

# OCaml IV: Lists and IO

CAS CS 320: Principles of Programming Languages

Thursday, February 1, 2024

## Administrivia

- Homework 1 is due today by 11:59 pm.
- Homework 2 is posted today and due on Thursday, Feb 8, by 11:59 pm.

## Administrivia

- Homework 1 is due today by 11:59 pm.
- Homework 2 is posted today and due on Thursday, Feb 8, by 11:59 pm.

## Reading Assignment

- OCP Section 3.1: **Lists**

## building lists (OCP 3.1.1) : three syntactic forms

```
(* empty list, also called "nil" *)
```

```
[]
```

```
(* prepending elt to lst with "cons" :: *)
```

```
elt :: lst
```

```
(* list with n expressions, all of the same type *)
```

```
[e_1; e_2; . . . ; e_n]
```

## building lists (OCP 3.1.1) : three syntactic forms

```
(* empty list, also called "nil" *)  
[]  
  
(* prepending elt to lst with "cons" :: *)  
elt :: lst  
  
(* list with n expressions, all of the same type *)  
[e_1; e_2; . . . ; e_n]
```

Specific examples:

```
# let lst1 = [] ;;  
# let lst2 = 'a' :: lst1 ;;  
# let lst3 = 'b' :: lst2 ;;  
# let lst4 = 'c' :: lst3 ;;  
val lst4 : char list = ['c'; 'b'; 'a']
```

## some notational conventions (OCP 3.1.1)

- metavariable **e** usually denotes an **expression** that can be evaluated further.

metavariable **v** usually denotes a **value**, i.e. an expression that **cannot** be evaluated further.

## some notational conventions (OCP 3.1.1)

- metavariable **e** usually denotes an **expression** that can be evaluated further.

metavariable **v** usually denotes a **value**, i.e. an expression that **cannot** be evaluated further.

- **[e<sub>1</sub>;e<sub>2</sub>;e<sub>3</sub>]** is sugar for **(e<sub>1</sub> :: (e<sub>2</sub> :: (e<sub>3</sub> :: [])))**.

for example: **['c' ; 'b' ; 'a']** is syntactic sugar for **('c' :: ('b' :: ('a' :: [])))**.

## some notational conventions (OCP 3.1.1)

- metavariable **e** usually denotes an **expression** that can be evaluated further.

metavariable **v** usually denotes a **value**, i.e. an expression that **cannot** be evaluated further.

- **[e<sub>1</sub>;e<sub>2</sub>;e<sub>3</sub>]** is sugar for **(e<sub>1</sub> :: (e<sub>2</sub> :: (e<sub>3</sub> :: [])))**.

for example: **['c' ; 'b' ; 'a']** is syntactic sugar for **('c' :: ('b' :: ('a' :: [])))**.

- **e ==> v** means **e** evaluates to **v** in finitely many, possibly 0, steps.

for example, **3+1 ==> 4**, **2\*(3+1) ==> 8**, and **7 ==> 7**.



examples of expressions which are **not** values

$2 + 1$

$2 * 2$

$"a" \wedge "b"$

$((2+1) :: ((2*2) :: (5 :: [])))$

## examples of expressions which are **not** values

`2 + 1`

`2 * 2`

`"a" ^ "b"`

`((2+1) :: ((2*2) :: (5 :: [])))`

## examples of expressions which are values

`3`

不用计算的直接是一个variable

`4`

`"ab"`

`(3 :: (4 :: (5 :: [])))`

## dynamic semantics (OCP 3.1.1)

evaluation rules:

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{(e_1 :: e_2) \Rightarrow v_1 :: v_2}$$

cons

$$\frac{e_i \Rightarrow v_i \text{ for all } i \text{ in } \{1, \dots, n\}}{[e_1; \dots ; e_n] \Rightarrow [v_1; \dots ; v_n]}$$

## dynamic semantics (OCP 3.1.1)

evaluation rules:

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{(e_1 :: e_2) \Rightarrow v_1 :: v_2}$$

$$\frac{e_i \Rightarrow v_i \text{ for all } i \text{ in } \{1, \dots, n\}}{[e_1; \dots ; e_n] \Rightarrow [v_1; \dots ; v_n]}$$

somewhat less transparent:

$$\frac{e_i \Rightarrow v_i \text{ for all } i \text{ in } \{1, \dots, n\}}{(e_1 :: ( \dots :: (e_n :: []))) \Rightarrow (v_1 :: ( \dots :: (v_n :: [])))}$$

example (OCP 3.1.1)

simple enough that we all agree that:

**[2+1 ; 2+2 ; (2+3)\*2] ==> [3 ; 4 ; 10]**

## example (OCP 3.1.1)

simple enough that we all agree that:

$$[2+1 ; 2+2 ; (2+3)*2] ==> [3 ; 4 ; 10]$$

we use evaluation rules to produce a formal evaluation:

$$\frac{\begin{array}{c} 2+2 ==> 4 \quad (2+3)*2 ==> 10 \\ \hline 2+1 ==> 3 \quad [2+2 ; (2+3)*2] ==> [4 ; 10] \\ \hline [2+1 ; 2+2 ; (2+3)*2] ==> [3 ; 4 ; 10] \end{array}}$$

remark: this formal evaluation is **not** unique, because (like in the book OCP) we have allowed " $==>$ " to mean "zero or more finitely many steps".

## static semantics (OCP 3.1.1)

typing rules:

$$\frac{[] : \text{'a list}}{[] : \text{t list}}$$

$$\frac{e : \text{t} \quad [] : \text{t list}}{[e] : \text{t list}}$$

$$\frac{e_1 : \text{t} \quad e_2 : \text{t list}}{(e_1 :: e_2) : \text{t list}}$$

$$\frac{e_i : \text{t} \text{ for all } i \text{ in } \{1, \dots, n\}}{[e_1; \dots; e_n] : \text{t list}}$$

example (OCP 3.1.1)

simple enough that we all agree that:

```
[3 ; 4 ; 10] : int list
```



## example (OCP 3.1.1)

simple enough that we all agree that:

`[3 ; 4 ; 10] : int list`

we use typing rules to formally derive a typing:

				<code>[] : 'a list</code>
			<code>10 : int</code>	<hr/> <code>[] : int list</code>
	<code>4 : int</code>		<hr/> <code>[10] : int list</code>	
<code>3 : int</code>	<hr/> <code>[4 ; 10] : int list</code>			
<hr/>				
<code>[3 ; 4 ; 10] : int list</code>				

example (OCP 3.1.1)

simple enough that we all agree that:

```
[3 ; 4 ; 10] : int list
```

we use typing rules to formally derive a typing:

```

                                [] : 'a list'
                                -----
                                10 : int    [] : int list
                                -----
                                4 : int      [10] : int list
                                -----
                                3 : int      [4 ; 10] : int list
                                -----
                                [3 ; 4 ; 10] : int list

```

**question:** Is the typing derivation uniquely defined?

## accessing lists (OCP 3.1.2)

- two ways of building lists: with **nil** "[]" and **cons** "::"
- to take apart a list into its component pieces, we have to say what to do with the list if it is **empty** [], and what to do if it is **non-empty** of the form **(elt :: lst)**.

## accessing lists (OCP 3.1.2)

- two ways of building lists: with **nil** `[]` and **cons** `::`
- to take apart a list into its component pieces, we have to say what to do with the list if it is **empty** `[]`, and what to do if it is **non-empty** of the form **(elt :: lst)**.
- best way to do this is with **pattern matching**.
- example: function **length** computes the length of a list:

```
let rec length lst =  
  match lst with  
  | [] -> 0  
  | (h :: t) -> 1 + length t
```

## another example (OCP 3.1.2)

function **append** *splices* (i.e. *concatenates*) two lists:

```
let rec append lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | h :: t -> h :: append t lst2
```

## yet another example (OCP 3.1.2)

function **sum** adds all the entries in a list of integers:

```
let rec sum lst =  
  match lst with  
  | [] -> 0  
  | h :: t -> h + sum t
```

## (non) mutating lists (OCP 3.1.3)

- lists in OCaml are *immutable*, i.e., there is no way to change an element of a list from one value to another.
- instead, in OCaml, we create new lists out of old lists.

## (non) mutating lists (OCP 3.1.3)

- lists in OCaml are *immutable*, i.e., there is no way to change an element of a list from one value to another.
- instead, in OCaml, we create new lists out of old lists.
- example: **inc\_fst** increments first entry in integer list:

```
let inc_fst lst =  
  match lst with  
  | [] -> []  
  | h :: t -> (h + 1) :: t
```



## (non) mutating lists (OCP 3.1.3)

- lists in OCaml are *immutable*, i.e., there is no way to change an element of a list from one value to another.
- instead, in OCaml, we create new lists out of old lists.
- another example: **inc\_snd** increments second entry in list:

```
let inc_snd lst =  
  match lst with  
  | [] -> []  
  | [h] -> [h]  
  | h1 :: (h2 :: t) -> h1 :: (1 + h2) :: t)
```

## pattern matching with lists (OCP 3.1.4)

syntax: `match e with p_1 -> e_1 | . . . | p_n -> e_n`

dynamic semantics:

(1) evaluate **e** to a value **v**

(2) if **p\_i** is the first pattern to match **v**, then evaluate **e\_i** to value **v\_i** and return **v\_i**

**remark**: it is a little more complicated to set up the dynamic semantics of **match-with** with formal evaluation rules (see OCP 3.1.4 for details)

## pattern matching with lists (OCP 3.1.4)

syntax: `match e with p_1 -> e_1 | . . . | p_n -> e_n`

static semantics:

$\frac{e : t_a \quad p_i : t_a \text{ and } e_i : t_b \text{ for all } i \text{ in } \{1, \dots, n\}}{\text{match } e \text{ with } p_1 \rightarrow e_1 \mid . . . \mid p_n \rightarrow e_n : t_b}$
---

## deep pattern matching (OCP 3.1.5)

patterns can be *nested*, which allows us to look deeply into the structure of a list.

### *Examples:*

`_ :: []` matches all lists with ?? element

`_ :: _` matches all lists with ?? elements

`_ :: _ :: []` matches all lists with ?? elements

`_ :: _ :: _ :: _` matches all lists with ?? elements

## deep pattern matching (OCP 3.1.5)

patterns can be *nested*, which allows us to look deeply into the structure of a list.

### *Examples:*

`_ :: []` matches all lists with **1** element

`_ :: _` matches all lists with **1 or more** elements

`_ :: _ :: []` matches all lists with **2** elements

`_ :: _ :: _ :: _` matches all lists with **3 or more** elements

## tail recursion (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.
- example of a **non-tail recursive** function:

```
(* factorial function with pattern matching *)  
let rec fact n =  
    match n with  
    | 0 -> 1  
    | n -> n * fact (n - 1) ;;
```

- several advantages of tail-recursion ... **what are they?**

## tail recursion (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.
- one way (*not the only one*) of implementing factorial function with **tail recursion** is to use a **help** function:

```
let fact n =  
  let rec helper n current =  
    match n with  
    | 0 -> current  
    | n -> helper (n-1) (n*current) (* why not (current*n)?? *)  
  in helper n 1 ;;
```

## tail recursion (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.
- example of a function already in **tail recursive** form – no need to transform it:

```
(* GCD function with pattern matching *)  
let rec gcd n m =  
    match m with  
    | 0 -> n  
    | m -> gcd m (n mod m) ;;
```



## tail recursion (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.
- example of a function not in **tail recursive** form:

```
(* power function with pattern matching *)  
let rec pow x y =  
    match y with  
    | 0 -> 1  
    | y -> x * pow x (y - 1) ;;
```

## tail recursion (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.
- transforming the power function into **tail recursive** form:

```
(* power function in tail-recursive form *)  
let pow x y =  
  let rec helper x y acc =  
    match y with  
    | 0 -> acc  
    | y -> helper x (y-1) (x*acc) (* why not (acc*n) ?? *)  
in helper x y 1 ;;
```

## tail recursion (OCP 3.1.9)

- a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call.
- example of a function not in **tail recursive** form:

```
(* function foo is not tail-recursive *)  
let rec foo n =  
  match n with  
  | 0 -> 1  
  | n -> n/(foo (n-1));; (* substitute "/" for "*" in fact *)
```

**problem:** define function `foo_tr` in tail-recursive form which is equivalent to function `foo`.

( THIS PAGE INTENTIONALLY LEFT BLANK )