Problem Set
CVO 103, Spring 2018

Hakjoo Oh

Due: 06/12 (in class)

Problem 1 The Fibonacci numbers can be defined as follows:

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

Write in OCaml the function

```
fib: int -> int
```

that computes the Fibonacci numbers.

Problem 2 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 3 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:

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

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:

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

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

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

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

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

Problem 4 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 5 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). For example,

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

Problem 6 Write a 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 7 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 8 Use `product` to define 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 9 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,

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

evaluates to $2 \times n$.

Problem 10 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 11 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 12 Write the function `filter`

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

Given a predicate `p` and a list `l`, `filter p l` returns all the elements of the list l that satisfy the predicate p. The order of the elements in the input list is preserved. For example,

```
# filter (fun x -> x mod 2 = 0) [1;2;3;4;5];;
- : int list = [2; 4]
# filter (fun x -> x > 0) [5;-1;0;2;-9];;
- : int list = [5; 2]
# filter (fun x -> x * x > 25) [1;2;3;4;5;6;7;8];;
- : int list = [6; 7; 8]
```

Problem 13 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 14 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 15 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 16 We can define the propositional formula as follows:

```
type formula =
    True
  | False
  | Neg of formula
  | Or of formula * formula
  | And of formula * formula
  | Imply of formula * formula
  | Equiv of formula * formula
```

1. Write a function

   ```
   eval : formula -> bool
   ```

   that evaluates a given propositional formula.

2. Write a function

   ```
   eval : formula -> formula list
   ```

   that computes the set of all subformulas of a given propositoinal formula.

Problem 17 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 18 Natural numbers can be defined as follows:

```
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 19 Let us define simple arithmetic expressions:

```
type exp =
 Const of int
|Minus of exp
|Plus of exp * exp
|Mult of exp * exp
```

Write the function

```
calc: exp -> int
```

that computes the value of expressions. For example,

```
calc (Plus (Const 1, Const 2))
```

evaluates to 3.

Problem 20 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 21 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

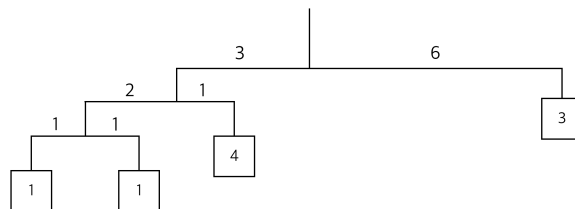
```
mem 4 t2
```

evaluates to *false*.

Problem 22 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 23 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 24 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")))`