Lycée Louis-le-Grand
Année 2003–2004

Arbres

option informatique

1 Sommaire

- arbres binaires ;
- parcours d'un arbre ;
- arbres de recherche ;
- tas et tri ;
- arbres n-aires ;
- arbres et expressions.

2 Arbres binaires

Un arbre binaire possède des nœuds et des feuilles. Certains disent plutôt nœuds internes et nœuds externes.
On peut définir la structure d'arbre binaire de façon récursive.
Si $\mathcal{N}$ (*resp.* $\mathcal{F}$) désigne l'ensemble des valeurs des nœuds (*resp.* des valeurs des feuilles), l'ensemble $\mathcal{A}(\mathcal{N}, \mathcal{F})$ des arbres binaires est défini par :

- toute feuille est un arbre : $\mathcal{F} \subset \mathcal{A}(\mathcal{N}, \mathcal{F})$;
- si α et β sont deux arbres, et si $n \in \mathcal{N}$, alors (n, α, β) est un arbre.

Le typage Camlcorrespondant est le suivant :

```
type ('n,'f) arbre_binaire =
    | Feuille of 'f
    | Noeud of 'n * ('n,'f) arbre_binaire * ('n,'f) arbre_binaire ;;
```

Un exemple d'arbre binaire du type `(int,string) arbre_binaire` est :

```
Noeud(1,Feuille "a",
       Noeud(6,Noeud(4,Feuille "c",
                     Feuille "d")
              Feuille "b"))
```

qui correspond au dessin de la page suivante, où on a choisi de représenter les nœuds par des ronds et les feuilles par des carrés.

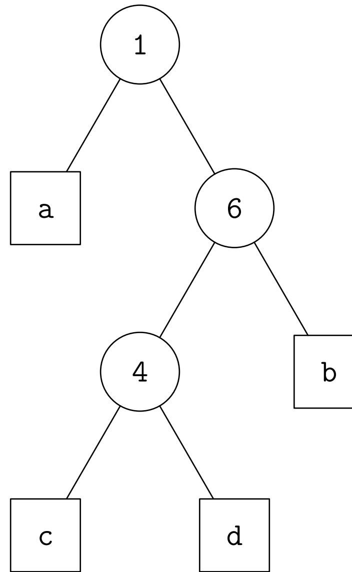


Figure 1 Un premier exemple d'arbre binaire

Squelette d'un arbre Le squelette d'un arbre définit sa géométrie : on l'obtient en supprimant toute information aux nœuds et feuilles, et en supprimant les feuilles. Voici le squelette de l'arbre précédent:

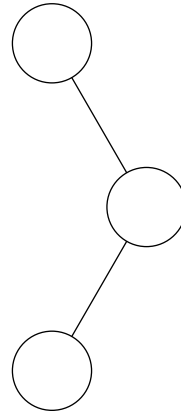


Figure 2 Son squelette

On type aisément les squelettes d'arbres binaires ainsi :

```
type squelette_binaire =
    | Rien
    | Jointure of squelette_binaire * squelette_binaire ;;
```

Exercice écrire la fonction Caml qui renvoie le squelette de l'arbre binaire argument.
Exercice écrire la fonction Caml qui renvoie le squelette symétrique du squelette argument dans la symétrie par rapport à une verticale.
Exercice écrire la fonction Caml qui teste l'égalité de deux squelettes.

```
let rec décharne = function
    | Feuille _ -> Rien
    |Noeud(_,g,d) -> Jointure(décharne g, décharne d) ;;

let rec symétrie = function
    | Rien -> Rien
    |Jointure(g,d) -> Jointure(symétrie d,symétrie g) ;;

let rec égalité_squelettes a b = match (a,b) with
    |(Rien,Rien) -> true
    | (Jointure(g,d),Jointure(g',d'))
        -> (égalité_squelettes g g') && (égalité_squelettes d d')
    | _ -> false ;;
```

Propriétés combinatoires

Vocabulaire Nous appellerons taille d'un arbre binaire le nombre de ses nœuds **internes**. C'est aussi la taille de son squelette.
On peut donc écrire :

```
let rec taille = function
    | Feuille -> 0
    | Noeud(_,g,d) -> 1 + (taille g) + (taille d) ;;
```

Sa hauteur (on parle aussi de profondeur) est définie inductivement par : toute feuille est de hauteur nulle, la hauteur d'un arbre (n, α, β) est égale au maximum des hauteurs de ses fils α et β augmenté d'une unité : la hauteur mesure donc l'imbrication de la structure.

```
let rec hauteur = function
    | Feuille -> 0
    | Noeud(_,g,d) -> 1 + (max (hauteur g) (hauteur d)) ;;
```

Nota Bene Remarque : la hauteur d'un arbre est égale à la longueur du trajet le plus long de la racine à une feuille (comptée par le nombre d'arêtes parcourues).

On peut également parler de la profondeur d'un nœud (ou d'une feuille) d'un arbre donné : il s'agit alors de la longueur du trajet de la racine jusqu'à ce nœud.
C'est donc aussi la différence des hauteurs de l'arbre entier et du sous-arbre dont le nœud choisi est la racine.
Dans l'exemple de la figure 1, l'arbre est de taille 3 et de hauteur 3. Le nœud interne 4 est de profondeur 2, la feuille b est à profondeur 2.

Le nombre de feuilles se déduit facilement du nombre de nœuds, c'est-à-dire de la taille :

Théorème

[Feuilles et nœuds d'un arbre binaire] Le nombre de feuilles d'un arbre binaire de taille n est égal à $n+1$.

✧ D'aucuns peuvent utiliser une preuve inductive.
Je préfère dire que si f est le nombre de feuilles, $n+f$ compte le nombre de nœuds et feuilles qui sont, sauf la racine, fils d'un nœud interne : $n+f-1=2n$. D'où le résultat : $f=n+1$. ✦

Nous allons voir que taille et profondeur d'un arbre binaire sont étroitement liées.

Théorème

[Hauteur d'un arbre binaire] Soit h la hauteur d'un arbre binaire de taille n. On dispose de :

$$1 + \lfloor \lg n \rfloor \leqslant h \leqslant n.$$

✧ La hauteur est la longueur du plus long chemin de la racine à une feuille. Au lieu de compter les arêtes, on peut compter les nœuds (internes) de départ, et on a bien $h \leqslant n$.
Un arbre binaire de hauteur h est dit complet si toutes ses feuilles sont à la profondeur h ou $h - 1$. On vérifie facilement pour un tel arbre la relation $h = 1 + \lfloor \lg n \rfloor$.
À un arbre binaire non complet on peut ajouter suffisament de nœuds (et de feuilles) pour qu'il soit complet : la taille passe alors de n à $n' = 2^h - 1 \geqslant n$, donc $\lfloor \lg n \rfloor < h$ et ainsi $\lfloor \lg n \rfloor + 1 \leqslant h$. ✦

On en déduit aussitôt le

Corollaire Soit h la hauteur d'un squelette binaire de taille n. On dispose de :

$$\lfloor \lg n \rfloor \leqslant h \leqslant n - 1.$$

Ici encore les bornes sont optimales (c'est-à-dire qu'il existe des squelettes pour lesquels elles sont atteintes).

Exercice écrire une fonction Caml qui renvoie un squelette binaire complet de taille donnée.

Exercice écrire une fonction Caml qui décide si un squelette binaire est ou non complet.

```
let rec squelette_complet = function
    | 0 -> Rien
    | n -> let g = squelette_complet ((n - 1)/2)
           and d = squelette_complet (n - 1 - (n - 1)/2)
           in Jointure(g,d) ;;

let rec est_complet = function
    | Rien -> true
    | Jointure(g,d)
        -> let hg = hauteur g and hd = hauteur d
           in if hg = hd then
             (est_complet g) && (est_complet d)
           else if hg = 1 + hd then
             (est_complet g) && (est_vraiment_complet d)
           else if hg + 1 = hd then
             (est_vraiment_complet g) && (est_complet d)
           else false
and est_vraiment_complet = function
    | Rien -> true
    | Jointure(g,d) -> ((hauteur g) = (hauteur d))
                   && (est_vraiment_complet g) && (est_vraiment_complet d) ;;
```

En fait il est beaucoup plus facile (et efficace) d'écrire d'abord une fonction qui renvoie une liste des profondeurs des feuilles.

```
let liste a =
    let rec aux n l = function
        | Rien -> n :: l
        | Jointure(g,d) -> aux (n + 1) (aux (n + 1) l d) g
    in
    aux 0 []a ;;

let est_complet a =
    let rec minmax = function
        | [] -> failwith "Liste vide"
        | [ t ] -> (t,t)
        | t :: q -> let (m,M) = minmax q
                    in
                    (min m t, max M t)
    in
    let (m,M) = minmax (liste a)
    in
    1 >= (M - m) ;;
```

3 Dénombrement

Soit C_n le nombre de squelettes binaires de taille n: $C_0 = 1$, $C_1 = 1$, $C_2 = 2\dots$

On dispose de la récurrence : $\forall n \geqslant 1, C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$, d'où pour la série génératrice $C(z) = \sum_{n=0}^{+\infty} C_n z^n$, de la relation :

$$C(z) = C_0 + z \sum_{n=1}^{+\infty} C_n z^{n-1} = 1 + z \sum_{n=1}^{+\infty} \sum_{k=0}^{n-1} C_k C_{n-1-k} z^{n-1} = 1 + zC(z)^2.$$

On résout cette équation (en tenant compte que $C(0) = C_0 = 1$) :

$$C(z) = \frac{1 - \sqrt{1-4z}}{2z} = \sum_{n=0}^{+\infty} C_n z^n,$$

et on en déduit le

Théorème

[Nombres de Catalan] Le n-ième nombre de Catalan $C_n = \frac{1}{n+1}\binom{2n}{n}$ compte les squelettes binaires de taille n.

On a $C_n \sim \frac{4^n}{n\sqrt{\pi n}}$ quand $n \longrightarrow +\infty$.

Hauteur d'un arbre binaire On peut tenter de prolonger l'étude précédente, en notant $c_{n,h}$ le nombre d'arbres binaires de taille n et de hauteur h.

On peut poser alors $P(u, z) = \sum_{n=0}^{+\infty} \sum_{h=0}^{n} c_{n,h} u^h z^n$, appelée fonction génératrice double associée aux $c_{n,h}$.

Mais l'expression inductive de la hauteur d'un arbre ne conduit à aucune équation explicite en terme de fonction génératrice qui permette son évaluation.

Posons en revanche $T^{[h]}(z) = \sum_{n=0}^{+\infty} \left(\sum_{k=0}^{h} c_{n,h} \right) z^n$, dénombrant ainsi les arbres binaires de taille n et de hauteur au plus égale à h, au nombre de $c'_{n,h} = \sum_{k=0}^{h} c_{n,h}$.

Cette fois nous avons une récurrence facile : $c'_{n,h+1} = \sum_{k=0}^{n-1} c'_{k,h} c'_{n-1-k,h}$, qui nous permet d'écrire par une méthode analogue à ce qu'on a fait plus haut : $T^{[h+1]}(z) = 1 + zT^{[h]}(z)^2$.
On peut alors démontrer le

Théorème

[Hauteur moyenne d'un arbre binaire — admis] La hauteur moyenne d'un arbre binaire de taille n vaut $2\sqrt{\pi n} + O(n^{1/4+\varepsilon})$ pour tout $\varepsilon > 0$.

Utilisation de séries génératrices doubles

Nombre moyen de nœuds sans descendance

Nous cherchons ici à déterminer, pour un squelette binaire de taille n, le nombre moyen de nœuds qui n'ont ni fils droit ni fils gauche.
Pour cela, nous notons $t_{n,k}$ le nombre de ces squelettes binaires de taille n possédant k tels nœuds, et nous introduisons la SGD

$$T(u,z) = \sum_{k,n} t_{n,k} u^k z^n.$$

Bien sûr nous disposons alors de $T(1,z) = C(z) = \sum_n C_n z^n$.

Mais $\frac{\partial T}{\partial u}(1,z) = \sum_n (\sum_k k\, t_{n,k}) z^n$, et le nombre moyen cherché s'écrit :

$$M_n = \frac{[z^n]\frac{\partial T}{\partial u}(1,z)}{[z^n]T(1,z)}.$$

Or $t_{0,0} = 1$, $t_{1,0} = 0$, $t_{1,1} = 1$ et bien sûr, si $k > n$, $t_{n,k} = 0$.
En outre, pour $n \geqslant 1$, on dispose de la récurrence suivante :

$$t_{n+1,k} = \sum_{n_1+n_2=n} \sum_{k_1+k_2=k} t_{n_1,k_1} t_{n_2,k_2},$$

ce qui revient à écrire sur la série génératrice double :

$$T(u,z) = 1 + uz + zT(u,z)^2 - z.$$

Dérivant par rapport à u, il vient $\dfrac{\partial T}{\partial u}(1, z) = z + 2zT(1, z)\dfrac{\partial T}{\partial u}(1, z)$, d'où

$$\frac{\partial T}{\partial u}(1, z) = \frac{z}{1 - 2zT(1, z)} = \frac{z}{1 - 2zC(z)} = \frac{z}{\sqrt{1 - 4z}}.$$

Finalement :

$$M_n = \frac{[z^n]\frac{z}{\sqrt{1-4z}}}{\frac{1}{n+1}\binom{2n}{n}} = \frac{\binom{2n-2}{n-1}}{\frac{1}{n+1}\binom{2n}{n}} = \frac{n(n+1)}{2(2n-1)} \sim \frac{n}{4}.$$

Longueur de chemin interne moyenne

On appelle lci d'un squelette binaire la somme des profondeurs de ses nœuds.
On cherche à déterminer la lci moyenne d'un squelette de taille n.
On procèdera de façon analogue à ce qu'on a fait pour les nœuds sans descendance, en notant $\ell_{n,k}$ le nombre de squelettes de taille n et de lci égale à k.
On introduit la SGD $L(u,z) = \sum_{k,n} \ell_{n,k} u^k z^n$.

On a toujours $L(1,z) = C(z)$, et la moyenne de la lci est donnée par :

$$L_n = \frac{[z^n]\frac{\partial L}{\partial u}(1,z)}{[z^n]L(1,z)} = \frac{[z^n]\frac{\partial L}{\partial u}(1,z)}{\frac{1}{n+1}\binom{2n}{n}}.$$

Un squelette s de taille $|s| > 0$ a un fils gauche g et un fils droit d.
On dispose de $|s| = |g| + |d| + 1$ et $lci(s) = lci(g) + lci(d) + |g| + |d|$.
Ces relations se traduisent directement par :

$$\begin{aligned} L(u,z) &= 1 + \sum_{g,d} u^{lci(g)+lci(d)+|g|+|d|} z^{|g|+|d|+1} \\ &= 1 + z \sum_{g} (uz)^{|g|} u^{lci(g)} \times \sum_{d} (uz)^{|d|} u^{lci(d)} \\ &= 1 + zL(u,uz)^2. \end{aligned}$$

En dérivant par rapport à u, on obtient

$$\frac{\partial L}{\partial u}(1,z) = \frac{2z^2C(z)C'(z)}{1-2zC(z)} = \frac{z}{1-4z} - \frac{1-z}{\sqrt{1-4z}} + 1,$$

$$L_n = \frac{(n+1)4^n}{\binom{2n}{n}} - 3n - 1 = n\sqrt{\pi n} - 3n + O(\sqrt{n}).$$

4 Parcours d'un arbre

À partir de la définition des arbres binaires, on définit de façon naturelle trois ordres de parcours récursifs, *qui travaillent en profondeur d'abord* :

ordre préfixe on lit la racine, puis on parcourt les arbres fils gauche et droit ;

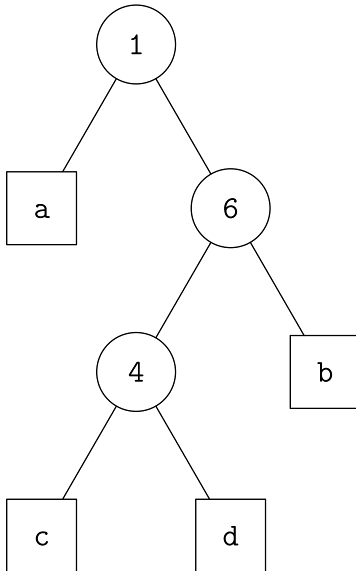
ordre infixe on parcourt l'arbre fils gauche, on lit la racine, puis on parcourt le fils droit ;

ordre suffixe on parcourt les arbres fils gauche et droit, puis on lit la racine.

Nota Bene Au lieu d'ordre infixe, on parle parfois d'ordre symétrique ; au lieu d'ordre suffixe, d'ordre postfixe.

Il faut ajouter l'ordre militaire, qui consiste à lire profondeur par profondeur, de gauche à droite. *C'est un parcours en largeur d'abord.*

Soit par exemple cet arbre :



Voici ses parcours :

militaire : 1a64bcd ;
préfixe : 1a64cdb ;
infixe : a1c4d6b ;
suffixe : acd4b61.

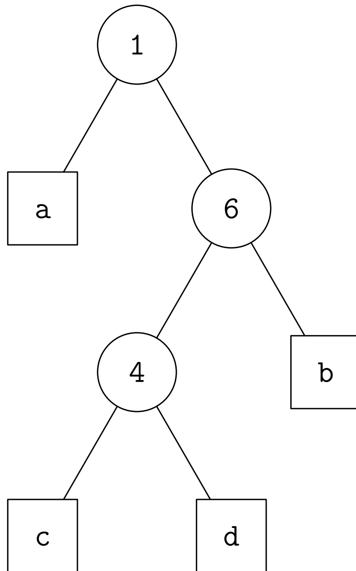
Exercice écrire en Caml ces quatre parcours : `parcours f1 f2 arbre` appliquera `f1` aux feuilles et `f2` aux nœuds.

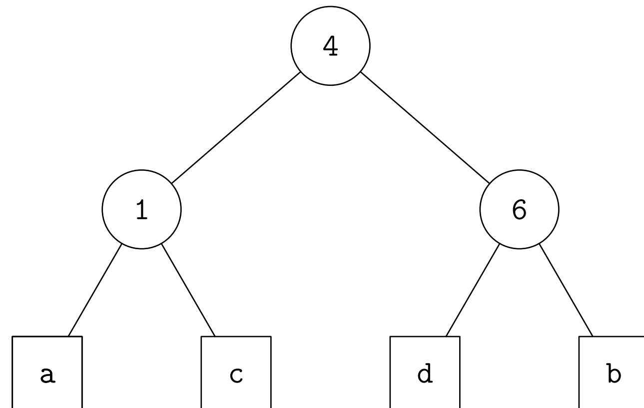
```
let rec parcours_préfixe f1 f2 = function
    | Feuille x -> f1 x
    | Noeud(x,g,d) -> f2 x ;
                     parcours_préfixe f1 f2 g ;
                     parcours_préfixe f1 f2 d ;;

let rec parcours_infixe f1 f2 = function
    | Feuille x -> f1 x
    | Noeud(x,g,d) -> parcours_infixe f1 f2 g ;
                     f2 x ;
                     parcours_infixe f1 f2 d ;;

let rec parcours_suffixe f1 f2 = function
    | Feuille x -> f1 x
    | Noeud(x,g,d) -> parcours_suffixe f1 f2 g ;
                     parcours_suffixe f1 f2 d ;
                     f2 x ;;
```

```
let parcours_militaire f1 f2 a =
    let rec parcours_rec = function
        | [] -> ()
        | Feuille(x) :: reste
            -> f1 x ; parcours_rec reste
        | Noeud(n,g,d) :: reste
            -> f2 n ; parcours_rec (reste @ [ g ; d ])
    in
    parcours_rec [a] ;;
```

Reconstruction de l'arbre

On peut reconstruire un arbre à partir de la donnée de son parcours préfixe, ou bien de son parcours suffixe.
En revanche, deux arbres distincts peuvent avoir le même parcours infixe :





On suppose qu'on se donne un parcours d'un arbre par une liste d'objets du

```
type ('n,'f) listing_d'arbre = F of 'f | N of 'n ;;
```

voici alors comment récupérer l'arbre depuis son parcours préfixe :

```
let recompose_préfixe l =
    let rec aux = function
        | (F f) :: reste -> (Feuille f), reste
        | (N n) :: reste -> let g, reste'  = aux reste  in
                            let d, reste'' = aux reste' in
                            Noeud(n,g,d), reste''
        | [] -> failwith "Description préfixe incorrecte"
    in match aux l with
        | a,[] -> a
        | _ -> failwith "Description préfixe incorrecte" ;;
```

Exercice faire de même pour le parcours suffixe.

```
let recompose_suffixe l =
    let rec aux ss_arbres parcours = match (ss_arbres, parcours) with
        | pile, (F f) :: q -> aux (Feuille(f) :: pile) q
        | d :: g :: pile, (N n) :: q -> aux (Noeud(n,g,d) :: pile) q
        | [ arbre ],[] -> arbre
        | _ -> failwith "Description suffixe incorrecte"
    in
    aux [] l ;;
```

Bien sûr :

Exercice deux arbres distincts peuvent-ils avoir le même parcours militaire ?
si non, écrire la fonction Caml `recompose_militaire`.

On utilisera ici une procédure assez analogue à `recompose_suffixe` appliquée au *miroir* du parcours militaire.

```
let recompose_militaire l =
    let rec aux ss_arbres parcours = match (ss_arbres, parcours) with
        | file, (F f) :: q -> aux (file @ [ Feuille f ]) q
        | d :: g :: file, (N n) :: q
            -> aux (file @ [ Noeud(n,g,d) ]) q
        | [ arbre ],[ ] -> arbre
        | _ -> failwith "Description militaire incorrecte"
    in
    aux [ ] (rev l) ;;
```

Mais il est vrai que cela mériterait une gestion plus efficace des files d'attentes. . .

5 Arbres de recherche

Soit E un ensemble quelconque. On le munit d'une relation d'ordre total en lui associant une valuation $v : E \longrightarrow \mathbb{Z}$ et en posant

$$x \preccurlyeq y \iff v(x) \leqslant v(y).$$

On s'intéresse au problème de la gestion d'une structure *dynamique* permettant la recherche, l'ajout et la suppression d'un élément de E.
Une méthode naïve consiste à utiliser une liste triée, pour lesquelles les trois fonctions de base s'écrivent aisément. Mais chacune d'elles a un coût en $O(n)$ où n est la taille courante de la structure.
Nous allons voir que la structure d'arbre binaire permet *presque* d'améliorer grandement les performances. . .

Utilisation des listes triées

Nous écrivons facilement les trois fonctions souhaitées :

```
let rec cherche v x = function
    | [] -> false
    | t :: q -> (t = x) ||(cherche v x q) ;;

let rec ajoute v x = function
    | [] -> [ x ]
    | t :: q -> if t = x then t :: q
                else if (v t) < (v x) then t :: (ajoute v x q)
                else x :: t :: q ;;

let rec supprime v x = function
    | [] -> []
    | t :: q -> if t = x then q
                else if (v t) < (v x) then t :: (supprime v x q)
                else t :: q ;;
```

Utilisation des arbres binaires

À une partie X de E, nous associons un arbre binaire de recherche dont les feuilles sont les éléments de E et dont les nœuds sont des entiers.
Un arbre binaire de recherche est ou bien une feuille ou bien un arbre binaire (n, g, d) tel que pour toute feuille $x \in g$ on ait $v(x) \leqslant n$ et pour toute feuille $y \in d$ on ait $v(y) > n$.
Ainsi chaque nœud de l'arbre permet, lors d'une recherche, de s'orienter vers le fils gauche ou le fils droit.

Nota Bene On pourrait souhaiter que $n = \max_{x \in g} v(x)$. On vérifiera attentivement que cette condition est compatible avec les fonctions d'ajout et de recherche, mais pas avec la suppression, à moins de la compliquer inutilement.
D'ailleurs imposer cette condition n'améliorerait pas les performances de la structure.

Recherche d'une clé

```
let cherche v x arbre =
    let vx = v x
    in
    let rec aux = function
        | Feuille y -> x = y
        | Noeud(n,g,d) -> if vx <= n then aux g
                          else aux d
    in
    aux arbre ;;
```

Ajout d'une clé

```
let ajout v x arbre =
    let vx = v x
    in
    let rec aux = function
        | Feuille y -> if x = y then Feuille y
                       else if vx <= v y then
                                Noeud(vx, Feuille x, Feuille y)
                       else Noeud(v y, Feuille y, Feuille x)
        | Noeud(n,g,d) -> if vx <= n then Noeud(n, aux g, d)
                          else Noeud(n, g, aux d)
    in
    aux arbre ;;
```

Suppression d'une clé

```
let supprime v x arbre =
    let vx = v x
    in
    let rec aux = function
        | Feuille y -> if x = y then failwith "Arbre vide !"
                       else Feuille y
        | Noeud(n,Feuille y,d) when x = y -> d
        | Noeud(n,g,Feuille y) when x = y -> g
        | Noeud(n,g,d) -> if vx <= n then Noeud(n,aux g,d)
                           else Noeud(n,g,aux d)
    in
    aux arbre ;;
```

On vérifie sans difficulté le :

Théorème

[Coût des opérations sur les arbres de recherche] Chacune des opérations (ajout, suppression ou recherche) sur un arbre binaire de recherche de hauteur h est un $O(h)$.

Or on peut démontrer le (très difficile):

Théorème

[Hauteur moyenne des arbres de recherche — admis] La hauteur moyenne d'un arbre de recherche de taille n est asymptotiquement équivalente à $c \lg n$ où $c \approx 4{,}31107\ldots$ est la solution supérieure à 2 de l'équation $c \ln(2e/c) = 1$.

Remarque on comparera avec la hauteur moyenne d'un arbre binaire de taille n, qui, on le rappelle, est équivalente à $2\sqrt{\pi n}$.

Toutefois, il existe bien des façons de s'assurer qu'un arbre de recherche reste équilibré, c'est-à-dire que sa hauteur soit un $O(\lg n)$.

Nota Bene Toutes ces méthodes, bien que classiques, sont hors programme.

Arbres de recherche : informations aux nœuds

Dans le cas où $E = \mathbb{Z}$, on peut poser $v = \mathrm{Id}$. Dans ce cas, il n'est plus utile de confiner les clés aux feuilles : les nœuds peuvent assurer les deux fonctions de balise et de support de l'information.
On utilisera donc le type suivant :

```
type arbre_recherche_int = Vide
       | Bifurcation of int * arbre_recherche_int * arbre_recherche_int ;;
```

La fonction de recherche d'une clé s'écrit donc maintenant :

```
let rec recherche x = function
    | Vide -> false
    | Bifurcation(n,g,d)
         -> (n = x)
            || (x < n && (recherche x g))
            || recherche x d ;;
```

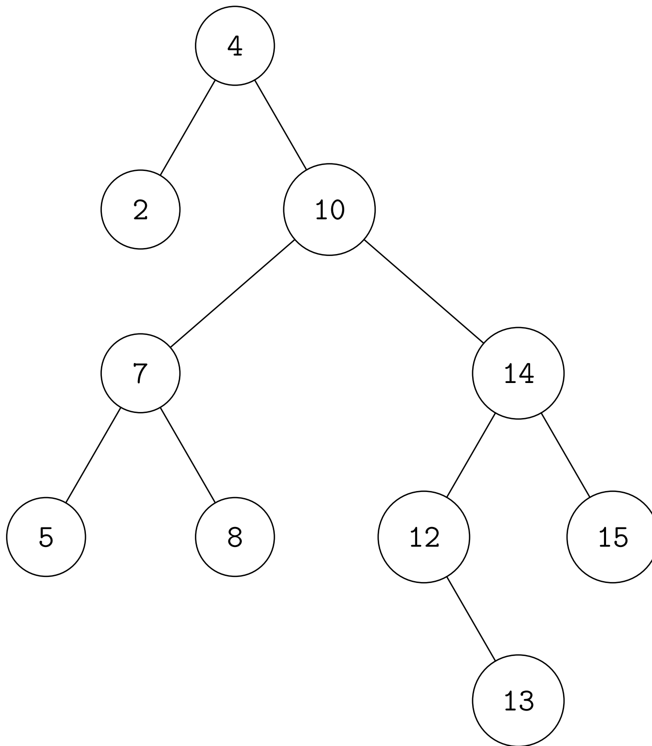
L'ajout peut se faire comme précédemment, aux feuilles :

```
let rec ajout x = function
    | Vide -> Bifurcation(x,Vide,Vide)
    | Bifurcation(n,g,d)
        -> if x < n then Bifurcation(n,ajout x g,d)
           else if x > n then Bifurcation(n,g,ajout x d)
           else Bifurcation(n,g,d) ;;
```

Mais on peut aussi procéder en ajoutant la nouvelle clé à la racine :

```
let rec ajout x arbre =
    let rec partage = function
        | Vide -> Vide,Vide
        | Bifurcation(n,g,d)
           -> if x = n then g,d
              else if x < n then let g',d' = partage g
                                 in
                                 g',Bifurcation(n,d',d)
              else let g',d' = partage d
                   in
                   Bifurcation(n,g,g'),d'
   in
   let g,d = partage arbre
   in
   Bifurcation(x,g,d) ;;
```

La suppression d'une clé est plus délicate :



Quand on supprime la clé 10, on a le choix de la remplacer par la plus grande clé du sous-arbre gauche (ici 8) ou par la plus petite du sous-arbre droit (ici 12; mais il faudra penser à replacer la clé 13).

On obtient le programme suivant :

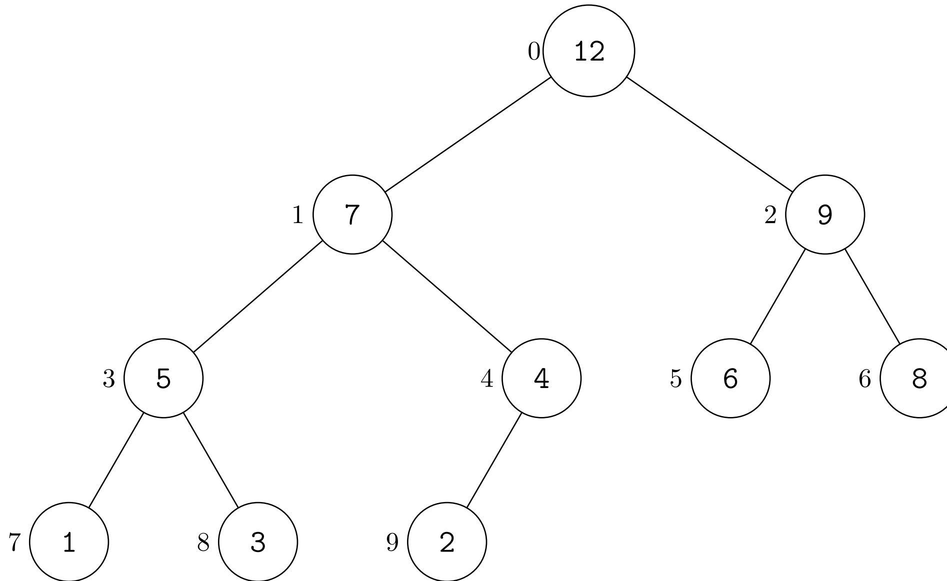
```
let rec supprime x arbre =
    let rec bonne_feuille = function
        | Vide -> failwith "Pas de feuille du tout !"
        | Bifurcation(n,g,Vide) -> n,g
        | Bifurcation(n,g,d)
           -> let x,d' = bonne_feuille d
              in
              x,Bifurcation(n,g,d')
   in
   match arbre with
      | Vide -> Vide
      | Bifurcation(n,g,d) when n > x -> Bifurcation(n,supprime x g,d)
      | Bifurcation(n,g,d) when n < x -> Bifurcation(n,g,supprime x d)
      | Bifurcation(n,g,Vide) -> g
      | Bifurcation(n,Vide,d) -> d
      | Bifurcation(n,g,d) -> let x,g' = bonne_feuille g
                              in
                              Bifurcation(x,g',d) ;;
```

6 Tas et tri

Vocabulaire Un arbre de priorité (on dit aussi arbre tournoi) est un arbre où l'information figure aux nœuds. En outre, on munit l'ensemble E des valeurs utiles d'une valuation $v : E \longrightarrow \mathbb{Z}$, et on garantit que la valuation d'un nœud est supérieure aux valuations de tous ses nœuds-fils que ce soit à gauche ou à droite. Il ne s'agit donc pas du tout d'un arbre de recherche.
Pour alléger les notations, on supposera dans la suite que $E = \mathbb{Z}$, de sorte qu'on n'a plus besoin de valuation v.
On utilisera donc le typage suivant :

```
type tournoi = Vide | Noeud of int * tournoi * tournoi ;;
```

Un tas est un arbre de hauteur h dont toutes les feuilles sont à profondeur $h-1$ ou h, les feuilles de profondeur h étant bien tassées sur la gauche, comme ici :



L'intérêt de la structure apparaît quand on numérote les nœuds dans l'ordre militaire en commençant par 0 : on observe que les fils d'un nœud numéro i sont numérotés $2i+1$ et $2i+2$, et donc que le père d'un nœud numéroté i est numéroté $\lfloor (i-1)/2 \rfloor$: un tableau permet donc de ranger la structure.

Percolation Soit v un vecteur qui représente un tas. On suppose que les fils gauche et droit de la racine sont des arbres tournois. Percoler v, c'est faire descendre à sa place la racine, de sorte que le tas final soit lui-même un arbre tournoi.
L'idée est simple : tant que la clé se trouve dans une position où l'un de ses fils est plus grand qu'elle, on l'échange avec le plus grand de ses fils et on recommence éventuellement.
La percolation tourne donc en $O(h)$ où h est la hauteur du tas. Mais un tas est équilibré, donc il s'agit toujours d'un $O(\lg n)$.
Nous proposons ci-dessous deux versions de la percolation, une récursive, l'autre itérative, puisque l'utilisation des vecteurs autorise à quitter le monde purement fonctionnel. . .

```
let percolation_récursive tas =
    let fils_gauche i = try tas.(2 * i + 1) with _ -> -1
    and fils_droit i = try tas.(2 * i + 2) with _ -> -1
    and échange i j = let a = tas.(i)
                    in (tas.(i) <- tas.(j) ; tas.(j) <- a)
    in
    let rec percole_encore i =
        let x,fg,fd = tas.(i),(fils_gauche i),(fils_droit i)
        in
        if x < fg && fg >= fd then
        begin
            échange i (2 * i + 1) ;
            percole_encore (2 * i + 1)
        end
        else if x < fd && fd >= fg then
        begin
            échange i (2 * i + 2) ;
            percole_encore (2 * i + 2)
        end
    in
    percole_encore 0 ;
    tas ;;
```

```
let percolation_itérative tas =
    let n = vect_length tas
    and x = tas.(0)
    and ix = ref 0
    and fils_gauche i = try tas.(2 * i + 1) with _ -> -1
    and fils_droit i = try tas.(2 * i + 2) with _ -> -1
    in
    while fils_gauche !ix > x || fils_droit !ix > x do
        if fils_gauche !ix > fils_droit !ix then
        begin
            tas.(!ix) <- tas.(2 * !ix + 1) ;
            ix := 2 * !ix + 1
        end
        else
        begin
            tas.(!ix) <- tas.(2 * !ix + 2) ;
            ix := 2 * !ix + 2
        end
    done ;
    tas.(!ix) <- x ;
    tas ;;
```

À l'aide de cette fonction de percolation (en fait un peu modifiée afin qu'elle ne s'applique qu'à un sous-arbre de l'arbre complet), on peut écrire la fonction `réorganise` qui à partir d'un vecteur quelconque renvoie un vecteur représentant un tas.

Les éléments qui figurent les feuilles (c'est-à-dire numérotés de $n - \lfloor n/2 \rfloor$ à $n - 1$ où n est la taille du vecteur) n'ont évidemment pas besoin d'être percolés. En revanche, il faut percoler tous les autres, en partant du bas, c'est-à-dire à rebours de la numérotation.

L'algorithme de réorganisation percole $O(n/2)$ nœuds, à un coût unitaire majoré par un $O(\lg n)$. Il tourne donc finalement à coup sûr en $O(n \lg n)$.

```
let percolation tas i j =
    let x = tas.(i)
    and ix = ref i
    and fils_gauche i = if 2 * i + 1 < j then tas.(2 * i + 1) else -1
    and fils_droit i = if 2 * i + 2 < j then tas.(2 * i + 2) else -1
    in
    while fils_gauche !ix > x || fils_droit !ix > x do
        if fils_gauche !ix > fils_droit !ix then
        begin tas.(!ix) <- tas.(2 * !ix + 1) ; ix := 2 * !ix + 1 end
        else
        begin tas.(!ix) <- tas.(2 * !ix + 2) ; ix := 2 * !ix + 2 end
    done ;
    tas.(!ix) <- x ;
    tas ;;

let réorganise tas =
    let n = vect_length tas
    in
    for i = (n - n/2 - 1) downto 0 do percolation tas i n done ;
    tas ;;
```

Le tri-tas Nous disposons maintenant de tout ce qui nous est utile pour réaliser un tri, appelé tri-tas, ou, en anglais, heap sort : étant donné un vecteur, on commence par le réorganiser; le maximum est alors à la racine, et on le met à sa place, en l'échangeant avec le dernier élément du vecteur. Il faut alors percoler la nouvelle racine, et recommencer ainsi jusqu'à ce qu'on ait extrait successivement $n-1$ maximums.

```
let tri_par_tas tas =
    let n = vect_length tas
    and échange p q = let a = tas.(p)
                      in ( tas.(p) <- tas.(q) ; tas.(q) <- a )
    in
    réorganise tas ;
    for i = n - 1 downto 1 do
        échange 0 i ;
        percolation tas 0 i
    done ;
    tas ;;
```

Rappelons que la réorganisation coûte $O(n \lg n)$ et que chaque percolation coûte $O(\lg n)$.

Théorème

[Performance du heap-sort] Le tri par tas d'un vecteur de taille n a un coût en $O(n \lg n)$.

Finalement notre tri par tas est un excellent algorithme de tri sur les vecteurs : le détour par les arbres a permis d'écrire un programme qui serait incompréhensible sans cette interprétation, mais qui n'est pas difficile quand on a la clé.

7 Arbres n-aires

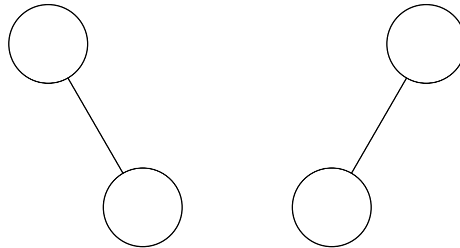
Il s'agit d'une généralisation des arbres binaires.
Un arbre général (ou arbre n-aire) sur un ensemble $\mathcal{X}$ de valeurs est un couple $(x, (a_1, \ldots, a_k))$ où $x \in \mathcal{X}$, k est un entier éventuellement nul et chaque a_i est un arbre général sur $\mathcal{X}$.
Le typage Caml correspondant est :

```
type 'a arbre = Père of 'a * 'a arbre list ;;
```

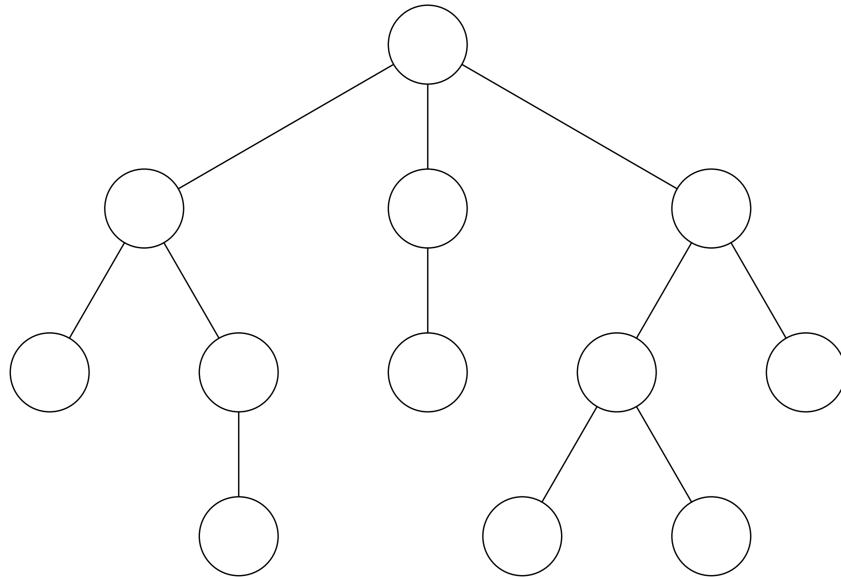
L'analogue d'une feuille est donc ici un nœud muni d'une liste vide de fils.
On définit de façon analogue les squelettes (d'arbres) généraux qu'on type ainsi:

```
type squelette = Joint of squelette list ;;
```

Toutefois, un squelette binaire n'est pas tout à fait un cas particulier de squelette général, puisqu'on n'a plus de notion de gauche et droite : quand il y a un seul fils à un nœud, on ne dit pas qu'il est gauche ni droit. Ainsi les deux squelettes binaires suivants correspondent au même squelette général.



On définit la profondeur et la taille d'un squelette ou d'un arbre général d'une façon analogue à ce qu'on a pu faire dans le cas des squelettes binaires.
On trouvera ci-dessous un schéma d'un squelette général, de profondeur 3 et de taille 12.



Voici comment programmer la taille et la profondeur d'un squelette général:

```
let rec it_list f x = function       (* fonction standard Caml *)
    | [] -> x
    | t :: q -> it_list f (f x t) q ;;

let rec taille = function
    | Joint fils -> it_list (fun n a -> n + (taille a)) 1 fils ;;

let rec profondeur = function
    | Joint fils -> it_list (fun p a -> max p (profondeur a))
                            (-1)
                            fils
                    + 1 ;;
```

Dénombrement

Soit G_n le nombre de squelettes généraux de taille n, et soit $G(z) = \sum_{n \in \mathbb{N}} G_n z^n$ la série génératrice associée.

On a $G_0 = 0$, $G_1 = 1$ et, pour $n \geqslant 0$: $G_{n+1} = \sum_{i \geqslant 0} \sum_{k_1 + \cdots + k_i = n} G_{k_1} G_{k_2} \ldots G_{k_i}$.

On peut traduire cette relation de récurrence sur la fonction génératrice : $G(z) = z(1 + G(z) + G(z)^2 + G(z)^3 + \cdots) = \dfrac{z}{1 - G(z)}$.

On en déduit : $G(z) = \dfrac{1 - \sqrt{1 - 4z}}{2}$, or on se rappelle que la série génératrice des nombres de Catalan s'écrit $C(z) = \dfrac{1 - \sqrt{1 - 4z}}{2z}$, donc $G(z) = zC(z)$, ce qui se traduit par le:

Théorème

[Dénombrement des squelettes généraux] Le nombre G_n de squelettes généraux de taille n vaut $C_{n-1} = \dfrac{1}{n} \dbinom{2n-2}{n-1}$.

On peut d'ailleurs exhiber une bijection entre les ensembles des squelettes binaires de taille $n+1$ et des squelettes généraux de taille n, qu'on peut même programmer en Caml.

```
let rec bin_de_gnal = function
    | Joint [] -> Vide
    | Joint (t :: q)
       -> Jointure (bin_de_gnal t,bin_de_gnal (Joint q)) ;;

let rec gnal_de_bin = function
    | Vide -> Joint []
    | Jointure(g,d)
       -> let (Joint l) = gnal_de_bin d
          in
          Joint ((gnal_de_bin g) :: l) ;;
```

Parcours d'un arbre général On définit bien sûr de façon analogue à ce qu'on a fait pour les arbres binaires les parcours militaire, préfixe et suffixe d'un arbre (ou d'un squelette) général. En revanche, bien entendu, il n'y a plus de parcours infixe !

Par exemple :

```
let rec parcours_préfixe f = function
    | Père(x,fils) -> f x ;
                      do_list parcours_préfixe f fils ;;
```

et, de même :

```
let parcours_militaire f a =
    let rec aux = function
        | [] -> ()
        | Père(x,fils) :: q -> f x ;
                               aux (q @ fils)
    in
    aux [ a ] ;;
```

En revanche, si on veut par exemple écrire la reconstitution d'un arbre général à partir de son parcours préfixe, le nombre variable de fils pour chaque nœud interdit un simple décalquage de la fonction qu'on avait écrite pour les arbres binaires.
On écrira le type d'un parcours d'un arbre du type `'a arbre` ainsi: `type 'a parcours == 'a list ;;` tout simplement.
En revanche on suppose connue une fonction `arité : 'a -> int` qui fournisse le nombre (éventuellement nul) de fils de chaque nœud.
On peut alors écrire la fonction souhaitée

```
reconstitue_préfixe : ('a -> int) -> 'a parcours -> 'a arbre
```

```
let reconstitue_préfixe arité parcours =
    let rec aux = function
        | [] -> failwith "Erreur de syntaxe"
        | x :: q -> let n = arité x
                    in
                    let fils, reste = cherche n q
                    in
                    Père(x,fils), q
    and cherche n parcours = match n with
        | 0 -> [], parcours
        | _ -> let fils, reste = aux parcours
               in
               let autres_fils, autre_reste = cherche (n-1) reste
               in
               (fils :: autres_fils), autre_reste
    in
    match aux parcours with
       | a,[] -> a
       | failwith "Erreur de syntaxe" ;;
```

La reconstitution à partir du parcours suffixe est encore plus compliquée à écrire:

```
let reconstitue_suffixe arité parcours =
    let rec aux pile parcours = match (pile,parcours) with
        | ([ a ], []) ->  a
        | (_,[]) -> failwith "Erreur de syntaxe"
        | (pile, x :: q) -> let fils, pile' = dépile (arité x) pile []
                            in
                            aux (Père(x,fils) :: pile') q
    and dépile n pile fils = match n with
        | 0 -> fils,pile
        | _ -> match pile with
                  | [] -> failwith "Erreur de syntaxe"
                  | f :: p -> dépile (n-1) p (f :: fils)
    in
    aux [] parcours ;;
```

8 Arbres et expressions

Syntaxe abstraite

L'ensemble $\mathcal{E}$ des expressions arithmétiques se définit de façon récursive.
On considère un ensemble $\mathcal{C}$ de constantes, un ensemble $\mathcal{V}$ de variables, un ensemble fini $\mathcal{O}$ d'opérateurs binaires et un ensemble fini $\mathcal{F}$ d'opérateurs unaires ou fonctions. Par exemple : $\mathcal{C} = \mathbb{R}$, $\mathcal{V} = \{x_1, x_2, \ldots\}$, $\mathcal{O} = \{+, -, \times, /\}$ et $\mathcal{F} = \{\sin, \cos, \tan, \sqrt{\ }, \ln\}$.
Alors toute constante est une expression, toute variable est une expression, et, si $c \in \mathcal{O}$ et $f \in \mathcal{F}$ et e_1 et e_2 sont deux expressions, $(e_1\ c\ e_2)$ et $(f\ e_1)$ sont des expressions.
En pratique, les règles usuelles de priorité entre opérateurs et d'associativité permettent de réduire le nombre de parenthèses utiles.
Ainsi, on écrira $\sin(\pi/4) + 3 \times \cos(2 \times \pi/5)$ et non pas

$$((\sin(\pi/4)) + (3 \times (\cos(2 \times (\pi/5))))).$$

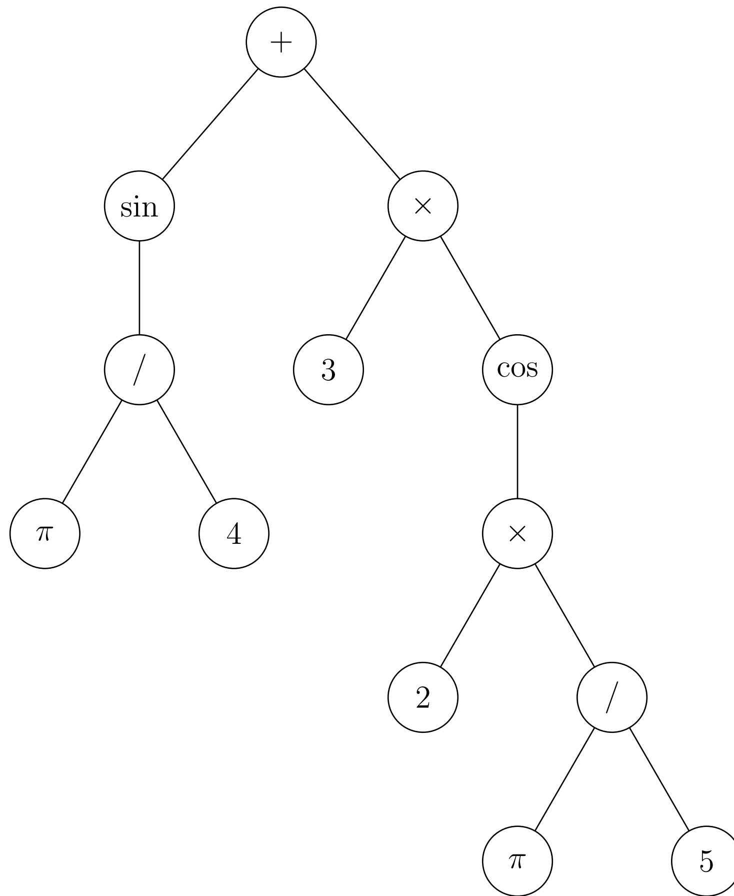
La grammaire des expressions peut donc être écrite ainsi :

$$\begin{aligned} expression ::= &\ constante \\ &\mid variable \\ &\mid (expression\ op\ expression) \\ &\mid (fonction\ expression) \end{aligned}$$

Syntaxe concrète : arbres d'expression

On associe naturellement à une expression arithmétique un arbre général : aux variables et constantes correspondent les feuilles de l'arbre, aux opérateurs binaires des nœuds binaires, et aux fonctions des nœuds unaires.
Par exemple, voici page suivante l'arbre de l'expression

$$\sin(\pi/4) + 3 \times \cos(2 \times \pi/5).$$


+
sin
×
/
3
cos
π
4
×
2
/
π
5

Syntaxe concrète : Caml

Le typage Caml correspondant est immédiat:

```
type ('c,'v,'o,'f) expr =
    | Constante of 'c
    | Variable of 'v
    | Terme2 of ('c,'v,'o,'f) expr * 'o  * ('c,'v,'o,'f) expr
    | Terme1 of 'f * ('c,'v,'o,'f) expr ;;
```

mais comme en pratique nos opérateurs binaires sont tous associatifs à gauche, on préfèrera :

```
type ('c,'v,'o,'f) expression =
    | Constante of 'c
    | Variable of 'v
    | Terme of 'o * ('c,'v,'o,'f) expression list
    | Applique of 'f * ('c,'v,'o,'f) expression ;;
```

et par exemple on définira

```
type fonction = Sin | Cos | Tan | Sqrt | Ln ;;
type expr == (float,string,char,fonction) expression ;;
```

On a choisi de représenter les variables par leurs noms, qui sont des chaînes de caractères ; et les opérateurs binaires par leur symbole, qui est un caractère. On représentera par exemple l'expression $1.3 + \sqrt{2} + x$ par :

```
let exemple = Terme(‘+‘,[Constante 1.3 ;
                          Applique(Sqrt,Constante 2.0) ;
                          Variable "x"]) ;;
```

Sémantique des expressions

Notion de contexte Un contexte (d'évaluation) est simplement une application φ de $\mathcal{V}$, l'ensemble des variables, dans $\mathcal{C}$, l'ensemble des valeurs. Cependant on notera $[\varphi]\, v$ au lieu de $\varphi(v)$, ce qui se justifiera bientôt.
En Caml, un contexte est souvent représenté par une liste de couples (v, c) variable-valeur, c'est-à-dire une liste associative de type `('v * 'c) list` et on écrira souvent un contexte sous cette forme : $[(v_1, c_1); (v_2, c_2); \ldots]\, v = c_k$ dès que $v = v_k$.
Dans la suite, l'ensemble des contextes est (naturellement) noté $\mathcal{C}^{\mathcal{V}}$.

À tout opérateur binaire o on associe son interprétation $\tilde{o} : \mathcal{C} \times \mathcal{C} \longrightarrow \mathcal{C}$ et, de même, à toute fonction f on associe son interprétation $\tilde{f} : \mathcal{C} \longrightarrow \mathcal{C}$.
Ainsi au **symbole** $\sin$ on associe son interprétation : le sinus, etc.
On définit alors une sémantique en introduisant la fonction d'évaluation définie sur $\mathcal{C}^{\mathcal{V}} \times \mathcal{E}$ de la façon suivante :

- $[\varphi]\, c = c$ pour tout contexte φ et toute constante c ;
- $[\varphi]\, v = \varphi(v)$ pour tout contexte φ et toute variable v ;
- $[\varphi]\, (e_1\, o\, e_2) = \tilde{o}([\varphi]\, e_1, [\varphi]\, e_2)$, pour tout contexte φ, toutes expressions e_1 et e_2, et tout opérateur o d'interprétation $\tilde{o}$;
- $[\varphi]\, (f\, e) = \tilde{f}([\varphi]\, e)$, pour tout contexte φ, toute expression e et toute fonction f d'interprétation $\tilde{f}$.

On traduit ceci immédiatement en Caml:

```
let rec assoc v = function
    | [] -> failwith "Contexte incomplet"
    | (w,x) :: q when w = v -> x
    | _ :: q -> assoc v q
and compose f (t :: q) = match q with
    | [] -> t
    | t' :: q' -> compose f ((f t t') :: q) ;;

let rec eval contexte = function
    | Constante x -> x
    | Variable v -> assoc v contexte
    | Terme(`+`,l) -> compose (fun x y -> x +. y) (map (eval contexte) l)
    | Terme(`-`,l) -> compose (fun x y -> x -. y) (map (eval contexte) l)
    | Terme(`*`,l) -> compose (fun x y -> x *. y) (map (eval contexte) l)
    | Terme(`/`,l) -> compose (fun x y -> x /. y) (map (eval contexte) l)
    | Applique(f,e) -> let x = eval contexte e in match f with
                           | Sin -> sin x | Cos -> cos x | Tan -> tan x
                           | Sqrt -> sqrt x | Ln -> log x ;;
```

On laisse en exercice au lecteur l'écriture d'une fonction `dérive : expr -> string -> expr`
Ce n'est pas très compliqué, et même plutôt amusant !
Là où cela se compliquerait, c'est si l'on demandait d'écrire une fonction de **simplification** des expressions, problème difficile qui dépasse très largement le cadre et les ambitions de ce cours.