Homework 1
COSE212, Fall 2019

Hakjoo Oh

Due: 9/30, 24:00

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):

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \\ & & & & \cdots & & & & \end{array}$$

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

```
dfact : int -> int
```

that computes double-factorials. Given a non-negative integer n, its double-factorial, denoted $n!!$, is the product of all the integers of the same parity as n from 1 to n. That is, when n is even

$$n!! = \prod_{k=1}^{n/2}(2k) = n \cdot (n-2) \cdot (n-4) \cdots 4 \cdot 2$$

and when n is odd,

$$n!! = \prod_{k=1}^{(n+1)/2}(2k-1) = n \cdot (n-2) \cdot (n-4) \cdots 3 \cdot 1$$

For example, $7!! = 1 \times 3 \times 5 \times 7 = 105$ and $6!! = 2 * 4 * 6 = 48$.

Problem 4 Consider the task of computing the exponential of a given number. We would like to write a function that takes as arguments a base b and a positive integer exponent n to compute b^n. Read the remaining problem description carefully and devise an algorithm that has time complexity of $\Theta(\log n)$.

One simple way to implement the function is via the following recursive definition:

$$\begin{array}{rcl} b^0 & = & 1 \\ b^n & = & b \cdot b^{n-1} \end{array}$$

which translates into the OCaml code:

```
let rec expt b n =
  if n = 0 then 1
  else b * (expt b (n-1))
```

However, this algorithm is slow; it takes $\Theta(n)$ steps.

We can improve the algorithm by using successive squaring. For instance, rather than computing b^8 as

$$b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b))))))$$

we can compute it using three multiplications as follows:

$$\begin{array}{rcl} b^2 & = & b \cdot b \\ b^4 & = & b^2 \cdot b^2 \\ b^8 & = & b^4 \cdot b^4 \end{array}$$

This method works only for exponents that are powers of 2. We can generalize the idea via the following recursive rules:

$$\begin{array}{rcll} b^n & = & (b^{n/2})^2 & \text{if } n \text{ is even} \\ b^n & = & b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{array}$$

Use the rules to write a function `fastexpt` that computes exponentials in $\Theta(\log n)$ steps:

```
fastexpt: int -> int -> int
```

Problem 5 Define the function `iter`:

```
iter : int * (int -> int) -> (int -> int)
```

such that

$$\texttt{iter}(n, f) = \underbrace{f \circ \cdots \circ f}_{n}.$$

When $n = 0$, $\texttt{iter}(n, f)$ is defined to be the identity function. When $n > 0$, $\texttt{iter}(n, f)$ is the function that applies f repeatedly n times. For instance,

$$\texttt{iter(}n\texttt{, fun x -> 2+x) 0}$$

evaluates to $2 \times n$.

Problem 6 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 7 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 8 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 9 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 10 Write a higher-order function

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

which removes elements of a list while they satisfy a predicate. For example,

```
drop (fun x -> x mod 2 = 1) [1;3;5;6;7]
```

evaluates to `[6;7]` and

```
drop (fun x-> x > 5) [1;3;7]
```

evaluates to `[1;3;7]`.