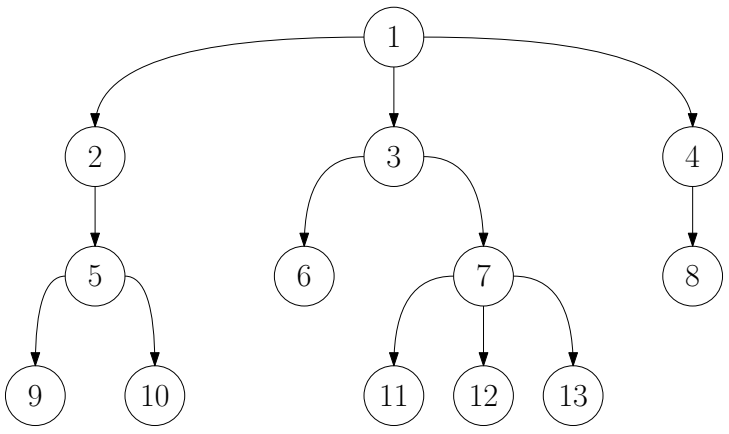
Cours : Arbres

Table des matières

1 Introduction

1.1 Premier exemple



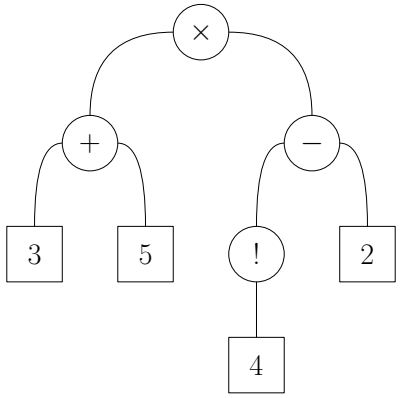
L'arbre T

— Un *arbre* est une structure hiérarchique constituée de *nœuds* qui peuvent éventuellement porter des *étiquettes*.
— L'arbre ci-dessus contient 13 nœuds, étiquetés avec les entiers de 1 à 13.
— Chaque nœud a un certain nombre (éventuellement nul) d'*enfants*. Ici :
  — les enfants du nœud 1 sont les nœuds 2, 3 et 4 ;
  — le seul enfant du nœud 2 est le nœud 5 ;
  — le nœud 9 n'a pas d'enfant.
— L'*arité* d'un nœud est son nombre d'enfants. Le nœud 1 est d'arité 3, le nœud 2 d'arité 1 et le nœud 9 d'arité 0.
— Un nœud d'arité 0 est appelé *feuille* ; un nœud d'arité non nulle est appelé *nœud interne*.

— Si x est un enfant de y, on dit que y est le *père* de x.
— Dans un arbre, il existe un unique nœud qui n'a pas de père : on appelle ce nœud la *racine* de l'arbre.
— Les *ancêtres* du nœud 7 sont les nœuds 3 et 1 : son père, le père de son père, et ainsi de suite jusqu'à la racine de l'arbre.
— Les enfants d'un même père sont dit *frères* : par exemple, 11 et 13 sont les frères de 12.
— La *profondeur* d'un nœud est la longueur du chemin qui relie la racine à ce nœud. La racine est à profondeur 0, ses enfants sont à profondeur 1, ses petits-enfants à profondeur 2...
— La *hauteur* de l'arbre est la profondeur maximum d'un nœud. Ici, la hauteur vaut 3.

1.2 Expression arithmétique

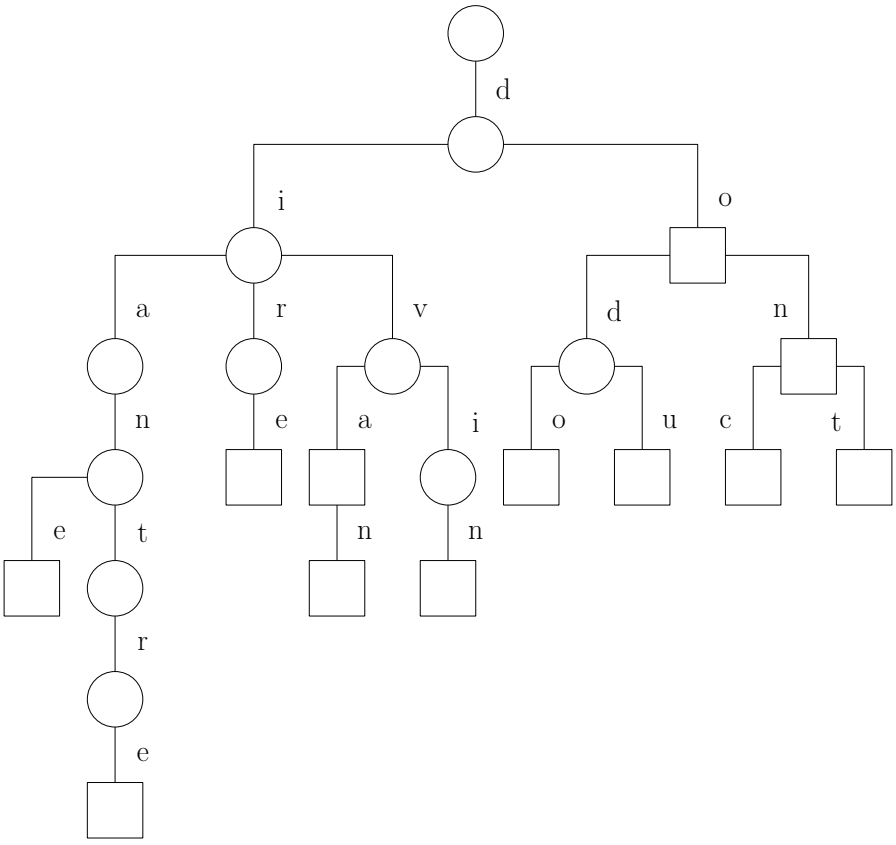
Une expression arithmétique se voit naturellement comme un arbre ; il serait même plus approprié de dire qu'une expression arithmétique *est* un arbre, que l'on a pour habitude de représenter « à plat » en s'aidant de parenthèses.



L'arbre de l'expression $(3 + 5) \times (4! - 2)$.

1.3 Trie

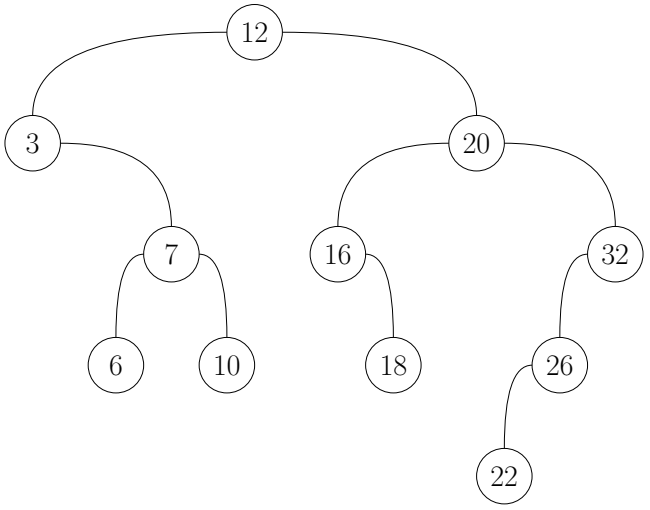
Dans la variante ci-dessous, les étiquettes sont portées par les arêtes et non par les nœuds. Implicitement, un nœud est étiqueté par le mot qu'on lit en lisant les caractères situés sur le chemin depuis la racine. L'arbre représente un ensemble de mots : l'étiquette d'un nœud appartient à cet ensemble si le nœud correspondant est représenté par un carré.



Un *trie* pour l'ensemble {diane, diantre, dire, diva, divan, divin, do, dodo, dodu, don, donc, dont}.

1.4 Arbre binaire de recherche

Un *arbre binaire de recherche* est une structure de données permettant de réaliser les types abstraits Ensemble et Dictionnaire (entre autres). La propriété fondamentale d'un tel arbre est que, pour chaque nœud x, tous les nœuds du sous-arbre gauche de x portent une étiquette inférieure à celle de x et tous ceux du sous-arbre droit une étiquette supérieure à celle de x.

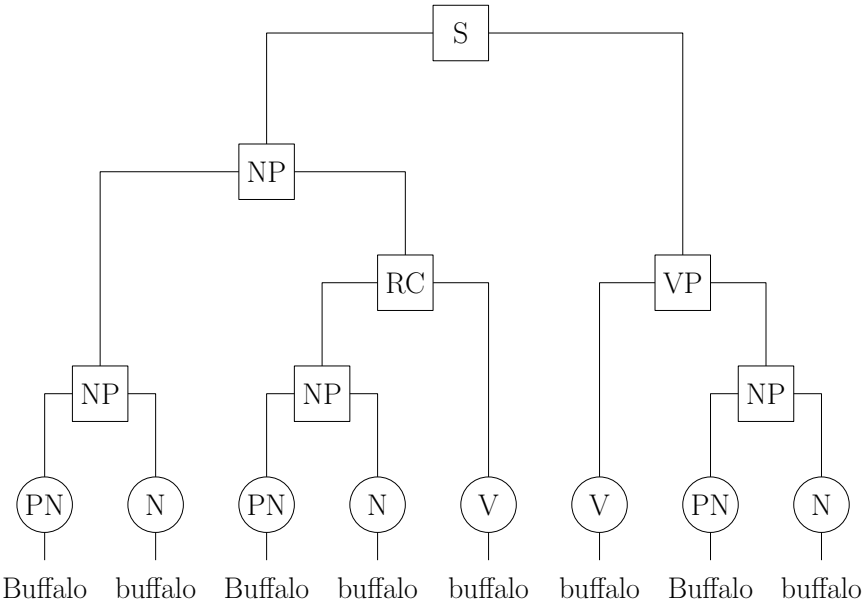


Un ABR pour l'ensemble $\{3, 6, 7, 10, 12, 16, 18, 20, 22, 26, 32\}$.

On observe une structure d'arbre un peu différente ici : certains nœuds n'ont qu'un seul fils, mais ce fils est un fils gauche ou un fils droit (dans les deux exemples précédents, un nœud d'arité 1 avait juste un fils qui n'était ni un fils droit ni un fils gauche).

1.5 Arbre syntaxique

De même que l'on peut faire apparaître la structure d'une expression arithmétique en rendant explicite l'arbre qui lui est sous-jacent, on peut faire apparaître la *structure syntaxique* d'une « phrase » à l'aide d'un arbre. La « phrase » en question peut aussi bien être une vraie phrase, issue d'une langue naturelle, qu'un fragment généré par une grammaire formelle (par exemple un extrait de code OCaml ou C).



Un exemple typique d'arbre syntaxique, qui permet de rendre limpide une phrase qui pouvait sembler quelque peu déroutante de prime abord.

La phrase ci-dessus est plus facile à comprendre en remplaçant les mots « buffalo » par des synonymes ou des mots de la même famille. On peut ainsi obtenir la phrase « Chicago cats (which) Denver dogs bully, bully Minnesota mice ».

2 Définitions

Remarque importante Il est très courant que les définitions varient légèrement d'une fois sur l'autre, parce que l'on choisit ce qui est le plus adapté à un problème donné. Par conséquent, il faudra être très attentif à bien lire les sujets (de TD, de TP, de devoir, de concours) pour identifier la définition précise avec laquelle on travaille.

2.1 Arbre binaire strict

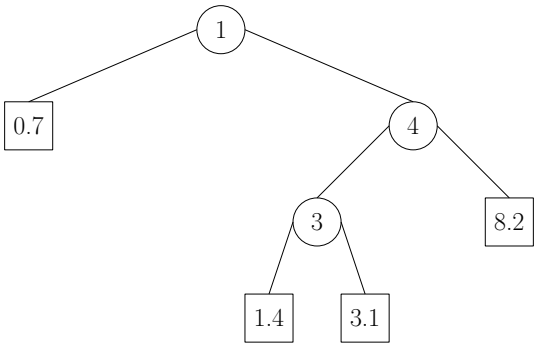
Définition 2.1: Arbre binaire strict

On considère un ensemble $\mathcal{F}$ (ensemble des étiquettes des feuilles) et un ensemble $\mathcal{N}$ (ensemble des étiquettes des nœuds internes). On peut alors définir l'ensemble $\mathcal{A}(\mathcal{N},\mathcal{F})$ des arbres binaires stricts comme suit :
- les feuilles sont des arbres : $\mathcal{F} \subset \mathcal{A}(\mathcal{N},\mathcal{F})$;
- si g et d sont deux arbres et si $x \in \mathcal{N}$, alors $N(x,g,d) \in \mathcal{A}(\mathcal{N},\mathcal{F})$.

On a une correspondance immédiate avec une définition de type en OCaml :

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

```
let arbre_exemple =
  Noeud(1,
        Feuille 0.7,
        Noeud(4,
              Noeud(3,
                    Feuille 1.4,
                    Feuille 3.1),
              Feuille 8.2))
```



Un exemple d'arbre binaire strict de type `int, float`.

Remarques

⇨ Dans la définition ci-dessus, il faut comprendre implicitement que $\mathcal{A}(\mathcal{N},\mathcal{F})$ est *le plus petit ensemble* (au sens de l'inclusion) vérifiant les deux conditions.

⇨ Autrement dit, un arbre binaire strict est un arbre dans lequel tous les nœuds internes sont d'arité 2.

Induction structurelle L'ensemble des arbres binaires stricts est un cas particulier d'*ensemble inductif* : tout arbre est soit une feuille, soit de la forme $N(x,g,d)$ où g et d sont des arbres plus « simples ». Nous étudierons un peu la théorie des ensembles inductifs ultérieurement, mais pour l'instant il faut comprendre les implications pratiques.

Preuve par induction structurelle Si P est un prédicat sur les arbres tel que :
- pour tout $f \in \mathcal{F}$, $P(f)$;
- pour tous $x \in \mathcal{N}$ et $g,d \in \mathcal{A}(\mathcal{F},\mathcal{N})$, $(P(g) \quad \text{et} \quad P(d)) \Longrightarrow P(N(x,g,d))$,

alors P est vérifié pour tout arbre de $\mathcal{A}(\mathcal{F},\mathcal{N})$.

Définition par induction On peut définir une fonction φ sur $\mathcal{A}(\mathcal{F},\mathcal{N})$ en :
- donnant sa valeur sur les feuilles (cas de base) ;
- définissant $\varphi(N(x,g,d))$ en fonction de x, $\varphi(g)$ et $\varphi(d)$.

Exercice 1

⇨ On considère un ensemble $\mathcal{N}$ d'étiquettes, et l'on définit $\mathcal{T}(\mathcal{N})$ de la manière suivante :
- l'arbre vide $\perp$ appartient à $\mathcal{T}(\mathcal{N})$;
- si g et d sont dans $\mathcal{T}(\mathcal{N})$ et si $n \in \mathcal{N}$, alors $(n,g,d) \in \mathcal{T}(\mathcal{N})$.

1. Proposer un type OCaml permettant de représenter cette famille d'arbres.
2. Montrer que cette définition peut être vue comme un cas particulier de la précédente.

2.2 Vocabulaire

Définition 2.2: Profondeur, hauteur

La *hauteur* d'un arbre binaire strict est définie récursivement par :
- $h(T) \coloneqq 0$ si T est une feuille.
- $h(N(n, g, d)) \coloneqq 1 + \max(h(g), h(d))$.

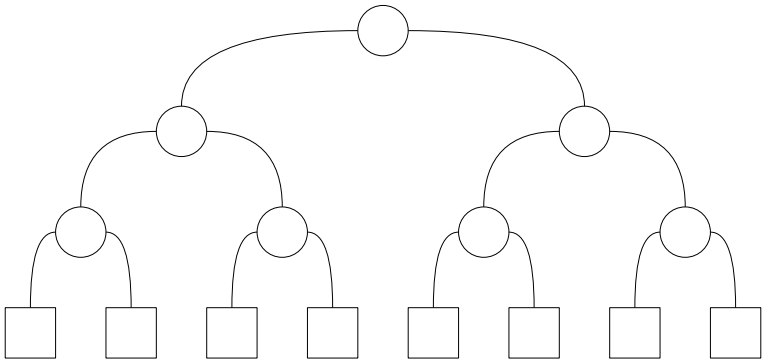
La *profondeur* d'un nœud x dans un arbre binaire est la longueur du chemin qui relie la racine à ce nœud.

Remarque

⇨ La hauteur d'un arbre est donc la longueur maximale d'un chemin reliant la racine à une feuille.

Définition 2.3: Arbre binaire parfait

Un *arbre binaire parfait* est :
- Soit une feuille.
- Soit de la forme $N(x, g, d)$, où g et d sont deux arbres binaires parfaits de même hauteur.



Proposition 2.4

Un arbre binaire strict est parfait si et seulement si toutes ses feuilles sont à la même profondeur.

Démonstration. On procède par induction structurelle.
- Si T est une feuille, c'est évident.
- Si $T = N(x, g, d)$, alors g et d sont deux arbres binaires parfaits de même hauteur h, par définition. Cela signifie que la feuille la moins profonde de g est à profondeur h dans g, et donc, par hypothèse d'induction, que toutes les feuilles de g sont à profondeur h dans g, et donc à profondeur $h + 1$ dans T. Il en va de même pour les feuilles de d, or les feuilles de T sont exactement celles de g et celles de d : elles sont donc toutes à la même profondeur $h + 1$.

□

2.3 Arbre binaire non strict

Définition 2.5: Arbre binaire non strict

On considère un ensemble $\mathcal{E}$ (ensemble des étiquettes des nœuds). L'ensemble $\mathcal{A}(\mathcal{E})$ des arbres binaires non stricts étiquetés par $\mathcal{E}$ est défini inductivement par :
- L'arbre vide $\perp$ appartient à $\mathcal{A}(\mathcal{E})$.
- Si g et d appartiennent à $\mathcal{A}(\mathcal{E})$ et $x \in \mathcal{E}$, alors l'arbre $N(x, g, d)$ appartient à $\mathcal{A}(\mathcal{E})$.

Remarques

⇨ Quand on parle d'*arbre binaire*, sans préciser, cela peut signifier arbre binaire strict ou non strict suivant le contexte...

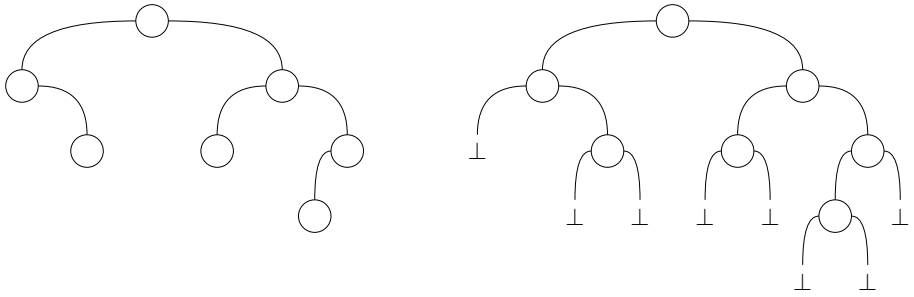
⇨ Un arbre binaire non strict est donc un arbre dans lequel :
- Les nœuds sont d'arité zéro, un ou deux.
- L'unique fils d'un nœud d'arité un est soit un fils gauche, soit un fils droit.

Deux arbres binaires non stricts *différents*.

Le type OCaml qui correspond naturellement à cette définition est le suivant :

```
type 'a arbre_bin_ns =
  | V
  | N of 'a * 'a arbre_bin_ns * 'a arbre_bin_ns
```

Définis ainsi, les arbres binaires non stricts peuvent en fait être vus comme un cas particulier d'arbre binaire strict : il suffit de considérer la définition des arbres binaires stricts et de choisir comme ensemble d'étiquettes pour les feuilles un singleton $\{\bot\}$. En particulier, on peut remarquer que le type `'a arbre_bin_ns` défini ci-dessus est isomorphe au type `('n, 'f) arbre_bin` défini plus dans lequel on a fixé `'f = unit` (ou n'importe quel type à un seul élément).



Un arbre binaire non strict et sa représentation stricte.

Définition 2.6

Dans un arbre binaire non strict :
- L'arité d'un nœud est son nombre d'enfants non vides.
- Une feuille est un nœud d'arité zéro, autrement dit un nœud de la forme $N(x, \bot, \bot)$.
- La hauteur d'un arbre réduit à une feuille vaut 0, ce qui pousse à définir la hauteur de l'arbre vide comme valant -1.

Remarque

⇨ Ces définitions ne correspondent pas à ce qu'on obtiendrait en considérant les arbres binaires non stricts comme des cas particuliers d'arbres binaires stricts. À nouveau, il peut y avoir des variations : il faut être très attentif aux définitions du sujet.

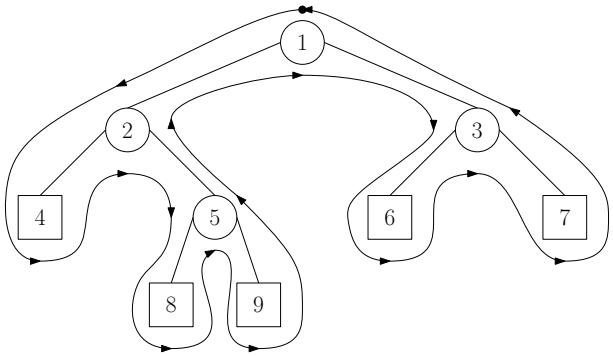
3 Parcours d'arbres binaires

Parcourir un arbre, c'est décrire l'ensemble de ses nœuds (internes ou non) dans un certain ordre, typiquement pour effectuer sur chacun d'entre eux une opération par exemple, afficher l'étiquette à l'écran.

D'une certaine manière, un parcours commence nécessairement à la racine (du point de vue informatique, il n'y a pas vraiment de différence entre l'arbre et sa racine), mais cela ne signifie pas pour autant que l'on souhaite effectuer l'opération en premier sur la racine. Tous les parcours que nous allons décrire se font de gauche à droite.

3.1 Parcours en profondeur

Dans un parcours en profondeur, on commence par descendre le plus profondément possible (jusqu'à une feuille, donc) avant de remonter pour parcourir le reste de l'arbre.



Principe du parcours en profondeur.

On peut faire deux remarques sur le parcours en profondeur :
— Il est naturellement récursif, puisqu'il s'agit de traiter la racine et de parcourir récursivement les sous-arbres gauche et droit (les feuilles étant les cas de base).
— On passe trois fois par chaque nœud interne :
  — Avant d'explorer son fils gauche.
  — Entre l'exploration du fils gauche et celle du fils droit.
  — Après l'exploration du fils droit.

Cette deuxième remarque induit trois ordres différents sur les nœuds :

ordre préfixe : On classe les nœuds par l'instant de leur première visite (pour les feuilles, on considère que les trois visites se font simultanément). Si l'on considère que l'on effectue un traitement sur chaque nœud, cela revient à traiter (récursivement) :
— d'abord la racine,
— puis le sous-arbre gauche,
— puis le sous-arbre droit.
L'ordre induit sur les étiquettes de l'exemple est 1, 2, 4, 5, 8, 9, 3, 6, 7.

ordre infixe : On traite d'abord le sous-arbre gauche, puis le nœud, puis le sous-arbre droit. Cela revient à classer les nœuds dans l'ordre de leur deuxième visite, et l'ordre induit sur les étiquettes de l'exemple est 4, 2, 8, 5, 9, 1, 6, 3, 7

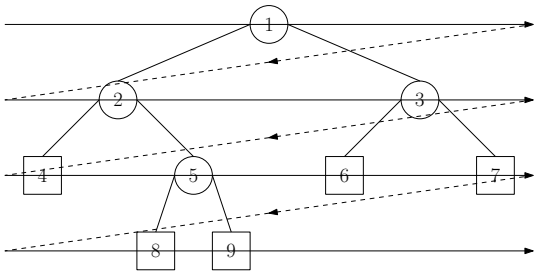
ordre postfixe : On traite d'abord le sous-arbre gauche, puis le sous-arbre droit, puis le nœud. Cela revient à classer les nœuds dans l'ordre de leur troisième visite, et l'ordre induit sur les étiquettes de l'exemple est 4, 8, 9, 5, 2, 6, 7, 3, 1

Remarque

⇨ La notion de parcours en profondeur n'est pas limitée aux arbres binaires : en particulier, l'arbre d'appels d'une fonction récursive est parcouru en profondeur lors de l'exécution (et il n'a aucune raison d'être binaire en général). De même, la notion d'ordre préfixe et postfixe garde un sens en présence de nœuds d'arité quelconque (mais pas celle d'ordre infixe).

3.2 Parcours en largeur

Dans un parcours en largeur, on parcourt les nœuds par profondeur croissante (et de gauche à droite pour une profondeur donnée). Ce parcours est un peu moins utilisé que les variantes du parcours en profondeur.



Principe du parcours en largeur.

L'ordre induit sur les étiquettes de l'exemple est 1, 2, 3, 4, 5, 6, 7, 8, 9

3.3 Unicité

Si l'on dispose de l'énumération des étiquettes dans un ordre défini, on peut se demander s'il est possible de reconstruire l'arbre correspondant (essentiellement, s'il y a un unique arbre donnant cette énumération). Si, comme c'est le cas dans les exemples ci-dessus, les nœuds internes ne sont pas différenciés des feuilles dans l'énumération, la

réponse est non. Cependant, c'est possible dans tous les ordres *sauf l'ordre infixe* si les nœuds internes sont distingués ($\boxed{1}$, $\boxed{2}$, 4, $\boxed{5}$, 8, 9, $\boxed{3}$, 6, 7 par exemple). Nous ferons la démonstration dans un cas plus général, mais l'on peut déjà considérer l'exemple suivant :

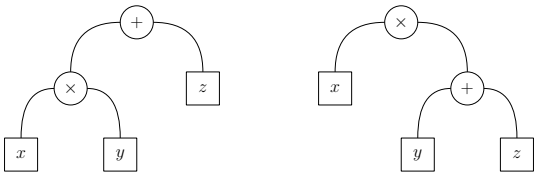


FIGURE 1 : Les deux arbres ci-dessus admettent exactement le même parcours en profondeur infixe : $x \times y + z$. En revanche, leurs parcours préfixes (ou suffixes) sont distincts.

3.4 Programmation en OCaml

Travailler avec des arbres est extrêmement facile en OCaml : c'est l'une des tâches pour lesquelles le langage est particulièrement bien adapté.

```
type 'a bintree =
  | E
  | N of 'a bintree * 'a * 'a bintree
```

```
let rec preorder (t : 'a bintree) (f : 'a -> unit) : unit =
  match t with
  | E -> ()
  | N (l, x, r) ->
      f x;
      preorder l f;
      preorder r f
```

```
let rec inorder (t : 'a bintree) (f : 'a -> unit) : unit =
  match t with
  | E -> ()
  | N (l, x, r) ->
      inorder l f;
      f x;
      inorder r f
```

```
let rec postorder (t : 'a bintree) (f : 'a -> unit) : unit =
  match t with
  | E -> ()
  | N (l, x, r) ->
      postorder l f;
      postorder r f;
      f x
```

```
let preorder_iterative (t : 'a bintree) (f : 'a -> unit) : unit =
  let st = Stack.create () in
  Stack.push t st;
  while not (Stack.is_empty st) do
    match Stack.pop st with
    | E -> ()
    | N (l, x, r) ->
        f x;
        Stack.push r st;
        Stack.push l st
  done
```

```
let inorder_iterative (t : 'a bintree) (f : 'a -> unit) : unit =
  let st = Stack.create () in
  Stack.push t st;
  while not (Stack.is_empty st) do
    match Stack.pop st with
    | E -> ()
    | N (E, x, E) -> f x
    | N (l, x, r) ->
        Stack.push r st;
        Stack.push (N (E, x, E)) st;
        Stack.push l st
  done
```

```
let postorder_iterative (t : 'a bintree) (f : 'a -> unit) : unit =
  let st = Stack.create () in
  Stack.push t st;
  while not (Stack.is_empty st) do
    match Stack.pop st with
    | E -> ()
    | N (E, x, E) -> f x
    | N (l, x, r) ->
        Stack.push (N (E, x, E)) st;
        Stack.push r st;
        Stack.push l st
  done
```

```
let breadthorder (t : 'a bintree) (f : 'a -> unit) : unit =
  let q = Queue.create () in
  Queue.push t q;
  while not (Queue.is_empty q) do
    match Queue.pop q with
    | E -> ()
    | N (l, x, r) ->
        f x;
        Queue.push l q;
        Queue.push r q
  done
```

4 Un peu de dénombrement

4.1 Relations entre hauteur, nombre de nœuds, nombre de feuilles

Proposition 4.1

Soit t un arbre binaire strict possédant n nœuds internes et f feuilles. Alors $f = n + 1$.

Démonstration. On procède par *induction structurelle* sur l'arbre A :
- si A est une feuille, alors $f = 1$, $n = 0$ et la propriété est vérifiée ;
- sinon, A est de la forme (x, A_g, A_d).
  On a alors $f_g = n_g + 1$ et $f_d = n_d + 1$ par hypothèse d'induction, $f = f_g + f_d$ et $n = n_g + n_d + 1$.
  On en déduit :

$$f = n_g + 1 + n_d + 1 = (n_g + n_d + 1) + 1 = n + 1.$$

□

Remarque

⇨ Cette relation est fausse en général pour un arbre binaire non strict.

Exercice 2

⇨ Soit $k \in \mathbb{N}^*$. On appelle *arbre k-aire strict* un arbre dont tous les nœuds internes ont exactement k fils. Trouver une relation entre le nombre de nœuds internes n_t d'un arbre k-aire strict t et son nombre de feuilles f_t, et la

prouver.

Solution. On trouve $n = 1 + (k-1)n$. □

Proposition 4.2

Soit t un arbre binaire strict de hauteur h et f son nombre de feuilles. Alors

$$h + 1 \leqslant f \leqslant 2^h.$$

De plus, $f = 2^h$ si et seulement si t est parfait.

Remarque

⇨ Si on note n le nombre de nœuds internes de l'arbre strict t, on a $h \leqslant n \leqslant 2^h - 1$. Si m est le nombre de nœuds de t, alors $2h + 1 \leqslant m \leqslant 2^{h+1} - 1$.

⇨ Si n est le nombre de noeuds d'un arbre binaire non strict t, alors $h + 1 \leqslant n \leqslant 2^{h+1} - 1$.

⇨ De nombreux algorithmes sur des arbres ayant n nœuds ont une complexité proportionnelle à la hauteur de l'arbre. D'après ce qui précède, si l'arbre est parfait, cette complexité vaut essentiellement $\log_2 n$. Bien sûr, dans le cas d'un arbre très déséquilibré, cette hauteur peut être de l'ordre de n.

Exercices 3

⇨ On appelle *arbre binaire complet de hauteur h* un arbre binaire non strict dont toutes les feuilles sont de profondeur h ou $h-1$. Donner et justifier pour un tel arbre :

1. un encadrement du nombre total de nœuds en fonction de la hauteur.
2. un encadrement de la hauteur en fonction du nombre total de nœuds.

⇨ On considère un arbre binaire non strict dont chaque nœud est colorié soit en rouge, soit en noir et qui vérifie les propriétés suivantes :
- Un nœud rouge ne peut avoir de fils rouge.
- Tous les chemins partant de la racine et descendant jusqu'à une feuille contiennent le même nombre de nœuds noirs.

Montrer qu'il existe une constante A telle que $h \leqslant A \log n$.

4.2 Nombre d'arbres binaires de taille fixée

Définition 4.3

Les *nombres de* CATALAN sont définis par :
- $c_0 \coloneqq 1$,
- $\forall n \in \mathbb{N}, \quad c_{n+1} \coloneqq \sum_{k=0}^{n} c_k c_{n-k}$.

Remarques

⇨ Ces nombres apparaissent naturellement dans de nombreux problèmes de combinatoire.

⇨ La manière « naturelle » de calculer récursivement les nombres de Catalan est très *inefficace*. Comment faire mieux ?

Proposition 4.4

Il y a c_n arbres binaires stricts non étiquetés possédant n nœuds internes.

Démonstration. On procède par récurrence sur le nombre n de nœuds de l'arbre.
- Il y a exactement un arbre binaire sans nœud interne : celui réduit à une feuille.
- Un arbre binaire à $n+1$ nœuds internes est constitué de :
  - une racine (non étiquetée) ;
  - un sous-arbre gauche ayant k nœuds internes (avec $0 \leqslant k \leqslant n$), choisi librement : par hypothèse de récurrence, on a c_k choix ;
  - un sous-arbre droit ayant $n-k$ nœuds internes, choisi librement et indépendamment du sous-arbre gauche : c_{n-k} choix.

  À k fixé, on a donc $c_k c_{n-k}$ arbres à $n+1$ nœuds internes, donc au total $\sum_{k=0}^{n} c_k c_{n-k} = c_{n+1}$ arbres.

□

Proposition 4.5

$$\forall n \in \mathbb{N}, \quad c_n = \frac{1}{n+1}\binom{2n}{n}.$$

Démonstration. — Démontrons le deuxième point par récurrence, en admettant temporairement le premier :

Initialisation Pour $n = 0$, on a bien $\frac{1}{1}\binom{0}{0} = 1 = c_0$.

Hérédité Supposons la propriété vraie au rang n.

$$\begin{aligned}
(n+2)c_{n+1} &= 2(2n+1)\cdot c_n && \text{d'après (??)}\\
&= \frac{2(2n+1)}{n+1}\cdot\binom{2n}{n} && \text{d'après } H_n\\
&= \frac{2(2n+1)}{n+1}\cdot\frac{(n+1)^2}{(2n+1)(2n+2)}\cdot\binom{2n+2}{n+1}\\
&= \frac{2(n+1)}{2n+2}\cdot\binom{2n+2}{n+1}\\
&= \binom{2n+2}{n+1}
\end{aligned}$$

On a donc H_{n+1}, ce qui achève la récurrence.

— Pour le premier point, on remarque que :
- le membre de gauche correspond au nombre de couples (A, f) où A est un arbre à $n+1$ nœuds internes et f est une feuille de A ;
- le membre de droite correspond au nombre de triplets (B, x, ε) où B est un arbre à n nœuds internes, x un nœud (interne ou non) de B et ε est choisi dans l'ensemble $\{\leftarrow, \rightarrow\}$.

On considère alors l'application *efface* qui à un couple (A, f) associe le triplet (B, x, ε) où :
- B est l'arbre obtenu à partir de A en supprimant la feuille f ainsi que son père (le frère de f se trouve alors remonté d'un cran) ;
- x est le nœud remonté (l'ancien frère de f) ;
- ε vaut $\leftarrow$ si f était un fils gauche, $\rightarrow$ si f était un fils droit.

Cette application est une bijection, de réciproque *insere* qui à un triplet (B, x, ε) associe le couple (A, f) où :
- A est obtenu à partir de B en créant un nouveau nœud à l'emplacement de x, ayant pour fils gauche (si $\varepsilon = \leftarrow$) une feuille et comme fils droit x (avec le sous-arbre associé). Les fils sont inversés si $\varepsilon = \rightarrow$;
- f est la feuille nouvellement créée.

On en déduit l'égalité des cardinaux, et donc le point (??).

□

Exercice 4

⇨ On rappelle la formule de Stirling : $n! \sim \sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^n$. En déduire un équivalent simple de c_n.

Solution. On calcule :

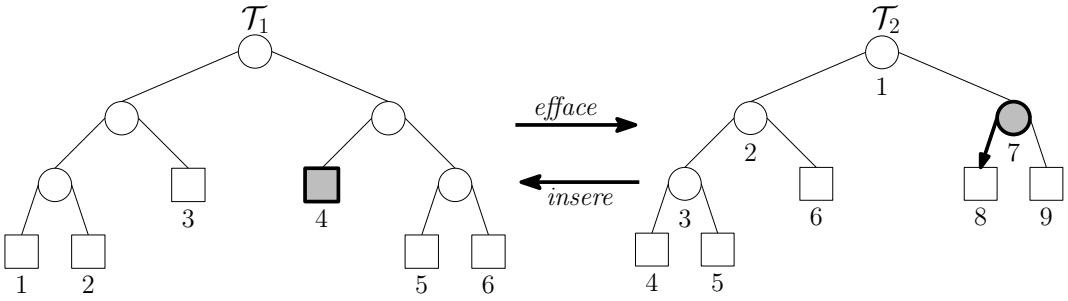
$$\begin{aligned}
c_n &= \frac{1}{n+1}\binom{2n}{n}\\
&\sim \frac{1}{n}\cdot\frac{(2n)!}{(n!)^2}\\
&\sim \frac{1}{n}\cdot\frac{\sqrt{4\pi n}}{2\pi n}\cdot\left(\frac{2n}{e}\right)^{2n}\cdot\left(\frac{e}{n}\right)^{2n}\\
&\sim \frac{1}{n^{3/2}\sqrt{\pi}}\cdot 2^{2n}\\
&\sim \frac{4^n}{n^{3/2}\sqrt{\pi}}
\end{aligned}$$

□

Exemple

⇨ Pour rendre effectives les définitions ci-dessus, il faut choisir une numérotation des nœuds et des feuilles : on choisit

l'ordre préfixe. On obtient alors :



$\text{efface}(\mathcal{T}_1, 4) = (\mathcal{T}_2, 7, \leftarrow)$ et $\text{insere}(\mathcal{T}_2, 7, \leftarrow) = (\mathcal{T}_1, 4)$.

Exercice 5

⇨ En reprenant les arbres ci-dessus, calculer

1. $\text{efface}(\mathcal{T}_1, 2)$,
2. $\text{insere}(\mathcal{T}_2, 6, \rightarrow)$,
3. $\text{efface}(\mathcal{T}_2, 5)$,
4. $\text{insere}(\mathcal{T}_1, 1, \leftarrow)$.

Attention, les numéros correspondent à des feuilles quand on applique *efface* et à des nœuds (internes ou non) quand on applique *insere*.

5 Arbres non binaires

5.1 Lecture unique

On s'intéresse à des arbres dont les nœuds peuvent avoir une arité quelconque : chaque nœud porte une étiquette et possède une liste d'enfants. Une feuille est simplement un nœud dont la liste d'enfants est vide. Un type permettant de représenter de tels arbres pourrait être :

```
type 'a arbre = N of 'a * 'a arbre list
```

On définit le parcours préfixe d'un tel arbre par :
— $\text{prefixe}(N(x, [\,])) = [(x, 0)]$
— $\text{prefixe}(N(x, [a_1, \ldots, a_k])) = [(x, k)] \mathbin{@} \text{prefixe}(a_1) \mathbin{@} \cdots \mathbin{@} \text{prefixe}(a_k)$

Définition 5.1

Étant donnés $x_1, \ldots, x_n \in \mathcal{E}$ et $k_1, \ldots, k_n \in \mathbb{N}$, on considère la suite finie

$$u := [(x_1, k_1), \ldots, (x_n, k_n)].$$

— On appelle *préfixe* de u toute suite de la forme $[(x_1, k_1), \ldots, (x_j, k_j)]$ où $0 \leqslant j \leqslant n$. On dit que le préfixe est *strict* lorsque $j < n$.
— On appelle *facteur* de u toute suite de la forme $[(x_i, k_i), \ldots, (x_{j-1}, k_{j-1})]$ où $1 \leqslant i \leqslant j \leqslant n+1$.

Définition 5.2: Suite équilibrée

On définit le *poids* d'une suite finie $u := [(x_1, k_1), \ldots, (x_n, k_n)]$ par

$$\text{poids}(u) := \sum_{i=1}^{n} (k_i - 1).$$

On dit que la suite u est équilibrée lorsque
— pour tout préfixe strict v de u, $\text{poids}(v) \geqslant 0$.
— $\text{poids}(u) = -1$.

Proposition 5.3

Pour tout arbre t, $\text{prefixe}(t)$ est équilibré.

Proposition 5.4

En tout point d'une suite équilibrée commence un unique facteur équilibré.

Proposition 5.5

Une suite u est équilibrée si et seulement si il existe un arbre t tel que $u = \text{prefixe}(t)$. Dans ce cas, t est unique.

Remarques

⇨ Autrement dit, un arbre est uniquement défini par son parcours préfixe, à condition que ce parcours contienne l'arité de chacun des nœuds.

⇨ Dans le cas d'un arbre binaire strict, zéro et deux sont les seules valeurs possibles de l'arité : il suffit donc de noter pour chaque nœud s'il s'agit d'une feuille ou d'un nœud interne.

⇨ Le résultat correspondant pour le parcours postfixe se prouve exactement de la même manière.

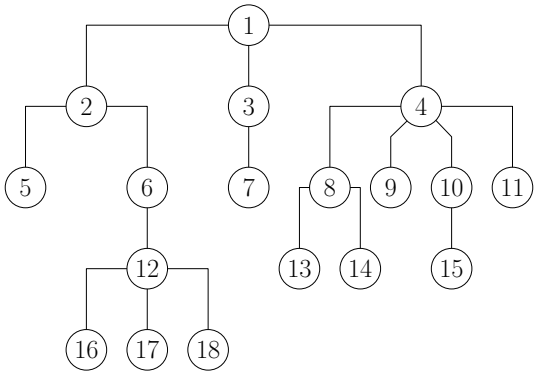
5.2 Transformation en arbre binaire

Il existe une manière standard de transformer un arbre d'arité quelconque en arbre binaire, de manière réversible (une bijection de l'ensemble des arbres dans l'ensemble des arbres binaires, essentiellement) : c'est la représentation LCRS (*Left child, right sibling*, c'est-à-dire fils gauche, frère droit).

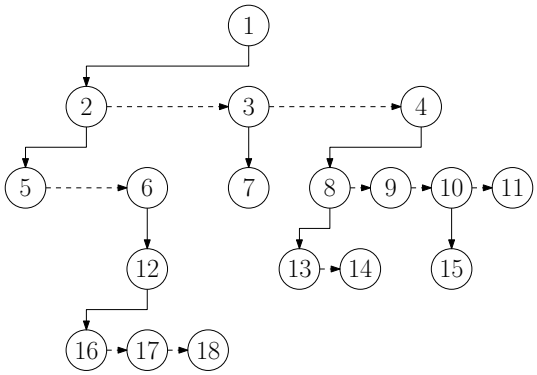
Une manière de comprendre cette représentation est d'imaginer qu'un nœud de l'arbre initial est constitué d'une étiquette et d'une liste chaînée d'enfants (éventuellement vide). Dans ce cas :

— chaque nœud contient un lien vers la liste chaînée de ses enfants, c'est-à-dire vers une cellule contenant son fils le plus à gauche ;

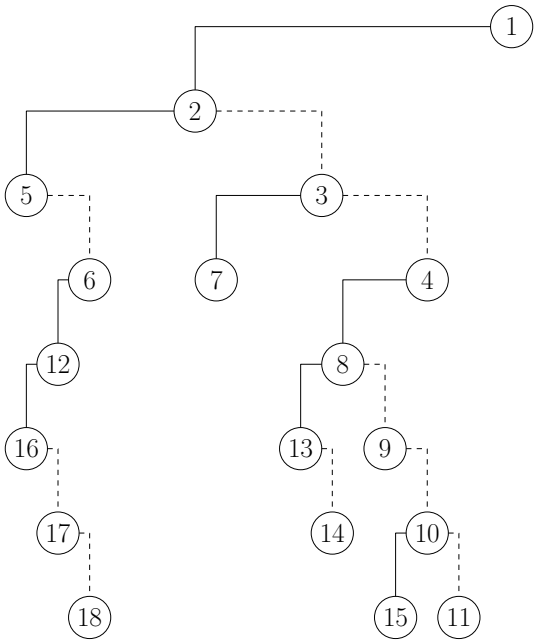
— et chaque nœud, sauf la racine, appartient à l'une de ces listes chaînées. Dans cette liste, la cellule contenant le nœud contient également un lien vers le « frère droit » du nœud (qui est dans la cellule suivante de la liste).



Un arbre d'arité quelconque.



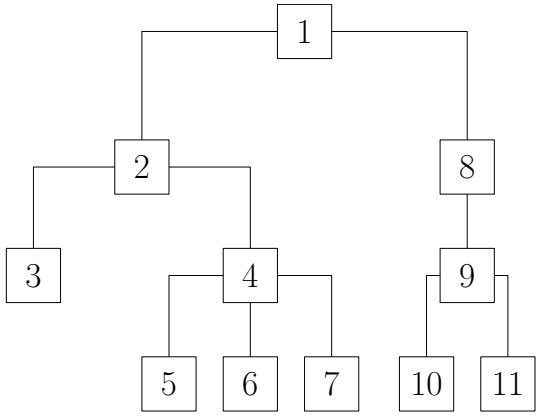
Représentation avec liste chaînée d'enfants.



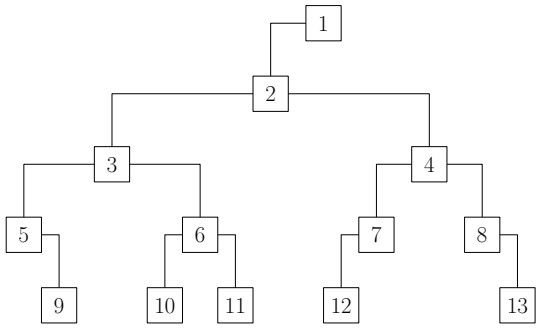
Transformation en arbre binaire.

Exercices 6

⇨ 1. Appliquer la transformation décrite ci-dessus à l'arbre suivant :



2. Appliquer la transformation inverse à l'arbre suivant :



⇨ On considère les deux types suivants :

— le type `'a arbre` représente un arbre d'arité quelconque (une feuille correspond à un nœud de la forme `Nn (x, [])`) :

```
type 'a arbre =
  | Nn of 'a * 'a arbre list
```

— le type `'a binaire` correspond à un arbre binaire (non strict) :

```
type 'a binaire =
  | V
  | N of 'a * 'a binaire * 'a binaire
```

1. Écrire une fonction `vers_binaire` réalisant la transformation de la partie dans le sens « arité quelconque vers binaire ».

2. Écrire une fonction `vers_naire` réalisant la transformation inverse.

Solution. 1.

```
let vers_binaire (arbre : 'a arbre) : 'a binaire =
  let rec aux (courant : 'a arbre) (freres : 'a arbre liste)
      : 'a binaire =
    match courant, freres with
    | Nn (x, []), [] -> N (x, V, V)
    | Nn (x, e :: es), [] -> N (x, aux e es, V)
    | Nn (x, []), f :: fs -> N (x, V, aux f fs)
    | Nn (x, e :: es), f :: fs -> N (x, aux e es, aux f fs) in
  aux arbre []
```

2.

```
let vers_naire (arbre : 'a binaire) : 'a arbre =
  let rec vers_liste = function
    | V -> []
    | N (x, g, d) -> Nn (x, vers_liste g) :: vers_liste d in
  match vers_liste arbre with
  | [arbre_naire] -> arbre_naire
  | _ -> failwith "argument invalide"
```

□