat
natmul : nat -> nat -> nat
```

For example,

```
# let two = SUCC (SUCC ZERO);;
val two : nat = SUCC (SUCC ZERO)
# let three = SUCC (SUCC (SUCC ZERO));;
val three : nat = SUCC (SUCC (SUCC ZERO))
# natmul two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC (SUCC ZERO)))))
# natadd two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC ZERO))))
```

Problem 6 Consider the following propositional formula:

```
type formula =
 | True
 | False
 | Not of formula
 | AndAlso of formula * formula
 | OrElse of formula * formula
 | Imply of formula * formula
 | Equal of exp * exp
and exp =
 | Num of int
 | Plus of exp * exp
 | Minus of exp * exp
```

Write the function

```
eval : formula -> bool
```

that computes the truth value of a given formula. For example,

```
eval (Imply (Imply (True,False), True))
```

evaluates to *true*, and

```
eval (Equal (Num 1, Plus (Num 1, Num 2)))
```

evaluates to *false*.