# OCaml Exercises

Hakjoo Oh
Korea University

**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** 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** 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 4** 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 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** 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 7** 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 8** 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 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 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 11** 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 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 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 $i$th element of $a$ before the $i$th element of $b$. If $b$ does not have an $i$th 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 14** 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 15** 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 16** 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 17** 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 18** Consider the inductive definition of binary trees:

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

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 19** 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 20** 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.