Homework 1
COSE212, Fall 2022

Hakjoo Oh

Due: 9/26, 23:59

Academic Integrity / Assignment Policy

- All assignments must be your own work.
- Discussion with fellow students is encouraged including how to approach the problem. However, your code must be your own.
  - Discussion must be limited to general discussion and must not involve details of how to write code.
  - You must write your code by yourself and must not look at someone else's code (including ones on the web).
  - Do not allow other students to copy your code.
  - Do not post your code on the public web.
- **Violating above rules gets you 0 points for the entire HW score.**

Problem 1 Consider the following triangle (it is called Pascal's triangle):

$$
\begin{array}{ccccccc}
 & & & 1 & & & \\
 & & 1 & & 1 & & \\
 & 1 & & 2 & & 1 & \\
1 & & 3 & & 3 & & 1 \\
 & \cdots & & & & &
\end{array}
$$

where the numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it. Write a function

```
pascal: int * int -> int
```

that computes elements of Pascal's triangle. For example, `pascal` should behave

as follows:

```
pascal (0,0) = 1
pascal (1,0) = 1
pascal (1,1) = 1
pascal (2,1) = 2
pascal (4,2) = 6
```

Problem 2 Write a function

```
prime: int -> bool
```

that checks whether a number is prime (n is prime if and only if n is its own smallest divisor except for 1). For example,

```
prime 2 = true
prime 3 = true
prime 4 = false
prime 17 = true
```

Problem 3 Write a function

```
range : int -> int -> int list
```

that takes two integers n and m, and creates a list of integers from n to m. For example, `range 3 7` produces `[3;4;5;6;7]`. When $n > m$, an empty list is returned. For example, `range 5 4` produces `[]`.

Problem 4 Write a function

```
suml: int list list -> int
```

which takes a list of lists of integers and sums the integers included in all the lists. For example, `suml [[1;2;3]; []; [-1; 5; 2]; [7]]` produces 19.

Problem 5 Write a function

```
lst2int : int list -> int
```

which converts a list of integers to an integer. For example;

```
lst2int [2;3;4;5] = 2345.
```

Problem 6 Define the function `binarize`:

```
binarize: int -> int list
```

that converts a decimal number to its binary representation. For example,

```
binarize 2 = [1; 0]
binarize 3 = [1; 1]
binarize 8 = [1; 0; 0; 0]
binarize 17 = [1; 0; 0; 0; 1]
```

Problem 7 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.

Problem 8 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 9 Consider the inductive definition of binary trees:

$$\overline{n}\ n \in \mathbb{Z} \qquad \frac{t}{(t, \mathbf{nil})} \qquad \frac{t}{(\mathbf{nil}, t)} \qquad \frac{t_1 \qquad t_2}{(t_1, t_2)}$$

which can be defined in OCaml as follows:

```
type btree =
  | Leaf of int
  | Left of btree
  | Right of btree
  | LeftRight of btree * btree
```

For example, binary tree $((1, 2), \mathbf{nil})$ is represented by

```
Left (LeftRight (Leaf 1, Leaf 2))
```

Write a function that exchanges the left and right subtrees all the ways down. For example, mirroring the tree $((1, 2), \mathbf{nil})$ produces $(\mathbf{nil}, (2, 1))$; that is,

```
mirror (Left (LeftRight (Leaf 1, Leaf 2)))
```

evaluates to

```
Right (LeftRight (Leaf 2, Leaf 1)).
```

Problem 10 Natural numbers are defined inductively:

$$\frac{}{0} \qquad \frac{n}{n+1}$$

In OCaml, the inductive definition can be defined by the following a data type:

```
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 11 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

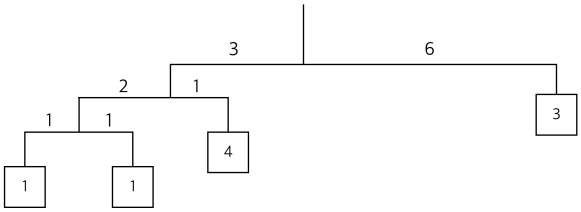
```
eval (Equal (Num 1, Plus (Num 1, Num 2)))
```

evaluates to *false*.

Problem 12 A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. In OCaml datatype, a binary mobile can be defined as follows:

```
type mobile = branch * branch     (* left and rigth branches *)
and branch = SimpleBranch of length * weight
           | CompoundBranch of length * mobile
and length = int
and weight = int
```

A branch is either a simple branch, which is constructed from a length together with a weight, or a compound branch, which is constructed from a length together with another mobile. For instance, the mobile



is represented by the following:

```
(CompoundBranch (3,
  (CompoundBranch (2, (SimpleBranch (1, 1), SimpleBranch (1, 1))),
   SimpleBranch (1, 4))),
 SimpleBranch (6, 3))
```

Define the function

```
balanced : mobile -> bool
```

that tests whether a binary mobile is balanced. A mobile is said to be *balanced* if the torque applied by its top-left branch is equal to that applied by its top-right branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. For example, the example mobile above is balanced.