

## **Cours 4 : Définition de types, types récurifs généraux et arbres**

---

## Objectif

- Pour l'instant : que des types prédéfinis
- Objectif : définir ses propres types

## Structure des types en OCAML

- Un type se définit comme un ensemble de "situations différentes" représentées chacune par un identifiant spécial: un **constructeur**.
- Un filtrage par cas est alors possible (comme sur les listes).
- Les constructeurs peuvent avoir des paramètres (syntaxe similaire aux fonctions).
- Les types peuvent être récursifs.
- Les constructeurs sont appelés ainsi car ils permettent de **construire** une donnée plus grande à partir de données plus petites.

## Définition de types

---

Si le type de données à définir ne comporte qu'un seul cas non récursif, alors l'usage des constructeurs peut être évité.

## Exemple

```
type 'a file = 'a list * 'a list
```

## Alias

Il s'agit d'un **alias** de type : l'utilisation du type `'a file` ci-dessus est identique à l'utilisation de `'a list * 'a list`.

## Utilité

Ils sont principalement utilisés :

- dans les modules
  - pour masquer une implantation (cf TD1)
  - pour définir un type exigé par l'interface du module (cf TD4)
- à des fins de documentation.

Un type peut être défini à l'aide de plusieurs constructeurs sans paramètre (similaire aux énumérations).

### Exemple

```
type couleur = Bleu | Blanc | Vert
```

```
type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche
```

- Attention : **B**leu, **B**lanc, **V**ert, ... deviennent des mots clés.
- Les constructeurs commencent par une majuscule, alors que les identificateurs commencent par une minuscule.

Le mécanisme de filtrage s'applique aux types définis.

## Filtrage

```
(* Fonction qui verifie si le jour est un vendredi 13 *)
```

```
#let chance (j, d) =
```

```
  match j, d with
```

```
    | Vendredi, 13 -> true
```

```
    | _              -> false;;
```

```
val chance : jour * int -> bool
```

```
#let chance (j, d) =
```

```
  j = Vendredi && d = 13;;
```

```
val chance : jour * int -> bool
```

# Constructeurs avec arguments



Les constructeurs peuvent prendre des arguments (similaire aux fonctions).

## Exemple

```
type num = Entier of int | Flottant of float  
type complexe = Polaire of float * float | Cartesien of float * float
```

## Définition d'expressions

```
# Entier 1;;  
- : num = Entier 1  
  
# Polaire (1.0, atan 1.);;  
- : complexe = Polaire (1.0, 0.78...)  
  
# Cartesien (sqrt 2., sqrt 2.);;  
- : complexe = Cartesien (1.41..., 1.41...)
```

Le mécanisme de filtrage s'applique aux types définis.

## Filtrage

```
(* conversion : complexe -> complexe *)  
(* Conversion de complexe, de polaire vers cartésien *)  
let conversion c =  
  match c with  
  | Polaire (mod, angle) -> Cartésien (mod *. cos angle, mod *. sin angle)  
  | -                      -> c  
  
(* egal_complexe : complexe -> complexe -> bool *)  
(* Teste l'egalite entre deux complexes *)  
let egal_complexe c1 c2 =  
  conversion c1 = conversion c2
```



# Constructeurs avec arguments

Un type peut mélanger des constructeurs constants et avec arguments

## Exemple du type `option`

- le type `option` est une **structure de données** servant à spécifier un choix. entre une situation normale et une situation exceptionnelle.
- On peut faire ce parallèle avec les autres structures de données:
  - une paire encode 2 valeurs.
  - un  $n$ -uplet encode  $n$  valeurs.
  - une liste encode de 0 à  $n$  valeurs.
  - une option encode 0 ou 1 valeur.

## Définition du type `option`

```
type 'a option =  
| None  
| Some of 'a
```

# Constructeurs avec arguments

## Utilisation du type `option`

Quand une fonction peut échouer, il est possible de :

- Renvoyer une valeur d'erreur - mauvaise idée!
- Lever une exception.

Le type `option` est une alternative à l'utilisation d'exceptions.

## Exemple d'utilisation du type `option`

```
(* Recherche dans une liste associative *)
(* 'a -> ('a * 'b) list -> 'b *)
let rec assoc cle l =
  match l with
  | [] -> throw Not_found
  |(c,v)::q ->
    if (c=cle) then v else assoc cle q
(* appel de assoc *)
try
  let v = assoc ... in (* code A *)
with
  Not_found -> (* code B *)
```

```
(* Recherche dans une liste associative *)
(* 'a -> ('a * 'b) list -> 'b option *)
let rec assoc cle l =
  match l with
  | [] -> None
  |(c,v)::q ->
    if (c=cle) then Some v else assoc cle q
(* appel de assoc *)
match assoc ... with
| Some v -> (* code A *)
| None -> (* code B *)
```

# Types récurrents

Les types peuvent également être récurrents (similaire aux fonctions récurrentes).

## Exemple

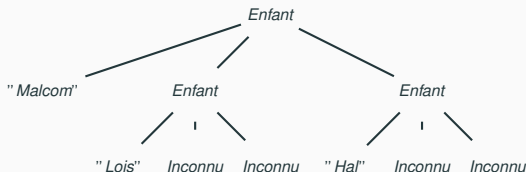
*(\* type pour pouvoir représenter des arbres généalogiques \*)*

**type** **personne** = Inconnu | **Enfant** **of** **string** \* **personne** \* **personne**

## Définition d'expressions

**#** **Enfant** ("Malcolm", **Enfant**("Lois", Inconnu, Inconnu), **Enfant**("Hal", Inconnu, Inconnu));;  
- : **personne** =

**Enfant** ("Malcolm", **Enfant** ("Lois", Inconnu, Inconnu),  
**Enfant** ("Hal", Inconnu, Inconnu))



# Constructeurs avec arguments

Le mécanisme de filtrage s'applique aux types récurifs.

## Filtrage

*(\* retourne le nom d'une personne \*)*

```
let nom personne =  
  match personne with  
  | Inconnu      -> failwith "nom: inconnu"  
  | Enfant (n, _, _) -> n
```

*(\* retourne le nombre maximum de generations d'ancetres connues \*)*

```
let rec max_generations personne =  
  match personne with  
  | Inconnu      -> 0  
  | Enfant (_, parent1, parent2) ->  
    1 + max (max_generations parent1) (max_generations parent2)
```

# Types rékursifs paramétrés

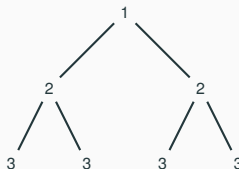
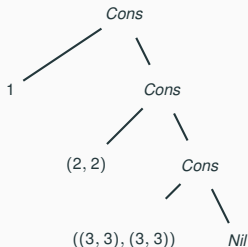
Les paramètres ne sont même pas nécessairement homogènes.

## Exemple

```
type 'a power_list = Nil | Cons of 'a * ('a * 'a) power_list
```

## Définition d'expressions

```
# Cons (1, Cons((2,2), Cons(((3,3),(3,3)), Nil )));;  
- : int power_list = Cons (1, Cons ((2, 2), Cons (((3, 3), (3, 3)), Nil )))
```



## Arbre binaire

---

## Spécification

Un arbre binaire contient :

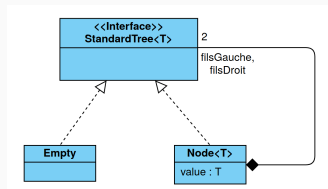
- des nœuds
- des branches (au plus 2 par nœuds)
- des données
  - dans les nœuds
  - dans les branches
  - dans les feuilles

Tous les mélanges sont bien sûr possibles, avec des éléments de types différents.

# Arbre binaire ♪♪♪

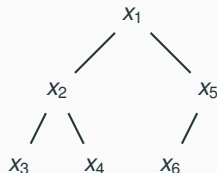
## Arbre binaire standard (données dans les nœuds)

```
type 'a standard_tree =  
  | Empty  
  | Node of 'a*'a standard_tree*'a standard_tree
```



## Arbre binaire standard - Exemple

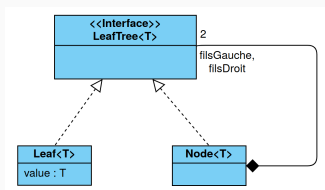
```
Node ("x1",  
  Node ("x2",  
    Node ("x3",Empty,Empty),  
    Node ("x4",Empty,Empty)  
  ),  
  Node ("x5",  
    Node ("x6",Empty,Empty),  
    Empty  
  )  
)
```





## Arbre binaire avec données dans les feuilles

```
type 'a leaf_tree =  
| Leaf of 'a  
| Node of 'a leaf_tree * 'a leaf_tree
```



## Arbre binaire avec données dans les feuilles - Exemple

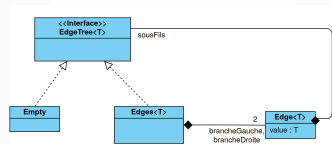
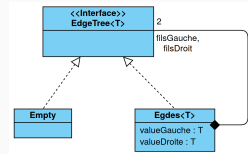
```
Node (Node (Leaf "x1",  
            Leaf "x2"),  
      Leaf "x3")
```



## Arbre binaire avec données dans les branches

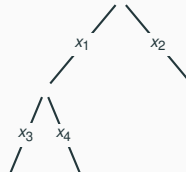
```
type 'a edge_tree =  
  | Empty  
  | Edges of ('a*'a edge_tree)*('a*'a edge_tree)
```

```
type 'a edge = 'a * 'a edge_tree  
and 'a edge_tree =  
  | Empty  
  | Edges of 'a edge * 'a edge
```



## Arbre binaire avec données dans les branches - Exemple

```
Edges ("x1",  
      Edges(("x3",Empty),  
            ("x4",Empty))),  
      ("x2",  
       Empty)  
)
```

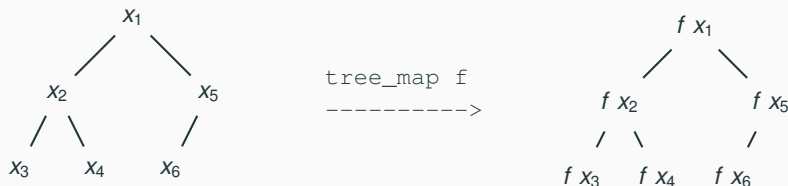


## Exemple de fonction

- Nous souhaitons définir une fonction qui compte le nombre d'éléments d'un arbre.
- La structure récursive du type `'a standard_tree` donne la structure récursive de la fonction.
- Analyse récursive : Si je sais calculer la taille des deux fils de la racine de l'arbre, comment est-ce que je calcule la taille de l'arbre ?
- Addition des deux tailles, et incrémentation (pour la racine).

```
(* cardinal : 'a standard_tree -> int *)  
(* Renvoie le nombre d'elements d'un arbre *)  
let rec cardinal arb =  
  match arb with  
  | Empty          -> 0  
  | Node (_, g, d) -> 1 + cardinal g + cardinal d
```

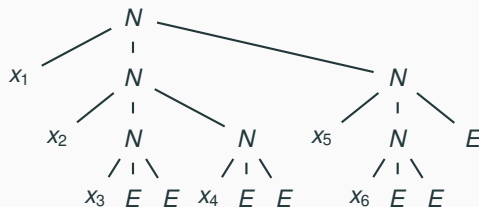
## Itérateur `tree_map`



## Itérateur `tree_map` - Code

```
(* tree_map : ('a -> 'b) -> 'a standard_tree -> 'b standard_tree *)  
let rec tree_map f arb =  
  match arb with  
  | Empty      -> Empty  
  | Node (n, g, d) -> Node (f n, tree_map f g, tree_map f d)
```

## Autre représentation de l'arbre



## Itérateur `tree_fold`



## Itérateur `tree_fold` - Code

```
(* tree_fold : ('a -> 'b -> 'b -> 'b) -> 'b -> 'a standard_tree -> 'b *)  
let rec tree_fold f e arb =  
  match arb with  
  | Empty      -> e  
  | Node (n, g, d) -> f n (tree_fold f e g) (tree_fold f e d)
```

## Itérateur `tree_fold` - Utilisation

- Ré-écrire la fonction `cardinal` à l'aide d'un itérateur
- Code:

## Itérateur `tree_fold` - Utilisation

- Ré-écrire la fonction `cardinal` à l'aide d'un itérateur
- Code:

```
(* cardinal : 'a standard_tree -> int *)  
(* Renvoie le nombre d'elements d'un arbre *)  
let cardinal arb =  
  tree_fold (fun _ cardinal_g cardinal_d -> 1 + cardinal_g + cardinal_d) 0 arb
```



## Itérateur `fold` - Généralisation

- `tree.fold` peut être généralisé en `fold` pour tout type OCaml.
- `fold` "remplace" les constructeurs du type par des appels de fonctions de même arité.
- Le nombre et le type des paramètres de l'itérateur `fold` dépendent donc du nombre et du type des constructeurs de la structure de données sur laquelle l'itérateur s'applique.

**Structure de données "efficace" :**  
**Arbres binaires à gauche**

---

## Contrainte structurelle

- Mise en place d'**invariants structurels** :
- Propriétés locales à chaque nœud qui ne peuvent être exprimées par le type seul et qui garantissent globalement l'efficacité des opérations.
- Exemples :
  - des propriétés numériques portant sur la taille, la profondeur, etc.
  - des propriétés d'ordre entre éléments.
  - des propriétés portant sur des données auxiliaires ajoutées (arbres colorés Rouge-Noir par exemple).
- Aide à l'obtention d'une représentation canonique.
- Nécessité que les types soient abstraits/privés.

# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Spécification

Les arbres binaires à gauches respectent le schéma de type standard des arbres binaires :

```
type 'a abg = Vide | Noeud of 'a abg * 'a * 'a abg
```

et reposent sur ces deux principes :

- **Invariant 1** : Les éléments de l'arbre sont ordonnés en tas ("heap ordered"), i.e. pour tout sous-arbre non vide, l'élément à sa racine est toujours inférieur à chaque élément présent dans ses fils gauche et droit.
- **Invariant 2** : Les branches penchent à gauche, i.e. pour tout sous-arbre non vide, son fils gauche est au moins aussi profond que son fils droit.

## Arbres binaires à gauche - Complexité

La complexité du pire cas des opérations suivantes est logarithmique:

- insertion
- union
- retrait de l'élément minimal

## Arbres binaires à gauche - Union

L'union entre deux arbres binaires à gauches non vides (seul cas non trivial) peut se décomposer en deux phases :

- Décomposition : on insère l'arbre de racine la plus grande dans le fils droit de l'autre arbre, afin de respecter l'invariant 1.
- Recomposition : on échange les fils gauche et droit du résultat si besoin est, afin de respecter l'invariant 2.

# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Implantation

```
type 'a abg = Vide | Noeud of 'a abg * 'a * 'a abg
```

# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Implantation

```
type 'a abg = Vide | Noeud of 'a abg * 'a * 'a abg
```

*(\* profondeur 'a abg -> int \*)*

*(\* Calcule la profondeur maximale d'un arbre binaire a gauche \*)*

# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Implantation

```
type 'a abg = Vide | Noeud of 'a abg * 'a * 'a abg
```

```
(* profondeur 'a abg -> int *)
```

```
(* Calcule la profondeur maximale d'un arbre binaire a gauche *)
```

```
(* On s'intéresse à la branche la plus à gauche, c'est par définition la plus profonde *)
```

```
let rec profondeur a = match a with
```

```
  | Vide          -> 0
```

```
  | Noeud(g,r,d) -> 1 + profondeur g
```



# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Implantation

```
type 'a abg = Vide | Noeud of 'a abg * 'a * 'a abg
```

```
(* profondeur 'a abg -> int *)
```

```
(* Calcule la profondeur maximale d'un arbre binaire a gauche *)
```

```
(* On s'interesse a la branche la plus a gauche, c'est par definition la plus profonde *)
```

```
let rec profondeur a = match a with
```

```
  | Vide          -> 0
```

```
  | Noeud(g,r,d) -> 1 + profondeur g
```

```
(* noeud : 'a abg -> 'a -> 'a abg -> 'a abg *)
```

```
(* Constructeur abstrait qui construit un noeud en permutant les 2 branches si necessaire *)
```

```
(* afin de respecter l'invariant 2 *)
```

# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Implantation

```
type 'a abg = Vide | Noeud of 'a abg * 'a * 'a abg
```

```
(* profondeur 'a abg -> int *)
```

```
(* Calcule la profondeur maximale d'un arbre binaire a gauche *)
```

```
(* On s'intéresse à la branche la plus à gauche, c'est par définition la plus profonde *)
```

```
let rec profondeur a = match a with
```

```
  | Vide          -> 0
```

```
  | Noeud(g,r,d) -> 1 + profondeur g
```

```
(* noeud : 'a abg -> 'a -> 'a abg -> 'a abg *)
```

```
(* Constructeur abstrait qui construit un noeud en permutant les 2 branches si nécessaire *)
```

```
(* afin de respecter l'invariant 2 *)
```

```
let noeud g r d =
```

```
  if profondeur g < profondeur d then Noeud (d, r, g) else Noeud (g, r, d)
```

# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Implantation

```
type 'a abg = Vide | Noeud of 'a abg * 'a * 'a abg
```

```
(* profondeur 'a abg -> int *)
```

```
(* Calcule la profondeur maximale d'un arbre binaire a gauche *)
```

```
(* On s'interesse a la branche la plus a gauche, c'est par definition la plus profonde *)
```

```
let rec profondeur a = match a with
```

```
  | Vide          -> 0
```

```
  | Noeud(g,r,d) -> 1 + profondeur g
```

```
(* noeud : 'a abg -> 'a -> 'a abg -> 'a abg *)
```

```
(* Constructeur abstrait qui construit un noeud en permutant les 2 branches si necessaire *)
```

```
(* afin de respecter l'invariant 2 *)
```

```
let noeud g r d =
```

```
  if profondeur g < profondeur d then Noeud (d, r, g) else Noeud (g, r, d)
```

```
(* union : 'a abg -> 'a abg -> 'a abg. Calcule l'union de deux arbres binaires a gauche *)
```

# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Implantation

```
type 'a abg = Vide | Noeud of 'a abg * 'a * 'a abg
```

```
(* profondeur 'a abg -> int *)
```

```
(* Calcule la profondeur maximale d'un arbre binaire a gauche *)
```

```
(* On s'interesse a la branche la plus a gauche, c'est par definition la plus profonde *)
```

```
let rec profondeur a = match a with
```

```
| Vide -> 0
```

```
| Noeud(g,r,d) -> 1 + profondeur g
```

```
(* noeud : 'a abg -> 'a -> 'a abg -> 'a abg *)
```

```
(* Constructeur abstrait qui construit un noeud en permutant les 2 branches si necessaire *)
```

```
(* afin de respecter l'invariant 2 *)
```

```
let noeud g r d =
```

```
if profondeur g < profondeur d then Noeud(d, r, g) else Noeud(g, r, d)
```

```
(* union : 'a abg -> 'a abg -> 'a abg. Calcule l'union de deux arbres binaires a gauche *)
```

```
let rec union abr1 abr2 = match abr1, abr2 with
```

```
| Vide, _ -> abr2
```

```
| _, Vide -> abr1
```

```
| Noeud(g1,r1,d1), Noeud(g2,r2,d2) ->
```

```
if (r1 > r2) then noeud g2 r2 (union abr1 d2)
```

```
else noeud g1 r1 (union abr2 d1)
```

## Arbres binaires à gauche - Implantation

(*\* ajout : 'a -> 'a abg -> 'a abg \**)

(*\* Ajoute un element a un arbre binaire a gauche \**)

## Arbres binaires à gauche - Implantation

*(\* ajout : 'a -> 'a abg -> 'a abg \*)*

*(\* Ajoute un element a un arbre binaire a gauche \*)*

**let** ajout e arb =

union (Noeud(Vide, e, Vide)) arb

# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Implantation

```
(* ajout : 'a -> 'a abg -> 'a abg *)
```

```
(* Ajoute un element a un arbre binaire a gauche *)
```

```
let ajout e arb =
```

```
  union (Noeud(Vide, e, Vide)) arb
```

```
(* minimum : 'a abg -> 'a *)
```

```
(* Renvoie le minimum d'un arbre binaire a gauche *)
```

```
(* Erreur si l'arbre est vide *)
```

# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Implantation

```
(* ajout : 'a -> 'a abg -> 'a abg *)
```

```
(* Ajoute un element a un arbre binaire a gauche *)
```

```
let ajout e arb =
```

```
  union (Noeud(Vide, e, Vide)) arb
```

```
(* minimum : 'a abg -> 'a *)
```

```
(* Renvoie le minimum d'un arbre binaire a gauche *)
```

```
(* Erreur si l'arbre est vide *)
```

```
let minimum abr = match abr with
```

```
  | Vide          -> failwith "Arbre vide!"
```

```
  | Noeud(g,r,d) -> r
```



# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Implantation

```
(* ajout : 'a -> 'a abg -> 'a abg *)
```

```
(* Ajoute un element a un arbre binaire a gauche *)
```

```
let ajout e arb =
```

```
  union (Noeud(Vide, e, Vide)) arb
```

```
(* minimum : 'a abg -> 'a *)
```

```
(* Renvoie le minimum d'un arbre binaire a gauche *)
```

```
(* Erreur si l'arbre est vide *)
```

```
let minimum abr = match abr with
```

```
  | Vide          -> failwith "Arbre vide!"
```

```
  | Noeud(g,r,d) -> r
```

```
(* retrait_min : 'a abg -> 'a abg *)
```

```
(* Retire son minimum a un arbre binaire a gauche *)
```

```
(* Erreur si l'arbre est vide *)
```

# Structure de données "efficace" : Arbres binaires à gauche

## Arbres binaires à gauche - Implantation

```
(* ajout : 'a -> 'a abg -> 'a abg *)
```

```
(* Ajoute un element a un arbre binaire a gauche *)
```

```
let ajout e arb =
```

```
  union (Noeud(Vide, e, Vide)) arb
```

```
(* minimum : 'a abg -> 'a *)
```

```
(* Renvoie le minimum d'un arbre binaire a gauche *)
```

```
(* Erreur si l'arbre est vide *)
```

```
let minimum abr = match abr with
```

```
  | Vide          -> failwith "Arbre vide!"
```

```
  | Noeud(g,r,d) -> r
```

```
(* retrait_min : 'a abg -> 'a abg *)
```

```
(* Retire son minimum a un arbre binaire a gauche *)
```

```
(* Erreur si l'arbre est vide *)
```

```
let retrait_min abr = match abr with
```

```
  | Vide          -> failwith "Arbre vide!"
```

```
  | Noeud (g,r,d) -> union g d
```

## **Parcours d'arbres binaires**

---

## Spécification

- Un parcours des éléments d'un arbre consiste à présenter ses éléments séquentiellement ;
- en vue d'itérer un traitement particulier sur cette séquence.
- Contrairement aux listes, où un seul type de parcours existe,
- le cas des arbres donne lieu à de multiples possibilités.
- On envisagera néanmoins uniquement les parcours de gauche à droite.

## Analyse fonctionnelle

1. Construire la liste (séquence) des éléments, dans l'ordre où le traitement à itérer les trouverait.
2. Appliquer itérativement ce traitement sur la liste obtenue (avec un `List.foldLeft` par exemple).

On ne s'intéressera donc qu'à la construction de la dite liste.

## Deux types de parcours

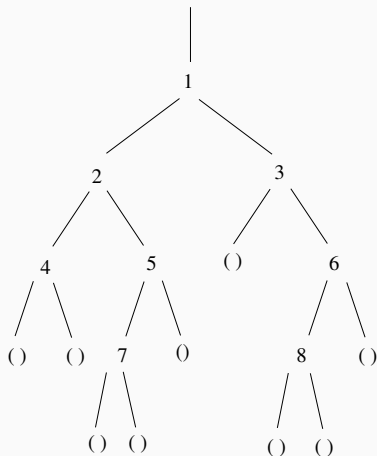
1. en profondeur
2. en largeur

## Parcours en profondeur

Le parcours en profondeur consiste, pour un arbre non vide:

- à explorer la branche gauche complètement (depuis la racine jusqu'aux feuilles),
- puis à explorer la branche droite.
- Le traitement de l'élément à la racine de l'arbre donne lieu à trois possibilités selon que:
  - on traite la racine avant ses fils. Parcours **préfixe**.
  - on traite la racine après ses fils. Parcours **postfixe**.
  - on traite la racine entre son fils gauche et son fils droit. Parcours **infixe**.

## Parcours en profondeur - Exemple



- Préfixe:

[1; 2; 4; 5; 7; 3; 6; 8]

- Postfixe:

[4; 7; 5; 2; 8; 6; 3; 1]

- Infixe:

[4; 2; 7; 5; 1; 3; 8; 6]

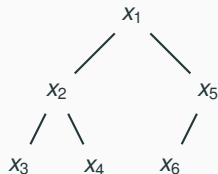
## Parcours en profondeur - Implantation

- ```
let parcours_profondeur_prefixe a =  
  tree_fold (fun racine ppp_g ppp_d -> racine::(ppp_g@ppp_d)) [] a  
  
let parcours_profondeur_postfixe a =  
  tree_fold (fun racine ppp_g ppp_d -> ppp_g@(ppp_d@[racine])) [] a  
  
let parcours_profondeur_infixe a =  
  tree_fold (fun racine ppi_g ppi_d -> ppi_g@(racine::ppi_d)) [] a
```
- Les appels récurifs utilisent implicitement la pile d'appels.
- Il est également possible d'utiliser une pile utilisateur plus simple pour réaliser un parcours en profondeur "à la main".



## Parcours en profondeur - Pile explicite

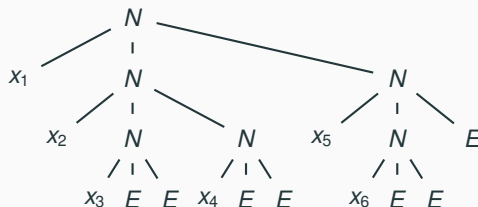
- Arbre :



- Linéarisation :  $[x_1; x_2; x_3; x_4; x_5; x_6]$

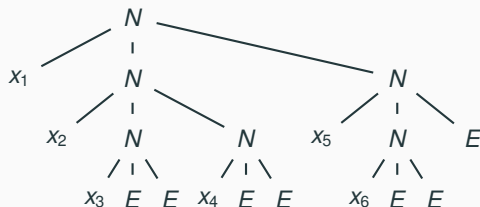
## Parcours en profondeur - Pile explicite

- Autre représentation de l'arbre :



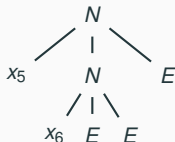
## Parcours en profondeur - Pile explicite

- Initialement dans la pile (un unique élément):



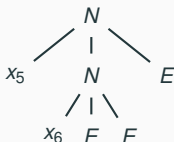
## Parcours en profondeur - Pile explicite

- On dépile le sommet de pile, on garde la racine du nœud et on empile les deux fils.
- Résultat :  $x_1::$ (appel récursif avec dans la pile )



## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, on garde la racine du nœud et on emplit les deux fils.
- Résultat :  $x_1 :: (x_2 :: (\text{appel récursif avec dans la pile}))$

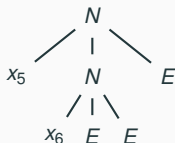


## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, on garde la racine du nœud et on empile les deux fils.
- Résultat :  $x_1 :: (x_2 :: (x_3 :: (\text{appel récursif avec dans la pile})))$ 
  - $E$
  - $E$
  -



•



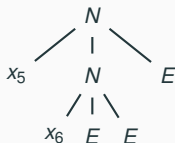
## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, il contient *Empty*, continue avec le reste de la pile.
- Résultat :  $x_1 :: (x_2 :: (x_3 :: (\text{appel récursif avec dans la pile } )))$

- $E$
- 

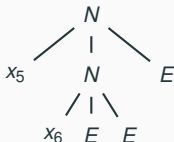


- 



## Parcours en profondeur - Pile explicite

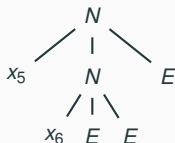
- Pour traiter l'appel récursif, on dépile le sommet de pile, il contient *Empty*, continue avec le reste de la pile.
- Résultat :  $x_1 :: (x_2 :: (x_3 :: (\text{appel récursif avec dans la pile } )))$





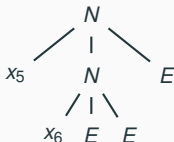
## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, on garde la racine du nœud et on empile les deux fils.
- Résultat :  $x_1::(x_2::(x_3::(x_4::(\text{appel récursif avec dans la pile } )))$ 
  - $E$
  - $E$
  -



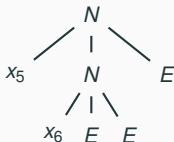
## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, il contient *Empty*, continue avec le reste de la pile.
- Résultat :  $x_1 :: (x_2 :: (x_3 :: (x_4 :: (\text{appel récursif avec dans la pile } )))$ 
  - *E*
  -



## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, il contient *Empty*, continue avec le reste de la pile.
- Résultat :  $x_1 :: (x_2 :: (x_3 :: (x_4 :: (\text{appel récursif avec dans la pile }))))$



## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, on garde la racine du nœud et on empile les deux fils.
- Résultat :  $x_1::(x_2::(x_3::(x_4::(x_5::(\text{appel récursif avec dans la pile }))))$

•



•  $E$

## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, on garde la racine du nœud et on emplit les deux fils.
- Résultat :  $x_1 :: (x_2 :: (x_3 :: (x_4 :: (x_5 :: (x_6 :: (\text{appel récursif avec dans la pile }))))))$ 
  - $E$
  - $E$
  - $E$

## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, il contient *Empty*, continue avec le reste de la pile.
- Résultat :  $x_1 :: (x_2 :: (x_3 :: (x_4 :: (x_5 :: (x_6 :: (\text{appel récursif avec dans la pile } ))))))$ 
  - *E*
  - *E*

## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, il contient *Empty*, continue avec le reste de la pile.
- Résultat :  $x_1 :: (x_2 :: (x_3 :: (x_4 :: (x_5 :: (x_6 :: (\text{appel récursif avec dans la pile }))))))$ 
  - *E*

## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, il contient *Empty*, la pile est vide, le calcul est fini.
- Résultat :  $x_1 :: (x_2 :: (x_3 :: (x_4 :: (x_5 :: (x_6 :: (\text{appel récursif avec pile vide}))))))$



## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, il contient *Empty*, la pile est vide, le calcul est fini.
- Résultat :  $x_1 :: (x_2 :: (x_3 :: (x_4 :: (x_5 :: (x_6 :: (\text{appel récursif avec pile vide}))))))$
- $x_1 :: (x_2 :: (x_3 :: (x_4 :: (x_5 :: (x_6 :: ( [] ))))))$

## Parcours en profondeur - Pile explicite

- Pour traiter l'appel récursif, on dépile le sommet de pile, il contient *Empty*, la pile est vide, le calcul est fini.
- Résultat :  $x_1 :: (x_2 :: (x_3 :: (x_4 :: (x_5 :: (x_6 :: (\text{appel récursif avec pile vide}))))))$
- $x_1 :: (x_2 :: (x_3 :: (x_4 :: (x_5 :: (x_6 :: ( [] ))))))$
- $[x_1; x_2; x_3; x_4; x_5; x_6]$

## Parcours en profondeur - Pile explicite

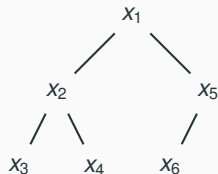
```
let parcours_prefixe arb =  
let rec parcours pile =  
  match pile with  
  | [] -> []  
  | Empty::q -> parcours q  
  | Node (n, g, d)::q -> n::parcours (g::d::q)  
in parcours [arb]
```

## Parcours en largeur

- Le parcours en largeur consiste à parcourir les nœuds de l'arbre, "ligne" par "ligne".
- Cela ne correspond pas du tout à la récursivité naturelle.
- Utilisation d'une file au lieu d'une pile.

## Parcours en largeur - File explicite

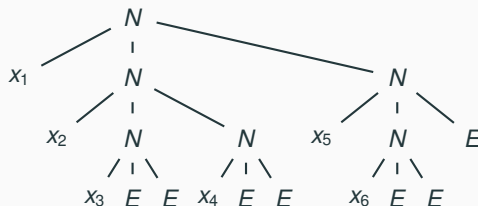
- Arbre :



- Linéarisation :  $[x_1; x_2; x_5; x_3; x_4; x_6]$

## Parcours en largeur - File explicite

- Initialement dans la file :



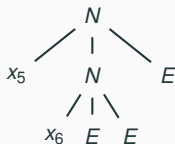
## Parcours en largeur - File explicite

- On défile le sommet de file, on garde la racine du nœud et on enfile les deux fils.
- Résultat :  $x_1 :: (\text{appel récursif avec dans la file})$



## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, on garde la racine du nœud et on enfile les deux fils.
- Résultat :  $x_1 :: (x_2 :: (\text{appel récursif avec dans la file}))$





## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, on garde la racine du nœud et on enfile les deux fils.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (\text{appel récursif avec dans la file } )))$



- $E$

## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, on garde la racine du nœud et on enfile les deux fils.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (\text{appel récursif avec dans la file } )))$ )

•



•



- $E$
- $E$
- $E$

## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, on garde la racine du nœud et on enfile les deux fils.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (\text{appel récursif avec dans la file }))))$

•



- $E$
- $E$
- $E$
- $E$
- $E$

## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, on garde la racine du nœud et on enfile les deux fils.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: (\text{appel récursif avec dans la file }))))))$ 
  - $E$
  - $E$
  - $E$
  - $E$
  - $E$
  - $E$
  - $E$

## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, il contient *Empty*, on continue avec le reste de la file.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: (\text{appel récursif avec dans la file }))))))$ 
  - $E$
  - $E$
  - $E$
  - $E$
  - $E$
  - $E$

## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, il contient *Empty*, on continue avec le reste de la file.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: (\text{appel récursif avec dans la file }))))))$ 
  - *E*
  - *E*
  - *E*
  - *E*
  - *E*

## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, il contient *Empty*, on continue avec le reste de la file.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: (\text{appel récursif avec dans la file }))))))$ 
  - $E$
  - $E$
  - $E$
  - $E$

## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, il contient *Empty*, on continue avec le reste de la file.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: (\text{appel récursif avec dans la file }))))))$ 
  - $E$
  - $E$
  - $E$



## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, il contient *Empty*, on continue avec le reste de la file.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: (\text{appel récursif avec dans la file }))))))$ 
  - $E$
  - $E$

## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, il contient *Empty*, on continue avec le reste de la file.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: (\text{appel récursif avec dans la file }))))))$ 
  - $E$

## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, il contient *Empty*, la file est vide, le calcul est fini.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: (\text{appel récursif avec file vide }))))))$

## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, il contient *Empty*, la file est vide, le calcul est fini.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: (\text{appel récursif avec file vide }))))))$
- $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: ( [] ))))))$

## Parcours en largeur - File explicite

- Pour traiter l'appel récursif, on défile le sommet de file, il contient *Empty*, la file est vide, le calcul est fini.
- Résultat :  $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: (\text{appel récursif avec file vide }))))))$
- $x_1 :: (x_2 :: (x_5 :: (x_3 :: (x_4 :: (x_6 :: ( [] ))))))$
- $[x_1; x_2; x_5; x_3; x_4; x_6]$

## Parcours en largeur - File explicite

- ```
let parcours_largeur arb =  
let rec parcours file =  
  match file with  
  | []                -> []  
  | Empty::q          -> parcours q  
  | Node (n, g, d)::q -> n::parcours (q@[g; d])  
in parcours [arb]
```
- On utilise ici une version naïve et inefficace de la file par concaténation à droite.

## Arbre n-aire

---

**Exercice** : Écrire le fichier `Arbre_naire.ml` correspondant à l'interface suivante.

## `Arbre_naire.mli`

*(\* un arbre dont le nombre de fils d'un noeud est quelconque : arbre n-aire \*)*

**type** 'a arbre\_naire

*(\* cons prend en paramètre un élément e et une liste d'arbres n-aires l \*)*

*(\* et construit l'arbre n-aire qui a pour racine e et pour fils les éléments de l \*)*

**val** cons : 'a -> 'a arbre\_naire list -> 'a arbre\_naire

*(\* racine renvoie la racine d'un arbre n-aire \*)*

**val** racine : 'a arbre\_naire -> 'a

*(\* fils renvoie la liste des fils d'un arbre n-aire \*)*

**val** fils : 'a arbre\_naire -> 'a arbre\_naire list

*(\* map prend en paramètre une fonction f et un arbre n-aire et*

*renvoie l'arbre n-aire où f a été appliquée à toutes les données dans les noeuds \*)*

**val** map : ('a -> 'b) -> 'a arbre\_naire -> 'b arbre\_naire

*(\* itérateur fold sur les arbres n-aires \*)*

**val** fold : ('a -> 'b list -> 'b) -> 'a arbre\_naire -> 'b



## Implantation Arbre\_naire.ml

```
type 'a arbre_naire =  
  | Noeud of 'a * 'a arbre_naire list  
  
let cons racine fils = Noeud (racine, fils )  
  
let racine (Noeud (racine, _)) = racine  
  
let fils (Noeud (_, fils )) = fils  
  
(* principe : une fonction / constante pour chaque constructeur: Noeud *)  
let rec fold fNoeud (Noeud (racine, fils )) =  
  fNoeud racine (List.map (fold fNoeud) fils )  
  
let rec map f (Noeud (r, fils )) =  
  Noeud (f r, List.map (map f) fils )  
  
(* ou bien *)  
let map f arb =  
  fold (fun r liste_map_fils -> Noeud (f r, liste_map_fils )) arb
```

## Utilisation

- Écrire la fonction `cardinal`, pour les arbres n-aires, à l'aide d'un itérateur
- Code:

## Utilisation

- Écrire la fonction `cardinal`, pour les arbres n-aires, à l'aide d'un itérateur
- Code:

```
(* cardinal : 'a arbre_naire -> int *)  
(* Renvoie le nombre d'elements d'un arbre *)  
let cardinal arb =  
  fold (fun _ liste_cardinal_fils -> 1 + List.fold_right (+) liste_cardinal_fils 0) arb
```

## **Problème : écriture d'un évaluateur d'expression**

---

# Problème : écriture d'un évaluateur d'expression

## Description du problème

- La structure d'arbre générique vu précédemment peut ne pas correspondre au besoin.
- Un type *ad hoc* peut être utilisé pour représenter une structure arborescente avec plus de précision.
- Illustration : évaluateur d'une sous-partie des expressions OCAML.

## Grammaire des expressions

1.  $E \rightarrow \text{let } id = E \text{ in } E$
2.  $E \rightarrow (E)$
3.  $E \rightarrow E + E$
4.  $E \rightarrow E - E$
5.  $E \rightarrow -E$
6.  $E \rightarrow id$
7.  $E \rightarrow \text{entier}$

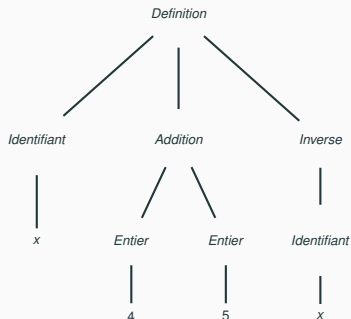
# Problème : écriture d'un évaluateur d'expression

## Représentation arborescente des expressions

- Les expressions se représentent naturellement sous forme d'arbre.
- Par exemple, l'expression :

let  $x = 4+5$  in  $-x$

- peut être représentée par l'arbre :



# Problème : écriture d'un évaluateur d'expression

## Type OCAML des expressions

```
type expression =  
  | Definition of string * expression * expression  
  | Addition of expression * expression  
  | Soustraction of expression * expression  
  | Inverse of expression  
  | Identifiant of string  
  | Entier of int
```

## Evaluateur sans les définitions

```
(* evaluate : expression -> int *)  
let rec evaluate exp =  
  match exp with  
  | Addition (e1,e2) -> (evaluate e1)+(evaluate e2)  
  | Soustraction (e1,e2) -> (evaluate e1)-(evaluate e2)  
  | Inverse e -> -(evaluate e)  
  | Entier i -> i
```

# Problème : écriture d'un évaluateur d'expression

## Evaluateur avec les définitions

- Besoin de gérer un environnement.
- Le type de la fonction `evaluate` ne doit pas être modifié.
- Besoin d'une fonction auxiliaire.

```
(* evaluate : expression -> int *)  
let evaluate exp=  
  let rec aux exp env =  
    match exp with  
    | Definition (i,def,e) -> aux e ((i,aux def env)::env)  
    | Addition (e1,e2) -> (aux e1 env)+(aux e2 env)  
    | Soustraction (e1,e2) -> (aux e1 env)-(aux e2 env)  
    | Inverse e -> -(aux e env)  
    | Identifiant id -> List.assoc id env  
    | Entier i -> i  
  in aux exp []  
  
# evaluate (Definition ("x",Addition ((Entier 4),(Entier 5)) , Inverse (Identifiant "x" )))  
- : int = -9
```