EC3204: Programming Languages and Compilers (Fall 2023)
Homework 1: Functional Programming in OCaml

100 points in total, 5% of the total score
Due: 9/23, 23:59 (submit via GIST LMS)

Instructor: Sunbeom So

Important Notes

- **Evaluation criteria**
  For each problem, the correctness of your code will be evaluated using testcases:

$$\frac{\#\text{Passed}}{\#\text{Total}} \times \text{point per problem}$$

  "Total" indicates a set of testcases prepared by the instructor (undisclosed before the evaluation), and "Passed" indicates testcases whose expected outputs match with the outputs produced by your implementations.

- **No Plagiarism**
  Cheating (i.e., copying assignments by any means) will get you 0 points for the entire HWs. See the slides for Lecture 0. Code-clone checking will be conducted irregularly.

- **Executable**
  Before you submit your code, please make sure that your code can be successfully executed by the OCaml interpreter. That is, the command `ocaml yourfile.ml` should not report any errors. Otherwise, you will get 0 points for that HW.

- **No Template Changes**
  For each problem, your job is to complete the "`(* TODO *)`" parts in provided templates. You can define new helper functions for it. However, you should not modify existing code templates such as variant type declarations.

- **No Printing Functions**
  The submitted files should not contain printing functions such as `print_int`.

- **File Extension**
  The submitted files should have `.ml` extensions, not the others (e.g., `.pdf`).

- **File Naming Rule**
  Submit 8 files in total: `p1_range.ml`, $\cdots$, `p8_reach.ml`. The files should contain your implementations for problems 1–8 respectively. Do not change the file names. Do not zip the files.

Problem 1 (10pt) Implement 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` should return `[3;4;5;6;7]`. If $n > m$, the empty list should be returned. For example, `range 5 4` should return `[]`.

Problem 2 (10pt) Implement a function

```
double: ('a -> 'a) -> 'a -> 'a
```

that takes a function as an argument and returns a function that applies the passed function twice. For example,

```
# use "yourfile.ml";;
...
# let inc x = x + 1;;
val inc : int -> int = <fun>
# (double inc) 1;;
- : int = 3
# ((double double) inc) 0;;
- : int = 4
# ((double (double double)) inc) 5;;
- : int = 21
```

Problem 3 (10pt) The set of the natural numbers can be inductively defined as follows:

$$n \rightarrow 0 \mid n+1$$

In OCaml, the equivalent set can be represented using a variant type declaration:

```
type nat = ZERO | SUCC of nat
```

where `SUCC n` indicates a successor of n. For example, `SUCC ZERO` and `SUCC (SUCC ZERO)` mean 1 and 2, respectively.

Your job is to implement two functions that add and multiply natural numbers:

```
natadd : nat -> nat -> nat
natmul : nat -> nat -> nat
```

For example,

```
# use "yourfile.ml";;
...
# let two = SUCC (SUCC ZERO);;
val two : nat = SUCC (SUCC ZERO)
# let three = SUCC (SUCC (SUCC ZERO));;
val three : nat = SUCC (SUCC (SUCC ZERO))
```

```
# natadd two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC ZERO))))
# natmul two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC (SUCC ZERO)))))
```

Problem 4 (10pt) 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
```

Your job is to implement a function

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

that computes the truth value of a given formula. For example,

```
let f1 = True in
let f2 = Equal (Num 1, Plus (Num 1, Num 2)) in
eval (Imply (Imply (f1, f2), True))
```

should output *true*.

Problem 5 (10pt) Implement a higher-order function

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

such that `sigma f a b` computes

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

For example,

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

should evaluate to 55, and

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

should evaluate to 140. If a $>$ b, raise `InvalidInput` exception.

Problem 6 (15pt) Implement 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, $x^2 + 2x + 1$ can be represented by

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

If we differentiate it with "x", we obtain $2x + 2$ that can be represented by

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

Here, note that the representation for $2x + 2$ is not unique. For example, $2x + 2$ can be also represented by

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

Both of the representations will be considered correct, because they are semantically equivalent despite their syntactic difference.

Problem 7 (15pt) 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 example,

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

is represented by

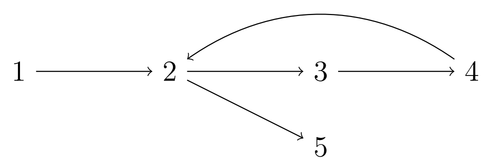
```
SIGMA (INT 1, INT 10, SUB (MUL (X, X), INT 1))
```

and evaluating it using `calculator` should output 375.

Problem 8 (20pt) A directed graph can be represented as follows:

```
type graph = (vertex * vertex) list
and vertex = int
```

For example, the following graph



is represented by `[(1,2);(2,3);(3,4);(4,2);(2,5)]`.

Your job is to implement a function

```
reach : graph * vertex -> vertex list
```

that returns the set of vertices that are reachable from the vertex given as the second argument. For example,

```
reach ([(1,2);(2,3);(3,4);(4,2);(2,5)], 1) = [1;2;3;4;5]
reach ([(1,2);(2,3);(3,4);(4,2);(2,5)], 2) = [2;3;4;5]
reach ([(1,2);(2,3);(3,4);(4,2);(2,5)], 3) = [2;3;4;5]
reach ([(1,2);(2,3);(3,4);(4,2);(2,5)], 4) = [2;3;4;5]
reach ([(1,2);(2,3);(3,4);(4,2);(2,5)], 5) = [5]
```