Révisions OCaml

Quentin Fortier

August 25, 2022

Expressions

Chaque **expression** en OCaml possède un **type** et une **valeur**.

Si e1 et e2 sont des expressions, `let a = e1 in e2` remplace a par e1 dans e2.

Exemple :

```
let a = 3 in let b = a*4 in b + a
-> let a = 3 in (let b = a*4 in (b + a))
-> let b = 3*4 in (b + 3)
-> 12 + 3
-> 15
```

Expressions

Une expression qui renvoie `()` (de type `unit`) est une **instruction**.

Exemples :

```
let r = ref 3 in
r := 7 (* instruction *)
```

Si `e1` et `e2` sont des expressions de valeurs entières et `e3` est une instruction, alors `for i = e1 to e2 do e3 done` est une instruction.

Si `e1` et `e2` sont des instructions alors `e1; e2` est une instruction.

Persistance

Exercice

Que vaut `l` après avoir exécuté le code suivant?

```
let l = [1; 2; 3] in
0::l
```

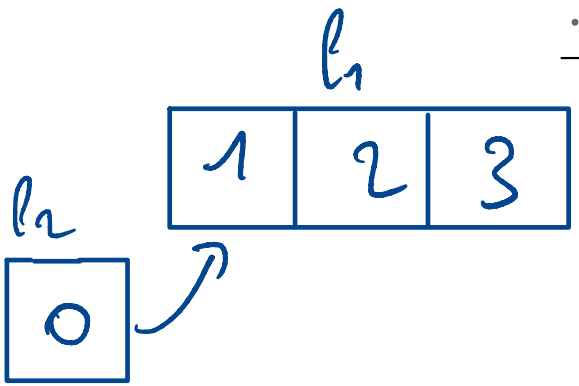
l vaut toujours [1;2;3]

Persistance

Exercice

Combien d'octets en mémoire sont utilisés dans le code ci-dessous ?

```
let l1 = [1; 2; 3] in
let l2 = 0::l1 in
...
```



Fonctions

Exercice

Quels sont les types et complexités des fonctions suivantes ?

(test d'appartenance) `List.mem`

(renvoie la longueur) `List.length`

(renverse une liste) `List.rev`

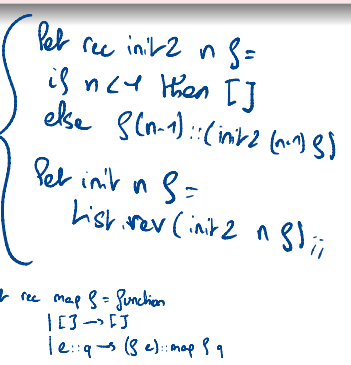
(renvoie liste vérifiant une condition) `List.filter`

(`List.init`

`List.map`

`List.iter`

`List.fold_left`



`Array.length`

`Array.make`

`Array.make_matrix`

`Array.copy`

Fonctions : Fonctions classiques

```
List.mem : 'a -> 'a list -> bool (* O(n) *)
List.length : 'a list -> int (* O(n) *)
List.rev : 'a list -> 'a list (* O(n) *)
List.filter : ('a -> bool) -> 'a list -> 'a list (* O(n) *)
List.init : int -> (int -> 'a) -> 'a list (* O(n) *)
List.map : (a -> b) -> 'a list -> 'b list (* O(n) *)
List.iter : (a -> unit) -> 'a list -> unit (* O(n) *)
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a (* O(n) *
```

```
Array.length : 'a array -> int (* O(1) *)
Array.make : int -> 'a -> 'a array (* O(n) *)
Array.make_matrix : int -> int -> 'a -> 'a array array (* O(np) *)
```

Remarque : Les fonctions de `List` ci-dessus existent aussi dans `Array`.

Fonctions : Fonctions classiques

`List.map f [e1; e2; ...]` renvoie `[f e1; f e2; ...]`.
`List.filter f l` renvoie la liste des éléments `a` de `l` tels que `f a` est `true`.
`List.init n f` renvoie `[f 0; f 1; ... f (n-1)]`.

Exercice

1. Réimplémenter ces fonctions.
2. Calculer la liste des carrés des entiers pairs entre 0 et 10.

2) List.map (fun i -> i*i) (List.filter (fun i -> i mod 2 = 0) (List.init n (fun i -> i)))

Fonctions : Fonctions classiques

`|>` est une notation pour appeler plusieurs fonctions à la suite.

`e |> f` est équivalent à `f e`.

`e |> f |> g` est équivalent à `g (f e)`.

Exercice

Calculer la liste des carrés des entiers pairs entre 0 et 10, en utilisant `|>`.

```
List.init n (fun i -> i) |> List.filter (fun i -> i mod 2 = 0)
                         |> List.map (fun i -> i * i)
```

Fonctions : Application partielle

Si `f` est une fonction à deux arguments, alors `f a` est une fonction à un argument, qui fixe le premier argument de `f`.

Exemple :

```
let sum x y = x + y in
let f = sum 42 in (* f est la fonction y -> 42 + y *)
f 3 (* 45 *)
```

Remarque : `(+)` (version préfixe de `+`) est la même fonction que `sum`.
De même pour `(=)`, `(mod)`...

Exercice

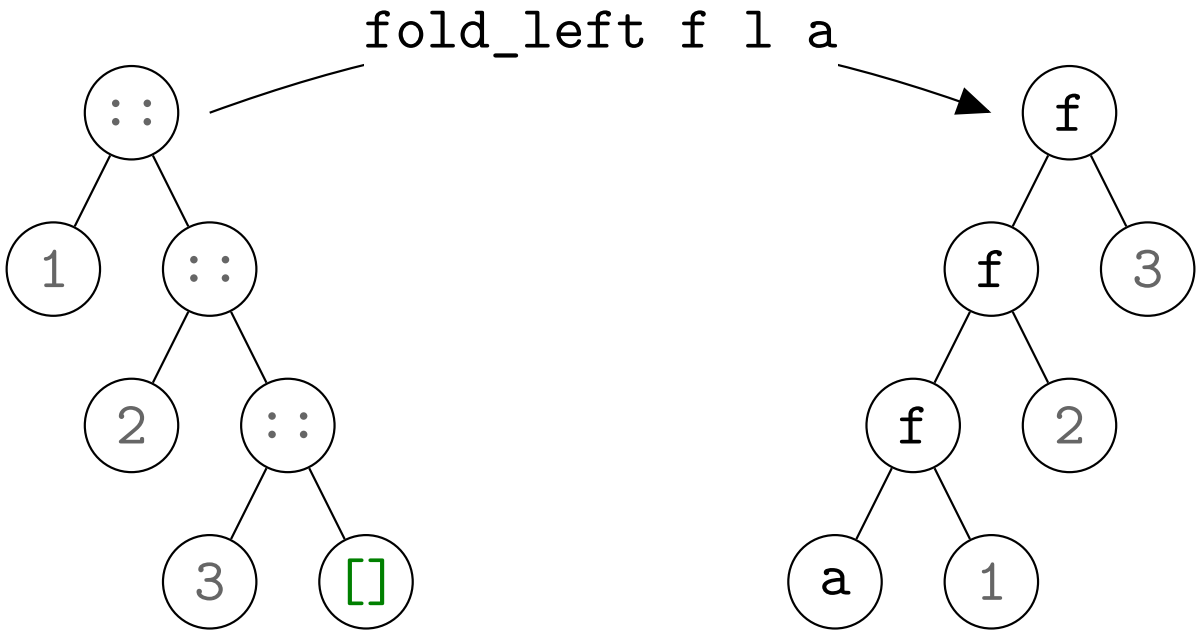
Que vaut `List.filter ((<) 0) l` ?

↳ Une liste contenant tous les éléments strictement positifs de l

List.fold

`List.fold_left f acc l` renvoie :

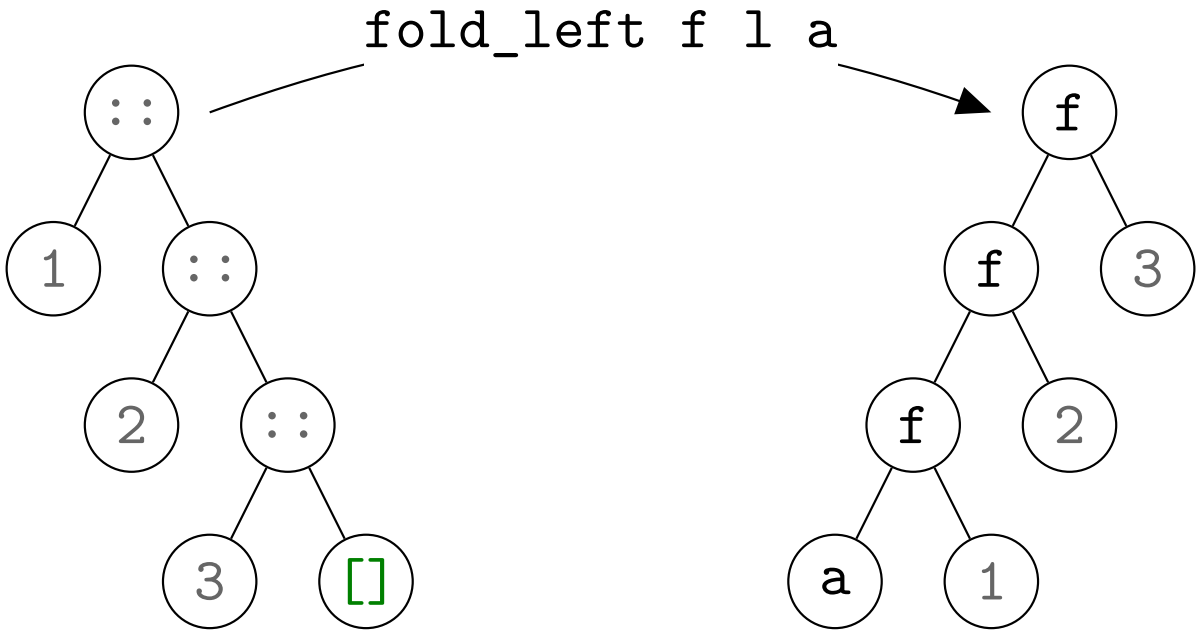
- Si `l = []` : `acc`.
- Si `l = [e1; e2; ...]` : `f (... (f (f acc e1) e2) ...) l`



List.fold

`List.fold_left f acc l` renvoie :

- Si `l = []` : `acc`.
- Si `l = [e1; e2; ...]` : `f (... (f (f acc e1) e2) ...) l`



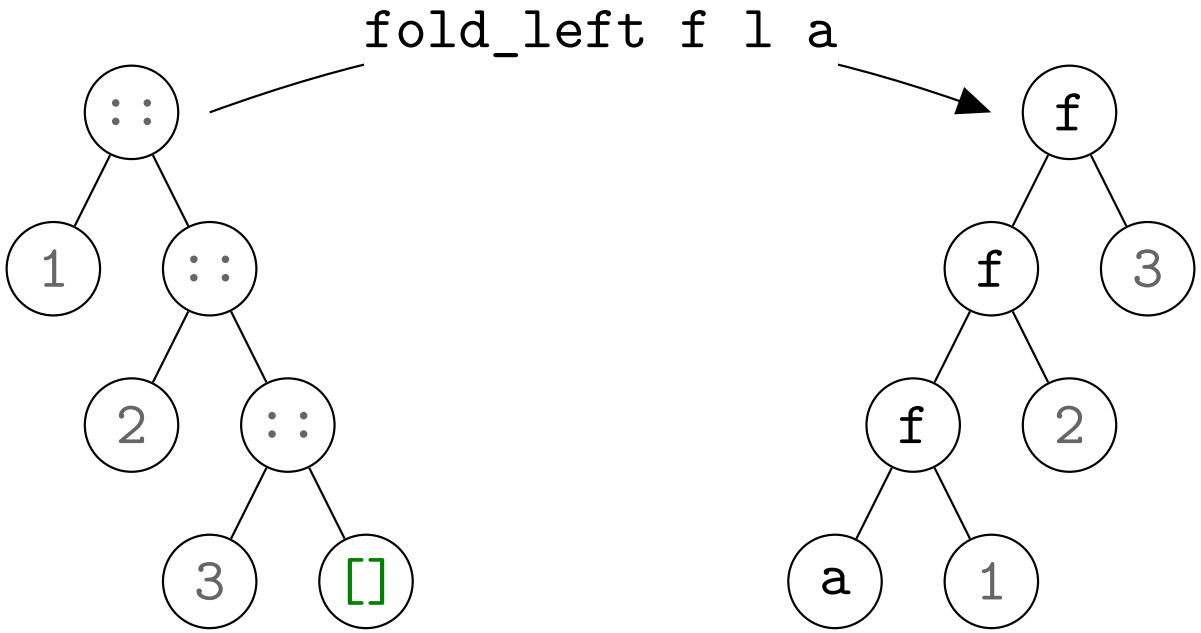
Exercice

Que renvoie `List.fold_left (+) 0 [3; 1; 4]` ?

List.fold

`List.fold_left f acc l` renvoie :

- Si `l = []` : `acc`.
- Si `l = [e1; e2; ...]` : `f (... (f (f acc e1) e2) ...) l`



Exercice

Redéfinir `List.rev` à l'aide de `List.fold_left`.

Récursivité terminale

Exercice

1. Expliquer pourquoi le code ci-dessous donne un message d'erreur.
   `Stack overflow during evaluation (looping recursion?).`
2. Comment pourrait-on résoudre ce problème ?

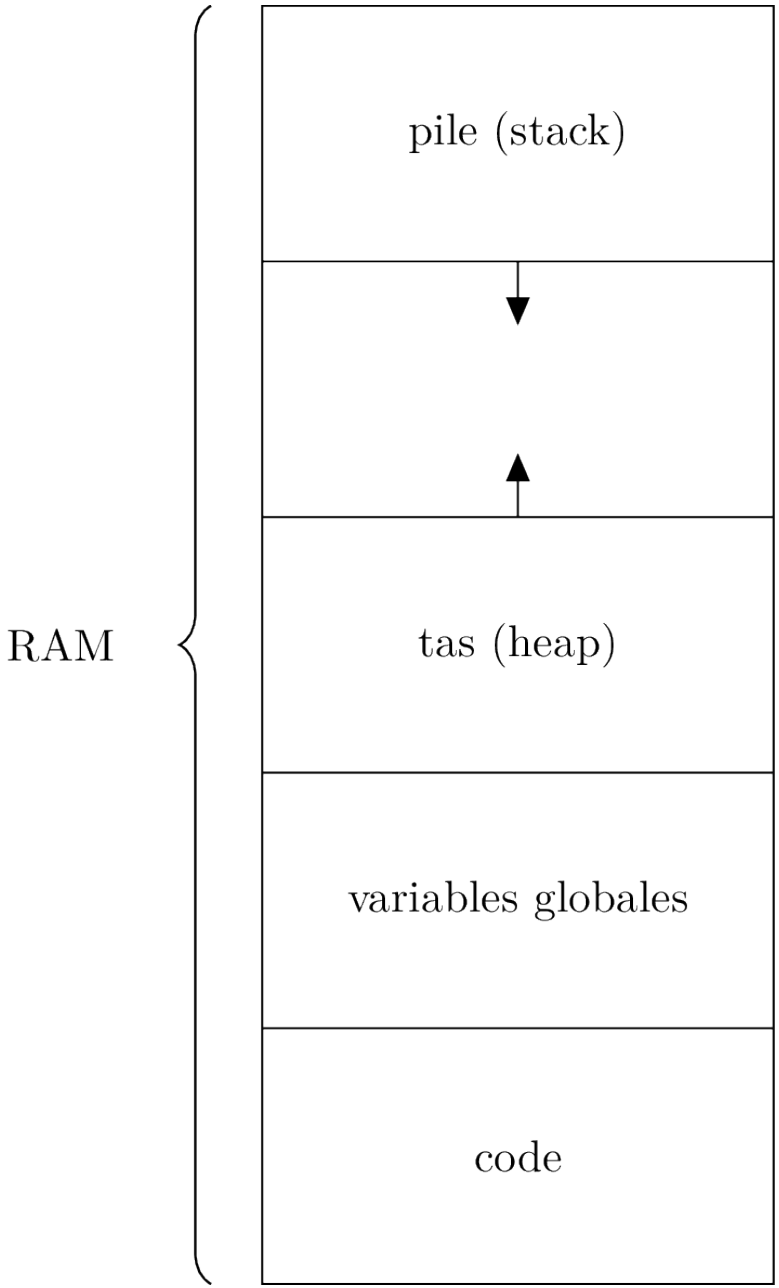
```
let rec sum = function
    | [] -> 0
    | e::q -> e + sum q;;

sum (List.init 1000000 (fun i -> i));;
```

2) let rec sum acc = function
   | [] -> acc
   | e::q -> sum (acc+e) :: q

ou

let sum l =
List.fold_left (+) l 0

Occupation d'un processus dans la mémoire RAM



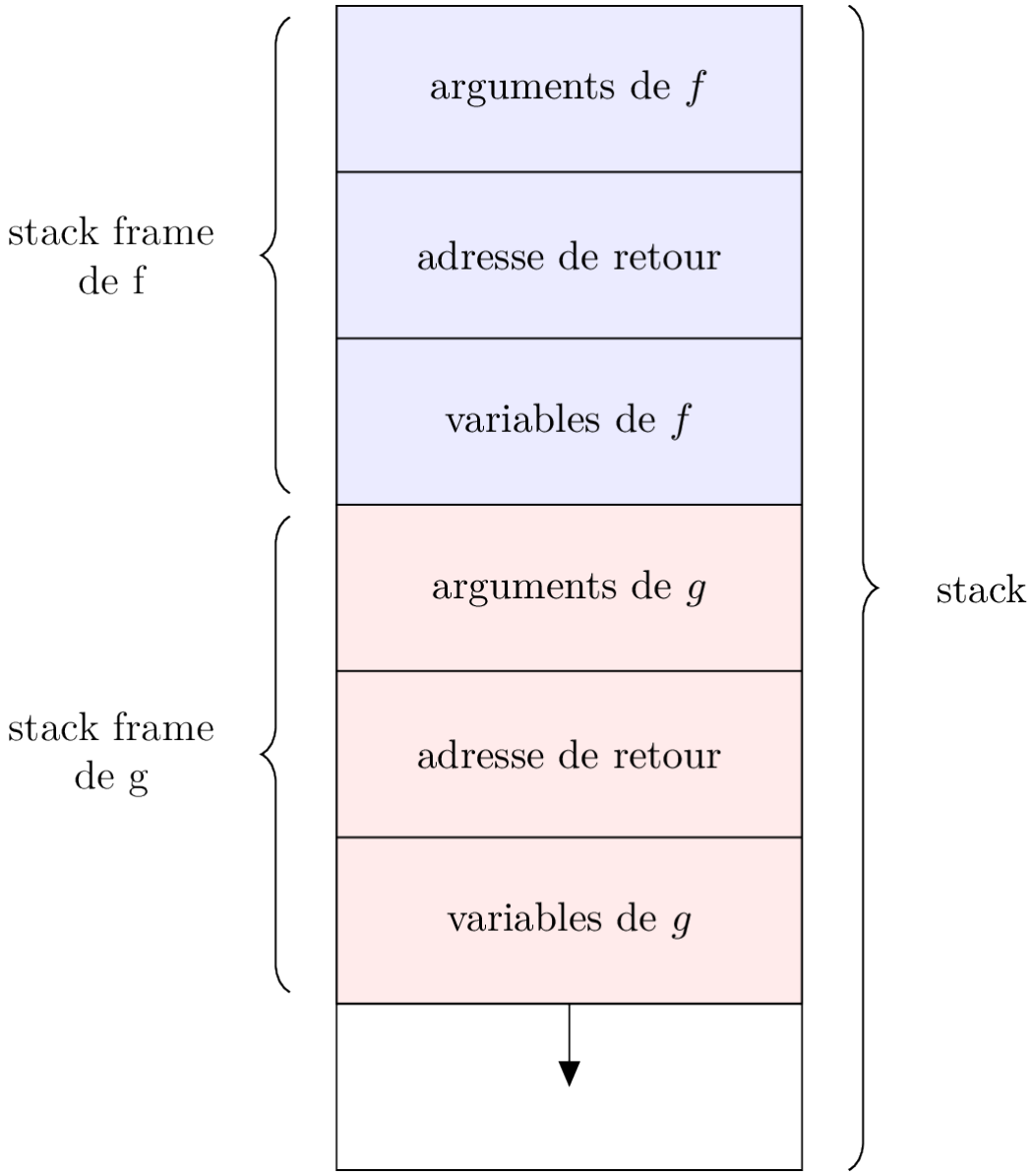
Pile d'appel

Chaque appel de fonction f est empilé sur la pile d'appel avec :

- Les valeurs de ses arguments.
- L'adresse mémoire de la prochaine instruction à exécuter, lorsque l'appel à f sera terminé.
- Les variables locales de f.

Pile d'appel

Par exemple, si une fonction f appelle une fonction g :



Pile d'appel

L'erreur Stack overflow vient d'un trop grand nombre de variables ou d'appels de fonctions sur la pile.

Cela arrive notamment dans le cas d'une fonction récursive qui ne termine pas.