Homework 1: OCaml Exercises
COSE312, Spring 2025

Hakjoo Oh

Due: 3/28, 23:59

Problem 1 Write a function

```
smallest_divisor: int -> int
```

that finds the smallest integral divisor (greater than 1) of a given number n. For example,

```
smallest_divisor 15 = 3
smallest_divisor 121 = 11
smallest_divisor 141 = 3
smallest_divisor 199 = 199
```

Ensure that your algorithm runs in $\Theta(\sqrt{n})$ steps.

Problem 2 Write a higher-order function

```
product : (int -> int) -> int -> int -> int
```

such that `product f a b` computes

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

For instance,

```
product (fun x -> x) 1 5
```

evaulates to 120. In general, we can use `product` to define the factorial function:

```
fact n = product (fun x -> x) 1 n
```

Problem 3 Define the function `iter`:

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

such that

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

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

```
iter(n, fun x -> 2+x) 0
```

evaluates to $2 \times n$.

Problem 4 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 5 Write a function `drop`:

```
drop : 'a list -> int -> 'a list
```

that takes a list l and an integer n to take all but the first n elements of l. For example,

```
drop [1;2;3;4;5] 2 = [3; 4; 5]
drop [1;2] 3 = []
drop ["C"; "Java"; "OCaml"] 2 = ["OCaml"]
```

Problem 6 Write a function

```
zip: 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,

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

Problem 7 Write a function

```
unzip: ('a * 'b) list -> 'a list * 'b list
```

that converts a list of pairs to a pair of lists. For example,

```
unzip [(1,"one");(2,"two");(3,"three")] = ([1;2;3],["one";"two";"three"])
```

Problem 8 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 9 Write a function

```
concat: 'a list list -> 'a list
```

which makes a list consisting of all the elements of a list of lists. For example,

```
concat [[1;2];[3;4;5]] = [1;2;3;4;5]
```

Problem 10 Write a function

```
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 11 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 12 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 13 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]
```

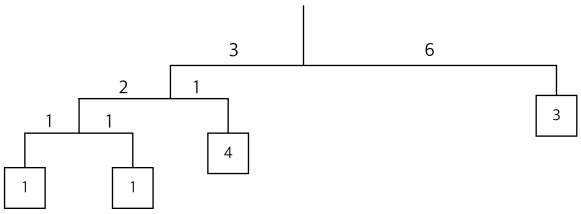
Note that the representation of $2x + 2$ in `aexp` is not unique. For instance, the following also represents $2x + 2$:

```
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 14 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.

Problem 15 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.