List Processing in OCAML

Given a list of integers *ns*, suppose we want to return a new list of the same length in which each element is one more than the corresponding element of *ns*. oHere's one way to express this in Java (using the `IntList` class you've seen in CS111 and CS230).

```
public static IntList incList (IntList ns) {
  if (IntList.isEmpty(ns)) {
    return IntList.empty();
  } else {
    return IntList.prepend(1 + IntList.head(ns), incList(IntList.tails(ns)));
  }
}
```

What are the corresponding list manipulation operators in OCAML?

Java	OCAML
`empty()`	`[]`
`prepend(`*x*`,`*ys*`)`	*x*`::`*ys*
`head(`*xs*`)`	`List.hd(`*xs*`)`
`tail(`*xs*`)`	`List.tl(`*xs*`)`
`isEmpty(`*xs*`)`	*xs* `= []`

In OCAML, `[`E_1`;`E_2`;...;`E_n`]` is syntactic sugar for E_1`::`E_2`::...::`E_n`::[]`. E.g., we can use this sugar to express a list of the first four positive integers as `[1;2;3;4]`.

Here is an OCAML transliteration of the Java `incList` given above:

```
let rec incList ns =
  if ns = [] then
    []
  else
    (1+List.hd(ns))::(incList (List.tl ns))
```

However, in practice, `List.hd` and `List.tl` are rarely used to process lists in OCAML. Instead, we normally use OCAML's powerful pattern matching facility (the `match ... with` construct) to perform a case analysis on a list:

```
let rec incList ns =
  match ns with
    [] -> []
  | n::ns' -> (n+1)::(incList ns')
```

The expression between `match` and `with` (`ns` in this case) is called the **discriminant**. Following `with` is a sequence of clauses of the form *`pattern`* `->` *`body`*. The value of the discriminant is compared against the pattern of each clause in the sequence until a match is found. The body of the matching clause is then evaluated in a context that uses the names bound by the pattern-matching process. The result of evaluating the body is returned as the value of the `match` expression. In the `incList` example, the clause `[] -> []` means "if `ns` is the empty list, then return the empty list." The clause `n::ns' -> (n+1)::(incList ns')` means "if `ns` is a non-empty list whose head is `n` and whose tail is `ns'`, then return the list that results from prepending `n+1` to the result of recursively processing `ns'`."

The following `process` function is a contrived example to illustrate pattern matching:

```
let rec process ps =
  match ps with
    [(c,d);(e,f)] -> [(d,f);(c,e)]
  | p1::p2::p3::ps' -> p3::(process(p1::p2::ps'))
  | _ -> ps
```

The underscore pattern, `_`, is a special pattern that matches anything without binding the underscore symbol to a value. Here are some sample uses of `process`:

```
# process [];;
- : ('a * 'a) list = []

# process [(1,2)];;
- : (int * int) list = [(1, 2)]

# process [(1,2);(3,4)];;
- : (int * int) list = [(2, 4); (1, 3)]

# process [(1,2);(3,4);(5,6)];;
- : (int * int) list = [(5, 6); (2, 4); (1, 3)]

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

Patterns cannot contain duplicates, but can have `when` guards:

```
let condswap xs =
  match xs with
    x1::x2::x3::xs' when x1 = x3 -> x2 :: x1 :: x3 :: xs'
  | _ -> xs;;

# condswap [1;2;1;4];;
- : int list = [2; 1; 1; 4]

# condswap [1;2;3;4];;
- : int list = [1; 2; 3; 4]
```

Subpatterns can be named by `as` patterns:

```
let condswap xs =
  match xs with
    x1::x2::((x3::_) as xs'')  when x1 = x3 -> x2 :: x1 :: xs''
  | _ -> xs;;
val condswap : 'a list -> 'a list = <fun>
```

In class, we will write the following functions:

`val sum : int list -> int`
`sum` *`ns`* returns the sum of all the integers in a list of integers *ns*.

```
# sum [];;
- : int = 0
# sum [3];;
- : int = 3
# sum [3;2;7;5];;
- : int = 17
```

`val range : int * int -> int list`
`range (`*`lo`*`, `*`hi`*`)` returns a list of integers from *lo* up to *hi*, inclusive. The list is empty if $lo > hi$.

```
# range (3,7);;
- : int list = [3; 4; 5; 6; 7]
# range (5,5);;
- : int list = [5]
# range (6,5);;
- : int list = []
```

`val squares : int list -> int list`

`squares` *ns* returns a list of the squares of the corresponding integers in the list *ns*.

```
# squares [3;1;5;4;2];;
- : int list = [9; 1; 25; 16; 4]
# squares [3];;
- : int list = [9]
# squares [];;
- : int list = []
```

`val evens : int list -> int list`

`evens` *ns* returns a list of the even integers in the list *ns* in the same relative order that they appear in *ns*. (`x mod y` gives the remainder of dividing the integer x by the integer y.)

```
# evens [3;1;4;2;5;8;9;6];;
- : int list = [4; 2; 8; 6]
# evens [3;1;5;9];;
- : int list = []
# evens [6;256;100];;
- : int list = [6; 256; 100]
# evens [];;
- : int list = []
```

Note: A key benefit of defining list-processing functions like `sum`, `range`, `squares`, `evens` is that they can be easily composed in mix-and-match ways to solve more complex problems. For example:

```
let sumOfSquaredEvensBetween (lo,hi) =
  sum(squares(evens(range(lo,hi))))
```

`val remove : 'a * 'a list -> 'a list`

`remove (`*x*`, `*ys*`)` returns a list of all the elements in *ys* except for occurrences of *x*. The relative order of non-*x* elements is preserved.

```
# remove (5, [5;4;5;3;4;2;3;4;5;1;3;5;4;2;5]);;
- : int list = [4; 3; 4; 2; 3; 4; 1; 3; 4; 2]
# remove (5, [1;2;3;4]);;
- : int list = [1; 2; 3; 4]
# remove (5, []);;
- : int list = []
```

`val isMember : 'a * 'a list -> bool`

`isMember(`*x*`,`*ys*`)` returns `true` if *x* is an element of the list `ys` (as determined by `=`) and `false` otherwise.

```
# isMember(3,[5;2;3;1;4]);;
- : bool = true
# isMember(6,[5;2;3;1;4]);;
- : bool = false
# isMember("be",["to";"be";"or";"not";"to";"be"]);;
- : bool = true
# isMember("two",["to";"be";"or";"not";"to";"be"]);;
- : bool = false
# isMember((2,"two"), [(3,"three");(1,"one");(2,"two");(4,"four")]);;
- : bool = true
# isMember((2,"too"), [(3,"three");(1,"one");(2,"two");(4,"four")]);;
- : bool = false
```

`val removeDups : 'a list -> 'a list`

`removeDups` *xs* returns a list containing one occurrence of each element in *xs*. The order of elements in the returned list is unspecified. *Note:* There are *many* ways to define this function!

```
# removeDups [5;4;5;3;4;2;3;4;5;1;3;5;4;2;5];;
- : int list = [1; 3; 4; 2; 5] (* order doesn't matter *)
# removeDups ["do";"be";"do";"be";"do"];;
- : string list = ["be"; "do"] (* order doesn't matter *)
# removeDups ['a';'b';'r';'a';'c';'a';'d';'a';'b';'r';'a'];;
- : char list = ['c'; 'd'; 'b'; 'r'; 'a'] (* order doesn't matter *)
# removeDups [];;
- : '_a list = []
```

`val isSorted : 'a list -> bool`

`isSorted` *xs* returns `true` if the list `xs` is sorted from low to high according to `<=`, and false otherwise.

```
# isSorted [];;
- : bool = true
# isSorted [3];;
- : bool = true
# isSorted [3;1;4;2];;
- : bool = false
# isSorted [1;2;3;4];;
- : bool = true
# isSorted [false;true];;
- : bool = true
# isSorted [true;false];;
- : bool = false
# isSorted ['a';'b';'c'];;
- : bool = true
# isSorted ['c';'a';'b'];;
- : bool = false
# isSorted ["one";"two";"three"];;
- : bool = false
# isSorted ["one";"three";"two"];;
- : bool = true
# isSorted [(1,"bar");(2,"baz");(3,"foo")];;
- : bool = true
# isSorted [(1,"bar");(3,"baz");(2,"foo")];;
- : bool = false
# isSorted [(1,"foo");(2,"bar");(3,"baz")];;
- : bool = true
# isSorted [[];[1];[1;2];[1;3;2];[1;3;4];[1;4];[2]];;
- : bool = true
# isSorted [[];[1];[1;2;3];[1;2];[1;3;4];[1;4];[2]];;
- : bool = false
```

`val flatten : 'a list list -> 'a list`

`flatten` *xss* returns a list containing all of the elements of the lists in the list of list `xss` in the same order. Use the infix `@` operator or prefix `List.append` operator to append two lists. *Note:* The `flatten` function is called `List.flatten` in the OCAML standard library.

```
# flatten [[4;2];[3;1;5;8];[7];[6;0;9]];;
- : int list = [4; 2; 3; 1; 5; 8; 7; 6; 0; 9]
# flatten [["foo"];["bar";"baz"];["quux"]];;
- : string list = ["foo"; "bar"; "baz"; "quux"]
# flatten [["foo"]];;
- : string list = ["foo"]
# flatten [];;
- : 'a list = []
```

`val reverse : 'a list -> 'a list`

`reverse` *xs* returns a list containing the elements of the list *xs* in reverse order. *Note:* This function is called `List.rev` in the OCAML standard library.

```
# reverse [3;1;5;4;2];;
- : int list = [2; 4; 5; 1; 3]
# reverse ["foo";"bar";"baz"];;
- : string list = ["baz"; "bar"; "foo"]
# reverse ["foo"];;
- : string list = ["foo"]
# reverse [];;
- : 'a list = []
```

`val zip : 'a list * 'b list -> ('a * 'b) list`
`zip (`*`xs`*`,`*`ys`*`)` returns a list of pairs containing the corresponding elements of the lists *xs* and *ys*. The length of the resulting list is the length of the shorter of *xs* and *ys*. *Note:* A curried version of this function is called `List.combine` in the OCAML standard library..

```
# zip ([1;2;3],['a';'b';'c']);;
- : (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]
# zip ([1;2;3;4;5],['a';'b';'c']);;
- : (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]
# zip ([1;2;3],['a';'b';'c';'d';'e']);;
- : (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]
# zip ([],['a';'b';'c'']);;
- : ('a * char) list = []
# zip ([1;2;3],[]);;
- : (int * 'a) list = []
```

`val unzip : ('a * 'b) list -> 'a list * 'b list`
`unzip` *`ps`* takes a list of pairs *ps* and returns a pair of lists, the first of which contains all the first components of *ps*, and the second of which contains all the second components of *ps*. *Note:* This function is called `List.split` in the OCAML standard library.

```
# unzip [(1, 'a'); (2, 'b'); (3, 'c')];;
- : int list * char list = ([1; 2; 3], ['a'; 'b'; 'c'])
# unzip [(2, 'b')];;
- : int list * char list = ([2], ['b'])
# unzip [];;
- : 'a list * 'b list = ([], [])
```

`val mapcons : 'a * 'a list list -> 'a list list`
`mapcons (`*x*`,`*zss*`)` returns a list containing the result of prepending *x* to each list in the list of lists *zss*.

```
# mapcons (5,[[4;1];[3];[2;1;3];[]]);;
- : int list list = [[5; 4; 1]; [5; 3]; [5; 2; 1; 3]; [5]]
# mapcons ("foo", [[]]);;
- : string list list = [["foo"]]
# mapcons ("foo", []);;
- : string list list = []
```

`val subsets : 'a list -> 'a list list`
Assume that *xs* is a list without duplicates, and thus represents a set of elements. `subsets` *xs* returns a list of lists containing all subsets of *xs*. The elements of each subset must appear in the same relative order as in *xs*, but the order of the subsets themselves is unspecified. *Hint:* `mapcons` is helpful here.

```
# subsets [];;
- : '_a list list = [[]]
# subsets [4];;
- : int list list = [[]; [4]]
# subsets [3;4];;
- : int list list = [[]; [4]; [3]; [3; 4]]
# subsets [2;3;4];;
- : int list list = [[]; [4]; [3]; [3; 4]; [2]; [2; 4]; [2; 3]; [2; 3; 4]]
# subsets [1;2;3;4];;
- : int list list =
[[];  [4];    [3];    [3; 4];    [2];    [2; 4];    [2; 3];    [2; 3; 4];
 [1]; [1; 4]; [1; 3]; [1; 3; 4]; [1; 2]; [1; 2; 4]; [1; 2; 3]; [1; 2; 3; 4]]
# subsets ['a';'b';'c';'d'];;
- : char list list =
[[]; ['d']; ['c']; ['c'; 'd']; ['b']; ['b'; 'd']; ['b'; 'c'];
 ['b'; 'c'; 'd']; ['a']; ['a'; 'd']; ['a'; 'c']; ['a'; 'c'; 'd']; ['a'; 'b'];
 ['a'; 'b'; 'd']; ['a'; 'b'; 'c']; ['a'; 'b'; 'c'; 'd']]
```

`val decimal : int list -> int`

Assume that *bs* is a list of zeroes and ones. `decimal bs` returns an integer that is the decimal representation of the number represented in binary by *bs*. An empty list of bits is assumed to denote 0.

```
# decimal [];;
- : int = 0
# decimal [0];;
- : int = 0
# decimal [1];;
- : int = 1
# decimal [1;0];;
- : int = 2
# decimal [1;0;0];;
- : int = 4
# decimal [1;0;1];;
- : int = 5
# decimal [1;0;1;0];;
- : int = 10
# decimal [1;0;1;1];;
- : int = 11
# decimal [1;0;1;1;0];;
- : int = 22
# decimal [1;0;1;1;1];;
- : int = 23
# decimal [1;0;1;1;1;0];;
- : int = 46
```