CS 321 Programming Languages

Intro to OCaml – Lists

Baris Aktemur

Özyeğin University

Last update made on Monday 2nd October, 2017 at 19:27.

Some of the contents here are taken from Elsa Gunter and Sam Kamin's OCaml notes available at `http://courses.engr.illinois.edu/cs421`

Lists

- First example of a recursive datatype (aka algebraic datatype).
- Unlike tuples, lists are homogeneous in type (all elements same type).
- A list has two forms.
  - Empty: written as `[]`.
  - Non-empty: with a head element and a tail, written as `x::xs`.
- The tail of a list is a list of the same type.
- `::` operation, read as "cons", combines a head element and a tail.
- Syntactic sugar:
  - `[x]` is `x::[]`
  - `[x1;x2;x3;...;xn]` is `x1::x2::x3::...::xn::[]`

Lists

```
# [];;
- : 'a list = []
# [1];;
- : int list = [1]
# 1::[];;
- : int list = [1]
# [1;2;3;4];;
- : int list = [1; 2; 3; 4]
# 1::2::[3;4];;
- : int list = [1; 2; 3; 4]
# let a::b = [1;2;3;4];; (* Pattern matching on lists *)
(* A warning suppressed *)
val a : int = 1
val b : int list = [2; 3; 4]
```


Lists are homogeneous

```
# let badList = [1; 2.3; 5];;

Error: This expression has type float but an expression was expected of type
         int
```

Lists

```
# [2.5; 3.8; 0.77];;
- : float list = [2.5; 3.8; 0.77]

# [true; false; true];;
- : bool list = [true; false; true]

# ['a'; 'b'; 'c'; 'd'];;
- : char list = ['a'; 'b'; 'c'; 'd']

# ["hello"; "world"];;
- : string list = ["hello"; "world"]

# [[1;2]; [3;4;5]; []; [6]];;
- : int list list = [[1; 2]; [3; 4; 5]; []; [6]]

# [2,3];;
- : (int * int) list = [(2, 3)]
```


Lists are homogeneous

Which of the following lists is invalid?

1. `[2; 3; 4; 6]`
2. `[2.3; 4.5; 6.7]`
3. `[2,3; 4,5; 6,7]`
4. `[2,3.4; 4,5.6; 6.8,7]`
5. `[["hi"; "there"]; ["whatcha"]; []; ["doin"]]`

Exercise

What are the types? (or flag error)

```
# ['a'; 'b'];;

# ['a'; 'b'; "c"];;

# [(1,2); (3,4)];;

# [(1,2); (3,4,5)];;

# [(1,[2]); (3,[4;5])];;

# ['a', 'b'];;
```


Exercise

Provide values (other than empty list) to form lists of given types.

```
# ???;;
- : int list
# ???;;
- : int list list
# ???;;
- : (int * string) list
# ???;;
- : string list list
# ???;;
- : (int * string list) list
# ???;;
- : (int * string list) list list
```

Purity

cons operation is "pure"; it does not destroy/modify existing lists, but rather it constructs a new one.

```
# let lst = [2;3;4;5];;
val lst : int list = [2; 3; 4; 5]

# 1::lst;;
- : int list = [1; 2; 3; 4; 5]
# lst;;
- : int list = [2; 3; 4; 5]
```


Lists

```
# let a::b::c = [1;2;3;4];; (* Pattern matching on lists *)
(* A warning suppressed *)
val c : int list = [3; 4]
val b : int = 2
val a : int = 1
# let a::_::c = [1;2;3;4];; (* Pattern matching on lists *)
(* A warning suppressed *)
val c : int list = [3; 4]
val a : int = 1

# let list1 = [1;2;3;4];;
val list1 : int list = [1; 2; 3; 4]
# let list2 = [5;6;7];;
val list2 : int list = [5; 6; 7]
# list1@list2;; (* Append lists *)
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

match expression for pattern-matching

```
# let rec power x n =
    match n with
    | 0 -> 1
    | m -> x * power x (n - 1);;

val power : int -> int -> int = <fun>
```


match expression for pattern-matching

```
# let foo triple =
    match triple with
    | (0, x, y) -> 1
    | (x, 0, y) -> 2
    | (x, y, 0) -> 3
    | _ -> 4;;
val foo : int * int * int -> int = <fun>

# foo (0,3,4);;
- : int = 1
# foo (3,0,4);;
- : int = 2
# foo (3,0,0);;
- : int = 2
# foo (0,0,0);;
- : int = 1
# foo (1,2,3);;
- : int = 4
```

match expression for pattern-matching

```
# let incomplete n =
    match n with
    | 0 -> "zero"
    | 1 -> "one";;

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
2

val incomplete : int -> string = <fun>
```


match expression for pattern-matching

```
# let redundant n =
    match n with
    | 0 -> "zero"
    | m -> "number"
    | 1 -> "one";;

    | 1 -> "one";;
  ----^

Warning 11: this match case is unused.

val redundant : int -> string = <fun>
```

Functions on lists

```
# let headOf lst =
    match lst with
    | [] -> failwith "Empty list doesn't have a head"
    | x::_ -> x;; (* don't care the tail *)
val headOf : 'a list -> 'a = <fun>

# let tailOf lst =
    match lst with
    | [] -> failwith "Empty list doesn't have a tail"
    | _::xs -> xs;; (* don't care the head *)
val tailOf : 'a list -> 'a list = <fun>
```


Functions on lists

```
# headOf [3;5;7;9];;
- : int = 3
# tailOf [3;5;7;9];;
- : int list = [5; 7; 9]
# tailOf(tailOf [3;5;7;9]);;
- : int list = [7; 9]
# headOf(tailOf [3;5;7;9]);;
- : int = 5

(* head and tail already defined in the library *)
# List.hd [1;2;3];;
- : int = 1
# List.tl [1;2;3];;
- : int list = [2; 3]
```

Functions on lists

```
# let secondElementOf lst =
    match lst with
    | x::y::rest -> y;;
```

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(_::[]|[])
```

```
val secondElementOf : 'a list -> 'a
# secondElementOf [1;2;3;4;5];;
- : int = 2
# secondElementOf [1];;
```

```
Exception: Match_failure ("//toplevel//", 20, 4).
```


Exercise

Write a function to compute the sum of the first two elements of an `int list`. You can assume the list is of length at least 2.

```
# let addfirsttwo ...
```

```
# addfirsttwo [5; 3; 2; 6];;
- : int = 8
```

Functions on lists

```
# let rec lengthOf lst =
    match lst with
    | [] -> 0
    | x::xs -> 1 + lengthOf xs;;

val lengthOf : 'a list -> int

# lengthOf [];;
- : int = 0
# lengthOf [1;2;3;4;5];;
- : int = 5
# List.length [1;2;3;4];; (* defined in the library *)
- : int = 4
```


Exercise

```
# let rec tally lst = ???;;
val tally : int list -> int

# tally [];;
- : int = 0
# tally [1;2;3;4;5;6];;
- : int = 21
```

Exercise

Write a function to compute the sum of the *lengths* of the first two elements of an `(int list) list`. You can assume the list is of length at least 2:

```
# let addfirsttwolengths ...


# addfirsttwolengths [[5; 3]; [2]; [6; 2; 5; 3]];;
- : int = 3
```


Exercise

```
# let rec numZeros lst =




val numZeros : int list -> int

# numZeros [];;
- : int = 0
# numZeros [1;2;3;4;5];;
- : int = 0
# numZeros [1;2;0;4;0];;
- : int = 2
```

Exercise

```
# let rec numZeros lst =
    match lst with
    | [] -> 0
    | 0::xs -> 1 + numZeros xs
    | _::xs -> numZeros xs;;
val numZeros : int list -> int

# numZeros [];;
- : int = 0
# numZeros [1;2;3;4;5];;
- : int = 0
# numZeros [1;2;0;4;0];;
- : int = 2
```


Functions on lists

```
(* Alternatively *)
# let rec numZeros lst =
    match lst with
    | [] -> 0
    | x::xs -> (if x = 0 then 1 else 0) + numZeros xs;;

val numZeros : int list -> int
```

Functions on lists

```
(* Yet another alternative *)
# let rec numZeros lst =
    match lst with
    | [] -> 0
    | x::xs when x = 0 -> 1 + numZeros xs
    | x::xs -> numZeros xs;;

val numZeros : int list -> int
```


Exercise

```
# let rec doubleUp lst = ???;;


val doubleUp : 'a list -> 'a list

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

Functions on lists

```
# let rec poorRev lst =
    match lst with
    | [] -> []
    | x::xs -> poorRev xs @ [x];;

val poorRev : 'a list -> 'a list

# poorRev [1;2;3;4;5];;
- : int list = [5; 4; 3; 2; 1]
```


Exercise – zipAdd

Define a function `zipAdd` that takes two integer lists, and returns a list that contains the sum of corresponding elements in its input lists. You may assume that the input lists are of the same length.

```
# let rec zipAdd lst1 lst2 =
```

```
val zipAdd : lst1:int list -> lst2:int list -> int list

# zipAdd [1;2;3;4;5] [6;7;8;9;10];;
- : int list = [7; 9; 11; 13; 15]
```

Exercise – zipAdd

Define a function zipAdd that takes two integer lists, and returns a list that contains the sum of corresponding elements in its input lists. You may assume that the input lists are of the same length.

```
# let rec zipAdd lst1 lst2 =
    (* pattern-match a pair of lists! *)
    match lst1, lst2 with
    | [],[] -> []
    | x::xs, y::ys -> (x+y)::zipAdd xs ys;;
   (* Incomplete pattern-matching warning omitted *)
val zipAdd : lst1:int list -> lst2:int list -> int list

# zipAdd [1;2;3;4;5] [6;7;8;9;10];;
- : int list = [7; 9; 11; 13; 15]
```


Exercise – zipAdd, Alternative implementation

You can write any expression corresponding to the case of a match, including another match.

```
# let rec zipAdd lst1 lst2 =
    match lst1 with
    | [] -> []
    | x::xs -> (match lst2 with
                | [] -> failwith "This shouldn't happen."
                | y::ys -> (x+y)::zipAdd xs ys);;

val zipAdd : lst1:int list -> lst2:int list -> int list

# zipAdd [1;2;3;4;5] [6;7;8;9;10];;
- : int list = [7; 9; 11; 13; 15]
```

Functions on lists – map

Define a function `map` such that `map f [x1; x2;...;xn]` computes `[f(x1); f(x2);...;f(xn)]`

```
# let rec map f lst =
    match lst with
    | [] -> []
    | x::xs -> f x :: map f xs;;

val map : ('a -> 'b) -> 'a list -> 'b list
```


Exercise: map

```
# let rec map f lst =
    match lst with
    | [] -> []
    | x::xs -> f x :: map f xs;;

val map : ('a -> 'b) -> 'a list -> 'b list

# map (fun n -> n + 2) [1;2;3;4;5];;
- : int list = [3; 4; 5; 6; 7]
# map (fun s -> s ^ "!") ["a"; "b"; "c"];;
- : string list = ["a!"; "b!"; "c!"]
# map (???) [1;2;3;4;5];;
- : int list = [1; 4; 9; 16; 25]
# map (???) [1;-2;3;-4;5;0;-99];;
- : int list = [1; 2; 3; 4; 5; 0; 99]

(* map is defined in the library's List module *)
# List.map abs [1;-2;3;-4;5;0;-99];;
- : int list = [1; 2; 3; 4; 5; 0; 99]
```

Iterating over lists – fold_left

```
# let rec fold_left f a lst =
    match lst with
    | [] -> a
    | x::xs -> fold_left f (f a x) xs;;

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

fold_left is an extremely important function that is used frequently.

fold_left f a [x_1;x_2;...;x_n] computes
f(...(f (f a x_1) x_2)...)x_n.

So, f takes two arguments: (1) the accumulated value over the list from the left, (2) the current element of the list. a is the initial value of accumulation, which is also the result if the list is empty.

Exercise: fold_left

```
(* tally *)
# fold_left (fun acc x -> acc + x) 0 [1;2;3;4;5;6];;
- : int = 21
(* Or, also as *)
# fold_left (+) 0 [1;2;3;4;5;6];;
- : int = 21

(* lengthOf *)
# fold_left (fun acc x -> acc + 1) 0 [4;9;0;45;3;6];;
- : int = 6

(* numZeros *)
# fold_left (fun acc x -> ???              ) 0 [4;9;0;45;0;0];;
- : int = 3

(* fold_left is already defined in the library *)
# List.fold_left (fun acc x -> acc + x) 0 [1;2;3;4;5;6];;
- : int = 21
```

Iterating over lists – fold_right

```
# let rec fold_right f lst a =
    match lst with
      [] -> a
    | x::xs -> f x (fold_right f xs a);;

val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

fold_right is the other extremely important function that is used frequently to iterate over lists.

fold_right f [x_1;x_2;...;x_n] a computes
f x_1(f x_2(... (f x_n a)...)).

So, f takes two arguments: (1) the current element of the list, (2) the accumulated value over the list from the right. a is the initial value of accumulation, which is also the result if the list is empty.

Exercise: fold_right

```
(* tally *)
# fold_right (fun x a -> ??? ) [1;2;3;4;5;6] 0;;
- : int = 21
# fold_right ( * ) [1;2;3;4;5;6] 1;;
- : int = 720

(* squareUp *)
# fold_right (fun x a -> ???  ) [1;2;3;4;5;6] [];;
- : int list = [1; 4; 9; 16; 25; 36]

(* fold_right is already defined in the library *)
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
# List.fold_right (fun x a -> if x=0 then a else x::a) [1;0;3;0;5;6;0] [];;
- : int list = [1; 3; 5; 6]
```

Exercise

```
(* Reversing a list, written with fold_left and fold_right *)
# fold_left (fun a x ->   ???   ) [] [1;2;3;4;5;6];;
- : int list = [6; 5; 4; 3; 2; 1]

# fold_right (fun x a ->   ???   ) [1;2;3;4;5;6] [];;
- : int list = [6; 5; 4; 3; 2; 1]

(* Which one is more efficient? *)
```


List functions

The `List` module contains very useful functions. See the API documentation at

`http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`

Some important functions, in addition to `map`, `fold_left` and `fold_right`.

- E.g.: `rev`, `flatten`, `mem`, `filter`.

Loops

Question

What happened to while/for loops?

Fact

In functional programming, you seldomly use while/for loops, which are highly associated with procedural/imperative programming. You excessively use recursion instead. Recursion is more powerful than simple loops, and if you form your recursion right, you don't compromise performance. (will talk about this soon)