

(6) Récursivité terminale

Programmation fonctionnelle (LU2IN019)

Licence d'informatique
2022/2023

Mathieu Jaume – Adrien Koutsos



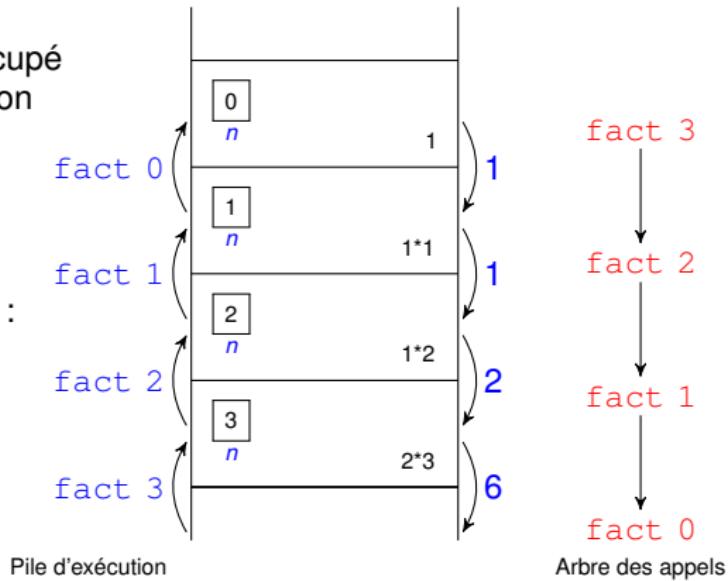
Récursivité : exemples

Espace mémoire (pile d'exécution)

```
let rec fact (n : int) : int =  
  if n=0 then 1 else (fact (n - 1)) * n  
  
fact 3 = (fact 2) * 3 = ((fact 1) * 2) * 3 = (((fact 0) * 1) * 2) * 3
```

l'espace mémoire occupé dans la pile d'exécution est proportionnel au nombre d'appels récursifs

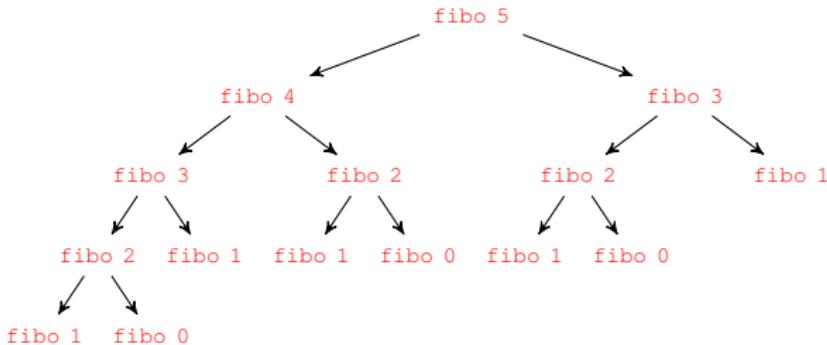
si n est « trop » grand :
Stack overflow



Récurivité : exemples

Duplication des calculs

```
let rec fibo (n : int) : int =
  if n=0 then 0
  else if n=1 then 1
  else (fibo (n - 1)) + (fibo (n - 2))
```



Arbre des appels

`(fibo 5)` est appelé/calculé 1 fois

`(fibo 4)` est appelé/calculé 1 fois

`(fibo 3)` est appelé/calculé 2 fois

`(fibo 2)` est appelé/calculé 3 fois

`(fibo 1)` est appelé/calculé 5 fois

`(fibo 0)` est appelé/calculé 3 fois

Récursivité : exemples

Terminaison : au moins un des cas de base (cas sans appel récursif) doit toujours être atteint au bout d'un **nombre fini d'appels récursifs**

- $f(0) = 2$ et pour $x > 0$, $f(x) = 2 + f(x + 1)$ est mal définie

```
let rec f (x : int) : int =
  if x=0 then 2 else 2 + (f (x + 1))

# f 3;;
Stack overflow during evaluation (looping recursion?).
```

- $g(0) = 2$ et pour $n > 0$, $g(n) = g(n + 1)$ est mal définie

```
let rec g (n : int) : int =
  if n=0 then 2 else g (n + 1)
```

```
# g 3;;
^CInterrupted.
```

(arrêt de l'exécution par l'utilisateur)

deux comportements différents : pourquoi ?

Définition de fonctions récursives terminales

- application de fonctions récursives
 - ▶ plusieurs évaluations du corps de la fonction dans des environnements d'évaluation différents (chaque environnement contient la valeur de l'argument utilisé lors de l'appel)
- complexité en espace mémoire / en temps de calcul ?
 - ▶ pile d'exécution : stockage de valeurs utilisées pour produire le résultat à partir des résultats des appels récursifs
Stack overflow during evaluation (looping recursion?).
 - ▶ duplication des calculs *(exemple : fonction fibo)*
- fonction **récursive terminale** : pas de calcul sur le résultat d'un appel récursif – la valeur renvoyée est le résultat de l'appel récursif
 - ▶ pas de stockage de valeurs dans la pile d'exécution
 - ★ si besoin, des paramètres de la fonction sont ajoutés et utilisés pour stocker ces valeurs (**accumulateurs**)
 - ▶ évaluation sans consommer d'espace sur la pile d'exécution possible : la gestion de la mémoire se déduit trivialement des transformations sur les paramètres

Fonctions récursives terminales : exemples

- factorielle

```
let rec fact (n : int) : int =
  if n=0 then 1 else (fact (n - 1)) * n
```

$$\text{fact } 3 = (\text{fact } 2) * 3 = ((\text{fact } 1) * 2) * 3 = (((\text{fact } 0) * 1) * 2) * 3$$

fact n'est pas récursive terminale

- PGCD de deux entiers

```
let rec pgcd (a : int) (b : int) : int =
  let r = a mod b in
  if r=0 then b
    else pgcd b r
```

$$\text{pgcd } 96 \ 36 = \text{pgcd } 36 \ 24 = \text{pgcd } 24 \ 12 = 12$$

pgcd est récursive terminale

Fonctions récursives terminales : exemples

- `let rec f (x : int) : int =
 if x=0 then 2 else 2 + (f (x + 1))`

n'est pas récursive terminale

~~ utilisation d'espace mémoire dans la pile

```
# f 3;;
```

Stack overflow during evaluation (looping recursion?).

- `let rec g (n:int) : int =
 if n=0 then 2 else g (n + 1)`

est récursive terminale

~~ pas d'utilisation d'espace mémoire dans la pile

```
# g 3;;
```

CInterrupted.

Fonctions récursives terminales sur les entiers

```
let rec fact (n : int) : int =  
  if n=0 then 1 else (fact (n - 1)) * n
```

$$\begin{array}{ccccccc} \text{fact } 3 & = & (\text{fact } 2) * \underbrace{3}_{\text{acc}} & = & ((\text{fact } 1) * \underbrace{2}_{\text{acc}}) * 3 & = & (((\text{fact } 0) * 1) * \underbrace{2}_{\text{acc}}) * 3 \\ 3 & \xrightarrow{\quad \underbrace{1}_{\text{acc}_0} \quad} & \sim 2 & \xrightarrow{\quad \underbrace{3 * 1}_{\text{acc}_1} \quad} & \sim 1 & \xrightarrow{\quad \underbrace{2 * 3 * 1}_{\text{acc}_2} \quad} & \sim 0 & \xrightarrow{\quad \underbrace{1 * 2 * 3 * 1}_{\text{acc}_3} \quad} \end{array}$$

- utilisation d'un accumulateur *acc* pour stocker le résultat en cours de construction

$$\boxed{n} \quad \boxed{\text{acc}} \rightsquigarrow \boxed{n - 1} \quad \boxed{n * \text{acc}}$$

- accumulateur ajouté en paramètre de la fonction récursive

```
let rec aux (n : int) (acc : int) : int =  
  if n=0 then acc  
  else aux (n - 1) (n * acc)
```

Fonctions récursives terminales sur les entiers

```
let rec aux (n : int) (acc : int) : int =  
  if n=0 then acc  
  else aux (n - 1) (n * acc)
```

- que calcule aux ?

$$\forall n \geq 1. \forall i \in \mathbb{N}. \text{aux } n \ i = i * 1 * 2 * \dots * n = i * n!$$

- par récurrence sur n

- si $n = 1$ alors $\text{aux } 1 \ i = \text{aux } 0 \ (1 * i) = i * 1 = i * 1!$
- si $n = k + 1 > 1$ alors :

$$\begin{aligned} & \text{aux } (k+1) \ i \\ = & \text{aux } k \ ((k+1) * i) && (\text{définition de aux}) \\ = & (k+1) * i * 1 * 2 * \dots * k && (\text{hyp. de récurrence}) \\ = & i * 1 * 2 * \dots * k * (k+1) = i * (k+1)! \end{aligned}$$

$$n! = \text{aux } n \ 1$$

Fonctions récursives terminales sur les entiers

- définition de fact à partir de aux

- ▶ aux est une fonction locale à la définition de fact

```
let fact (x : int) : int =
  let rec aux (n : int) (acc : int) : int =
    if n=0 then acc
    else aux (n - 1) (n * acc)
  in
  aux x 1
```

aux est récursive terminale

fact 3 = aux 3 1 = aux 2 3 = aux 1 6 = aux 0 6 = 6

fact 5 = aux 5 1 = aux 4 5 = aux 3 20 = aux 2 60
= aux 1 120 = aux 0 120 = 120

Fonctions récursives terminales sur les entiers

- suite de Fibonacci $F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$ (pour $n \geq 2$)

```
let rec fibo (n : int) : int =
    if n=0 then 0
    else if n=1 then 1
    else (fibo (n - 1)) + (fibo (n - 2))
```

fibo n'est pas récursive terminale

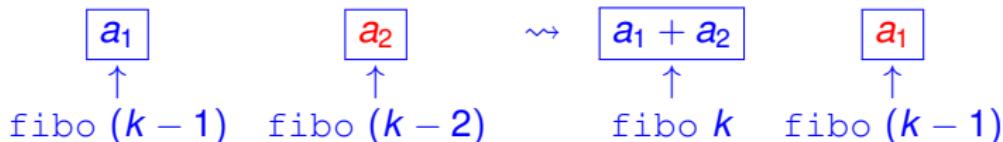
- pour calculer $(\text{fibo } n)$, il faut disposer des résultats des deux appels récursifs $(\text{fibo } (n - 1))$ et $(\text{fibo } (n - 2))$

- utilisation de deux accumulateurs

- ★ a_1 pour $(\text{fibo } (n - 1))$
- ★ a_2 pour $(\text{fibo } (n - 2))$

- initialisation : $a_1 = \text{fibo } 1$ et $a_2 = \text{fibo } 0$

- à chaque étape de calcul :



- calcul de $(\text{fibo } n)$: n étapes de calcul

Fonctions récursives terminales sur les entiers

- utilisation de deux accumulateurs a_1 et a_2 ajoutés en paramètre de la fonction récursive

- initialement : $a_1 = \text{fibo } 1$ et $a_2 = \text{fibo } 0$
- à chaque étape de calcul : a_1 $a_2 \rightsquigarrow a_1 + a_2$ a_1
- calcul de $(\text{fibo } n)$: n étapes de calcul

exemple : calcul de $(\text{fibo } 5)$



```
let rec aux (n : int) (a1 : int) (a2 : int) : int =
  if n=0 then a2
  else aux (n - 1) (a1 + a2) a1
```

Fonctions récursives terminales sur les entiers

- suite de Fibonacci $F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2}$ (pour $n \geq 2$)
- fonction aux

```
let rec aux (n : int) (a1 : int) (a2 : int) : int =
  if n=0 then a2
  else aux (n - 1) (a1 + a2) a1
```

- propriété de aux : $\forall n \in \mathbb{N} \ \forall i \in \mathbb{N}. \ (\text{aux } n \ F_{i+1} \ F_i) = F_{n+i}$
par récurrence sur n

- ▶ si $n = 0$ alors $(\text{aux } 0 \ F_{i+1} \ F_i) = F_i = F_{0+i}$
- ▶ si $n = k + 1$ alors :

$$\begin{aligned}& \text{aux } (k + 1) \ F_{i+1} \ F_i \\&= \text{aux } k \ (F_{i+1} + F_i) \ F_{i+1} \quad (\text{définition de aux}) \\&= \text{aux } k \ F_{i+2} \ F_{i+1} \quad (\text{définition de } F_n) \\&= F_{k+i+1} \quad (\text{hyp. de récurrence}) \\&= F_{(k+1)+i}\end{aligned}$$

$$F_n = \text{aux } n \ F_1 \ F_0$$

Fonctions récursives terminales sur les entiers

- définition de fibo à partir de aux

- ▶ aux est une fonction locale à la définition de fibo

```
let fibo (x : int) : int =
  let rec aux (n : int) (a1 : int) (a2 : int) : int =
    if n=0 then a2
    else aux (n - 1) (a1 + a2) a1
  in
  aux x 1 0
```

aux est récursive terminale

$$\begin{aligned} \text{fibo } 5 &= \text{aux } 5 \ 1 \ 0 = \text{aux } 4 \ 1 \ 1 = \text{aux } 3 \ 2 \ 1 = \text{aux } 2 \ 3 \ 2 \\ &= \text{aux } 1 \ 5 \ 3 = \text{aux } 0 \ 8 \ 5 = 5 \end{aligned}$$

Fonctions récursives terminales sur les entiers

- calcul de x^n : $x^0 = 1$ $x^{n+1} = x^n * x$ (pour $n \geq 1$)

```
let rec pow (x : int) (n : int) : int =
    if n=0 then 1 else (pow x (n - 1)) * x
```

pow n'est pas récursive terminale

$$\begin{array}{ccccccc} \text{pow } x \cdot 3 & = & (\text{pow } x \cdot 2) * \underbrace{x}_{\text{acc}} & = & ((\text{pow } x \cdot 1) * \underbrace{x}_{\text{acc}}) * x & = & (((\text{pow } x \cdot 0) * \underbrace{x}_{\text{acc}}) * x) * x \\ 3 \underbrace{1}_{\text{acc}_0} & \rightsquigarrow & 2 \underbrace{x * 1}_{\text{acc}_1} & \rightsquigarrow & 1 \underbrace{x * x * 1}_{\text{acc}_2} & \rightsquigarrow & 0 \underbrace{x * x * x * 1}_{\text{acc}_3} \end{array}$$

- utilisation d'un accumulateur *acc* pour stocker le résultat en cours de construction

$$\boxed{n} \quad \boxed{\text{acc}} \rightsquigarrow \boxed{n - 1} \quad \boxed{x * \text{acc}}$$

- accumulateur ajouté en paramètre de la fonction récursive

```
let rec aux (x : int) (n : int) (acc : int) : int =
    if n=0 then acc
    else aux x (n - 1) (x * acc)
```

Fonctions récursives terminales sur les entiers

```
let rec aux (x : int) (n : int) (acc : int) : int =  
  if n=0 then acc  
  else aux x (n - 1) (x * acc)
```

- que calcule aux ?

$$\forall x \in \mathbb{N} \quad \forall i \in \mathbb{N} \quad (\text{aux } x \ n \ i) = i * x^n$$

- par récurrence sur n

- si $n = 0$ alors $(\text{aux } x \ 0 \ i) = i = 1 * i = x^0 * i$
- si $n = k + 1 > 1$ alors :

$$\begin{aligned}& \text{aux } x \ (k+1) \ i \\&= \text{aux } x \ k \ (x * i) \quad (\text{définition de aux}) \\&= x * i * x^k \quad (\text{hyp. de récurrence}) \\&= i * x^{k+1}\end{aligned}$$

$$x^n = \text{aux } x \ n \ 1$$

Fonctions récursives terminales sur les entiers

définition de `pow` à partir de `aux`

- aux est une fonction locale à la définition de `pow`

```
let pow (y : int) (p : int) : int =
  let rec aux (x : int) (n : int) (acc : int) : int =
    if n=0 then acc
    else aux x (n - 1) (x * acc)
  in aux y p 1
```

- paramètres de `aux`

- le paramètre `y` de la fonction `pow` est accessible dans la définition de `pow` (et donc `y` est accessible dans la fonction locale `aux`)
- à chaque appel récursif le premier argument (paramètre `x`) est identique (et a la valeur de `y`)

```
let pow (y : int) (p : int) : int =
  let rec aux (n : int) (acc : int) : int =
    if n=0 then acc
    else aux (n - 1) (y * acc)
  in aux p 1
```

Fonctions récursives terminales sur les entiers

- définition de pow à partir de aux

```
let pow (y : int) (p : int) : int =
  let rec aux (n : int) (acc : int) : int =
    if n=0 then acc
    else aux (n - 1) (y * acc)
  in aux p 1
```

aux est récursive terminale

pow 2 3 = aux 3 1 = aux 2 2 = aux 1 4 = aux 0 8 = 8

Itération et programmation fonctionnelle

Itération et programmation fonctionnelle

- calcul de $(\text{pow } x \ n) = (\text{aux } n \ 1)$
 - ▶ initialisation $\text{acc} = 1$
 - ▶ itérer n fois la fonction $\text{mult_x} : \text{acc} \mapsto x * \text{acc}$
- itération d'une fonction $f : A \rightarrow A \quad f^n = \underbrace{f \circ f \circ \dots \circ f}_{n \text{ fois}}$
 - ▶ $f^n : x \mapsto \begin{cases} x & \text{si } n = 0 \\ f(f^k(x)) & \text{si } n = k + 1 \end{cases}$
 - ```
let rec iter_f (n : int) (f : 'a -> 'a) (x : 'a) : 'a =
 if n=0 then x
 else f (iter_f (n - 1) f x)
```
  - ▶  $f^n = \begin{cases} \text{fonction identité sur } A & \text{si } n = 0 \\ f \circ f^k & \text{si } n = k + 1 \end{cases}$
  - ```
let compose (f:'b -> 'c) (g:'a -> 'b) : 'a -> 'c =
  fun x -> f (g x)
```
 - ```
let rec iter_f (n : int) (f : 'a -> 'a) : 'a -> 'a =
 if n=0 then (fun (x : 'a) : 'a -> x)
 else compose f (iter_f (n - 1) f)
```

# Itération et programmation fonctionnelle

---

- calcul de  $(\text{pow } x \ n) = (\text{aux } n \ 1)$ 
  - ▶ initialisation  $\text{acc} = 1$
  - ▶ itérer  $n$  fois la fonction  $\text{mult\_x} : \text{acc} \mapsto x * \text{acc}$
- itération d'une fonction  $f : A \rightarrow A$   $f^n = \underbrace{f \circ f \circ \dots \circ f}_{n \text{ fois}}$
- définition de  $\text{pow}$

```
let pow x n =
 let mult_x = fun a -> x*a in
 (iter_f n mult_x) 1
```

## Fonctions récursives terminales sur les listes

# Fonctions récursives terminales sur les listes

- somme des éléments d'une liste d'entiers

```
let rec somme (l : int list) : int =
 match l with
 | [] -> 0
 | h :: t -> h + somme t
```

$$\begin{aligned}\text{somme } [4; 1; 3] &= \underbrace{4}_{\text{acc}} + (\text{somme } [1; 3]) = \underbrace{4+1}_{\text{acc}} + (\text{somme } [3]) \\ &= \underbrace{4+1+3}_{\text{acc}} + (\text{somme } []) = \underbrace{4+1+3+0}_{\text{acc}} = 8\end{aligned}$$

- version récursive terminale

```
let somme (l : int list) : int =
 let rec aux (acc : int) (la : int list) : int =
 match la with
 | [] -> acc
 | h :: t -> aux (h + acc) t
 in aux 0 l
```

$$\begin{aligned}&\text{somme } [4; 1; 3] \\ &= \text{aux } 0 [4; 1; 3] = \text{aux } 4 [1; 3] = \text{aux } 5 [3] = \text{aux } 8 [] = 8\end{aligned}$$

# Fonctions récursives terminales sur les listes

- longueur d'une liste

```
let rec length (l : 'a list) : int =
 match l with
 | [] -> 0
 | h :: t -> 1 + (length t)
```

$$\begin{aligned} \text{length } [4; 1; 3] &= \underbrace{1}_{acc=1} + (\text{length } [1; 3]) \\ &= \underbrace{1+1}_{acc=2} + (\text{length } [3]) = \underbrace{1+1+1}_{acc=3} + (\text{length } []) = 3 \end{aligned}$$

- version récursive terminale

```
let length (l : 'a list) : int =
 let rec aux (acc : int) (la : 'a list) =
 match la with
 | [] -> acc
 | h :: t -> aux (acc + 1) t
 in aux 0 l
```

$$\begin{aligned} \text{length } [4; 1; 3] &= \text{aux } 0 [4; 1; 3] \\ &= \text{aux } 1 [1; 3] = \text{aux } 2 [3] = \text{aux } 3 [] = 3 \end{aligned}$$

# Fonctions récursives terminales sur les listes

- fonction qui multiplie par  $n$  tous les entiers d'une liste d'entiers

$$\text{mult\_l } [] \ n = []$$

$$\text{mult\_l } [e_1; e_2; \dots; e_n] \ n = (e_1 * n) :: (\text{mult\_l } [e_2; \dots; e_n] \ n)$$

```
let rec mult_l (l : int list) (n : int) : int list =
 match l with
 | [] -> []
 | h :: t -> (h * n) :: (mult_l t n)
```

$$\text{mult\_l } [4; 1; 3] \ 2 = \underbrace{8}_{\text{acc}} :: (\text{mult\_l } [1; 3] \ 2)$$

$$= \underbrace{8}_{\text{acc}} :: \underbrace{(2 :: (\text{mult\_l } [3] \ 2))}_{\text{acc}} = \underbrace{8}_{\text{acc}} :: \underbrace{(2 :: (6 :: (\text{mult\_l } [])))}_{\text{acc}}$$
$$= 8 :: (2 :: (6 :: [])) = [8; 2; 6]$$

problème : construction de l'accumulateur en ajoutant en fin de liste

# Fonctions récursives terminales sur les listes

- fonction qui multiplie par *n* tous les entiers d'une liste d'entiers

*version récursive terminale*

```
let mult_l (l : int list) (n : int) : int list =
 let rec aux (la : int list) (acc : int list) : int list =
 match la with
 | [] -> acc
 | h :: t -> aux t ((h * n) :: acc)
 in
 aux l []
```

$$\begin{aligned} \text{mult\_l } [4; 1; 3] \ 2 &= \text{aux } [4; 1; 3] [] = \text{aux } [1; 3] [8] \\ &= \text{aux } [3] [2; 8] = \text{aux } [] [6; 2; 8] \end{aligned}$$

construction de l'accumulateur en ajoutant en tête de liste mais obtention de la liste résultat « à l'envers »

# Fonctions récursives terminales sur les listes

- fonction qui multiplie par *n* tous les entiers d'une liste d'entiers

*version récursive terminale*

```
let mult_l (l : int list) (n : int) : int list =
 let rec aux (la : int list) (acc : int list) : int list =
 match la with
 | [] -> List.rev acc
 | h :: t -> aux t ((h * n) :: acc)
 in
 aux l []

mult_l [4;1;3] 2;;
- : int list = [8; 2; 6]
```

construction du résultat en « 2 passes » :

- 1 un parcours de la liste en argument pour construire la liste résultat « à l'envers »
- 2 un parcours pour « renverser » la liste

... on accepte de perdre en temps de calcul pour gagner en espace mémoire utilisé

## Récurivité terminale et arbres binaires

# Récursivité terminale et arbres binaires

- taille d'un arbre binaire

```
let rec size (t : 'a btree) : int =
 match t with
 | Empty -> 0
 | Node(e, g, d) -> 1 + (size g) + (size d)
```

non récursive terminale

- définition récursive terminale de la taille d'un arbre binaire

- définition d'une fonction locale récursive terminale avec pour paramètres :

- un accumulateur `acc` (résultat partiel du calcul)
    - la** donnée qu'il reste à traiter pour calculer le résultat final

- il reste à traiter 2 arbres lors du premier appel, puis potentiellement plusieurs arbres lors des appels suivants

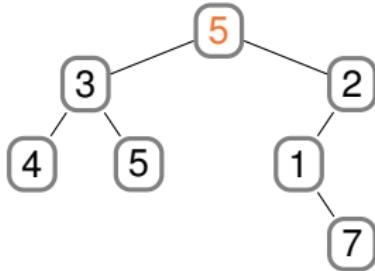
- la** donnée qu'il reste à traiter pour calculer le résultat final est une liste d'arbres binaires

# Récursivité terminale et arbres binaires

- définition récursive terminale de la taille d'un arbre binaire

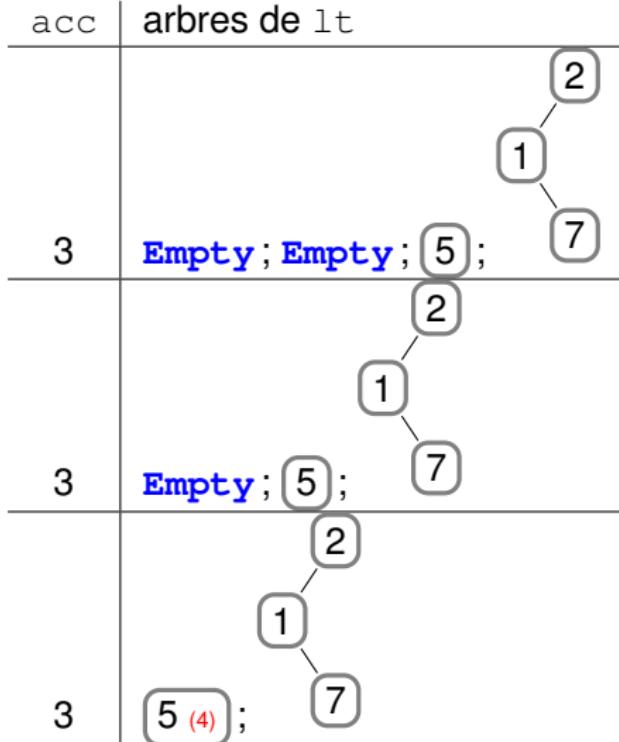
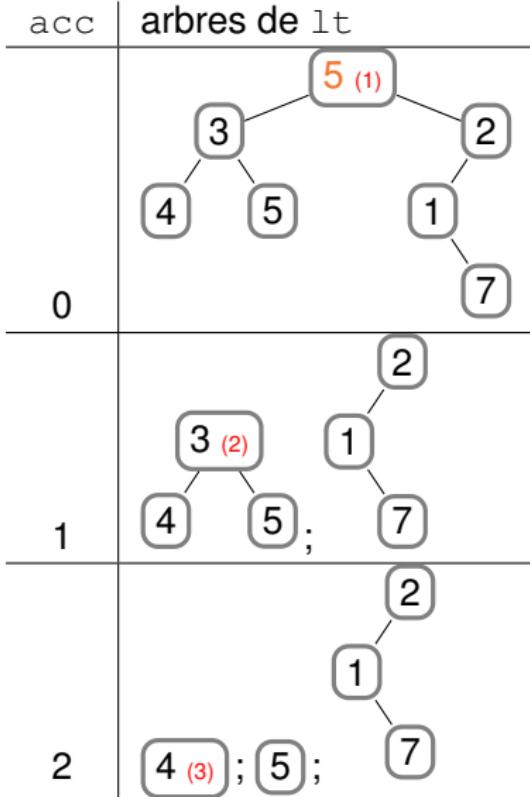
```
let size (t : 'a btree) : int =
 let rec aux (acc : int) (lt : 'a btree list) : int =
 match lt with
 | [] -> acc
 | Empty :: tlt -> aux acc tlt
 | Node (e,g,d) :: tlt -> aux (acc + 1) (g :: (d :: tlt))
 in
 aux 0 [t]
```

arbre t

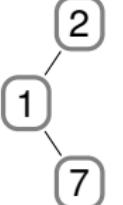
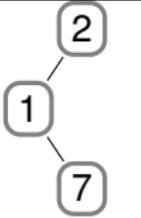
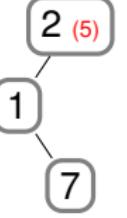


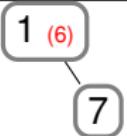
```
(size t);;
- : int 7
```

# Récursivité terminale et arbres binaires



# Récursivité terminale et arbres binaires

| acc | arbres de lt                                                                                                      |
|-----|-------------------------------------------------------------------------------------------------------------------|
| 4   | <br><code>Empty ; Empty ;</code> |
| 4   | <br><code>Empty ;</code>         |
| 4   | <br><code>Empty</code>           |

| acc | arbres de lt                                                                                                             |
|-----|--------------------------------------------------------------------------------------------------------------------------|
| 5   | <br><code>1 (6) ; Empty</code>          |
| 6   | <br><code>Empty ; 7 ; Empty</code>    |
| 6   | <br><code>7 (7) ; Empty</code>         |
| 7   | <br><code>Empty ; Empty ; Empty</code> |
| 7   | <br><code>Empty ; Empty</code>         |
| 7   | <br><code>Empty</code>                 |
| 7   |                                                                                                                          |

- la liste des arbres à traiter est utilisée comme une pile (*Last In First Out*)
  - ▶ les sommets sont énumérés dans l'ordre du parcours en profondeur