Homework 2
COSE212, Fall 2019

Hakjoo Oh

Due: 10/12, 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 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 2 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 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 inc) 2;;
- : int = 4
# ((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 Write a function

```
app: 'a list ->'a list -> 'a list
```

which appends the first list to the second list while removing duplicated elements. For instance, given two lists `[4;5;6;7]` and `[1;2;3;4]`, the function should output `[1;2;3;4;5;6;7]`:

```
app [4;5;6;7] [1;2;3;4] = [1;2;3;4;5;6;7].
```

Problem 5 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 6 Write a function `reduce` of the type:

```
reduce : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
```

Given a function `f` of type `'a -> 'b -> 'c -> 'c`, the expression

```
reduce f [x1;x2;...;xn] [y1;y2;...;yn] c1
```

evaluates to `f xn yn (... (f x2 y2 (f x1 y1 c1))...)`. For example,

```
reduce (fun x y z -> x * y + z) [1;2;3] [0;1;2] 0
```

evaluates to 8.

Problem 7 Write a function

```
zipper: int list * int list -> int list
```

which receives two lists a and b as arguments and combines the two lists by inserting the ith element of a before the ith element of b. If b does not have an ith element, append the excess elements of a in order. For example,

```
# zipper ([1;3;5],[2;4;6]);;
- : int list = [1; 2; 3; 4; 5; 6]
# zipper ([1;3],[2;4;6;8]);;
- : int list = [1; 2; 3; 4; 6; 8]
# zipper ([1;3;5;7],[2;4]);;
- : int list = [1; 2; 3; 4; 5; 7]
```

Problem 8 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 9 Binary numerals can be represented by lists of 0 and 1:

```
type digit = ZERO | ONE
type bin = digit list
```

For example, the binary representations of 11 and 30 are

```
[ONE;ZERO;ONE;ONE]
```

and

```
[ONE;ONE;ONE;ONE;ZERO],
```

respectively. Write a function

```
bmul: bin -> bin -> bin
```

that computes the binary product. For example,

```
bmul [ONE;ZERO;ONE;ONE] [ONE;ONE;ONE;ONE;ZERO]
```

evaluates to `[ONE;ZERO;ONE;ZERO;ZERO;ONE;ZERO;ONE;ZERO]`.

Problem 10 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 11 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 12 Consider the following language:

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

In this language[1], 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")))`

[1] Called "lambda calculus" (see `https://en.wikipedia.org/wiki/Lambda_calculus`)

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")))`