# Structures de données

P. Skler

## Table des matières

**I Structures abstraites** **2**

**II Définition de structure de données en OCaml** **3**

**III Listes** **4**

**IV Arbres binaires** **5**
1) Généralités . . . 5
2) Algorithme sur les arbres binaires . . . 6
a) Nombres de feuilles et de noeuds . . . 6
b) Hauteur d'un arbre . . . 7
3) Parcours d'un arbre binaire . . . 7

**V Piles** **9**
1) Généralités . . . 9
2) Implémentations en CAML : . . . 9
3) Application : évaluation d'une expression arithmétique postfixée . . . 10
a) Représentations d'une expression mathématique . . . 10
b) Syntaxe des expressions arithmétiques postfixées . . . 10

**VI Files** **14**
1) Généralités . . . 14
2) Parcours hiérarchique d'un arbre binaire . . . 16

# I Structures abstraites

**Définition I - 1 : Type de données**
Un type de données est une collection d'objets qui ont des propriétés identiques indépendamment de toute représentation.

**Définition I - 2 : Structure de données**
On appelle type de données abstrait ou structure de donnes abstraite un type de données muni d'une signature, c'est à dire de fonctions de manipulation typées) accompagnée de sa sémantique.

On distingue deux types de structures de données :
- structures **immuables** (ou persistantes) : on ne peut pas la modifier sans recréer complètement la structure
- structures **impératives** (ou modifiables ou encore mutable) : on peut modifier localement sans avoir à tout reconstruire.

La structure de liste est une structure persistante alors que les tableaux sont modifiables. Les références sont modifiables.

**Remarque :** Signature générique Les fonctions de manipulation des types de données sont de plusieurs classes : les constructeurs, les accesseurs, les transformateurs, les prédicats,
- Les fonctions de construction (ou constructeurs) : qui permettent de créer la donnée
- Les fonctions de sélection ou accesseurs : qui permettent d'avoir accès aux informations contenues dans la donnée
- Les transformateurs : qui permettent de modifier l'état de la structure
- Les prédicats : fonctions booléennes qui permettent de tester la nature de la donnée.

Ce contexte définit la signature d'un type ou d'une structure de données. Cette signature serait sans grand intérêt si on ne connaissait pas le rôle précis des fonctions de manipulation : c'est l'objet de la sémantique.

# II Définition de structure de données en OCaml

OCaml est un langage orienté objet : il permet au programmeur de définir de nouveaux types, ce qui lui permet de personnaliser ses structures de données.

Un première façon de faire est de définir un type **enregistrement**, ou produit. Un objet est alors une concaténation de différents **champs**, chaque champ ayant un nom et un type définis lors de la définition du type enregistrement. On peut ensuite lire les champs de l'objet. Si on veut qu'un champ soit modifiable après création, il faut le préciser à l'aide du mot-clé `mutable`.

```
type complexe = {re : float ; im : float};;
let i = {re = 0. ; im = 1. };;
let partiereellle z = z.re;;
let add z zb = {re = z.re+.zb.re; im= z.im+.zb.im};;
add i i;;
```

Beaucoup plus spécifique à OCaml, on peut définir un type **somme**. Un tel type est défini par une énumération de cas. Chaque cas comporte un nom, appelé **constructeur**, qui doit commencer par une majuscule :

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

Les constructeurs peuvent être associés à une ou plusieurs composantes, dont les types sont précisés :

```
type date = D of jour * int;;
let aujourdhui = D(Samedi,30);;
```

La ou les composantes d'un constructeur peuvent même être du type qu'on est en train de définir ! On parle alors de **type récursif** (pour qu'un tel type soit utilisable, il faut définir un ou plusieurs cas d'initialisation dans la définition) :

```
type couleur = Cyan | Magenta | Jaune | Melange of couleur * couleur ;;
let rouge = Melange (Magenta, Jaune);;
let orange = Melange (rouge , Jaune);;
```

Un objet d'un type somme est en général lu par filtrage.

# III Listes

Les listes sont déjà définies en OCaml, mais il est quand même intéressant de leur faire subir la même analyse que les autres structures de données de chapitre. Autrement dit :

- Comment définir le type de données liste ?
- Quelles sont les primitives associées à ce type ?
- Comment implémenter une structure de données associées ?

**Définition III - 1 : Définition récursive du type liste**

Soit $E$ un ensemble non vide. Une **liste d'éléments de** $E$ est :

- La liste vide, de longueur 0 ;
- Ou bien une liste de longueur $n \geq 1$, comme couple de la forme $\ell = (t, q)$, tel que $t \in E$, et $\ell$ est une liste d'éléments de $E$ de longueur $n-1$. $t$ et $q$ sont appelés respectivement **tête** et **queue** de $\ell$.

**Définition III - 2 : Primitives du type liste**

- Création de la liste vide.
- Création d'une nouvelle liste à partir d'un élément et d'une liste plus petite.
- Test pour savoir si la liste est vide.
- Extraction de la tête d'une liste non vide.
- Extraction de la queue d'une liste non vide.

```
type 'a liste = Listevide | Cons of 'a * ('a liste);;

let lst = Cons(5,Cons(5,Cons(4,Listevide)));;(*Exemple*)

let listevide () = Listevide;;

let conse t q = Cons(t,q);;

let estvide lst = match lst with
  | Listevide -> true
  | _ -> false;;

let tete lst = match lst with
  | Listevide -> failwith "liste vide"
  | Cons(t,q) -> t;;

let queue lst = match lst with
  | Listevide -> failwith "liste vide"
  | Cons(t,q) -> q;;
```

# IV Arbres binaires

## 1) Généralités

Un arbre est un graphe acyclique orienté (c'est à dire une collection de sommets et d'arêtes orientées ne formant pas de cycles), possédant une unique **racine**, et dans lequel tous les sommets, à l'exception de la racine, ont un unique parent.

Si une arête mène du sommet i au sommet j, on dit que i est le **père** de j, et en conséquence que j est le **fils** de i.

Il est d'usage de dessiner un arbre en plaçant un père au dessus de ses fils, si bien que l'on peut sans ambigüité représenter les arêtes par des traits simples.

Un sommet qui n'a pas de fils est appelé une feuille (ou noeud externe), les autres sommets sont appelés les **noeuds (ou noeuds internes)**.

Enfin, on notera que chaque noeud est la racine d'un arbre constitué de lui-même et de l'ensemble de ses descendants ; on parle alors de sous-arbre de l'arbre initial.

**Autre définition :**

Un arbre non vide est
- un ensemble fini d'objets : les noeuds
- liés par une relation « n est le fils de m » ou « m est le père de n » telle que
  - il existe un unique élément sans père : la racine
  - tout élément, sauf la racine, a un père unique
  - tout élément est descendant de la racine
- un noeud sans fils est une feuille de l'arbre ou noeud externe
- les autres sont les noeuds internes

**Sous arbre :**

L'ensemble des noeuds qui sont les descendants d'un noeud $n_0$ forme un arbre dont $n_0$ est la racine : c'est le sous-arbre de racine $n_0$.

Un arbre peut donc être défini récursivement comme
- soit l'arbre vide
- soit un ensemble d'arbres fils d'une racine.

L'ensemble des fils n'est pas, dans ce cas général, ordonné.

**Structure de données**

En réalité, il n'existe pas une structure de données associée aux arbres, mais des structures de données, que l'on définit en fonction de l'organisation que l'on souhaite. En outre, l'information peut être stockée dans les noeuds, dans les feuilles, voire dans les deux à la fois. Par la suite, on appellera étiquette l'information attachée à un noeud ou à une feuille.

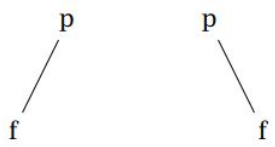
On appelle **arité** d'un noeud, le nombre de branches qui en partent.

On appelle arbre binaire un arbre dont chaque noeud a une arité d'au plus 2.

On donne alors la notion de **fils-gauche** et **fils-droit**.

**Différence :** Un arbre avec deux noeuds est défini de manière unique en tant qu'arbre général


mais admet deux structures d'arbre binaire :




On a alors la possibilité de l'implémenter de la façon suivante :

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

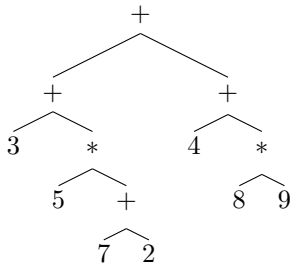
On parle d'arbre binaire stricte un arbre dont chaque noeud a une arité égale à 2.
Ce qui amène à la définition suivante : `Arbre = feuille + Arbre×noeud×Arbre`
On a alors l'implémentation suivante lorsqu'on veut différentier les feuilles et les noeuds :

```
type ('a,'b) arbre = Feuille of 'a | Noeud of 'b * ('a,'b) arbre * ('a,'b) arbre;;
```

L'expression

$$E = (3 + (5 \times (7 + 2))) + (4 + (8 \times 9)) .$$

serait représentée par l'arbre suivant :




On a alors avec la première implémentation

```
let exp= Noeud (‘+‘ , Noeud ( ‘*‘ , Noeud( ‘+‘, Feuille 2, Feuille 3), Feuille 5),
 Noeud(‘*‘,Feuille 7, Feuille 2));;
```

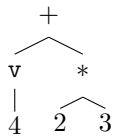
Lorsqu'on veut une définition plus générale ne distinguant pas les noeuds et les feuilles, un *arbre binaire étiqueté* : l'ensemble $\mathcal{A}$ des arbres binaires étiquetés par des éléments d'un ensemble non vide $E$ est défini inductivement par :

$$\mathcal{A} = \{\emptyset\} \cup \mathcal{A} \times E \times \mathcal{A}$$

(un arbre est soit vide, soit un triplet constitué d'un arbre (le *sous-arbre gauche* ou *fils gauche*), d'un élément de $E$ (l'*étiquette*) et d'un second arbre (le *sous-arbre droit* ou *fils droit*).
Comme on ne représente pas les sous-arbres vides (chaque feuille devrait être complétée par deux sous-arbres vides), on peut aussi utiliser l'implémentation suivantes :

```
type 'a arbre2 = Nil | N of 'a * 'a arbre2 * 'a arbre2;;
```

Avec l'exemple suivant : $\sqrt{4} + 2 * 3$.




Les noeuds sont des caractères, `v` représente la racine.

```
let exp2 = N( ‘+‘, N (‘v‘, N (‘4‘,Nil,Nil),Nil),
 N (‘*‘, N (‘2‘,Nil,Nil), N (‘3‘,Nil,Nil)));;
```

## 2) Algorithme sur les arbres binaires

### a) Nombres de feuilles et de noeuds

**Calcul du nombre de feuilles**

a) Pour un arbre binaire stricte

```
let rec nb_f arbre = match arbre with
        | Feuille _ -> 1
        | Noeud (_, fils_g, fils_d) -> (nb_f fils_g)+(nb_f fils_d);;
```

b) Pour un arbre binaire plus général

```
let rec nb_f2 arbre = match arbre with
        | Nil -> 0
        | N(_ ,Nil,Nil) ->1
        | N(_ ,fils_g,fils_d) -> (nb_f2 fils_g)+(nb_f2 fils_d);;
```

**Calcul du nombre de noeuds**

a) Pour un arbre binaire stricte

```
let rec nb_n arbre = match arbre with
        | Feuille _ -> 0
        | Noeud (_, fils_g, fils_d) -> 1+(nb_n fils_g)+(nb_n fils_d);;
```

b) Pour un arbre binaire plus général

```
let rec nb_n2 arbre = match arbre with
          | Nil -> 0
          | N(_ ,Nil,Nil) ->0
          | N(_ ,fils_g,fils_d) -> 1+(nb_n2 fils_g)+(nb_n2 fils_d);;
```

**Théorème IV - 1 :**
Si un arbre binaire strict possède $n$ noeuds et $f$ feuilles alors $f = n + 1$

**Preuve :** Si cet arbre est constitué d'une unique feuille, alors $n = 0$ et $f = 1$ et le résultat est bien vérifié. Dans le cas contraire, cet arbre est constitué d'une racine ayant deux fils. Notons $n_1$ et $f_1$ (respectivement $n_2$ et $f_2$) le nombre de noeuds et de feuilles du premier fils (respectivement du second), et supposons $f_1 = n_1 + 1$ et $f_2 = n_2 + 1$. Alors $n = 1 + n_1 + n_2$ et $f = f_1 + f_2$ donc $f = (n_1 + 1) + (_+1) = n_1 + n_2 + 2 = n + 1$. ■

### b) Hauteur d'un arbre

**Définition :** la profondeur d'un noeud ou d'une feuille est la distance qui sépare ce noeud ou cette feuille de la racine, et la hauteur de l'arbre est la profondeur maximale d'une feuille.

**Autre formulation :** La profondeur d'un noeud est le nombre d'ascendants stricts de ce noeud, la hauteur d'un arbre est la profondeur maximale de ses noeuds. La hauteur de l'arbre vide est $-1$.

a) Pour un arbre binaire stricte

```
let rec hauteur arbre = match arbre with
          | Feuille _ -> 0
          | Noeud (_, fils_g, fils_d)-> 1 + max (hauteur fils_g) (hauteur fils_d);;
```

b) Pour un arbre binaire plus général

```
let rec hauteur2 arbre = match arbre with
          | Nil ->-1
          | N (_, Nil, Nil) -> 0
          | N (_, fils_g, fils_d) -> 1+ max (hauteur2 fils_g) (hauteur2 fils_d);;
```

**Théorème IV - 2 :**
Si h désigne la hauteur d'un arbre binaire (strict ou non), le nombre de feuilles est majoré par $2^h$.

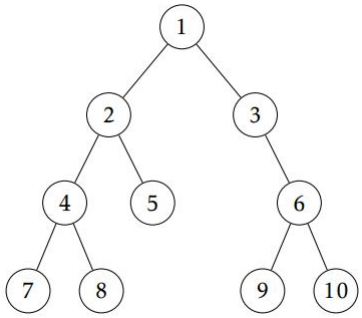
**Preuve :** Raisonnons par récurrence sur la hauteur de l'arbre.

Un arbre de hauteur nulle est réduit à une feuille (ou à l'arbre vide). Un arbre de hauteur $h \geqslant 1$ possède une racine avec deux fils. Chacun de ces deux fils est un arbre de hauteur inférieure ou égale à $h - 1$, donc par hypothèse de récurrence chacun possède au plus $2^{h-1}$ feuilles. L'arbre initial possède donc au plus $2 \times 2^{h-1} = 2^h$ feuilles. ■

**Corollaire IV - 1 :**
Un arbre binaire possédant $n$ feuilles a pour hauteur minimale $\lceil \log_2(n) \rceil$.

## 3) Parcours d'un arbre binaire

Pour la suite on utilise l'arbre suivant :



**Parcours en profondeur**

On utilise essentiellement trois types de parcours récursif pour *"visiter"* tous les nœuds d'un arbre binaire.

Le parcours **préfixe** consiste à parcourir l'arbre suivant l'ordre : r ! fils g. ! fils d. Dans l'exemple qui suit, il correspond au parcours de l'arbre suivant l'ordre 1 - 2 - 4 - 7 - 8 - 5 - 3 - 6 - 9 - 10 : c'est à dire suivant le premier passage à gauche d'un noeud lorsqu'on parcours le trajet représenté.

Le parcours **suffixe** consiste à parcourir l'arbre suivant l'ordre : fils g. ! fils d. ! r. Dans l'exemple précédent, il correspond au parcours 7 - 8 - 4 - 5 - 2 - 9 - 10 - 6 - 3 - 1, c'est à dire suivant le premier passage à droite d'un noeud.

Le parcours **infixe** (ou parcours symétrique) consiste à parcourir l'arbre suivant l'ordre : fils g. ! r ! fils d. Dans l'exemple précédent, il correspond au parcours 7 - 4 - 8 - 2 - 5 - 1 - 3 - 9 - 6 - 10, c'est à dire suivant le premier passage sous un noeud.

# V Piles

## 1) Généralités

Les piles sont fondées sur le principe du « dernier arrivé, premier sorti ». Une pile est ainsi, comme une liste, une structure LIFO (Last In, First Out), mais **sans parcours possible** (à moins de détruire la pile !) : on ne peut accéder qu'à l'élément situé au sommet.

**Définition V - 1 : Le type pile (stack en anglais)**

$E$ étant un ensemble non vide, l'ensemble $\mathcal{P}$ des *piles d'éléments de* $E$ est par définition égal à l'ensemble des listes d'éléments de $E$ ; cela est le point de vue mathématique. Le vocabulaire est un peu différent : *le sommet* d'une pile est la tête de la liste correspondante.
Du point de vue informatique, la *spécification* du type `Pile` diffère notablement de celle du type `Liste`.
Les *primitives* disponibles pour le type `Pile` sont les suivantes :
- créer la pile vide
- tester si elle est vide
- ajouter un élément : empiler
- retirer un élément et le renvoyer : dépiler

**Exemples d'utilisation :**
- gestion de l'historique des pages visitées dans un navigateur internet
- actions dans un traitement de texte
- exécution d'une fonction récursive : chaque appel récursif est ajouté dans une pile et lorsqu'on arrive à un cas de base ou cas terminal (plus de récursivité) on dépile pour passer aux appels antérieurs
- évaluation d'une expression arithmétique postfixée

## 2) Implémentations en CAML :

Même si elle ressemble à une liste, la structure de donnée est modifiable et donc il y a des effets de bords lors de la manipulation.

**Première méthode :** avec un enregistrement de liste qui permet de modifier la pile par effet de bord.

```
type 'a pile = {mutable contenu : 'a list};;
let pile_vide () = { contenu = [] };;
let empile x p = p.contenu <- x :: p.contenu;;
let depile p = match p.contenu with
       | [] -> failwith "pile vide"
       | x::q -> p.contenu <- q; x;;
let est_vide p = match p.contenu with
       | [] -> true
       | _ -> false;;
```

**Deuxième méthode :** On utilise des références de listes

```
type 'a pile = 'a list ref;;
let newpile() = (ref []:'a pile);;
let estvide (s : 'a pile) =!s = [];;
let push (s : 'a pile) x = s:= x::!s;;
let pop (s : 'a pile) = if estvide s then failwith "pile vide"
          else begin let x = List.hd !s in
           s: = List.tl !s;
           x;
          end;;
```

**Troisième méthode :** On utilise un tableau et un compteur indiquant le sommet de la pile. L'inconvénient est que la taille de la pile est limitée.

```
type 'a pile = { contenu : 'a array ; mutable sommet : int};;
```

Inconvénient : spécifier la taille maximale de la pile à créer.

```
let newpile ()={ contenu = Array.make 10 0; sommet = -1};;
let push p x = p.sommet<-p.sommet+1;p.contenu.(p.sommet)<- x;;
let pop p = p.sommet<-p.sommet -1;p.contenu(p.sommet +1);;
let estvide p = p.sommet = -1;;
```

**Dernière méthode :** on peut utiliser le module de Caml (librairie) `Stack`.

Les piles d'éléments de type `'a` ont pour type `'a t`.
La fonction `Stack.create`, de type `unit -> 'a t`, crée une nouvelle pile, vide au départ.
La fonction `Stack.push`, de type `'a -> 'a t -> unit`, permet d'empiler un élément au sommet d'une pile.
La fonction `Stack.pop`, de type `'a t -> 'a`, élimine le sommet de la pile et le renvoie.
L'exception `Empty` est déclenchée lorsqu'on tente d'appliquer `Stack.pop` à une pile vide.
La fonction `Stack.is_empty`, de type `'a t -> bool` renvoie `true` si la pile est vide `false` sinon.

## 3) Application : évaluation d'une expression arithmétique postfixée

### a) Représentations d'une expression mathématique

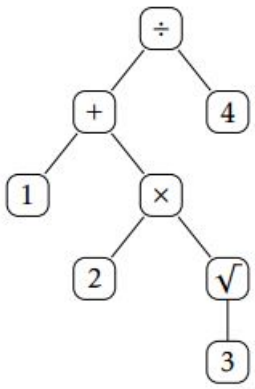
Suivant l'usage courant, une expression mathématique est représentée sous sa forme infixe : les opérateurs binaires se placent entre leurs deux arguments. Ceci nécessite l'usage de parenthèses pour préciser la hiérarchie des calculs : $(1+2)\times 3$ est différent de $1+(2\times 3)$.

Par exemple, la formule $\dfrac{1+2\sqrt{3}}{4}$ est représentée par la liste de caractères :
$(1+(2\times(\sqrt{}3)))\div 4$.

Une expression est dite postfixée lorsque les opérateurs suivent immédiatement la liste de leur(s) argument(s). Par exemple, cette même formule sera représentée sous forme postfixée par la liste :
$1\ 2\ 3\ \sqrt{}\ \times\ +\ 4\ \div$. L'usage des parenthèses est ici superflu puisque l'ordre dans lequel les opérations apparaissent est l'ordre dans lequel elles doivent être effectuées.

De même, une expression est dite préfixée lorsque les opérateurs précèdent immédiatement la liste de leurs arguments. Ainsi, la formule donnée en exemple sera représentée sous forme préfixée par : $\div\ +\ 1\ \times\ 2\ \sqrt{}\ 3\ 4$.

On peut noter que l'expression postfixée d'une formule mathématique correspond au parcours en profondeur postfixe de l'arbre associé à cette expression et l'expression préfixée au parcours en profondeur suffixe :




### b) Syntaxe des expressions arithmétiques postfixées

**Définitions - Notations**

On se propose de définir les *expressions arithmétiques postfixées* (syntaxiquement correctes !).

Soient $E$ un ensemble (de "nombres"), $\Omega$ un ensemble d'applications de $E\times E$ dans $E$ (*opérateurs binaires*, dits aussi d'*arité* 2) et $\mathcal{F}$ un ensemble d'applications de $E$ dans $E$ (*opérateurs unaires*).

On note $\mathcal{S}=E\cup\Omega\cup\mathcal{F}$ l'ensemble des *"symboles"* utilisés et $\mathcal{L}$ l'ensemble des listes non vides d'éléments de $\mathcal{S}$. Pour alléger, on écrit les éléments d'une telle liste simplement séparés par des espaces.

Si $A = a_1 \ \dots \ a_p$ et $B = b_1 \ \dots \ b_q$ sont deux éléments de $\mathcal{L}$, on note $A\ B$ le résultat de la *concaténation* des deux listes, à savoir

$$A\ B = a_1 \ \dots \ a_p \ b_1 \ \dots \ b_q \ .$$

L'ensemble $\mathcal{E}$ des expressions arithmétiques postfixées est le sous-ensemble de $\mathcal{L}$ défini récursivement comme l'ensemble des éléments de $\mathcal{L}$ s'écrivant sous l'une des formes suivantes :

- $x$ *(liste de longueur 1)*, où $x \in E$ ;
- $A\ B\ \omega$ , où $(A, B) \in \mathcal{E}^2$ et $\omega \in \Omega$ ;
- $A\ f$ , où $A \in \mathcal{E}$ et $f \in \mathcal{F}$.

Cette définition récursive signifie en fait que $\mathcal{E}$ est la plus petite des parties (c'est-à-dire l'intersection des parties) $\mathcal{P}$ de $\mathcal{L}$ vérifiant les propriétés suivantes :

- $\forall x \in E \quad x \in \mathcal{P}$ ;
- $\forall (A, B) \in \mathcal{P}^2 \quad \forall \omega \in \Omega \quad A\ B\ \omega \in \mathcal{P}$ ;
- $\forall A \in \mathcal{P} \quad \forall f \in \mathcal{F} \quad A\ f \in \mathcal{P}$.

*On peut bien sûr généraliser cette construction à des expressions comprenant des opérateurs d'arité supérieure à 2.*

**Définition V - 2 :**

Étant donné $A = a_1 \ \dots \ a_n$ élément de $\mathcal{L}$, on appelle :

- *poids* de $A$ l'entier $p(A) = \sum_{k=1}^{n} p(a_k)$ où l'on a posé, pour $a \in \mathcal{S}$, $p(a) = \begin{cases} 1 & \text{si } a \in E \\ -1 & \text{si } a \in \Omega \\ 0 & \text{si } a \in \mathcal{F} \end{cases}$ .
- *préfixes stricts* de $A$ les listes de la forme $a_1 \ \dots \ a_p$ , $1 \leq p < n$ .

**Caractérisation des expressions arithmétiques postfixées**

**Théorème V - 1 :**

un élément $A$ de $\mathcal{L}$ est dans $\mathcal{E}$ si et seulement si $A$ est de poids 1 et tout préfixe strict de $A$ est de poids supérieur ou égal à 1.
Si $A = a_1 \ \dots \ a_n$ est dans $\mathcal{E}$, avec $n > 1$, alors $A$ s'écrit, de manière unique,

- soit $A = B\ C\ \omega$ , avec $(B, C) \in \mathcal{E}^2$ et $\omega \in \Omega$,
- soit $A = B\ f$ , avec $B \in \mathcal{E}$ et $f \in \mathcal{F}$.

Dans les deux cas, $B$ est le plus long préfixe strict de poids 1 de $A$.

**Preuve :**

1. Condition nécessaire : on montre par récurrence sur $n$ que la propriété $\mathcal{P}_n$ : "si $A$ est un élément de $\mathcal{E}$ de longueur au plus égale à $n$, alors $A$ est de poids 1 et tout préfixe strict de $A$ est de poids supérieur ou égal à 1" est vraie pour tout $n$ de $\mathbb{N}^*$.
   (a) $\mathcal{P}_1$ est claire : un élément de $\mathcal{E}$ de longueur 1 est nécessairement de la forme $x$, $x \in E$.
   (b) soit $n > 1$ tel que $\mathcal{P}_{n-1}$ soit vraie ; il suffit de prouver le résultat pour $A \in \mathcal{E}$, de longueur $n$ :
   - si $A = B\ C\ \omega$ , avec $(B, C) \in \mathcal{E}^2$ et $\omega \in \Omega$, alors : $p(A) = p(B) + p(C) - 1 = 1 + 1 - 1 = 1$ cela grâce à l'hypothèse de récurrence appliquée à $B$ et à $C$.
     De plus, les préfixes stricts de $A$ sont $B$, $B\ C$, et les listes, soit de la forme $B'$, avec $B'$ préfixe strict de $B$, soit de la forme $B\ C'$, avec $C'$ préfixe strict de $C$ : dans ces quatre cas l'hypothèse de récurrence permet de conclure ;
   - si $A = B\ f$ , avec $B \in \mathcal{E}$ et $f \in \mathcal{F}$, alors : $p(A) = p(B) + 0 = 1$ ,
     cela grâce à l'hypothèse de récurrence appliquée à $B$.
     De plus, les préfixes stricts de $A$ sont $B$ et les préfixes stricts de $B$, d'où le résultat grâce à l'hypothèse de récurrence.
2. Soit $A$ élément de $\mathcal{E}$ de longueur strictement supérieure à 1 :
   (a) si $A = B\ f$, où $B \in \mathcal{E}$, $f \in \mathcal{F}$, alors $B$ est clairement le plus long préfixe strict de poids 1 de $A$
   (b) si $A = B\ C\ \omega$ , avec $(B, C) \in \mathcal{E}^2$ et $\omega \in \Omega$, alors $B$ est le plus long préfixe strict de poids 1 de $A$ (en effet $B$ est bien un préfixe strict de poids 1 de $A$ et tout préfixe strict de $A$ plus long que $B$ est de poids supérieur ou égal à 2 d'après la condition nécessaire établie ci-dessus).

   Il en résulte que l'écriture d'un élément $A$ de $\mathcal{E}$ de longueur strictement supérieure à 1 sous l'une des deux formes $B\ f$ ou $B\ C\ \omega$ est unique puisque $B$ est parfaitement défini, dans les deux cas, par la propriété précédente ; $C$ s'en déduit immédiatement dans le second cas (cette constatation correspond à la non-ambiguïté de l'écriture postfixée, contrairement à l'écriture infixée qui nécessite des parenthèses).

3. Condition suffisante : on montre par récurrence sur $n$ que la propriété $\mathcal{P}_n$ : "si $A$ est un élément de $\mathcal{L}$, de longueur au plus égale à $n$, tel que $A$ est de poids 1 et tout préfixe strict de $A$ est de poids supérieur ou égal à 1, alors $A$ appartient à $\mathcal{E}$" est vraie pour tout $n$ de $\mathbb{N}^*$.

   (a) $\mathcal{P}_1$ est vraie : si $A$ est un élément de $\mathcal{L}$ de longueur 1 et de poids 1, nécessairement, d'après la définition de $p$, $A$ est de la forme $x$, $x \in E$ et donc $A \in \mathcal{E}$ ;

   (b) soit $n > 1$ tel que $\mathcal{P}_{n-1}$ soit vraie ; soit $A = a_1 \ \dots \ a_n$ élément de longueur $n$ de $\mathcal{L}$, de poids 1 et dont les préfixes stricts sont de poids supérieur ou égal à 1. En particulier $p(a_1 \ \dots \ a_{n-1}) \geq 1$ et donc $p(a_n) \leq 0$, c'est-à-dire que $a_n$ est un opérateur, d'où les deux cas :
   - si $a_n = f \in \mathcal{F}$, alors $A' = a_1 \ \dots \ a_{n-1}$ est de poids 1 et ses préfixes stricts sont de poids supérieur ou égal à 1 (puisque ce sont des préfixes stricts de $A$ !) ; par conséquent $A' \in \mathcal{E}$ d'après $\mathcal{P}_{n-1}$ et donc $A = A'\ f \in \mathcal{E}$ ;
   - si $a_n = \omega \in \Omega$, alors $A' = a_1 \ \dots \ a_{n-1}$ est de poids 2 ; $a_1$ est nécessairement de poids 1 (supérieur ou égal à 1 car $a_1$ est un préfixe strict de $A$, inférieur ou égal à 1 par définition de $p$) et donc $A$ admet un préfixe strict $B$ de poids 1 et de longueur maximale (le plus grand élément de la partie non vide majorée de $\mathbb{N}$ formée des longueurs des préfixes stricts de $A$ de poids 1) ; $B$ est élément de $\mathcal{E}$ grâce à l'hypothèse de récurrence (ses préfixes stricts sont des préfixes stricts de $A$), en outre $B$ est un préfixe strict de $A'$ puisque $p(A') = 2$ ; $A'$ est donc de la forme $B\ C$, où $C$ est un élément de $\mathcal{L}$ de poids $p(A') - p(B)$, donc de poids 1 ; enfin, si $C'$ est un préfixe strict de $C$, alors $B\ C'$ est un préfixe strict de $A$ strictement plus long que $B$, donc de poids supérieur ou égal à 2 par définition de $B$, d'où
   $$p(C') = p(B\ C') - p(B) \geq 2 - 1 = 1 \ ;$$
   par conséquent l'hypothèse de récurrence s'applique également à $C$, donc $C$ et par suite $A = B\ C\ \omega$ sont des éléments de $\mathcal{E}$.

Cela achève la démonstration du théorème.

■

## Sémantique

### Définition de l'évaluation

On se propose maintenant d'*évaluer* les expressions arithmétiques postfixées (il s'agit de donner un sens à des objets syntaxiquement corrects).

**Théorème V - 2 :**

il existe une unique application $eval$ de $\mathcal{E}$ dans $E$ telle que :
- $eval(A) = x$ si $A = x$ , où $x \in E$ ;
- $eval(A) = \omega\left(eval(B), eval(C)\right)$ si $A = B\ C\ \omega$ , où $(B, C) \in \mathcal{E}^2$ et $\omega \in \Omega$ ;
- $eval(A) = f\left(eval(B)\right)$ si $A = B\ f$ , où $B \in \mathcal{E}$ et $f \in \mathcal{F}$.

**Preuve :**

Le résultat se prouve par récurrence sur la longueur de $A$, en utilisant le fait (vu au paragraphe précédent) que $A$ s'écrit de manière unique sous l'une des trois formes $x$, $B\ C\ \omega$, $B\ f$. ■

### Évaluation pratique à l'aide d'une pile

Étant donné un élément de $\mathcal{L}$, on peut dans un premier temps vérifier sa syntaxe à l'aide de la caractérisation du théorème de caractérisation, puis, s'il s'agit bien d'un élément de $\mathcal{E}$, l'évaluer récursivement suivant le principe ci-dessus.

En pratique, il est plus simple d'évaluer en vérifiant la syntaxe au fur et à mesure de l'introduction des éléments $a_1 \ \dots \ a_n$ ; les résultats intermédiaires sont stockés dans une *pile* (voir ci-dessous) ; la hauteur de cette pile après introduction de $a_1 \ \dots \ a_p$ est le poids de $a_1 \ \dots \ a_p$

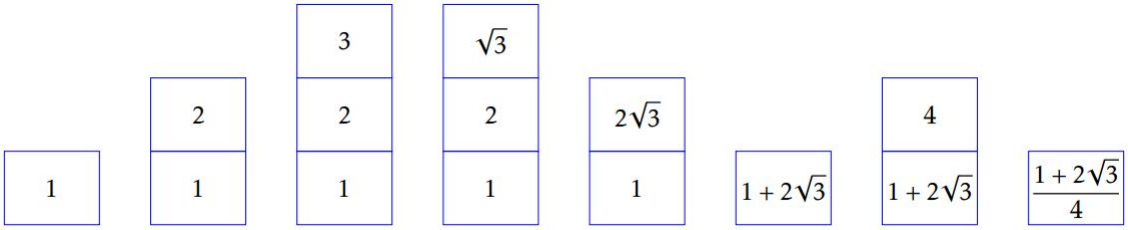
**Mise en oeuvre :**

On commence par définir le type suivant :

```
type lexeme = Nombre of float
            | Op_binaire of float -> float -> float
            | Op_unaire of float -> float ;;
```

L'évaluation d'une expression postfixée se déroule en parcourant la liste des lexèmes et en suivant les règles suivantes :
- si la tête de la liste est un nombre $a$, on l'empile ;
- si la tête est un opérateur unaire $f$ , on dépile le sommet $a$ et on empile $f(a)$ ;
- si la tête est un opérateur binaire $g$ , on dépile les deux éléments $a$ et $b$ les plus hauts, et on empile $g(a, b)$.

Par exemple, la pile associée à l'expression $1\ 2\ 3\ \sqrt{}\ \times\ +\ 4\ \div$ va évoluer comme suit :




La fonction correspondante est la suivante :

```
let evalue lst =
        let pile = Stack.create () in
        let rec aux lst = match lst with
           | [] -> Stack.pop pile
      | (Nombre a)::q -> Stack.push a pile ; aux q
      | (Op_unaire f)::q -> let a = Stack.pop pile in
             Stack.push (f a) pile ; aux q
      | (Op_binaire f)::q -> let b = Stack.pop pile in let a = Stack.pop pile in
             Stack.push (f a b) pile ; aux q
        in aux lst ;;
```

# VI Files

## 1) Généralités

**Définition VI - 1 : Le type file**

La file est une structure de données linéaire avec priorité FIFO (*first in, first out*). C'est à dire que l'élément retiré est le plus ancien dans la file.
Les primitives sont :
- créer une file vide
- tester si une file est vide
- ajouter (en dernier) un élément
- retire (le premier) un élément et le renvoyer

**Implémentations en CAML :** en dehors de la librairie citée en dernier il y a deux méthodes principales différentes :

**Implémentation persistante :** on utilise deux listes

```
type 'a file = {mutable debut : 'a list ; mutable fin : 'a list};;
let file_vide () = {debut = [] ; fin = []};;
let ajoute e f = f.fin <- e::f.fin;;
let est_vide f = (f.debut = [])&& (f.fin = []);;
let premier f =
       if (est_vide f) then failwith "file vide"
          else begin
             if (f.debut =[]) then begin
             f.debut <- List.rev f.fin;
             f.fin <- []
             end;
           let x = List.hd f.debut in
           f.debut <- List.tl f.debut;
           x
           end;;
```

**Implémentation impérative :** on utilise un tableau (tableau circulaire).

D'un point de vue mathématique, cela revient à considérer les indices modulo la longueur du tableau et à laisser tomber la contrainte **debut** $<$ **fin**.

On peut avoir un problème si l'indice de début est identique à celui de fin. Il faut distinguer la file vide avec le liste de taille maximale. On ajoute donc un booléen.

```
type 'a file ={ contenu : 'a array;
 mutable debut : int; mutable fin : int ; mutable vide : bool};;
```

La taille maximale intervient directement dans le code des fonctions de manipulation. On peut en faire une variable globale dès le début.

```
let fmax = 20 ;;
```

Un autre problème lié aux tableaux est le type de données. Il faut faire attention si on veut déclarer un type polymorphe.

```
let file_vide objet =
       { contenu = Array.make fmax objet; debut = 0; fin = 0; vide = true}
```

Pour la suite on va rester dans les entiers :

```
let file_vide ()= {contenu = Array.make fmax 0; debut = 0; fin =0; vide = true}
```

Les fonctions primitives sont alors :

```
let est_vide f = f.vide;;
let ajoute e f =
 if (f.fin = f.debut) && not(f.vide) then failwith "file pleine"
       else begin
          f.contenu.(f.fin) <- e;
          f.fin <- (f.fin +1) mod fmax;
          f.vide <- false;
```

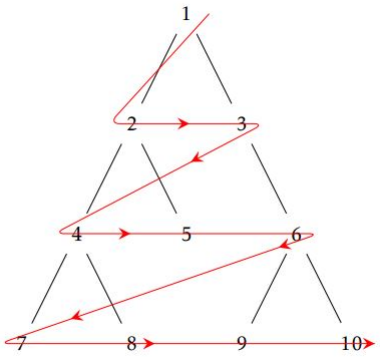
```
        f
        end;;
let premier f =
      if (f.vide) then failwith "file vide"
        else begin
         let x = f.contenu.(f.debut) in
         f.debut <- (f.debut +1) mod fmax;
         if (f.debut = f.fin) then f.vide <- true;
         x
         end;;
```

Remarque : une autre possibilité est de donner le nombre d'éléments de la file, le nombre maximal et l'emplacement de la tête de la file.

**Dernière méthode :** on peut utiliser le module de Caml (librairie) `Queue`.
Les files d'éléments de type `'a` ont pour type `'a t`.
La fonction `Queue.create`, de type `unit -> 'a t`, crée une nouvelle file, vide au départ.
La fonction `Queue.is_empty`, de type `'a t -> bool`, renvoie un booléen indiquant si la file est vide.
La fonction `Queue.add`, de type `'a -> 'a t -> unit`, permet d'ajouter un élément en queue de file.
La fonction `Queue.take`, de type `'a t -> 'a`, élimine la tête de la file et la renvoie.
La fonction `Queue.top`, de type `'a t -> 'a`, donne la tête de la file sans l'enlever de la file.

## 2) Parcours hiérarchique d'un arbre binaire

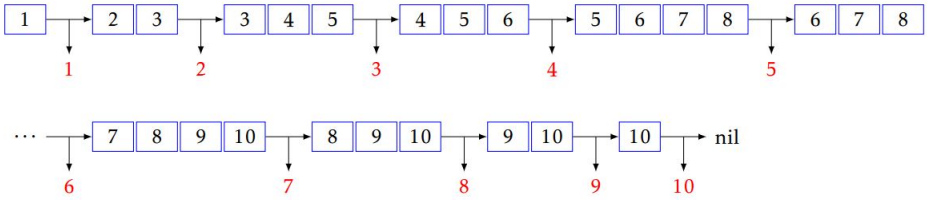
Le parcours hiérarchique, ou parcours en largeur, consiste à parcourir les nœuds et feuilles de l'arbre en les classant par profondeur croissante (et en général de la gauche vers la droite pour une profondeur donnée). Dans l'exemple qui suit, il correspond à l'ordre 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 :




La solution de ce problème consiste à utiliser une file dans laquelle figurera au départ le racine de l'arbre, et à adopter la règle suivante :

1. la tête de la file est traitée ;
2. les fils gauche et droit de cet élément, s'ils sont présents dans l'arbre, sont ajoutés en queue de la file ; et à procéder ainsi jusqu'à exhaustion de la file.

Dans l'exemple ci-dessus, la file va évoluer de la façon suivante :




Pour un arbre :

```
type 'a arbre = Nil | Noeud of 'a * 'a arbre * 'a arbre ;;
```

la fonction de parcours hiérarchique va prendre la forme suivante (dans le cas d'une étiquette de type int) :

```
let parcours arbre =
        let file = Queue.create () in
        Queue.add arbre file ;
        let rec aux () = match ( Queue.take file) with
           | Nil -> aux ()
           | Noeud (r, fils_g, fils_d) -> print_int r ;
                       Queue.add fils_g file ;
                       Queue.add fils_d file ;
                       aux ()
        in try aux () with empty -> () ;;
```

Sans la gestion de l'erreur (`try`) :

```
let parcours arbre =
          let file = Queue.create () in
          Queue.add arbre file ;
          let rec aux () = il not(Queue.is_empty file) then match ( Queue.take file) with
             | Nil -> aux ()
             | Noeud (r, fils_g, fils_d) -> print_int r ;
                          Queue.add fils_g file ;
                          Queue.add fils_d file ;
                          aux ()
          in try aux ()
```