Homework 1: OCaml Exercises
COSE312, Spring 2023

Hakjoo Oh

Due: 4/2, 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):

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

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

```
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:

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

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 Write a higher-order function

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

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

```
dropWhile (fun x -> x mod 2 = 0) [2;4;7;9]
```

evaluates to `[7;9]` and

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

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

Problem 13 Write a higher-order function

```
sigma : (int -> int) -> int -> int -> int
```

such that `sigma f a b` computes

$$\sum_{i=a}^{b} f(i).$$

For instance,

```
sigma (fun x -> x) 1 10
```

evaulates to 55 and

```
sigma (fun x -> x*x) 1 7
```

evaluates to 140.

Problem 14 Write a higher-order function

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

which decides if all elements of a list satisfy a predicate. For example,

```
forall (fun x -> x mod 2 = 0) [1;2;3]
```

evaluates to false while

```
forall (fun x -> x > 5) [7;8;9]
```

is true.

Problem 15 Write a function

```
uniq: 'a list -> 'a list
```

which removes duplicated elements from a given list so that the list contains unique elements. For instance,

```
uniq [5;6;5;4] = [5;6;4]
```

Problem 16 In class, we defined the function `reverse` as follows:

```
let rec reverse l =
  match l with
  | [] -> []
  | hd::tl -> (reverse tl) @ [hd]
```

The function is slow; its time complexity is $O(n^2)$. For instance, `reverse (range 1 100000)` may not terminate quickly on typical machines. However, list reversal can be implemented efficiently with time complexity $O(n)$. Write a function

```
fastrev : 'a list -> 'a list
```

that reverses a given list with in $O(n)$. For instance, `fastrev (range 1 100000)` should produce `[100000; 99999; ...; 1]` immediately.

Problem 17 Write a function

```
diff : aexp * string -> aexp
```

that differentiates the given algebraic expression with respect to the variable given as the second argument. The algebraic expression `aexp` is defined as follows:

```
type aexp =
 | Const of int
 | Var of string
 | Power of string * int
 | Times of aexp list
 | Sum of aexp list
```

For example, $x^2 + 2x + 1$ is represented by

```
Sum [Power ("x", 2); Times [Const 2; Var "x"]; Const 1]
```

and differentiating it (w.r.t. "x") gives $2x + 2$, which can be represented by

```
Sum [Times [Const 2; Var "x"]; Const 2]
```

Note that the representation of $2x + 2$ in `aexp` is not unique. For instance, the following also represents $2x + 2$:

```
Sum
 [Times [Const 2; Power ("x", 1)];
  Sum
   [Times [Const 0; Var "x"];
    Times [Const 2; Sum [Times [Const 1]; Times [Var "x"; Const 0]]]];
  Const 0]
```

Problem 18 Consider the following expressions:

```
type exp = X
         | INT of int
         | ADD of exp * exp
         | SUB of exp * exp
         | MUL of exp * exp
         | DIV of exp * exp
         | SIGMA of exp * exp * exp
```

Implement a calculator for the expressions:

```
calculator : exp -> int
```

For instance,

$$\sum_{x=1}^{10}(x * x - 1)$$

is represented by

```
SIGMA(INT 1, INT 10, SUB(MUL(X, X), INT 1))
```

and evaluating it should give 375.

Problem 19 Consider the following language:

```
type exp = V of var
         | P of var * exp
         | C of exp * exp
and var = string
```

In this language, a program is simply a variable, a procedure, or a procedure call. Write a checker function

```
check : exp -> bool
```

that checks if a given program is well-formed. A program is said to be *well-formed* if and only if the program does not contain free variables; i.e., every variable name is bound by some procedure that encompasses the variable. For example, well-formed programs are:

- `P ("a", V "a")`
- `P ("a", P ("a", V "a"))`
- `P ("a", P ("b", C (V "a", V "b")))`
- `P ("a", C (V "a", P ("b", V "a")))`

Ill-formed ones are:

- `P ("a", V "b")`

- `P ("a", C (V "a", P ("b", V "c")))`
- `P ("a", P ("b", C (V "a", V "c")))`

Problem 20 Re-define the following functions using `fold_right` and `fold_left`.

1. ```
   let rec length l =
     match l with
     [] -> 0
     |h::t -> 1 + length t
   ```
2. ```
   let rec reverse l =
     match l with
     | [] -> []
     | hd::tl -> (reverse tl)@[hd]
   ```
3. ```
   let rec is_all_pos l =
   match l with
   | [] -> true
   | hd::tl -> (hd > 0) && (is_all_pos tl)
   ```
4. ```
   let rec map f l =
     match l with
     | [] -> []
     | hd::tl -> (f hd)::(map f tl)
   ```
5. ```
   let rec filter p l =
     match l with
     | [] -> []
     | hd::tl ->
       if p hd then hd::(filter p tl)
       else filter p tl
   ```
   ```