CSE216
Foundations of Computer Science

State University of New York, Korea

Review on types

Taken from

https://courses.grainger.illinois.edu/cs421/su2009/exams/mt-sampleqns.pdf

Thanks!

1. Give the types of each of the following Ocaml functions:

(a)
```
let alwaysfour x = 4
```

(b)
```
let add x y = x + y
```

(c)
```
let concat x y = x ^ y
```

(d)
```
let addmult x y = (x + y, x * y)
```

```
(e) let rec f x = if x=[] then [] else hd x @ f (tl x)

(f) let rec copy x = if x=[] then [] else hd x :: copy (tl x)

(g) let b (x,y) = x+y

(h) let c (x,y) = x
```

```
(i) let d x = match x with (a,b) -> a
```

```
(j) let e x = hd x + 1
```

```
(k) let f x y = match x with
        [] -> 0 | a::b -> a+y
```

```
(l) let g (a,b) (c,d) = (a+d, b^c)
```

(Recall that ^ is the string concatenation operation.)

```
(m) let rec h x = match x with
          [] -> 0 | (a,b)::r -> a + (h r)
```

Review on Algebraic Datatype

Taken from SBU cse216. Thanks!

1. Think of a binary tree to represent basic arithmetic operations. In such a tree, the leaf nodes will be of a numeric type while the internal non-leaf nodes will hold the arithmetic operations of addition, subtraction, multiplication, and division. This means, we need to define a binary tree data type over two distinct data types. Define such a binary tree in OCaml.

2. Next, write down a function that takes a binary tree of the type you defined, and returns a tuple with its first element being a list of operators, and its second element being a list of the numeric values in the nodes. That is, the function's type should be

   ```
   val function name : ('a, 'b) tree -> 'b list * 'a list = <fun>
   ```

   where 'a is the operator's type and 'b is the numeric type.

3. Suppose we define the binary tree data type as follows:

```
type 'a binary_tree =
      | Empty
      | Node of 'a * 'a binary_tree * 'a binary_tree;;
```

   a) Write a function called **num_of_leaves** to count the number of leaves in a binary tree.
   b) Write a function called **get_all_leaves** to collect all the leaves in a list.

Additional review exercises on algebraic datatypes

Taken from

https://github.com/jcollard/ocaml-exercises/blob/master/OCamlTutorial.md

Thanks!

Recursive Data Structures: Lists

For this part of the tutorial, we will work with a type declaration for lists of integers. There are two kinds of lists: the empty list and lists that have a single integer and a a reference to the rest of the list. We can specify the shape of lists using a type declaration with two constructors:

```
type intlist =
  | Cons of int * intlist
  | Empty
```

For example, here is a list of numbers from -1 through 4:

```
let from_minus_1_to_4 =
  Cons (-1, Cons (0, Cons (1, Cons (2, Cons (3, (Cons (4, Empty)))))))
```

Exercise 9

Write a function `all_positive lst`, which returns `true` if all the integers in `lst` are positive.

```
val all_positive : intlist -> bool
```

Exercise 10

Write a function `all_even lst`, which returns `true` if all the integers in `lst` are even numbers.

```
val all_even : intlist -> bool
```

Exercise 11

Write the function `is_sorted lst` to determine if the integers in `lst` are in sorted (ascending).

```
val is_sorted : intlist -> bool
```

Hint: You will need to write a recursive helper function.

Exercise 12

Write the function `insert_sorted n lst`, which inserts `n` into the sorted list `lst` and preserves the sort-order.

Exercise 13

Write the `insertion_sort` function, using `insert_sorted` as a helper.

```
val insertion_sort : intlist -> intlist
```

For the exercises below, use the following type declaration that represents arithmetic expressions.

```
type exp =
  | Int of int
  | Add of exp * exp
  | Mul of exp * exp
```

Exercise 14

Encode the following arithmetic expressions as `exp` s:

1. 10 + 5
2. (2 + 3) * 5
3. 3 * 0 * 3 * 5

Exercise 15

Write the function `eval e`, which reduces expressions to integer values:

```
val eval : exp -> int
```

Exercise 16

Write the function `print e`, which returns a string representing `e`:

```
val print : exp -> string
```

Review on list recursion

Taken from

https://courses.grainger.illinois.edu/cs421/su2009/exams/mt-sampleqns.pdf

Thanks!

(a) `contains : 'a -> 'a list -> bool` such that `contains x lst` returns true if and only if `lst` has `x` as one of its elements. Do not use any pre-existing functions. E.g.

```
contains 4 [3;4;5]  =  true
```

(b) `evens: 'a list -> 'a list` returns the 2nd, 4th, etc. elements of its argument. E.g.

```
evens [13;5;9;0;7;8] = [5; 0; 8]
```

(c) Implement the Ocaml function `partition: int list -> (int list) list`, which divides a list into "runs" of the same integer, e.g.

```
partition [9;9;5;6;6;6;3] = [[9;9]; [5]; [6;6;6]; [3]]
```

(You may define auxiliary functions, but it is not actually necessary.)

(d) `genlist m n = [m; m+1; ... ; n]  (or [] if m>n)`

link