Cours d'informatique

Spéciales MP et MP*
Lycée Thiers Marseille

Y.Lemaire

9 mai 2000

Table des matières

Liste des algorithmes

Liste des programmes

Notations

$\blacksquare$	« Ce Qu'il Fallait Démontrer ».
$\wedge$, $\vee$, $\neg$	*et* , *ou* , *non*.
$\lfloor x \rfloor$	*Partie entière* du nombre réel x = le plus grand entier $\leqslant$ à x.
$\lceil x \rceil$	*Partie entière supérieure* du nombre réel x = le plus petit entier $\geqslant$ à x.
N, **Z**, **R**, etc.	L'ensemble des *entiers naturels*, des *entiers relatifs*, des *nombres réels*, etc.
$[\vert p,q \vert]$	L'ensemble des entiers n tels que $p \leqslant n \leqslant q$.
$u_n = \mathrm{O}(v_n)$	La suite (u_n) est *dominée* par (v_n) : $\exists M,\exists N : \forall n, n \geqslant N \Rightarrow \vert u_n \vert \leqslant M \vert v_n \vert$
$u_n = \mathrm{o}(v_n)$	(u_n) est *négligeable* devant (v_n) : $\exists (\varepsilon_n)_n : \varepsilon_n \to 0$ et $\forall n, u_n = \varepsilon_n v_n$.
$u_n = \Theta(v_n)$	(u_n) est un « *théta* » de (v_n) : $u_n = \mathrm{O}(v_n)$ et $v_n = \mathrm{O}(u_n)$.
$u_n \sim v_n$	(u_n) est *équivalente* à (v_n) : $u_n = v_n + \mathrm{o}(v_n)$.
$\vert u \vert$	La longueur du mot u.
$\vert \mathcal{A} \vert$	Le langage reconnu par l'automate $\mathcal{A}$.
$\vert E \vert$	Le nombre d'éléments de l'ensemble fini E.
$\mathcal{P}(E)$	L'ensemble des parties de E.
$E \setminus F$	L'ensemble des éléments de E qui n'appartiennent pas à F.
E^F	L'ensemble des applications de F dans E.

Chapitre 1

Preuve et évaluation d'un programme

Ce chapitre est dédié à la mise au point de programmes simples. La première partie fournit les principes permettant de prouver la validité d'un programme. On donne ensuite des exemples de calculs de complexités de programmes. Au passage, on revoit des algorithmes étudiés en première année.

Les algorithmes seront parfois décrits dans un pseudo-langage proche de CAML : **soit**, **dans**, **et**, **tant que**, **pour**, **faire**, etc. devront être traduits en CAML respectivement par `let`, `in`, `and`, `while`, `for`, `do`, etc.

Pour affecter une valeur a à une variable x, on écrira $x \leftarrow a$.

1.1 Preuve de validité d'un programme

1.1.1 Objets d'un programme

Les *objets* définis dans un programme sont essentiellement de deux sortes.

- les objets non fonctionnels : entiers, booléens, réels, tableaux, listes, arbres, etc. qui sont éventuellement variables;
- les objets fonctionnels : fonctions.

1.1.2 Spécifications d'un bloc d'instructions

Le *contexte* d'un bloc de programme est l'ensemble des objets auxquels ce bloc peut accéder; soit pour utiliser les valeurs de ces objets, soit, éventuellement, pour les modifier.

Soit un bloc d'instructions P de contexte E et soient deux propriétés C_1 et C_2 portant sur les objets de E. On dit que P satisfait la *précondition* C_1 et la *postcondition* C_2 quand la propriété suivante est vérifiée :

Si le contexte E vérifie C_1, alors, aprés éxécution des instructions du bloc P, il vérifie C_2.

On peut schématiser cette propriété par :

$$C_1 \xrightarrow{P} C_2$$

Une propriété I qui vérifie $I \xrightarrow{P} I$ est appelée un *invariant* de P.

1.1.3 Spécifications d'une fonction

On appelle *spécifications* d'une fonction

soit $f(x) =$<expression>

la donnée

- du type T_1 de l'argument x;
- du type T_2 de la valeur renvoyée;
- d'une condition C_1, appelée *précondition*, portant sur x et, éventuellement, sur les objets du contexte de la fonction;
- d'une condition C_2, appelée *postcondition*, portant sur la valeur renvoyée et, éventuellement, sur les objets du contexte de la fonction.

On dit que la fonction f *satisfait* ces spécifications si, pour tout x, de type T_1, vérifiant la condition C_1, un appel de $f(x)$ renvoie une valeur de type T_2 vérifiant la condition C_2.

1.1.4 Affectation

On a

$$C(a) \xrightarrow{x \leftarrow a} C(x)$$

Par exemple, si x est une variable entière et si c est une valeur entière, on a

$$(c \geqslant 1) \xrightarrow{x \leftarrow c+1} (x \geqslant 2)$$

1.1.5 Séquence

Soient P et Q deux blocs d'instructions. Notons $P\,;Q$ ou simplement $P\ Q$, le bloc obtenu en ajoutant les instructions du bloc Q à la suite de celles de P :

$$\text{Si } C_1 \xrightarrow{P} C_2 \text{ et } C_2 \xrightarrow{Q} C_3, \text{ alors } C_1 \xrightarrow{P\ Q} C_3$$

1.1.6 Répétition (boucle *for*)

Considérons l'instruction :

$$\textbf{pour } k = p \textbf{ à } q \textbf{ faire } \text{action}(k)$$

(l'instruction, ou le bloc d'instructions, action(k) est appelé le *corps* de la boucle).

Soit une propriété $I(k)$ qui soit un invariant de cette boucle; c'est à dire que, pour tout $k \in [|p,q|]$, action(k) satisfait la précondition $I(k-1)$ et la postcondition $I(k)$. Alors la boucle satisfait la précondition $I(p-1)$ et la postcondition $I(q)$. Autrement dit

$$\forall k, I(k-1) \xrightarrow{\text{action}(k)} I(k) \Rightarrow I(p-1) \xrightarrow{\textbf{pour } k=p \textbf{ à } q \textbf{ faire } \text{action}(k)} I(q)$$

Cela peut être considéré comme une conséquence de la propriété des séquences car la boucle peut aussi s'écrire :

$$\text{action}(p)\,;\,\text{action}(p+1)\,;\,\ldots\,;\,\text{action}(q),$$

et on a : $I(p-1) \xrightarrow{\text{action}(p)} I(p) \xrightarrow{\text{action}(p+1)} I(p+1) \cdots \xrightarrow{\text{action}(q)} I(q)$.

Ainsi, pour montrer que la propriété $I(q)$ est vérifiée après l'éxécution de la boucle, il suffit de montrer

- que $I(p-1)$ est vérifiée avant l'éxécution de la boucle;
- et que $I(k)$ est un invariant de la boucle.

Exemple 1.1 *Exponentiation lente*

Montrons que la fonction `puiss : int -> int -> int` suivante

```
let puiss x n = let y = ref 1 in for k = 1 to n do y := !y * x done; !y;;
```

est telle que `puiss` x n renvoie x^n :

La propriété $y = x^k$ est un invariant de la boucle car $(y = x^{k-1}) \xrightarrow{y \leftarrow y \times x} (y = x^k)$. De plus, avant l'éxécution de la boucle, on a $y = 1 = x^0$; donc, après la boucle, on a $y = x^n$.

Dans la pratique, pour démontrer la validité d'un bloc d'instructions, il est préférable, autant que possible, de faire apparaître la preuve dans les commentaires (entre `(* *)`) du bloc, comme dans le programme 1.1.

Programme 1.1 Exponentiation lente

```
let puiss x n = (* int -> int -> int, puiss x n = x^n *)
   let y = ref 1 in   (* y = x^0 *)
   for k = 1 to n do  (* y = x^(k-1) *)
      y := !y * x     (* y = x^k *)
   done;              (* y = x^n *)
   !y;;
```

Dans cet exemple, les commentaires sont d'ailleurs trop nombreux. Le seul commentaire indispensable est `(* y = x^(k-1) *)` qui permet d'en déduire les autres.

COMPLEXITÉ : Un appel de `puiss` x n effectue n multiplications.

Il faut bien comprendre que l'obligation de prouver, en introduisant le bon invariant de boucle, qu'une boucle effectue bien le travail qu'on lui demande n'est pas une contrainte; c'est aussi une aide à la mise au point de cette boucle. Ce point de vue est expliqué dans l'exemple suivant.

Exemple 1.2 *Evaluation d'un polynome*

$P(X) = \sum_{i=0}^{n} p_i X^i \in \mathbf{R}[X]$ et $x \in \mathbf{R}$ étant donnés, on se propose de calculer $P(x) = \sum_{i=0}^{n} p_i x^i$.
On introduit une variable réelle y et on essaie d'écrire une boucle

pour $k = 1$ **à** n **faire** action(k)

admettant la propriété $(y = \sum_{i=0}^{k} p_i x^i)$ pour invariant, de manière à récupérer à la sortie de la boucle la valeur cherchée dans y.
Pour cela, il suffit de prendre l'instruction : $y \leftarrow y + p_k x^k$ pour action(k). Mais cette instruction est coûteuse car elle nécessite le calcul de x^k et elle ne tient pas compte du fait que x^{k-1} a été calculé à l'étape précédente. Il est préférable d'introduire une deuxième variable z et d'imposer l'invariant

$$I(k) : (z = x^k \wedge y = \textstyle\sum_{i=0}^{k} p_i x^i).$$

action(k) devient alors : $z \leftarrow x \times z$; $y \leftarrow y + p_k z$.
Pour que $I(n)$ soit vérifiée à la sortie de la boucle, il suffit donc que $I(0)$ soit vérifié à l'entrée : on écrit avant la boucle les instructions d'initialisation : $z \leftarrow 1$; $y \leftarrow p_0$.

En CAML, cela donne le programme 1.2.

Programme 1.2 Evaluation d'un polynome

```
let  eval_polynome p x =    (* float vect -> float -> float *)
(* eval_polynome p x renvoie   pn x^n + ... + p0 où n = longueur(p) - 1 *)
   let z = ref 1.
   and y = ref p.(0) in
   for k = 1 to vect_length p - 1 do
      z := x *. !z;
      y := !y +. p.(k) *. z
      (* z = x^k et y = pk x^k + ... + p0 *)
   done;
   !y;;
```

COMPLEXITÉ : Un appel de `eval_polynome` p x nécessite $2n$ multiplications et n additions.

La méthode du programme 1.3 (HÖRNER) est plus efficace : n multiplications et n additions.

1.1.7 Répétition (boucle *while*)

Considérons l'instruction :

tant que C **faire** action

(action est appelé le *corps* de la boucle).

Programme 1.3 Méthode de Hörner

```
let eval_polynome_Horner p x =
    let n = vect_length p - 1 in
    let y = ref p.(n) in
    for k = n-1 downto 0 do
        y := !y *. x +. p.(k)
        (*  y = pn x^(n-k) + ... + pk  *)
    done;
    !y;;
```

Correction partielle : Soit une propriété I qui soit un *invariant* de cette boucle, dans le sens où

$$I \wedge C \xrightarrow{\text{action}} I$$

alors, en supposant que la boucle *termine* (ne soit pas une boucle infinie), on a

$$I \xrightarrow{\textbf{tant que } C \textbf{ faire} \text{ action}} I \wedge \neg C$$

En d'autres termes, pour montrer qu'une propriété P est vérifiée à la fin de l'éxécution de la boucle (en supposant qu'elle termine), il suffit de trouver une propriété I telle que

- I est vérifiée à l'entrée dans la boucle;
- $I \wedge C \xrightarrow{\text{action}} I$ (I est un invariant);
- $I \wedge \neg C \Rightarrow P$.

Cela se comprend aisément : la propriété I est constamment vérifiée (avant et après chaque éxécution du corps de la boucle). En particulier, elle est vérifiée à la fin de l'éxécution de la boucle. De plus la propriété $\neg C$ est aussi vérifiée après la boucle car c'est la condition de sortie.

Arrêt : Pour prouver l'arrêt de la boucle, il suffit de trouver une fonction t (ou *taille*) du contexte à valeurs dans l'ensemble $\mathbf{N}$ des entiers naturels telle que toute éxécution du corps de la boucle fasse strictement décroître t; autrement dit, $t = k \xrightarrow{\text{action}} t < k$.

En effet, si la boucle était infinie, soit t_p la valeur de t après p éxécutions du corps de la boucle; la suite $(t_p)_{p\in\mathbf{N}}$ serait une suite strictement décroissante d'entiers naturels, ce qui ne se peut pas[1].

Exemple 1.3 *Exponentiation rapide*

Soit à calculer x^n avec $x \in \mathbf{Z}$ (mais on pourrait supposer que x appartient à un monoïde quelconque) et $n \in \mathbf{N}$. Dans le programme 1.1, effectuons le changement d'indice $p = n-k+1$ et transformons la boucle *for* en une boucle *while*. On obtient le programme 1.4.

Programme 1.4 Exponentiation lente avec boucle *while*

```
let puiss' x n =
   let y = ref 1 and p = ref n in
   while !p > 0 do
      (* invariant : y = x^(n-p) *)
      y := !y * x;
      p := !p - 1
   done;
   !y;;
```

Preuve d'arrêt : La variable entière p reste > 0 et décroît strictement dans le corps de la boucle.

Preuve de correction partielle : L'invariant étant donné en commentaire dans l'algorithme, la preuve de correction partielle consiste seulement à vérifier que

- la propriété $I : y = x^{n-p}$ est bien un invariant de la boucle : $x^{n-p} \times x = x^{n-(p-1)}$;

1. Cet argument montre que, plus généralement, on peut prendre une fonction taille à valeurs dans un ensemble muni d'un *bon ordre* (relation d'ordre pour laquelle toute partie non vide admet un plus petit élément).

- I est vérifiée à l'entrée : $1 = x^{n-n}$;
- si I est vérifiée et si, de plus $(p > 0)$ n'est pas vérifiée, alors $y = x^n : p = 0$.

En règle générale, il suffit de préciser l'invariant qui sert à faire la preuve de correction partielle car la preuve elle-même ne consiste qu'en vérifications.

L'inefficacité du programme 1.4 (nombre de multiplications $= n$) provient du fait que, pour maintenir l'invariant $y = x^{n-p}$, qui peut aussi s'écrire : $yx^p = x^n$, la variable p ne décroît que d'une unité à chaque éxécution du corps de la boucle. Introduisons une nouvelle variable z et tâchons de maintenir l'invariant $J : yz^p = x^n$ tout en diminuant p de motié :

- si p est pair, $yz^p = y(z^2)^{p/2}$ donc $J \xrightarrow{z \leftarrow z^2 \ ; \ p \leftarrow p/2} J$;
- si p est impair, $yz^p = yz(z^2)^{(p-1)/2}$ donc $J \xrightarrow{y \leftarrow yz \ ; \ z \leftarrow z^2 \ ; \ p \leftarrow (p-1)/2} J$.

Cela montre la correction partielle de la boucle du programme 1.5.

Programme 1.5 Exponentiation rapide

```
let puiss_rapide x n =
   let y = ref 1 and z = ref x and p = ref n in
   while !p > 0 do
      (* Invariant : y z^p = x^n *)
      if !p mod 2 = 1 then y := !y * !z;
      z := !z * !z;
      p := !p / 2
   done;
   !y;;
```

PREUVE D'ARRÊT : $p \in \mathbf{N}$ décroît strictement dans le corps de la boucle.

COMPLEXITÉ : Le nombre T_n de multiplications est majoré par $2U_n$ où U_n est le nombre d'éxécutions du corps de la boucle. U_n est aussi le nombre d'éxécutions du corps de la boucle simplifiée : `while !p > 0 do p := !p / 2 done`. On voit alors que $U_n = 1 + U_{\lfloor n/2 \rfloor}$, ce qui montre — voir le théorème 1.3 page 20 — que $U_n = \mathrm{O}(\log n)$ et $T_n = \mathrm{O}(\log n)$.

1.1.8 Fonction récursive

Soit une fonction récursive

soit récursivement $f(x)$ =<expression(x)> .

<expression(x)> contient donc des appels récursifs de la fonction f.

On désire prouver que f satisfait des spécifications données : Pour tout argument x correct (de type T_1, vérifiant la condition C_1), un appel de $f(x)$ est correct (renvoie une valeur de type T_2 vérifiant la condition C_2).

Pour cela, il faut faire la *preuve de correction partielle* et la *preuve d'arrêt*.

CORRECTION PARTIELLE : Il faut montrer que, pour tout argument correct x,

- les arguments des appels récursifs de f dans <expression(x)> sont corrects;
- l'hypothèse que les appels récursifs de f dans <expression(x)> sont corrects, entraîne que l'appel de $f(x)$ est correct.

ARRÊT : Pour prouver qu'un appel de f n'engendre pas une infinité d'appels récursifs, il suffit de trouver une fonction t (ou *taille*) de x (et éventuellement du contexte) à valeurs dans $\mathbf{N}$ telle que, pour tout argument correct x, les appels récursifs de f dans <expression(x)> ont des arguments de taille strictement inférieure à celle de x.

Exemple 1.4 *Factorielle*

Montrons que la fonction

soit récursivement $f(n) =$ **si** $n = 0$ **alors** 1 **sinon** $n \times f(n-1)$

satisfait la spécification :

- f est de type entier $\rightarrow$ entier;

- précondition : $n \geqslant 0$ [1];
- postcondition : $f(n) = n!$.

PREUVE DE CORRECTION PARTIELLE : Elle se déduit des propriétés $0! = 1$ et $\forall n \in \mathbf{N}^*, n! = n \times (n-1)!$:
Soit n un argument correct ($n \in \mathbf{N}$) :

- si $n = 0$, alors l'expression <e> : **si** $n = 0$ **alors** 1 **sinon** $n \times f(n-1)$ s'évalue en $1 = n!$ (sans appel récursif);
- si $n \geqslant 1$, <e> est équivalente à $n \times f(n-1)$; l'argument $n-1$ de l'appel récursif est correct ($n - 1 \in \mathbf{N}$) et, avec l'hypothèse que $f(n-1)$ renvoie bien $(n-1)!$, <e> s'évalue en $n \times (n-1)! = n!$.

PREUVE D'ARRÊT : Si l'appel $f(n)$ contient un appel récursif $f(n-1)$, l'argument $n-1$ de l'appel récursif est $<$ à celui, n, de l'appel principal.

COMPLEXITÉ : Le nombre T_n de multiplications effectuées par l'appel de $f(n)$ vérifie les propriétés : $T_0 = 0$ et $T_n = T_{n-1} + 1$; d'où $T_n = n$.

En pratique, pour prouver qu'une fonction récursive satisfait une spécification, il faut :

- donner cette spécification avec précision;
- fournir les propriétés qui permettent de faire la preuve de correction partielle (ici : $0! = 1$ et $\forall n \in \mathbf{N}^*, n! = n \times (n-1)!$); il est en général inutile de rédiger cette preuve comme on l'a fait ici;
- faire la preuve d'arrêt.

Exemple 1.5 *Exponentiation rapide*

Pour $x \in \mathbf{Z}$ et $n \in \mathbf{N}^*$, $x^n = (x^{n/2})^2$ si n est pair et $(x^{(n-1)/2})^2 x$ sinon; ce qui prouve la correction partielle de la sous fonction récursive p du programme 1.6.

Programme 1.6 Exponentiation rapide (version récursive)

```
let puiss_rapide_récursive x =
(* int -> int -> int. si n >= 0,  puiss_rapide_récursive  x n renvoie x^n *)
   let rec p = function
   (* int -> int.  si n > 0, p n renvoie x^n *)
      0 -> 1 |
      n -> let  y = p (n / 2) in
           if n mod 2 = 0 then  y * y else y * y * x in
   p;;
```

PREUVE D'ARRÊT : Supposons qu'un appel de $p\ n$ donne lieu au sous appel récursif $p(n/2)$. Alors n est non nul donc l'argument $n/2$ de p dans le sous appel est strictement inférieur à l'argument n de p dans l'appel principal.

COMPLEXITÉ : Le nombre T_n de multiplications effectuées par un appel de $p\ n$ vérifie $T_n \leqslant 2 + T_{\lfloor n/2 \rfloor}$ donc — voir le théorème 1.3 page 20 — $T_n = O(\log n)$.

1.2 Complexité d'un programme

La *complexité temporelle* d'un algorithme $\mathcal{A}$ est le nombre $c(\mathcal{A})$ d'opérations élémentaires effectuées durant l'éxécution de $\mathcal{A}$.

Mais cela suppose que soit définie la notion d'*opération élémentaire*.

Dans le cas d'un algorithme de tri par comparaisons, par exemple, on peut convenir qu'une opération élémentaire est une comparaison de deux éléments.

En l'absence de toute précision sur les opérations élémentaires, on entend par opération élémentaire toute opération réellement élémentaire effectuée par le processeur d'une machine

1. En réalité, pour prendre en compte le fait que les entiers CAML sont $<$ à 2^{30}, il faudrait prendre pour précondition la propriété : $0 \leqslant n \leqslant 12$.

donnée. La complexité est alors à peu de choses près proportionnelle au temps d'éxécution de $\mathcal{A}$ et on peut admettre que

- la complexité d'une opération arithmétique sur des entiers ou des réels est constante [1];
- la complexité d'une affectation est constante;
- la complexité d'une séquence $P\,;Q$ est la somme $c(P)+c(Q)$ des complexités de P et de Q;
- la complexité d'une boucle **pour** $i = 1$ **à** n **faire** action(i) est la somme $Cn + \sum_{i=1}^{n} c(\text{action}(i))$ d'une quantité proportionnelle à n ($C > 0$) et de la somme des complexités des action(i) [2];
- etc.

Considérons maintenant un algorithme $\mathcal{A}(x)$ qui dépend d'une *donnée* x. On suppose définie une application qui, à toute donnée x associe un élément $t(x)$ d'un ensemble N. Par exemple, si x est un entier, on a généralement $t(x) = |x|$ [3]; si x est un tableau, on prend pour $t(x)$ la longueur de x, etc.

Pour toute valeur $\nu \in N$, le nombre $T_\nu = \max\{c(\mathcal{A}(x)) \,|\, t(x) = \nu\}$ est appelé la *complexité dans le pire des cas* de l'algorithme [4].

Si $N = \mathbf{N}$ ou $\mathbf{R}^+$, on s'intéresse généralement au comportement asymptotique de T_ν quand $\nu \to +\infty$.

La *complexité spatiale* d'un algorithme est une mesure de l'espace mémoire utilisé par l'algorithme en plus de l'espace mémoire nécessaire pour stocker les données.

On peut encore définir la notion de complexité spatiale dans le pire des cas.

1.2.1 Récurrences « affines »

Soient $a \in \mathbf{R}^*$ et $(u_n)_{n>p} \in \mathbf{R}^{[\![p+1,+\infty[\![}$. On définit une suite $(T_n)_{n\geqslant p}$ par la donnée de T_p et de la relation de récurrence

$$\forall n > p, T_n = aT_{n-1} + u_n$$

Pour exprimer T_n, on introduit la suite $S_n = \frac{T_n}{a^n}$ qui vérifie la relation $\forall n > p, S_n = S_{n-1} + \frac{u_n}{a^n}$, d'où l'on tire $S_n = S_p + \sum_{k=p+1}^{n} \frac{u_k}{a^k}$, puis

$$\forall n \geqslant p, T_n = a^n \left(\frac{T_p}{a^p} + \sum_{k=p+1}^{n} \frac{u_k}{a^k} \right)$$

Exemple 1.6 *Tri par insertion d'une liste d'entiers*

Pour écrire la fonction `tri_insertion : int list -> int list` du programme 1.7 on écrit d'abord une fonction `insérer : int -> int list -> int list` telle que, si ℓ est une liste triée et si x est un entier, `insérer` x ℓ renvoie la liste obtenue en insérant x à sa place dans ℓ.

Proposition 1.1 *Le nombre maximum de comparaisons effectuées par le tri par insertion sur une liste de longueur n est $\frac{n(n-1)}{2} = \Theta(n^2)$.*

Preuve Soit T_n (resp. u_n) la complexité (au sens du nombre de comparaisons) maximale de `tri_insertion` (resp. `insérer`) sur une liste de longueur n.
On a $u_0 = 0$ et $u_n \leqslant u_{n-1} + 1$ donc $u_n \leqslant n$.
On a $T_0 = 0$ et $T_n \leqslant T_{n-1} + u_{n-1} = T_{n-1} + n - 1$ donc $T_n \leqslant \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.
On a exactement $T_n = \frac{n(n-1)}{2}$. En effet, soit t_n la complexité de `tri_insertion` sur la liste $L_n = [n; n-1; \ldots; 2; 1]$ (triée dans l'odre inverse). Le tri de L_n est constitué du tri de L_{n-1},

---

1. Voir quand même la section 1.2.4 où l'on tient compte de la taille des arguments.
2. La boucle `for i = 1 to 100 do done;;` a donc une complexité $100C$ non nulle.
3. En fait il serait préférable de choisir $t(x) = \log_2 |x|$ de manière que $t(x)$ soit une mesure de la taille de x (la quantité de mémoire occupée par x).
4. $\min\{c(\mathcal{A}(x)) \,|\, t(x) = \nu\}$ est la *complexité dans le meilleur cas* et, si une distribution de probabilité est définie sur $\{x \,|\, t(x) = \nu\}$, on peut aussi définir la *complexité en moyenne* comme la moyenne des $c(\mathcal{A}(x))$ pour $t(x) = \nu$.

Programme 1.7 Tri par insertion d'une liste d'entiers

```
let rec tri_insertion =
   let rec insérer a = function
      [] -> [a] |
      (b :: l) as l' -> if a <= b then a :: l' else b :: insérer a l  in
   function
      [] -> [] |
      a :: l -> insérer a (tri_insertion l);;
```

ce qui nécessite t_{n-1} comparaisons, puis de l'insertion de n dans la liste $[1;2;\dots;n-1]$ (la liste L_{n-1} une fois triée), ce qui nécessite $n-1$ comparaisons. On a donc $t_n = t_{n-1} + n - 1$ et, comme $t_0 = t_1 = 0$, $t_n = \frac{n(n-1)}{2}$. ∎

Proposition 1.2 *Si une suite $(T_n)_{n \geqslant p}$ vérifie la relation de récurrence*

$$\forall n > p, T_n = aT_{n-1} + Kb^n$$

où a,b,$K \in \mathbf{R}_+^*$, alors
- *si $b < a$, $T_n = \Theta(a^n)$;*
- *si $b = a$, $T_n = \Theta(na^n)$;*
- *si $b > a$, $T_n = \Theta(b^n)$.*

Preuve $T_n = a^n \left(\frac{1}{a^p}T_p + K\sum_{k=p+1}^{n}\left(\frac{b}{a}\right)^k\right)$ donc, si $b = a$, $T_n \sim Kna^n$ et si $b \neq a$, $T_n = a^n\left(\frac{1}{a^p}T_p + K\left(\frac{b}{a}\right)^{p+1}\frac{1-(b/a)^{n-p}}{1-b/a}\right)$ est un $\Theta(a^n)$ ou un $\Theta(b^n)$ suivant que $b < a$ ou $b > a$. ∎

Exemple 1.7 *Produit de deux polynomes*

Soit à multiplier deux polynomes P et Q de degrés $< n = 2^p$. La méthode consistant à calculer les coefficients de PQ par application directe des formules définissant ce produit nécessite de l'ordre de n^2 opérations. On décrit ici une méthode plus rapide. On peut écrire $P = X^{n/2}A + B$ et $Q = X^{n/2}C + D$ avec A, B, C et D de degrés $< n/2 = 2^{p-1}$. On peut alors calculer $PQ = X^nAC + X^{n/2}(AD+BC) + BD$ en n'effectuant que trois multiplications et quatre additions de polynomes de degrés $< 2^{p-1}$. Il suffit en effet de calculer successivement $U = AC$, $V = BD$ et $AD + BC = (A+B)(C+D) - U - V$. Il s'ensuit un algorithme récursif de calcul de PQ dont la complexité T_p vérifie $T_p = 3T_{p-1} + \mathrm{O}(2^p)$, soit, d'après la proposition 1.2, $T_p = \mathrm{O}(3^p) = \mathrm{O}\left(n^{\log_2 3}\right) = \mathrm{O}(n^{1,59})$.

Remarque 1.1 Un principe analogue au précédent permet de ramener le calcul du produit de deux matrices $n \times n$ avec $n = 2^p$ au calcul de 7 produits de matrices $n/2 \times n/2$ et de $\mathrm{O}(4^p)$ additions. Cela donne l'algorithme de STRASSEN de complexité $T_p = 7T_{p-1} + \mathrm{O}(4^p) = \mathrm{O}(7^p) = \mathrm{O}\left(n^{\log_2 7}\right) = \mathrm{O}(n^{2,81})$.

1.2.2 Récurrences « diviser pour régner »

Théorème 1.3 *Soient*
- *b un entier naturel $\geqslant 2$;*
- *deux suites $a_1(n)$ et $a_2(n)$ de réels $\geqslant 0$ telles que $a_1(n) + a_2(n) \leqslant a$ où a est une constante $\geqslant 1$;*
- *$(f(n))_n$ une suite $\geqslant 0$ telle que $f(n) = O(n^c)$ avec $c \in \mathbf{R}^+$.*

Si une suite $(T_n)_{n \geqslant p}$ de réels $\geqslant 0$ vérifie la relation

$$(1) \quad \forall n \geqslant q, T_n = a_1(n)T_{\lfloor n/b \rfloor} + a_2(n)T_{\lceil n/b \rceil} + f(n)$$

alors $T_n = O(g(n))$ où, avec $\omega = \log_b a$, $g(n) = \begin{cases} n^\omega & \text{si } c < \omega \\ n^\omega \log n & \text{si } c = \omega \\ n^c & \text{si } c > \omega \end{cases}$.

Preuve On peut supposer que q est assez grand pour que la suite $(T_n)_{n \geqslant p}$ soit définie par la donnée de $T_p, \dots, T_{q-1}$ et de la relation de récurrence (1).[1]

1. $q \geqslant \max(2,pb)$ de sorte que $n \geqslant q \Rightarrow p \leqslant \lfloor n/b \rfloor \leqslant \lceil n/b \rceil < n$.

Soit $K \geqslant 0$ vérifiant $\forall n, f(n) \leqslant Kn^c$. On définit une suite $(U_n)_{n \geqslant p}$ par les conditions $U_p = U_{p+1} = \cdots = U_{q-1} = \max\{T_i \mid i = p, \ldots q-1\}$ et $\forall n \geqslant q, U_n = aU_{\lceil n/b \rceil} + Kn^c$.
La suite (U_n) est croissante (vérifier par récurrence que $\forall n > p, U_n - Un - 1 \geqslant 0$).
$\forall n \geqslant p, T_n \leqslant U_n$ (par récurrence aussi en utilisant la croissance de (U_i)).
La suite $u_k = U_{b^k}$ vérifie $u_k = au_{k-1} + Kb^{kc}$ pour k assez grand. En appliquant la proposition 1.2, on obtient $U_{b^k} = \mathrm{O}(g(b^k))$. De plus on peut vérifier que $g(b^k) = \mathrm{O}(g(b^{k-1}))$. Soit donc K' tel que $\forall k, U_{b^k} \leqslant K'g(b^{k-1})$. Pour tout n, soit l'entier k défini par $b^{k-1} \leqslant n < b^k$; comme (U_n) et $(g(n))$ sont croissantes, on peut écrire $T_n \leqslant U_n \leqslant U_{b^k} \leqslant K'g(b^{k-1}) \leqslant K'g(n)$. ∎

Remarque 1.2 Dans le théorème précédent, on peut remplacer la relation (1) par la relation $T_n = \sum_{i=p}^{\lceil n/b \rceil} a_{i,n} T_i$ où les $a_{i,n}$ sont des réels $\geqslant 0$ tels que $\sum_{i=p}^{\lceil n/b \rceil} a_{i,n} \leqslant a$ avec $a \geqslant 1$; la démonstration s'applique sans changement.

Exemple 1.8 *Recherche dichotomique dans un tableau trié*
On se propose d'écrire une fonction
`recherche : ('a -> 'b) -> 'b -> 'a vect -> 'a`
spécifiée comme suit :
PRÉCONDITIONS : t `: 'a vect` est un tableau trié dans l'ordre croissant des clés de ses éléments; les clés étant fournies par une fonction *clé* `: 'a -> 'b`[1]; autrement dit, $clé(t_0) \leqslant clé(t_1) \leqslant \cdots \leqslant clé(t_{n-1})$ où n est la longueur de t. c `: 'b` est une clé possible.
POSTCONDITIONS : `recherche` *clé c t* renvoie l'unique indice $i \in [|-1, n-1|]$ tel que $clé(t_i) \leqslant c < clé(t_{i+1})$ (on convient de donner à $clé(t_0)$ (resp. $clé(t_n)$) une valeur < (resp. >) à toutes les clés possibles)[2].

Le programme 1.8 propose deux versions de la fonction `recherche`.

Programme 1.8 Recherche dans un tableau trié

```
let recherche_itérative clé c t =
   let u = ref (-1) and v = ref (vect_length t) in
   while !v <> !u + 1 do
   (* invariant : -1 <= u < v <= n et clé(t(u)) <= c < clé(t(v)) *)
   (* preuve d'arrêt : v - u décroît dans le corps de la boucle  *)
      let w = (!u + !v) / 2 in
      if c < clé t.(w) then v := w else u := w
   done;
   !u;;

let recherche_récursive clé c t =
  let rec rech u v =
  (* rech : int -> int -> int                                          *)
  (* précondition  -1 <= u < v <= n et clé(t(u)) <= c < clé(t(v))      *)
  (* postcondition rech u v renvoie i tel que clé(t(i)) <= c < clé(t(i+1))*)
  (* preuve d'arrêt : v - u diminue dans l'éventuel appel récursif     *)
    if v = u + 1
      then u
      else let w = (u + v) / 2 in
           if c < clé t.(w) then rech u w else rech w v in
  rech (-1) (vect_length t);;
```

Notons T_n le maximum du coût d'éxécution de la boucle de `recherche_itérative` quand, au départ, les variables u et v vérifient $v - u \leqslant n$. Soient u et v tels que $v - u = n$. Après une éxécution du corps de la boucle, on a $v - u \leqslant \lceil n/2 \rceil$. On en déduit que $T_n \leqslant T_{\lceil n/2 \rceil} + \mathrm{O}(1)$ puis (théorème 1.3) $T_n = \mathrm{O}(\log n)$.

La complexité T'_n de la sous fonction `rech` de la fonction `recherche_récursive` vérifie aussi $T'_n \leqslant T'_{\lceil n/2 \rceil} + \mathrm{O}(1)$ donc $T'_n = \mathrm{O}(\log n)$.[3]

1. Rappelons qu'en CAML, tout type (ici `'b`) est muni d'un ordre total (voir p. 96).
2. La fonction renvoie -1 si $c < clé(t_0)$ et $n-1$ si $clé(t_{n-1}) \leqslant c$.
3. On peut d'ailleurs déduire la version itérative de la version récursive par suppression de la récursivité terminale.

Exemple 1.9 *Tri fusion*

On étudie la complexité T_n de l'algorithme 1.1.

Algorithme 1.1 Tri fusion d'un tableau de n éléments

1 **soit récursivement** trier x =
2   **soit** u = le sous tableau des $\lfloor n/2 \rfloor$ premiers éléments de x
3   **et** v = le sous tableau des $\lceil n/2 \rceil$ derniers éléments de x **dans**
4   trier u
5   trier v
6   fusionner les deux tableaux triés u et v

Les coûts des tris des sous tableaux u et v (lignes 4 et 5) sont respectivement $T_{\lfloor n/2 \rfloor}$ et $T_{\lceil n/2 \rceil}$ et le coût de la fusion (ligne 6) est $\mathrm{O}(n)$; d'où la récurrence $T_n = T_{\lfloor n/2 \rfloor} + T_{\lceil n/2 \rceil} + \mathrm{O}(n)$ et (théorème 1.3) $T_n = \mathrm{O}(n \log n)$.

1.2.3 Un autre exemple de récurrence : le tri rapide

Le programme 1.9 fournit une fonction `tri_rapide : int vect -> unit` de tri d'un tableau d'entiers. Pour trier le tableau $x = (x_0, x_1, \ldots, x_{n-1})$ on écrit une procédure récursive `tri : int -> int -> unit` telle que, si $0 \leqslant p,q \leqslant n-1$, l'appel `tri` p q a pour effet de trier le sous tableau $(x_p, x_{p+1}, \ldots, x_q)$ de x.[1] Il suffira alors d'appeler `tri` 0 $(n-1)$ pour trier tout le tableau x.

Pour programmer `tri` p q :

- si $p \geqslant q$ il n'y a rien à faire;
- sinon on commence par *partitionner* $(x_p, x_{p+1}, \ldots, x_q)$ en choisissant un *pivot* $m \in \{x_p, x_{p+1}, \ldots, x_q\}$ (ici $m = x_p$) puis en effectuant des échanges dans le sous tableau $(x_p, \ldots, x_q)$ pour que les éléments $\leqslant m$ (resp. $> m$) soient à gauche (resp. à droite) de m (il existe $r \in [|p,q|]$ tel que $p \leqslant k < r \Rightarrow x_k \leqslant m$, $x_r = m$ et $r < k \leqslant q \Rightarrow x_k > m$). Il ne reste plus alors qu'à appeler récursivement `tri` p $(r-1)$ et `tri` $(r+1)$ q pour trier $(x_p, \ldots, x_q)$.

Preuve d'arrêt : Un appel de `tri` p q termine car, pour chacun des deux appels récursifs `tri` p' q' ($(p', q') = (p, r-1)$ ou $(r+1, q)$), on a $-1 \leqslant q' - p' < q - p$.

Programme 1.9 Tri rapide

```
let tri_rapide x =
   let échanger i j = let t = x.(i) in x.(i) <- x.(j); x.(j) <- t in
   let rec tri p q =
      if p < q
         then
            let r =
               let m = x.(p)
               and j = ref p in
               for i = p+1 to q do
                  if x.(i) <= m then ( incr j; échanger !j i )
                  (* p < k <= j => xk <= m et j < k <= i => xk > m *)
               done;
               échanger !j p;
               !j in
            tri p (r-1);
            tri (r+1) q in
   tri 0 (vect_length x - 1);;
```

1. Si $p > q$, le sous tableau est considéré comme vide et `tri` p q ne doit avoir aucun effet.

Proposition 1.4 *Le nombre maximum T_n de comparaisons effectuées par le tri rapide d'un tableau de n nombres vérifie $\frac{n(n-1)}{2} \leqslant T_n \leqslant \frac{n^2}{2}$; donc $T_n \sim \frac{1}{2}n^2$.* [1]

Preuve Le tri d'un tableau $x = (x_p, \ldots ,x_q)$ de $n = q - p + 1$ nombres commence par le partitionnement de x, ce qui nécessite $n - 1$ comparaisons. On applique ensuite le tri rapide aux deux sous tableaux issus du partitionnement : $y = Y(x) = (x_p, \ldots ,x_{r-1})$ et $z = Z(x) = (x_{r+1}, \ldots ,x_q)$. Le nombre A_x de comparaisons effectuées par le tri de x est donc $A_x = n-1+A_y+A_z$ ce qui conduit à l'inégalité $T_n \leqslant n-1+\max\{T_{i-1}+T_{n-i} \mid i = 1, \ldots ,n\}$ (avec $T_0 = T_1 = 0$). On montre alors par récurrence que $T_n \leqslant n^2/2$: soit $n > 1$ tel que $\forall i < n, T_i \leqslant i^2/2$. Alors pour tout $i \in [|1,n|], T_{i-1}+T_{n-i} \leqslant (i-1)^2/2+(n-i)^2/2 \leqslant (n-1)^2/2$ donc, d'après l'inégalité de récurrence, $T_n \leqslant n - 1 + (n-1)^2/2 \leqslant n^2/2$, ce qui termine la récurrence.
Soit maintenant U_n le nombre de comparaisons effectuées par le tri rapide d'un tableau de n nombres distincts et déja triés : $x = (x_p, \ldots ,x_q)$ avec $x_p < x_{p+1} < \cdots < x_q$. Dans ce cas le partitonnement de x conduit à $y = ()$ et $z = (x_{p+1}, \ldots ,x_q)$ avec $x_{p+1} < \cdots < x_q$ donc $U_n = n - 1 + U_{n-1}$; d'où $T_n \geqslant U_n = n(n-1)/2$. ■

1.2.4 Prise en compte de la taille des arguments dans les calculs arithmétiques

On vérifie sans peine que la fonction *pgcd* définie par [2]
soit récursivement *pgcd* a b = **si** $b = 0$ **alors** a **sinon** *pgcd* b $(a \bmod b)$
satisfait les spécifications suivantes
PRÉCONDITIONS a et b sont entiers et $a > b \geqslant 0$. [3]
POSTCONDITION *pgcd* a b renvoie PGCD(a,b).

La fonction *pgcd* est la version récursive de l'algorithme d'EUCLIDE. Si les arguments a et b sont de grands entiers, il est irréaliste de considérer que les divisions effectuées par l'algorithme ont un coût constant. On supposera que la division de a par b — $a = bq + r$ où $0 \leqslant r < b$ — est effectuée par l'algorithme ordinaire qui consiste à calculer les (tranches de) chiffres de q en commençant par le chiffre de poids fort, le calcul de chaque chiffre de q nécessitant de l'ordre de p opérations élémentaires où p est le nombre de chiffres de a (la *taille* de a). Comme $p = \Theta(\log a)$, la complexité de la division est $\mathrm{O}((1 + \log q) \log a)$.

Proposition 1.5 *Le coût de l'algorithme d'Euclide appliqué à deux arguments a et b avec $a > b \geqslant 0$ est $O(\log a)^2$.* [4]

Preuve On montre d'abord par induction que le nombre N de divisions effectuées par *pgcd* a b satisfait à $a \geqslant F_{N+2}$ et $b \geqslant F_{N+1}$ où $(F_n)_n$ est la suite de FIBONACCI définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$ — Si $b = 0$ on a bien $a \geqslant 1 = F_1$ et $b = 0 = F_0$. Sinon *pgcd* a b effectue une première division pour calculer $r = (a \bmod b) = a - bq$ puis $N - 1$ autres pour calculer *pgcd* b r. Si on suppose que $b \geqslant F_{N+1}$ et $r \geqslant F_N$ alors $a = bq + r \geqslant b + r \geqslant F_{N+1} + F_N = F_{N+2}$ —[5].
De $F_n = (\varphi^n + \overline{\varphi}^n)/\sqrt{5}$ où $\varphi = \left(1 + \sqrt{5}\right)/2$ et $\overline{\varphi} = \left(1 - \sqrt{5}\right)/2$, on tire alors que $N = \mathrm{O}(p)$ où $p = \log a$.
Soient $q_1, \ldots ,q_N$ les quotients des divisions effectuées par *pgcd* a b. La complexité totale T_a des divisions est dominée par $\sum_{i=1}^{N}(1 + \log q_i)p = \left(N + \log \prod_{i=1}^{N} q_i\right) p$; or $N = \mathrm{O}(p)$ et on voit par induction que $\prod_{i=1}^{N} q_i \leqslant a$ donc $T_a = (\mathrm{O}(p) + O(p))p = \mathrm{O}(p^2)$. ■

1.2.5 Arbre associé à un appel d'une fonction récursive

Cette section utilise le vocabulaire des *arbres* introduit au chapitre 3 mais elle peut être lue dés à présent si l'on se contente d'une idée intuitive de la notion d'arbre.

---

1. Au sens de la complexité dans le pire des cas, le tri rapide est équivalent au tri par insertion et inférieur au tri fusion. C'est de sa complexité en moyenne que le tri rapide tire son nom (voir p. 26).
2. $a \bmod b$ désigne le reste de la division euclidienne de a par b.
3. On pourrait remplacer les préconditions par : a et b sont des entiers $\geqslant 0$.
4. Il existe un algorithme plus performant (SCHÖNAGE) de complexité $\mathrm{O}((\log a)(\log\log a)^2(\log\log\log a))$.
5. Noter que si $a = F_{p+2}$ et $b = F_{p+1}$ alors $N = p$.

La fonction récursive
soit récursivement $\mathit{fib}(n) =$ **si** $n > 1$ **alors** $\mathit{fib}(n-1) + \mathit{fib}(n-2)$ **sinon** n
calcule (de manière notoirement inefficace) le n^{e} terme F_n de la suite de FIBONACCI définie dans la preuve de la proposition 1.5. L'arbre de la figure 1.1 permet de visualiser le déroulement de l'appel $\mathit{fib}(4)$.

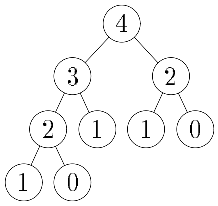


FIG. 1.1 – *Arbre des appels de la fonction* fib *pour l'appel principal* fib(4). *Chaque* nœud *d'étiquette* $p > 1$ *a deux fils étiquetés respectivement* $p-1$ *et* $p-2$ *qui correspondent aux deux appels récursifs* $\mathrm{fib}(p-1)$ *et* $\mathrm{fib}(p-2)$ *engendrés par l'appel de* $\mathrm{fib}(p)$. *Les nœuds d'étiquette* 0 *et* 1 *n'ont pas de fils (ce sont des* feuilles*) car l'appel de* fib(0) *ou de* fib(1) *n'engendre pas de sous appel récursif.*

De manière générale, une fonction récursive
soit récursivement $f(x) =$ <expression>
étant donnée, l'*arbre des appels récursifs* $\mathcal{A}_f(x)$ est défini récursivement par la règle :

> $\mathcal{A}_f(x)$ est l'arbre dont la racine est étiquetée par x et dont les fils sont les n arbres $\mathcal{A}_f(y_1), \dots, \mathcal{A}_f(y_n)$ où $y_1, \dots, y_n$ sont les arguments des sous appels récursifs produits par l'appel de $f(x)$; en particulier, si l'appel de $f(x)$ n'engendre pas de sous appel récursif, $n = 0$ et $\mathcal{A}_f(x)$ est réduit à une feuille.

Exemple 1.10 *Suites récurrentes*

On se donne un ensemble E, un élément a de E, deux suites d'entiers $(p(n))_{n>0}$ et $(q(n))_{n>0}$ telles que $\forall n > 0, 0 \leqslant p(n), q(n) < n$ et une application $f : E \times E \to E$.

On définit alors une suite $(u_n)_{n\geqslant 0}$ par $u_0 = a$ et $\forall n > 0, u_n = f(u_{p(n)}, u_{q(n)})$.

Pour calculer un terme u_n de la suite, on procède récursivement mais, pour ne pas recalculer plusieurs fois un même terme de la suite, on stocke dans un tableau t chaque terme calculé. On utilise aussi un tableau c de booléens tel que c_i indique si le i^{e} terme de la suite a été calculé. Voir le programme 1.10

Programme 1.10 Calcul du n^{e} terme de la suite récurrente $u_n = f(u_{p(n)}, u_{q(n)})$

```
let terme a p q f n =
(* terme : 'a -> ('a -> 'a) -> ('a -> 'a) -> ('a * 'a -> 'a) -> int -> 'a *)
(* terme a p q f n = le n-ème terme de la suite définie par a,p,q et f     *)
    let t = make_vect (n+1) a
    and c = make_vect (n+1) false in
    c.(0) <- true;
    let rec u k =
       if c.(k) then t.(k)
                else (t.(k) <- f(u(p k),u(q k)); c.(k) <- true; t.(k)) in
    u n;;
```

PREUVE DE VALIDITÉ : Le contexte de la sous fonction récursive `u` contient les arguments `a`, `p`, `q`, `f` de la fonction `terme` ainsi que les deux tableaux `t` et `c`. On considère la condition (C) suivante portant sur ce contexte :

$$(C) : \quad \texttt{c.(0)} \wedge (\forall k = 0, \dots, \texttt{n}, \texttt{c.}(k) \Rightarrow \texttt{t.}(k) = u_k).$$

On peut vérifier que la fonction `u` vérifie les spécifications suivantes :

Préconditions : (C) et $k \in [|0, n|]$.

Postconditions : (C) et `u` k renvoie u_k.

COMPLEXITÉ : Considérons l'arbre des sous appels récursifs d'un appel de `u` n. Chaque nœud admet 0 ou 2 fils et le nombre de nœuds internes (ayant 2 fils) est $\leqslant n$ — en effet, à chaque valeur de $k \in [|1,n|]$, il correspond au plus un nœud de l'arbre car, même si `u` k est appelé plusieurs fois, ce n'est que la première fois que l'appel donnera lieu à des sous appels récursifs —. Le nombre total de nœuds (que l'on peut considérer comme une mesure de la complexité temporelle du calcul) est donc $\leqslant 2n+1$, de sorte que le calcul est en temps $\mathrm{O}(n)$.

La complexité spatiale est $\Theta(n)$ à cause des tableaux `t` et `c`.

1.2.6 Complexité en moyenne de quelques algorithmes

Si E est un ensemble de n nombres on note $\mathfrak{S}(E)$ l'ensemble des $n!$ tableaux $x = (x_1, \dots ,x_n)$ dont le support $\{x_1, \dots ,x_n\}$ est E (les x_i sont donc distincts). On considère un algorithme $\mathcal{A}_x$ prenant un tableau $x \in \mathfrak{S}(E)$ en entrée et effectuant des comparaisons des x_i et éventuellement des échanges dans le tableau x; par exemple un algorithme de tri. Soit A_x la complexité de $\mathcal{A}_x$, la notion d'opération élémentaire ayant été préalablement précisée. On s'intéresse dans cette section à la *complexité moyenne* de $\mathcal{A}$, l'ensemble $\mathfrak{S}(E)$ étant muni d'une distribution uniforme. Autrement dit, on calcule (ou on étudie le comportement asymptotique de) la valeur moyenne (ou l'*espérance*) $t_n = \frac{1}{n!} \sum_{x \in \mathfrak{S}(E)} A_x$ des A_x.

Si $x = (x_1, \dots ,x_n) \in \mathfrak{S}(E)$, soit $x'_i \in [|1,n|]$ le rang de x_i (si x_i est le plus petit des x_j, $x'_i = 1$, si x_i est le deuxième plus petit des x_j, $x'_i = 2$, etc.). Vues les hypothèses faites sur l'algorithme $\mathcal{A}_x$ (comparaisons et échanges), A_x ne dépend que des positions relatives des x_i, de sorte que $A_x = A_{x'}$ où $x' = (x'_1, \dots ,x'_n)$. Ainsi on peut se restreindre, sans perte de généralité, au cas où $E = [|1,n|]$, auquel cas $\mathfrak{S}(E) = \mathfrak{S}_n$ est l'ensemble des permutations de $[|1,n|]$.

Exemple 1.11 *Analyse en moyenne d'un algorithme de recherche de maximum*

pour $x = (x_1, \dots ,x_n) \in \mathfrak{S}(E)$, l'algorithme 1.2 renvoie le maximum de ces n nombres.

Algorithme 1.2 Maximum des éléments d'un tableau de n nombres

1 **soit** *maximum* x =
2   **soit** $m \leftarrow x_1$ **dans**
3   **pour** $i = 2$ **à** n **faire**
4   **si** $x_i > m$ **alors**
5             $m \leftarrow x_i$
6   (* $m = \max\{x_1, \dots ,x_i\}$ *)
7   **renvoyer** m

On s'intéresse ici au nombre des changements d'affectation de la variable m (ligne 5) (on peut imaginer des situations analogues où ces changements d'affectation seraient coûteux). Ainsi, A_x est le nombre d'éxécutions de la ligne 5 dans le calcul de *maximum* x.

Proposition 1.6 *Le nombre moyen d'affectations dans l'algorithme de recherche du maximum d'un tableau de n nombres distincts est* $t_n = \sum_{i=2}^n \frac{1}{i} \sim \ln n$.

Preuve Pour $k \geqslant 0$, notons p_k^n la probabilité de l'évènement $(A_x = k)$: $p_k^n = \frac{|S_k^n|}{n!}$ où $S_k^n = \{x \in \mathfrak{S}_n \,|\, A_x = k\}$ et $|\,.\,| = \mathrm{cardinal}(\,.\,)$. On voit (ou on sait) que $t_n = \mathrm{E}(A_x) = \sum_{k=0}^{\infty} k p_k^n$ (étant entendu que les p_k^n sont nuls pour $k \geqslant n$, on peut ne faire varier k dans cette somme que de 0 à $n-1$).
La famille $(S_k^n(m))_{m=1,\dots,n}$ où $S_k^n(m) = \{x \in S_k^n \,|\, x_n = m\}$ est une partition de S_k^n.
Si $x = (x_1, \dots ,x_{n-1},n) \in S_k^n(n)$ alors le début de l'éxécution de la boucle du calcul de *maximum* x ($i = 2, \dots ,n-1$) coïncide avec l'éxécution de la boucle du calcul de *maximum* x' où $x' = (x_1, \dots ,x_{n-1})$ et se termine, pour $i = n$, par une éxécution de la ligne 5. On en déduit que $x' \in S_{k-1}^{n-1}$. Plus précisément, $x \mapsto x'$ est une bijection de $S_k^n(n)$ sur S_{k-1}^{n-1} et $|S_k^n(n)| = |S_{k-1}^{n-1}|$.
On voit de même que si $m < n$, $|S_k^n(m)| = |S_k^{n-1}|$.
Finalement, $|S_k^n| = \sum_{m=1}^n |S_k^n(m)| = |S_{k-1}^{n-1}| + (n-1)|S_k^{n-1}|$; soit, en divisant par $n!$:

$$(1) \quad p_k^n = \frac{1}{n} p_{k-1}^{n-1} + \frac{n-1}{n} p_k^{n-1}$$

On peut alors vérifier que cette formule reste valable pour tout $n > 0$ et $k \geqslant 0$, à condition de convenir que $p_k^1 = \delta_{0,k}$ (symbole de KRONECKER) et $p_k^n = 0$ si $k < 0$.
On introduit la *fonction génératrice* de la suite $(p_k^n)_{k\geqslant 0}$. C'est la fonction polynomiale $G_n(x) = \sum_{k=0}^{\infty} p_k^n x^k$. On déduit de la relation (1) que $G_n(x) = \frac{x+n-1}{n} G_{n-1}(x)$ puis $t_n = Gn'(1) = G'_{n-1}(1) + \frac{1}{n} G_{n-1}(1) = G'_{n-1}(1) + \frac{1}{n}$; d'où, par récurrence, $t_n = \sum_{i=2}^{n} \frac{1}{i}$. ■

Exemple 1.12 *Analyse en moyenne du tri par insertion*

Proposition 1.7 *Le nombre moyen de comparaisons effectuées par le tri par insertion d'un tableau (ou d'une liste; voir p. 19) de n nombres distincts est $\sim \frac{n^2}{4}$.*

Preuve On dit qu'un couple (i,j) est une *inversion* d'un tableau $x = (x_1, \dots, x_n)$ si $1 \leqslant i < j \leqslant n$ et $x_i > x_j$.
Le tri par insertion de x se déroule en deux étapes

- trier récursivement $(x_2, \dots, x_n)$, ce qui donne $y = (y_2, \dots, y_n)$;
- insérer x_1 à sa place dans le tableau trié y.

Le nombre de comparaisons effectuées à la deuxième étape est égal à $I = |\{i = 2, \dots, n \mid x_1 > y_i\}|$ ou à $I+1$ suivant que $x_1 > y_n$ ou non. Or, comme $\{y_2, \dots, y_n\} = \{x_2, \dots, x_n\}$, I est aussi le nombre d'inversions de x de la forme $(1,i)$. On en déduit par induction que $B_x \leqslant A_x \leqslant B_x + n - 1$ où A_x est le nombre total de comparaisons effectuées par le tri par insertion de x et B_x est le nombre d'inversions de x. En ajoutant ces inégalités puis en divisant par $n!$ on obtient $u_n \leqslant t_n \leqslant u_n + n - 1$ où u_n est le nombre moyen d'inversions d'une permutation $x \in \mathfrak{S}_n$. Il reste à prouver que $u_n \sim \frac{n^2}{4}$: si $x' = (x_n, \dots, x_2, x_1)$ est la permutation *miroir* d'une permutation $x = (x_1, x_2, \dots, x_n)$, les inversions de x' sont les couples (i,j) (avec $1 \leqslant i < j \leqslant n$) qui ne sont pas des inversions de x. La somme des nombres d'inversions de x et de x' est donc $|\{(i,j) \mid 1 \leqslant i < j \leqslant n\}| = \frac{n(n-1)}{2}$. En groupant deux par deux les permutations de $\mathfrak{S}_n$ on en déduit que $u_n = \frac{n(n-1)}{4} \sim \frac{n^2}{4}$. ■

Exercice 1.1 En utilisant la proposition 1.6 montrer que le nombre moyen de comparaisons du tri par insertion est $\frac{n(n+3)}{4} - \sum_{i=1}^{n} \frac{1}{i}$.

Exemple 1.13 *Analyse en moyenne du tri rapide*

Proposition 1.8 *Le nombre moyen de comparaisons effectuées par le tri rapide*[1] *d'un tableau de n nombres distincts est $\sim 2n \ln n$.*

Preuve La preuve de la proposition 1.4 (voir p. 23) avait permis d'établir la formule $A_x = n - 1 + A_{Y(x)} + A_{Z(x)}$ pour un tableau x de n nombres, $Y(x)$ et $Z(x)$ désignant les deux sous tableaux issus du partitionnement de x. On peut alors concevoir que la valeur moyenne t_n des A_x vérifie

$$(1) \quad t_n = n - 1 + \frac{1}{n} \sum_{i=1}^{n} (t_{i-1} + t_{n-i})$$

Etablissons formellement et de manière élémentaire[2] cette équation de récurrence.
En notant $\mathfrak{S}_n^i$ l'ensemble des permutations $x \in \mathfrak{S}_n$ telles que $x_1 = i$, on a $n! t_n = \sum_{x \in \mathfrak{S}_n} A_x = \sum_{i=1}^{n} \sum_{x \in \mathfrak{S}_n^i} A_x$; d'où, en posant $u_n^i = \sum_{x \in \mathfrak{S}_n^i} A_{Y(x)}$ et $v_n^i = \sum_{x \in \mathfrak{S}_n^i} A_{Z(x)}$:

$$(2) \quad t_n = n - 1 + \frac{1}{n!} \sum_{i=1}^{n} (u_n^i + v_n^i)$$

Fixons $i \in [|1,n|]$. Si $x \in \mathfrak{S}_n^i$, le pivot du partitionnement de x est $x_1 = i$, donc $Y(x) \in \mathfrak{S}([|1,i-1|])$ et $Z(x) \in \mathfrak{S}([|i,n|])$.
Pour $y \in \mathfrak{S}([|1,i-1|])$, on note $Y^{-1}(y) = \{x \in \mathfrak{S}_n^i \mid Y(x) = y\}$. Soit $y' \in \mathfrak{S}([|1,i-1|])$ la suite obtenue à partir de y en permutant deux éléments $a = y_\alpha$ et $b = y_\beta$: $y = (\dots, a, \dots, b, \dots)$ et $y' = (\dots, b, \dots, a, \dots)$. A tout x élément de $Y^{-1}(y)$ on associe x' en permutant a et b dans x. Comme le partitionnement de x ne fait intervenir que des comparaisons des x_j avec i et que a et b sont tous deux $<$ à i, $Y(x')$ s'obtiendra à partir de $Y(x) = y$ en permutant a et b donc $Y(x') = y'$ et $x \mapsto x'$ est une application de $Y^{-1}(y)$ dans $Y^{-1}(y')$. En inversant les rôles de y et de y', on voit que c'est même une bijection. $Y^{-1}(y)$ et $Y^{-1}(y')$ ont donc

1. Voir p. 22.
2. Autrement dit, sans utiliser la théorie des probabilités.

le même nombre d'éléments. Plus généralement, si y et y' sont deux suites quelconques de $\mathfrak{S}([|1,i-1|])$, on peut passer de y à y' par une suite convenable d'échanges de deux éléments (le groupe symétrique est engendré par les transpositions) donc $Y^{-1}(y)$ et $Y^{-1}(y')$ ont encore le même nombre d'éléments. Comme $|\mathfrak{S}_n^i| = (n-1)!$ et $|\mathfrak{S}([|1,i-1|])| = (i-1)!$, le cardinal commun aux ensembles $Y^{-1}(y)$ quand $y \in \mathfrak{S}([|1,i-1|])$ est $\frac{(n-1)!}{(i-1)!}$, d'où $u_n^i = \frac{(n-1)!}{(i-1)!}\sum_{y\in\mathfrak{S}([|1,i-1|])} A_y = (n-1)!t_{i-1}$. De même, on montre que $v_n^i = (n-1)!t_{n-i}$ et en reportant ces valeurs dans (2) on trouve bien la relation (1).
On a alors successivement :
$(1) \Rightarrow t_n = n-1+\frac{2}{n}\sum_{i=0}^{n-1} t_i \Rightarrow nt_n-(n-1)t_{n-1} = 2(n-1)+2t_{n-1} \Rightarrow nt_n-(n+1)t_{n-1} = 2(n-1) \Rightarrow \frac{t_n}{n+1}-\frac{t_{n-1}}{n} = \frac{4}{n+1}-\frac{2}{n} \Rightarrow \sum_{k=1}^n \frac{t_k}{k+1} - \sum_{k=1}^n \frac{t_{k-1}}{k} = \sum_{k=1}^n \frac{4}{k+1} - \sum_{k=1}^n \frac{2}{k} \Rightarrow t_n = 2(n+1)\sum_{k=1}^n \frac{1}{k} - 4n \Rightarrow t_n \sim 2n\ln n$. ■

Tri rapide stochastique

Le tri rapide est particulièrement inefficace si, à chaque étape de la récursion, le pivot est proche du plus petit ou du plus grand nombre du tableau (voir le cas d'un tableau trié ou d'un tableau trié en ordre inverse). A priori, le mieux qui puisse arriver est que, à chaque étape, le pivot partitionne le tableau x de longueur n en deux sous tableaux de longueurs $\lfloor n/2\rfloor$ et $\lceil n/2\rceil$. Dans ce cas, si on note $u_n = A_x$, on a $u_n = n-1+u_{\lfloor n/2\rfloor}+u_{\lceil n/2\rceil}$ et, d'après le th. 1.3, $u_n = \mathrm{O}(n\log n)$. La complexité dans le meilleur des cas est donc asymptotiquement du même ordre que la complexité en moyenne. On peut donc s'attendre à ce que A_x ne soit pas trop éloigné du meilleur des cas. Mais cela suppose que toutes les suites sont équiprobables. Si tel n'est pas le cas, on a intérêt à « forcer le hasard » en obligeant le pivot à être, avec la même probabilité un élément quelconque du tableau. Avant de commencer le partitionnement, on échange le premier élément du tableau avec un autre choisi au hasard. Dans le programme 1.9 on insère après `let r =` la ligne[1]

```
échanger p (random__int (q - p + 1) + p);;
```

1. Voir la section A.7.4 p. 106 pour l'utilisation du module `random`.

Chapitre 2

Notions de base

2.1 Mots et langages

2.1.1 Le monoïde libre des mots sur un alphabet

Un ensemble quelconque est aussi appelé un *alphabet*[1]. Un élément d'un alphabet est appelé une *lettre*.

Un *mot* sur un alphabet X est un n-uplet ($n \in \mathbf{N}$) $u = (x_1, \ldots ,x_n)$ de lettres $x_i \in X$. Le nombre n de lettres constituant le mot u est appelé la *longueur* de u et noté $|u|$. En particulier il y a un unique mot de longueur nulle, $()$, il est appelé le *mot vide* et noté ici μ, parfois λ. Les mots de longueur 1 sont exactement les lettres de X.

L'ensemble des mots sur X est noté X^* ; on le munit d'une loi de composition interne notée « . » et appelée le *produit de concaténation* définie comme suit : si $u = (x_1, \ldots ,x_n) \in X^*$ et $v = (y_1, \ldots ,y_m) \in X^*$ sont deux mots, on pose $u.v = (x_1, \ldots ,x_n,y_1, \ldots ,y_m)$ mais on omet la plupart du temps le point et on note $u.v = uv$.

Un *monoïde* est un ensemble muni d'une loi de composition interne associative et possédant un élément neutre. Il est clair que X^* muni de la concaténation est un monoïde admettant le mot vide μ pour élément neutre (c'est pourquoi on note aussi $\mu = 1_{X^*}$). X^* est appelé *monoïde libre* sur l'alphabet X.

Si $u = (x_1, \ldots ,x_n) \in X^*$, on voit que u est le produit de concaténation des lettres qui le composent, ce qui permet d'écrire $u = x_1x_2 \ldots x_n$. Il ne faut néanmoins pas oublier que, si $x_1,x_2, \ldots ,x_n \in X$ sont des lettres le produit $x_1x_2 \ldots x_n$ désigne le n-uplet $(x_1, \ldots ,x_n)$ et donc que l'écriture $u = x_1x_2 \ldots x_n$ est unique.

On dit qu'un mot $v \in X^*$ est *facteur* d'un mot $u \in X^*$ s'il existe deux mots $v',v'' \in X^*$ tels que $u = v'vv''$. En particulier, s'il existe v'' tel que $u = vv''$, on dit que v est un *facteur gauche* ou *préfixe* de u et s'il existe un mot v' tel que $u = v'v$, on dit que v est un *facteur droit* ou *suffixe* de u. La relation « est préfixe de » est une relation d'ordre sur X^*.

2.1.2 Définition d'un morphisme sur le monoïde libre par sa restriction aux lettres

Un *morphisme* d'un monoïde $(M,.)$ dans un monoïde $(N,\times)$ est une application $f : M \to N$ telle que $\forall x,y \in M, f(x.y) = f(x) \times f(y)$ et $f(1_M) = 1_N$.

Considérons un morphisme F du monoïde libre X^* dans un monoïde $(M,\times)$ (qui peut aussi être un monoïde libre Y^*). F est entièrement déterminé par sa restriction f à X. En effet, si $u = x_1 \ldots x_n \in X^*$ où $x_1, \ldots x_n \in X$, $F(u) = F(x_1) \times F(x_2) \times \cdots \times F(x_n) = f(x_1) \times f(x_2) \times \cdots \times f(x_n)$. Réciproquement, si on se donne une application quelconque $f : X \to M$, l'application $F : X^* \to M$ définie par

$$\forall x_1, \ldots x_n \in X, F(x_1 \ldots x_n) = f(x_1) \times \cdots \times f(x_n) \text{ et } F(\mu) = 1_M$$

est un morphisme (vérification facile) dont la restriction à X est f. On dit que F est le *morphisme défini sur les lettres* par la condition $\forall x \in X, F(x) = f(x)$.

1. Dans la plupart des situations, on suppose les alphabets finis mais ce n'est pas nécessaire ici.

Exemple 2.1 La longueur $|u|$ d'un mot u est l'image de u par le morphisme de X^* dans $(\mathbf{N},+)$ défini sur les lettres par $|x| = 1$.

Exemple 2.2 Le *nombre d'occurrences* $|u|_x$ d'une lettre fixée x dans un mot u est défini comme l'image de u par le morphisme de X^* dans $(\mathbf{N},+)$ défini sur les lettres par : $|y|_x = \delta_{x,y}$ (symbole de KRONECKER): sur $X = \{0,1\}$, $|01001101|_1 = 4$.

Exemple 2.3 Si $Y \subset X$, on définit la *projection* $p_Y : X^* \to Y^*$ comme le morphisme défini sur les lettres par $p_Y(x) = x$ si $x \in Y$ et $p_Y(x) = 1_{Y^*}$ sinon. $p_Y(u)$ est obtenu en supprimant de u les lettres qui ne sont pas dans Y: si $X = \{a,b,c\}$ et $Y = \{a,b\}$, $p_Y(aacbca) = aaba$.

2.1.3 Langages

Une partie de X^* est appelée un *langage* sur l'alphabet X. Un langage est donc un ensemble de mots muni d'aucune structure particulière. $\emptyset$ est un langage (le *langage vide*) à ne pas confondre avec le langage $\{\mu\}$ constitué du seul mot vide. Par abus de notation, le langage $\{u\}$ réduit à un mot est noté u. De même, on note μ pour $\{\mu\}$.

Opérations sur les langages

Si L et M sont deux langages sur le même alphabet X on définit

- leur *somme* $L + M = L \cup M$;
- leur *produit de concaténation* $LM = \{uv \mid u \in L \wedge v \in M\}$;
- la puissance p^{e} de L, $L^p = \{u_1 \dots u_p \mid u_1, \dots, u_p \in L\}$; en particulier, $L^0 = \mu$, $L^1 = L$ et $L^2 = LL$;
- l'*étoile* ou l'*itéré* de L, $L^* = \bigcup_{p \in \mathbf{N}} L^p$;
- l'*itéré strict* de L, $L^+ = LL^* = L^*L = \bigcup_{p \in \mathbf{N}^*} L^p$.

Noter que l'étoile du langage X constitué des mots d'une lettre est l'ensemble des produits de lettres, c.-à-d. l'ensemble de tous les mots. Ainsi la notation X^* pour désigner cet ensemble est conforme à la définition de l'étoile.

Proposition 2.1 *Si L, M et N sont des langages sur un alphabet X, on a :*

$L + M = M + L$	$(L + M) + N = L + (M + N)$	$L + L = L + \emptyset = L$
$(LM)N = L(MN)$	$L\mu = \mu L = L$	$L\emptyset = \emptyset L = \emptyset$
$L(M + N) = LM + LN$	$(L + M)N = LN + MN$	
$\emptyset^* = \mu^* = \mu$	$(L^*)^* = (\mu + L)^* = L^* + L = L^*$	$(L + M)^* = (L^*M^*)^*$
$L(ML)^* = (LM)^*L$	$(L^*M)^* = \mu + (L + M)^*M$	$(LM^*)^* = \mu + L(L + M)^*$

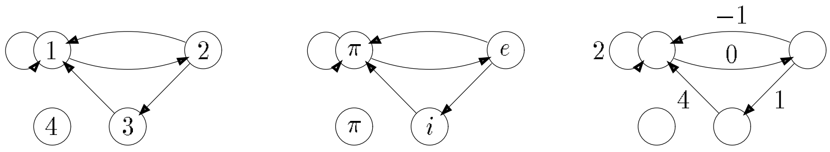
Preuve Montrons seulement la dernière formule, les autres sont laissées en exercice :
– Soit $u \in (LM^*)^*$. $u = u_1 \dots u_p$ où $\forall i, u_i \in LM^*$. Si $p = 0$, $u = \mu$. Sinon $u_i = v_i w_i$ avec $v_i \in L$ et $w_i \in M^*$ donc $u = v_1 x$ avec $v_1 \in L$ et $x = w_1 v_2 w_2 \dots v_p w_p \in (L + M)^*$ car les v_i et les w_i sont des produits de mots de $L + M$. Donc $u \in L(L + M)^*$.
– Réciproquement, soit $u \in \mu + L(L + M)^*$. Si $u = \mu$, $u \in (LM^*)^*$. Si $u \in L(L + M)^*$, $u = vw$ avec $v \in L$ et $w \in (L + M)^*$. w s'écrit donc $w = w_1 \dots w_p$ où $w_i \in L + M$. Si $1 \leqslant i_1 < i_2 < \dots < i_q \leqslant p$ sont les indices i tels que $w_i \in L$, l'écriture de u sous la forme $u = (vw_1 \dots w_{i_1 - 1})(w_{i_1} \dots w_{i_2 - 1}) \dots (w_{i_q} \dots w_p)$ montre que $u \in (LM^*)^{q+1} \subset (LM^*)^*$. ■

2.2 Graphes et arbres non ordonnés

2.2.1 Graphes orientés

Un *graphe* est un couple $G = (S,A)$ où S est un ensemble fini et A est un sous ensemble du produit cartésien $S \times S$. Les éléments de S sont appelés les *sommets* et ceux de A les *arcs* de G. L'*origine* et l'*extrémité* d'un arc $a = (x,y) \in A$ sont les sommets x et y respectivement. Une *boucle* est un arc dont l'origine et l'extrémité coïncident.

On représente un graphe en dessinant un rond par sommet et une flèche par arc selon des conventions évidentes. Le dessin obtenu est appelé la *représentation sagittale* du graphe. Voir la figure 2.1 a.



a. *graphe* b. *graphe étiqueté* c. *graphe valué*

FIG. 2.1 – *Représentation sagittale du graphe* $(\{1,2,3,4\},\{(1,2),(2,1),(2,3),(3,1),(1,1)\})$ *et (b. et c.) le même graphe muni d'étiquettes et de valuations*

Un *chemin* de *longueur* $k \geqslant 0$, d'*origine* $x \in S$ et d'*extrémité* $y \in S$ est une suite de sommets $(x_0,x_1,\dots,x_k)$ telle que $x = x_0$, $y = x_k$ et $\forall i \in [|1,k|], (x_{i-1},x_i) \in A$. Si un tel chemin existe, on dit que y est *accessible* à partir de x et que x est *coaccessible* à partir de y. Un chemin est dit *élémentaire* si les x_i sont distincts.

Le *degré sortant* (resp. *rentrant*) d'un sommet x est le nombre d'arcs d'origine x (resp. d'extrémité x).

Un *étiquetage* d'un graphe $G = (S,A)$ par un ensemble E est une application e de S dans E. La donnée de G et de e est alors appelée un *graphe étiqueté* et, si x est un sommet de G, on dit que $e(x)$ est l'*étiquette* de x. Par exemple, si l'on munit le graphe de la figure 2.1 a. de l'étiquetage $1 \mapsto \pi$, $2 \mapsto e$, $3 \mapsto i$ et $4 \mapsto \pi$, on obtient le graphe de la figure 2.1 b.

De même, on appelle *graphe valué* un graphe dont chaque arc, et non plus chaque sommet, est muni d'une valeur; voir la figure 2.1 c.

IMPLÉMENTATION : On suppose que les sommets d'un graphe $G = (S,A)$ sont numérotés de 0 à $n-1$, si bien que l'on peut supposer que $S = [|0,n-1|]$. Il existe essentiellement deux méthodes permettant de représenter G :

La *matrice d'adjacence* de G est la matrice $(g_{ij})_{i,j=0,\dots n-1}$ définie par $g_{ij} = 1$ si $(i,j) \in A$ et $g_{ij} = 0$ sinon. Elle peut être implémentée en CAML par le type `int vect vect`.

La représentation par *liste d'adjacence* est le tableau $(\ell_i)_{i=0,\dots,n-1}$ où ℓ_i est la liste des extrémités des arcs d'origine i (les sommets *adjacents* à i). Cette représentation amène donc à implémenter les graphes par le type CAML `int list vect`.

2.2.2 Arbres non ordonnés

Un *arbre non ordonné* est un graphe T vérifiant les conditions suivantes (figure 2.2) :

(i) T a un unique sommet de degré rentrant nul appelé sa *racine*;

(ii) Tout sommet $x \neq$ de la racine a un degré rentrant égal à 1, l'origine y de l'unique arc d'extrémité x est appelée le *père* de x et on dit que x est un *fils* de y;

(iii) Tout sommet est accessible à partir de la racine.

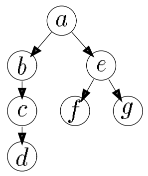


FIG. 2.2 – *Un arbre non ordonné* $(\{a,b,c,d,e,f,g\},\{(a,b),(a,e),(b,c),(c,d),(e,f),(e,g)\})$*; dans la représentation sagittale, on place la racine en haut et les fils d'un sommet sont placés sous ce sommet.*

Si $T_i = (S_i,A_i)$, $i = 1,\dots,n$ sont n arbres non ordonnés disjoints (les S_i sont disjoints), le graphe $(\bigcup_i S_i, \bigcup_i A_i)$ est appelé une *forêt non ordonnée*.

> **Exercice 2.1** Montrer qu'un graphe est une forêt si et seulement si les deux conditions suivantes sont vérifiées :
>
> – le degré rentrant d'un sommet est 0 ou 1;
> – tout chemin est élémentaire.

Les définitions qui suivent concernent un arbre non ordonné $T = (S,A)$ et seront illustrées avec l'arbre de la figure 2.2.

Le nombre de sommets est la *taille* de T; ex. 7.

Le *degré* d'un sommet est son degré sortant (le nombre de ses fils); ex. les degrés de a, b, c, d, e, f et g sont respectivement 2, 1, 1, 0, 2, 0 et 0.

Le *degré* de T est le maximum des degrés de ses sommets; ex. 2.

Un sommet de degré > 0 est dit *interne* et un sommet de degré 0 est appelé une *feuille* ou un sommet *terminal*; ex. les sommets internes sont a, b, c et e et les feuilles d, f et g.

Pour tout sommet x il y a un unique chemin de la racine à x — l'existence d'un tel chemin est assurée par (iii) et la non unicité contredirait (ii) —. Les sommets constituant ce chemin sont appelés les *ascendants* de x; ex. les ascendants de c sont a, b et c.

Les sommets accessibles à partir de x sont appelés les *descendants* de x; ex. les descendants de e sont e, f et g. En utilisant le chemin de la racine à x, on voit que y est descendant de x ssi x est ascendant de y.

Si S_x est l'ensemble des descendants de x et $A_x = \{(y,z) \in A \mid \{y,z\} \subset S_x\}$, alors on vérifie facilement que $T_x = (S_x,A_x)$ est un arbre non ordonné de racine x; on l'appelle le *sous arbre enraciné* en x; ex. $T_e = (\{e,f,g\},\{(e,f),(e,g)\})$.

Si S'_x est l'ensemble des descendants stricts (différents de x) de x et $A'_x = \{(y,z) \in A \mid \{y,z\} \subset S'_x\}$, alors $T'_x = (S'_x,A'_x)$ est une forêt constituée des sous arbres enracinés en les fils de x. Ces sous arbres sont appelés les *branches* de x.

La *profondeur* ou *hauteur* d'un sommet x est le nombre de ses ascendants stricts ($\neq x$); ex. c est de hauteur 2. La *profondeur* ou *hauteur* de T est le maximum des hauteurs de ses sommets; ex. 3.

Exercice 2.2 Soit S un ensemble fini, $r \in S$ un élément particulier et $p : S \setminus \{r\} \to S$ une application. Quelles conditions doit vérifier p pour que S soit l'ensemble des sommets d'un arbre de racine r pour lequel le père d'un sommet $x \neq r$ soit $p(x)$.

Exercice 2.3 Sur l'ensemble S des sommets d'un arbre non ordonné on définit une relation $\leqslant$ par $\forall x,y \in S, x \leqslant y \Leftrightarrow x$ est ascendant de y. Montrer que :

(i)′ $\leqslant$ est une relation d'ordre;

(ii)′ S a un plus petit élément pour $\leqslant$;

(iii)′ pour tout $x \in S$ différent du plus petit élément, l'ensemble des minorants stricts de x a un plus grand élément.

Réciproquement, montrer que si S est un ensemble fini muni d'une relation $\leqslant$ vérifiant (i)′, (ii)′ et (iii)′ alors S est l'ensemble des sommets d'un arbre non ordonné de racine $r = \min S$ pour lequel le père d'un sommet $x \neq r$ est le plus grand minorant strict de x.

Exemple : Soit S un langage fini sur un alphabet X tel que tout préfixe d'un mot de L appartient encore à L. Montrer que la relation « est préfixe de » sur S vérifie les propriétés (i)′, (ii)′ et (iii)′.

2.2.3 Graphes non orientés

Un graphe est dit *non orienté* s'il est symétrique — c.-à-d. (x,y) est un arc $\Rightarrow$ (y,x) aussi — et antiréflexif — c.-à-d. sans boucle —. Pour un tel graphe, les paires $\{x,y\}$ telles que (x,y) est un arc sont appelées les *arêtes*. On représente plutôt un graphe non orienté sous la forme $G = (S,A)$ où A est l'ensemble des arêtes et, dans la représentation sagittale, on dessine les arêtes sans flèche. Dans un graphe non orienté, on appelle *cycle* un chemin $(x_0,x_1,\ldots,x_k)$ tel que $x_k = x_0$, $k \geqslant 3$ et $x_1,\ldots,x_k$ sont distincts. Le graphe est dit *acyclique* s'il ne contient pas de cycle. Il est dit *connexe* si tout sommet est accessible à partir de tout autre sommet.

Exercice 2.4 Soit $G = (S,A)$ un graphe non orienté. Montrer l'équivalence des 6 propriétés suivantes

1. G est connexe et acyclique;
2. deux sommets quelconques de G sont reliés par un unique chemin élémentaire;
3. G est connexe, mais si une arête quelconque est ôtée de G, le graphe résultant n'est plus connexe;
4. G est connexe et $|A| = |S| - 1$;[1]

1. $|E|$ désigne le nombre d'éléments de E.

5. G est acyclique et $|A| = |S| - 1$;
6. G est acyclique mais si une arête quelconque est ajoutée à A, le graphe résultant contient un cycle.

Exercice 2.5 A un arbre non ordonné $T = (S,A)$ on associe le graphe non orienté $T' = (S,\{\{x,y\} \mid (x,y) \in A\})$ obtenu en supprimant les orientations des arcs de T. Montrer que T' est connexe et acyclique.
Réciproquement, montrer que si G est un graphe non orienté connexe et acyclique et si r est un sommet particulier de G, on peut orienter les arêtes de G de manière à en faire un arbre non ordonné de racine r.

2.2.4 Parcours d'un graphe

Etant donnée une application qui, à chaque sommet x d'un graphe orienté $G = (S,A)$, associe une action éxécutable $action(x)$, un parcours de G consiste à éxécuter $action(x)$ pour certains des sommets x de G — éventuellement tous —.

On suppose que G est représenté par liste d'adjacence ou par une structure analogue.

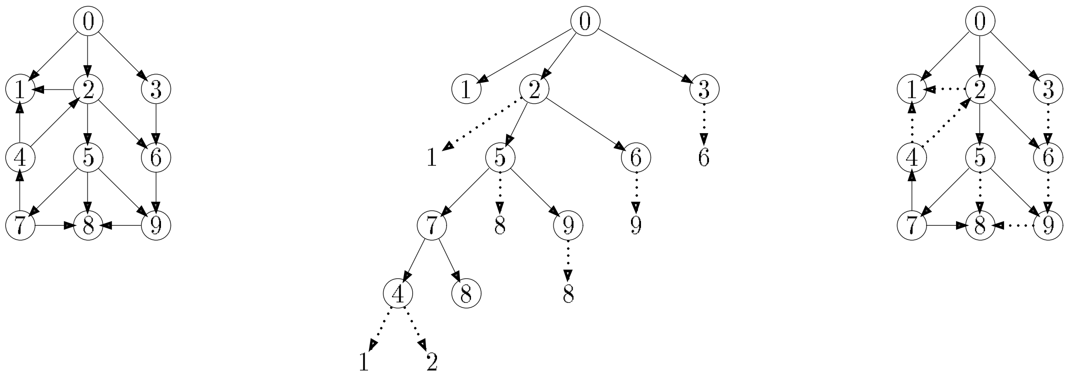
Parcours en profondeur

Etant donné un ensemble de sommets $S' \subset S$ d'un graphe $G = (S,A)$, on se propose de parcourir l'ensemble $acc(S')$ des sommets accessibles à partir d'un sommet de S'; autrement dit, on doit éxécuter $action(x)$ une et une seule fois pour chaque sommet x de $acc(S')$.

Algorithme 2.1 Parcours en profondeur d'un graphe

```
soit parcours_en_profondeur action (S,A) S' =
  soit marque = tableau de booléens indexé par S dans
  pour x ∈ S faire marque(x) ← faux
  soit récursivement explorer x =
    si non marque(x)
      alors
        action(x)
        marque(x) ← vrai
        pour y extrémité d'un arc d'origine x faire explorer y dans
  pour x ∈ S' faire explorer x
```

Etudions le comportement de la fonction $parcours\_en\_profondeur$ de l'algorithme 2.1 sur le graphe $G = (S,A)$ de la figure 2.3 a et l'ensemble de sommets $S' = \{0\}$.



a. Le graphe G b. Arbre des appels récursifs c. Le même arbre

FIG. 2.3 – *Parcours en profondeur d'un graphe G*

Si $marque(x)$ est faux, l'appel de $explorer\ x$ débute par l'éxécution de $action(x)$ et donne lieu en général à des sous appels récursifs. Qualifions un tel appel d'*effectif*. Un appel non effectif n'éxécute rien.

L'arbre des appels[1] de la fonction récursive *explorer* est donné par la figure 2.3 b où on a entouré d'un cercle les sommets effectifs — qui correspondent à des appels effectifs —. On voit que l'ensemble des étiquettes des sommets effectifs de l'arbre est précisément $S = acc(S')$, ce qui permet de vérifier, sur cet exemple, que $action(x)$ est bien éxécuté pour chaque sommet $x \in acc(S')$. L'ordre d'éxécution est 0, 1, 2, 5, 7, 4, 8, 9, 6, 3.

Si l'on supprime de l'arbre les sommets non effectifs qui sont des feuilles, on obtient l'arbre de la figure 2.3 c qui n'est autre que le graphe G privé de certains de ses arcs.[2] Les arcs supprimés sont en pointillés sur la figure; on peut aussi considérer qu'ils correspondent aux arcs de l'arbre initial ayant comme extrémité un sommet non effectif.

De manière informelle, on voit que le parcours consiste à suivre des chemins les plus longs — ou profonds — possibles puis à les remonter pour suivre des branches non encore explorées.

Proposition 2.2 *parcours_en_profondeur action* (S,A) S' *éxécute* $action(x)$ *une et une seule fois pour chaque sommet* x *accessible depuis* S'. *Sa complexité est* $O(|S| + |A|)$.

Preuve Considérons le prédicat

(I) Pour tout sommet x, $marque(x) = vrai$ ssi $action(x)$ a déjà été éxécuté, auquel cas $action(x)$ n'a été éxécuté qu'une fois.

On peut vérifier que *explorer* x admet (I) pour invariant et satisfait les spécifications suivantes :

- *explorer* x marque tous les sommets y pour lesquels, au moment de l'appel de *explorer* x, il existe un chemin formé de sommets non marqués joignant x à y.

On en déduit facilement la validité de la fonction *parcours_en_profondeur*.

La complexité est en $O(|S| + |A|)$ car la forêt des appels récursifs de *explorer* contient au plus $|S|$ sommets effectifs et $|A|$ sommets non effectifs. ■

IMPLÉMENTATION : La fonction
`parcours_en_profondeur : (int -> 'a) -> int list vect -> int list -> unit`
du programme 2.1 implémente le parcours en profondeur d'un graphe représenté par liste d'adjacence. Par exemple,

```
parcours_en_profondeur
   print_int [|[1;2;3];[];[5;6];[6];[1;2];[7;8;9];[9];[4;8];[];[8]|] [0];;
```

affiche `0125748963`.

Programme 2.1 Parcours en profondeur d'un graphe

```
let parcours_en_profondeur action g s =
   let n = vect_length g in
   let marque = make_vect n false in
   let rec explorer x =
      if not marque.(x)
         then
            (
            action x;
            marque.(x) <- true;
            do_list explorer g.(x)
            ) in
   do_list explorer s;;
```

Parcours en largeur

Etudions l'algorithme 2.2 dans lequel la file F est une file d'attente.[3] enfiler x F consiste donc à ajouter x à la fin de F et défiler F supprime et renvoie l'élément de début de F.

1. Voir la section 1.2.5 p. 23.
2. Le parcours en profondeur de G est le parcours préfixe de cet arbre au sens de la section 3.2.3 p. 52.
3. Voir section A.7.3 p. 106

Algorithme 2.2 Parcours en largeur d'un graphe

soit *parcours_en_largeur action* (S,A) $S' =$
  soit *marque* = tableau de booléens indexé par S **dans**
  pour $x \in S$ **faire** $marque(x) \leftarrow faux$
  soit F = file_vide **dans**
  soit *traiter* x =
    $action(x)$
    $marque(x) \leftarrow vrai$
    enfiler x F **dans**
  pour $x \in S'$ **faire** *traiter* x
  tant que $F \neq$ file_vide **faire**
    soit x = défiler F **dans**
    pour y extrémité d'un arc d'origine x **faire**
      si non $marque(y)$ **alors** *traiter* y

Comme un sommet n'est traité que s'il n'est pas marqué et que *traiter* x marque le sommet x, on voit déjà que $action(x)$ est éxécuté au plus une fois pour chaque sommet et que l'ensemble T des sommets traités est, à chaque instant, l'ensemble des sommets sur lesquels *action* a été éxécuté. Comme pour le parcours en profondeur, l'ensemble des sommets traités à la fin de l'éxécution de l'algorithme est l'ensemble $acc(S')$ des sommets accessibles depuis S'. La différence réside dans l'ordre de traitement des sommets.

Pour un sommet $x \in acc(S')$, appelons *distance* de S' à x le minimum $d(S',x)$ des longueurs des chemins d'origine appartenant à S' et d'extrémité x.

Le déroulement de l'algorithme sur le graphe de la figure 2.3 a avec $S' = \{0\}$, est illustré par la figure 2.4.

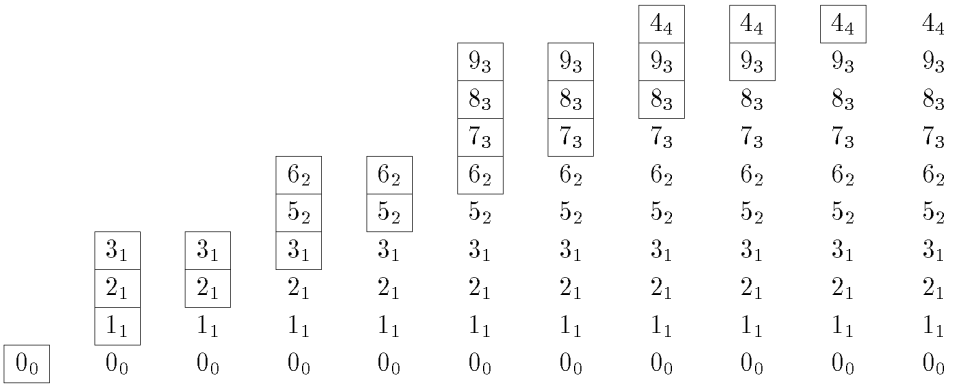


FIG. 2.4 – *Parcours en largeur du graphe de la figure 2.3 a. A chaque étape est représenté l'ensemble T des sommets traités; la file $F \subset T$ est symbolisée par un rectangle vertical (début de la file en bas et fin en haut). Chaque sommet comporte, en indice, sa distance à $S' = \{0\}$.*

Les sommets sont traités dans l'ordre croissant de leur distance à S' : 0_0, 1_1, 2_1, 3_1, 5_2, 6_2, 7_3, 8_3, 9_3, 4_4.

Proposition 2.3 *parcours_en_largeur action (S,A) S' éxécute $action(x)$ une et une seule fois pour chaque sommet x accessible depuis S' et dans un ordre tel que $d(S',x)$ est croissant. Sa complexité est $O(|S| + |A|)$.*

Preuve On démontre que les sommets traités sont exactement ceux de $acc(S')$ en vérifiant que la boucle satisfait l'invariant suivant :

- $S' \subset T \subset acc(S')$;
- $\forall x,y : \; x \in T \setminus F \;\wedge\; (x,y) \in A \Rightarrow y \in T$.

Notons $S_k = \{x \in acc(S') \,|\, d(S',x) = k\}$ et $k_{\max}$ la plus grande valeur de k pour laquelle S_k est non vide. On montre par récurrence que, pour tout $k \leqslant k_{\max}$, il existe une étape de l'algorithme où $T = \bigcup_{i \leqslant k} S_i$ et $F = S_k$, ce qui démontre l'affirmation de la proposition sur l'ordre de traitement des sommets.

Les sommets enfilés dans F durant le déroulement de l'algorithme sont ceux de $acc(S')$, et chacun de ces sommets est enfilé, et donc aussi défilé, une unique fois. La boucle **tant que** est donc exécutée $|acc(S')| \leqslant |S|$ fois (ce qui, en passant, fournit la preuve d'arrêt), le x du corps de la boucle décrivant l'ensemble $acc(S')$. De plus, si on note $A_x = \{y \mid (x,y) \in S\}$, le corps — ligne 13 — de la boucle des lignes 12 et 13 est éxécuté $|A_x|$ fois pour chaque $x \in acc(S')$. Au total, la ligne 13 est donc exécutée moins de $\sum_{x\in S} |A_x| = |A|$ fois, ce qui prouve la complexité en $O(|S| + |A|)$. ■

Exercice 2.6 Modifier le parcours en largeur pour qu'il renvoie le tableau des $d(S',x)$.

Exercice 2.7 En utilisant une pile, donner une version itérative du parcours en profondeur. Montrer que si, dans l'algorithme obtenu, on remplace la pile par une file d'attente, on obtient un parcours en largeur. Mais, à l'inverse, si, dans le parcours en largeur de l'algorithme 2.2, on remplace la file d'attente par une pile, montrer que l'algorithme obtenu n'est pas, en général, un parcours en profondeur.

Parcours des sommets coaccessibles d'un ensemble de sommets

Le graphe $G = (S,A)$ et $S' \subset S$ étant donnés, on se propose de parcourir l'ensemble $coacc(S')$ des sommets coaccessibles à partir d'un sommet de S'.

Une première méthode en $O(|S| + |A|)$ consiste à appliquer un parcours en profondeur ou en largeur à l'inverse $G^{-1} = (S,\{(y,x) \mid (x,y) \in A\})$ de G.[1]

L'algorithme 2.3 définit une autre méthode.

Algorithme 2.3 Parcours des sommets coaccessibles d'un ensemble de sommets

soit *parcours_des_coaccessibles action* (S,A) S' =
  soit t = tableau d'ensembles indexé par S
  et *marque* = tableau de booléens indexé par S **dans**
  pour $x \in S$ **faire** $t(x) \leftarrow \emptyset$
  pour $x \in S$ **faire** $marque(x) \leftarrow faux$
  soit *marquer* x =
    $action(x)$
    $marque(x) \leftarrow vrai$ **dans**
  soit récursivement *marquer_rec* x =
    marquer x
    pour $y \in t(x)$ **faire si** non $marque(y)$ **alors** *marquer_rec* y **dans**
  pour $x \in S'$ **faire** *marquer* x
  pour $x \in S \setminus S'$ **faire**
    si il existe y tel que $(x,y) \in A$ et $marque(y)$
      alors *marquer_rec* x
      sinon pour y tel que $(x,y) \in A$ **faire** $t(y) \leftarrow t(y) \cup \{x\}$

Proposition 2.4 *parcours_des_coaccessibles action (S,A) S' exécute $action(x)$ une et une seule fois pour chaque sommet x coaccessible depuis S'. Sa complexité est $O(d|S|)$ où $d = \max_{x\in S} |\{y \mid (x,y) \in A\}|$.*

Preuve Appelons sommet *traité* un sommet ou bien dans S' ou bien pour lequel le corps de la boucle principale (**pour** $x \in S \setminus S'$ etc.) a été éxécuté. On peut vérifier que la boucle principale admet l'invariant :

- tout sommet marqué est traité et appartient à $coacc(S')$;
- si $x \in t(y)$ alors x est traité et $(x,y) \in A$;
- si x est traité et non marqué alors, pour tout sommet y tel que $(x,y) \in A$, y n'est pas marqué et $x \in t(y)$.

On en déduit qu'à la fin de l'éxécution de la boucle, quand tous les sommets sont traités, les sommets marqués sont exactement ceux de $coacc(S')$.

1. Vérifier que le calcul de G^{-1} peut être effectué en $O(|S| + |A|)$.

Complexité : Si on ne tient pas compte des appels à *marquer_rec*, on obtient un temps d'éxécution en $O(d|S|)$. De plus le temps total des éxécutions de *marquer_rec* est dominé par la somme des cardinaux des ensembles $t(x)$ donc aussi par $d|S|$ car chaque sommet appartient à au plus d de ces ensembles. ■

Chapitre 3

Arbres

3.1 Arbres binaires

3.1.1 Définitions

Un ensemble E étant donné, on définit récursivement l'ensemble des *arbres binaires* sur E ou *étiquetés* par E à l'aide des deux règles suivantes :[1]

- nil est un arbre binaire sur E appelé l'*arbre vide*;
- si $x \in E$ et si U et V sont des arbres binaires sur E alors le triplet $T = (U,x,V)$ est un arbre binaire sur E; x, U et V sont appelés respectivement la *racine*, le *sous arbre gauche* et le *sous arbre droit* de l'arbre binaire T.

Notant A l'ensemble des arbres binaires sur E, cette définition peut aussi s'écrire :

$$A ::= \mathsf{nil} \mid A \times E \times A$$

Un arbre binaire sur un ensemble à un élément est dit *non étiqueté* ou *muet*.
Pour dessiner un arbre binaire, on écrit « nil » pour représenter l'arbre vide et, pour représenter l'arbre binaire (U,x,V), on dessine un cercle entourant x pour la racine, on dessine U (resp. V) en dessous et à gauche (resp. à droite) de la racine et on joint la racine au dessin de U (resp. V) par un segment de droite. En général, pour aérer le dessin, on ne fait pas figurer les arbres vides. Voir la figure 3.1.

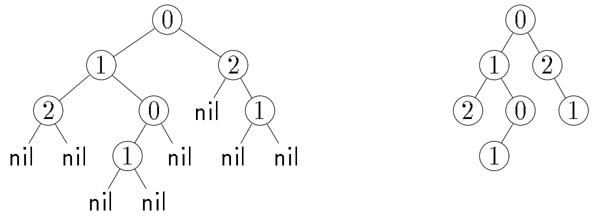


FIG. 3.1 – *Représentations complète et incomplète (sans figuration des arbres vides) de l'arbre binaire* $(((nil,2,nil),1,((nil,1,nil),0,nil)),0,(nil,2,(nil,1,nil)))$

Fonctions inductives

Vue la définition des arbres binaires, pour définir une application de l'ensemble A des arbres binaires sur E à valeurs dans un ensemble F, il suffit de connaître l'image de nil et de savoir exprimer l'image d'un arbre binaire (U,x,V) en fonction de x et des images de U et de V. Plus formellement, soient un élément a d'un ensemble F et une application $\varphi : F \times E \times F \to F$. On peut alors définir récursivement une application $f : A \to F$ par les règles :

- $f(\mathsf{nil}) = a$;
- $\forall U,V \in A, \forall x \in E, f(U,x,V) = \varphi(f(U),x,f(V))$.

[1] Cette définition et les notions qui suivent sont à rapprocher de la définition des listes donnée dans la section A.3.9 p. 98.

Une fonction f qui peut être définie de cette manière est dite *inductive*.

Si une fonction g est une fonction d'une fonction inductive f, on dit que f est un *prolongement inductif* de g.

Exemple 3.1 A tout arbre binaire T on associe par induction un langage[1] fini P_T sur l'alphabet à deux lettres $\{g,d\}$:

- $P_{\mathsf{nil}} = \mu$ le langage réduit au mot vide;
- $P_{(U,x,V)} = \mu + gP_U + dP_V = \{\mu\} \cup \{gu \mid u \in P_U\} \cup \{dv \mid v \in P_V\}$.

Un mot de P_T est appelé une *position* de T; par exemple
$P_T = \{\mu,g,gg,ggg,ggd,gd,gdg,gdgg,gdgd,gdd,d,dg,dd,ddg,ddd\}$ pour l'arbre binaire T de la figure 3.1.

Démonstrations par induction structurelle

Pour démontrer qu'une propriété $R(T)$ est vérifiée pour tout arbre binaire $T \in A$, on peut procéder par *induction* de la manière suivante :

- démontrer $R(\mathsf{nil})$;
- démontrer que $\forall U,V \in A, \forall x \in E, R(U) \wedge R(V) \Rightarrow R(U,x,V)$.

Exemple 3.2 Montrons par induction que, pour tout arbre binaire T, l'ensemble P_T des positions de T défini dans l'exemple 3.1 vérifie la propriété $R(P_T)$ suivante : tout préfixe d'un mot de P_T est dans P_T :

- Il est clair que le langage $P_{\mathsf{nil}} = \mu$ vérifie $R(\mu)$.
- Si U et V sont deux arbres binaires tels que $R(P_U)$ et $R(P_V)$, soit $u \in P_T = \mu + gP_U + dP_V$ et v un préfixe de u. Si $v = \mu$ alors $v \in P_T$. Sinon, supposons par exemple que $u \in gP_U$; alors $u = gu'$ où $u' \in P_U$ et v est de la forme $v = gv'$ où v' est un préfixe de u'. D'après l'hypothèse d'induction — $R(P_U)$ est vérifiée — le mot v' appartient à P_U; donc $v = gv' \in gP_U \subset P_T$. ■

Terminologie

Il est clair qu'un arbre binaire T sur E définit un arbre non ordonné T' étiqueté par $E \cup \{\mathsf{nil}\}$ au sens de la section 2.2.2 p. 31; il suffit pour s'en convaincre de comparer la représentation complète d'un arbre binaire (figure 3.1) et la représentation sagittale d'un arbre non ordonné (figure 2.2 p. 31). Tous les sommets internes de T' sont de degré 2 et les feuilles sont les sommets d'étiquette nil.[2]

Un *nœud* de l'arbre binaire T est un sommet interne de l'arbre non ordonné T'. Un nœud est donc étiqueté par un élément de E et les feuilles de T' (nil), qui sont appelées les *feuilles vides* de T ne sont donc pas considérées commes des nœuds de T.

Le *fils gauche* (resp. *droit*) d'un nœud N est la racine du sous arbre gauche de N si celui ci n'est pas vide, ou nil dans le cas contraire.

Une *feuille* est un nœud dont les deux fils sont nil. Un nœud qui n'est pas une feuille est un *nœud interne*.

La *taille* de T est le nombre de nœuds. Noter que les feuilles vides, n'étant pas des nœuds, ne sont pas comptées dans la taille.

La *hauteur* de T est hauteur$(T') - 1$. Si $T \neq \mathsf{nil}$ c'est donc le maximum des hauteurs des nœuds et la hauteur de nil est -1.

L'arbre binaire de la figure 3.1, de taille 7 et de hauteur 3, comporte 4 nœuds internes, 3 feuilles et 8 feuilles vides.

1. Voir la section 2.1 p. 29 pour les définitions relatives aux mots et aux langages.

2. On peut définir l'arbre non orienté T' de manière plus formelle: les sommets sont les positions de T définies dans l'exemple 3.1 et les arcs sont définis en utilisant l'exercice 2.3 p. 32.

3.1.2 Implémentation

Définition d'un type implémentant les arbres binaires

Pour définir un type CAML `'a arbrebinaire` représentant les arbres binaires sur un ensemble E lui-même représenté par le type `'a`, la définition récursive conduit naturellement à poser [1]

```
type 'a arbrebinaire = Nil | Noeud of 'a arbrebinaire * 'a * 'a arbrebinaire
```

Par exemple la valeur CAML `Noeud(Noeud(Nil,1,Nil),0,Nil)` représente l'arbre binaire sur $\mathbf{Z}$ $((\mathsf{nil},1,\mathsf{nil}),0,\mathsf{nil})$.

Implémentation d'une fonction inductive

La fonction qui, à un arbre binaire, associe sa taille est inductive; en effet : $\text{taille}(\mathsf{nil}) = 0$ et $\text{taille}(U,x,V) = 1 + \text{taille}(U) + \text{taille}(V)$.

La fonction hauteur est aussi inductive :
$\text{hauteur}(\mathsf{nil}) = -1$ et $\text{hauteur}(U,x,V) = 1 + \max(\text{hauteur}(U),\text{hauteur}(V))$;

Et aussi la fonction feuilles qui renvoie le nombre de feuilles :
$\text{feuilles}(\mathsf{nil}) = 0$ et $\text{feuilles}(U,x,V) = \delta + \text{feuilles}(U) + \text{feuilles}(V)$ où $\delta = 1$ ou 0 suivant que $U = V = \mathsf{nil}$ ou non.

Une fonction inductive s'implémente de manière quasi automatique. Le programme 3.1 fournit trois fonctions CAML de type `'a arbrebinaire -> int` qui implémentent les trois fonctions précédentes.

Programme 3.1 Taille, hauteur et nombre de feuilles d'un arbre binaire

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

let rec taille = function
   Nil -> 0 | Noeud(u,_,v) -> 1 + taille u + taille v;;

let rec hauteur = function
    Nil -> -1 | Noeud(u,_,v) -> 1 + max (hauteur u) (hauteur v);;

let rec feuilles = function
   Nil -> 0 |
   Noeud(Nil,_,Nil) -> 1 |
   Noeud(u,_,v) -> feuilles u + feuilles v;;
```

Un arbre binaire est dit *localement complet* si aucun des deux sous arbres d'un nœud interne n'est vide; il est dit *complet* si toutes les feuilles sont à la même hauteur; voir la figure 3.2.

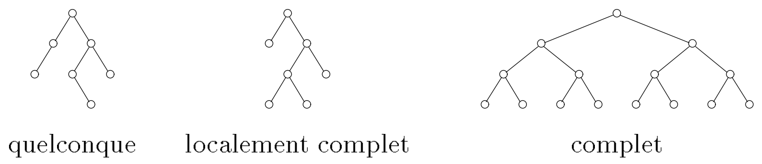


FIG. 3.2 – *(Locale) complétude d'un arbre binaire*

Pour tester qu'un arbre binaire (U,x,V) est localement complet, on vérifie que U et V sont localement complets tous deux nil ou tous deux $\neq \mathsf{nil}$. La fonction qui teste si un arbre binaire est localement complet n'est pas inductive car on ne peut pas décider de sa valeur en (U,x,V) quand on ne connait que sa valeur en U et en V; il faut aussi savoir si $U = \mathsf{nil}$ et si $V = \mathsf{nil}$. Mais on peut l'implémenter aussi facilement qu'une fonction inductive [2].

1. Voir la section A.3.12 p. 100 pour l'utilisation des types sommes.

2. Plus formellement, on obtient un prolongement inductif en considérant la fonction qui, à un arbre binaire T, associe le couple de booléens (T localement complet,T=nil).

La fonction « est_complet » qui teste si un arbre binaire est complet n'est pas inductive mais un arbre (U,x,V) est complet ssi U et V sont complets de même hauteur : la fonction qui à un arbre binaire T associe le couple (est_complet(T),hauteur(T)) est un prolongement inductif de la fonction est_complet.

Le programme 3.2 implémente ces deux fonctions.

Programme 3.2 Tests de complétude locale et de complétude d'un arbre binaire

```
let rec est_localement_complet = function (* 'a arbrebinaire -> bool *)
   Nil | Noeud(Nil,_,Nil) -> true |
   Noeud(Nil,_,_) | Noeud(_,_,Nil) -> false |
   Noeud(u,_,v) -> est_localement_complet u & est_localement_complet v;;

let est_complet t = (* est_complet : 'a arbrebinaire -> bool *)
   let rec aux = function (* aux(t) = (est_complet(t),hauteur(t))*)
      Nil -> true,-1 |
      Noeud(u,_,v) -> let ecu,hu = aux u and ecv,hv = aux v in
                      ecu & ecv & hu = hv , 1 + max hu hv in
   fst (aux t);;
```

Complexité d'une fonction inductive

Chacune des cinq fonctions des programmes 3.1 et 3.2 a une complexité en $O(n)$ où n est la taille de l'arbre binaire argument. De manière générale, on a :

Proposition 3.1 *La complexité d'une fonction f de la forme*
`let rec` f `= function Nil ->` a `| Noeud(`u`,`x`,`v`) ->` $\varphi(f(u),x,f(v))$
est O(taille), pourvu qu'un appel de la fonction φ ait une complexité O(1).

Preuve Une première démonstration consiste à remarquer que l'arbre des appels récursifs [1] de l'éxécution de $f(T)$ a la même structure [2] et donc la même taille que l'arbre T lui-même. La démonstration suivante se généralise à des situations plus compliquées [3]. Notant τ_T la complexité de l'appel $f(T)$, on montre l'existence de deux constantes A et B telles que la propriété $\tau_T \leqslant A\text{taille}(T) + B$ soit inductive [4] :
– pour que $\tau_{\mathsf{nil}} \leqslant A\text{taille}(\mathsf{nil}) + B$, il suffira de choisir $B \geqslant \tau_{\mathsf{nil}}$;
– si $\tau_U \leqslant A\text{taille}(U) + B$ et $\tau_V \leqslant A\text{taille}(V) + B$ alors, si C désigne un majorant de la complexité d'un appel de φ, $\tau_{(U,x,V)} \leqslant \tau_U + \tau_V + C \leqslant A\text{taille}(U) + B + A\text{taille}(V) + B + C = A\text{taille}(T) - A + 2B + C = A\text{taille}(T) + B$ si on choisit $A = B + C$. ■

Exercice 3.1 La fonction `f` suivante utilise la fonction `hauteur` du programme 3.1 pour tester la complétude d'un arbre binaire :

```
let rec f = function
   Nil -> true | Noeud(u,_,v) -> f u & f v & hauteur u = hauteur v;;
```

Montrer qu'elle est de complexité O(taille^2) et moins efficace que la fonction est_complet du programme 3.2.

Exercice 3.2 Dans les conditions de la proposition 3.1, on suppose que l'évaluation de $\varphi(f(u),x,f(v))$ ne nécessite l'évaluation que d'une seule des deux valeurs $f(u)$ ou $f(v)$. Montrer que la complexité devient O(hauteur).

3.1.3 Dénombrements dans les arbres binaires

Liens entre la taille, la hauteur et le nombre de feuilles dans un arbre binaire

Proposition 3.2 *Soient n, h et f la taille, la hauteur et le nombre de feuilles d'un arbre binaire T.*

1. *$f \leqslant (n+1)/2$ et $f = (n+1)/2$ ssi T est localement complet $\neq$ nil;*

---

1. Voir la section 1.2.5 p. 23.
2. La *structure* d'un arbre binaire T est l'arbre binaire muet obtenu à partir de T en supprimant les étiquettes.
3. Voir l'exercice 3.1.
4. Une propriété est *inductive* si on peut la démontrer par induction.

2. *$n+1 \leqslant 2^{h+1}$ et $n+1 = 2^{h+1}$ ssi T est complet;*
3. *$h \leqslant n-1$ et $h = n-1$ ssi chaque nœud a au moins un fils vide;*
4. *le nombre de feuilles vides est $n+1$.*

Preuve 1. On procède par induction :

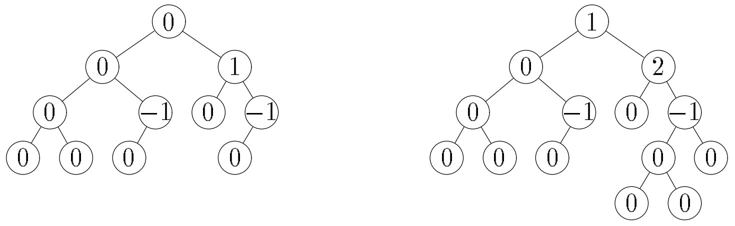
- Si $T = \mathsf{nil}$, $f = 0 < (n+1)/2 = 1/2$ et T n'est pas localement complet $\neq \mathsf{nil}$.
- Si la propriété est vérifiée pour les arbres binaires U et V, soit $T = (U,x,V) \neq \mathsf{nil}$:
  - si $U = V = \mathsf{nil}$, $f_T = 1 = (n_T+1)/2$ et T est localement complet;
  - si $U = \mathsf{nil} \neq V$, $f_T = f_V \leqslant (n_V+1)/2 = n_T/2 < (n_T+1)/2$ et T n'est pas localement complet;
  - si U et V sont $\neq \mathsf{nil}$, $f_T = f_U + f_V \leqslant (n_U+1)/2 + (n_V+1)/2 = (n_T+1)/2$ et $f_T = (n_T+1)/2 \Leftrightarrow f_U = (n_U+1)/2 \wedge f_V = (n_V+1)/2 \Leftrightarrow U$ et V sont localement complets $\Leftrightarrow T$ est localement complet.

2., 3. et 4. sont laissés en exercice. ■

Arbres binaires équilibrés

Si n est la taille et h la hauteur d'un arbre binaire, la propriété 3.2 2 montre que $\log n = \mathrm{O}(h)$. Une famille $\mathcal{F}$ d'arbres binaires est dite *équilibrée* si $h = \mathrm{O}(\log n)$ pour tous les arbres de la famille $\mathcal{F}$. Par exemple, la famille des arbres binaires complets est équilibrée car $h = \log_2(n+1) - 1$ pour un arbre binaire complet; mais la plupart des arbres binaires que l'on rencontre dans la pratique ne sont pas complets. On introduit ici une famille équilibrée plus vaste.

Le *déséquilibre* d'un arbre binaire non vide $T = (U,x,V)$ est défini par $\text{déséquilibre}(T) = \text{hauteur}(V) - \text{hauteur}(U)$. On dit que T est un *arbre AVL*[1] si le déséquilibre de tout sous arbre de T est égal à 0, 1 ou -1; voir figure 3.3.



AVL — non AVL

FIG. 3.3 – *Deux arbres binaires. L'étiquette d'un nœud est le déséquilibre de ce nœud. Le deuxième arbre n'est pas AVL car il contient l'étiquette* 2.

Exercice 3.3 Ecrire une fonction CAML qui teste, avec une complexité $\mathrm{O}(\text{taille}(T))$, si un arbre binaire T est AVL.[2]

Proposition 3.3 *La famille des arbres AVL est équilibrée.*

Preuve Pour $h \geqslant 1$, soit $N(h)$ le minimum des tailles des arbres AVL de hauteur h et soit $E(h)$ l'ensemble (non vide) des arbres AVL de hauteur h et de taille minimum $N(h)$.

Soit $T = (U,x,V) \in E(h)$ $(h \geqslant 1)$.

Alors un des deux arbres U ou V, disons U, est de hauteur $h-1$.

$U \in E(h-1)$ car sinon un arbre (U',x,V) avec $U' \in E(h-1)$ serait AVL, de hauteur h et de taille $<$ à $N(h) = \text{taille}(T)$.

Ainsi $N(h) = 1 + N(h-1) + \text{taille}(V)$ (noter que cela implique que N est strictement croissante).

Si $h \geqslant 2$, $V \in E(h-2)$ car sinon, que V soit de hauteur $h-1$ ou $h-2$, un arbre (U,x,V') avec $V' \in E(h-2)$ serait AVL, de hauteur h et de taille $<$ à $N(h)$.

Finalement $N(h) = 1 + N(h-1) + N(h-2)$ avec $N(0) = 1$ et $N(1) = 2$ d'où l'on déduit par récurrence que $N(h) > \left(\frac{3}{2}\right)^h$ puis $h < \log_{3/2} N(h) \leqslant \log_{3/2} n$. ■

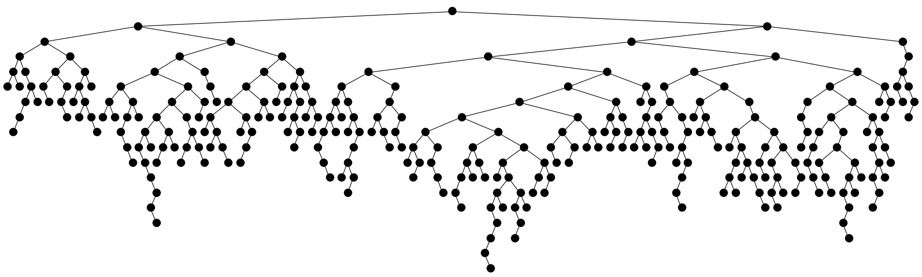


FIG. 3.4 – *Un arbre binaire aléatoire de taille* 300

Hauteur moyenne d'un nœud dans un arbre binaire construit aléatoirement

On se propose de résoudre le problème suivant : on choisit au hasard (à préciser) un nœud dans un arbre binaire de taille n lui-même choisi au hasard (à préciser). Quelle est l'espérance h_n de la hauteur du nœud?

Le choix au hasard de l'arbre s'effectue par *adjonction aux feuilles* de la manière suivante : on part d'un arbre binaire de taille 1; on choisit une des 2 feuilles vides — avec la probabilité 1/2 pour chacune — que l'on remplace par un nœud; dans l'arbre de taille 2 obtenu, on choisit une des 3 feuilles vides — avec la probabilité 1/3 pour chacune — que l'on remplace par un nœud; et ainsi de suite jusqu'à obtenir un arbre de taille n.

Le choix du nœud dans l'arbre de taille n se fait avec la probabilité $1/n$ pour chaque nœud.

Il est important de noter que cette manière de choisir au hasard un arbre binaire de taille n définit une distribution de probabilité sur l'ensemble de ces arbres qui n'est pas uniforme : tous les arbres binaires de taille n n'ont pas la même probabilité d'être choisis. Par exemple pour $n = 3$, l'arbre binaire complet de taille 3 a la probabilité 1/3 d'être choisi contre 1/6 pour chacun des 4 autres arbres binaires de taille 3; voir la figure 3.5.

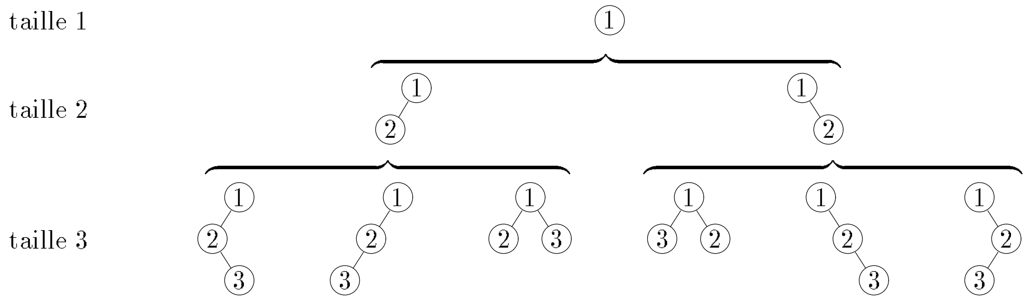


FIG. 3.5 – *Arbres numérotés de taille* 1, 2 *et* 3

Proposition 3.4 *La hauteur moyenne d'un nœud d'un arbre binaire de taille n construit aléatoirement est $\sim 2 \ln n$.*

Preuve Les arbre binaires étiquetés par $\mathbf{N}^*$ de la figure 3.5 seront appelés les arbres numérotés de taille 1, 2 et 3. Plus précisément,

- il y a un unique arbre numéroté de taille 1, c'est (nil,1,nil);
- un arbre numéroté de taille n s'obtient à partir d'un arbre numéroté de taille $n-1$ en remplaçant une des n feuilles vides par un nœud d'étiquette n.

Il y a $n!$ arbres numérotés de taille n et choisir au hasard un arbre binaire muet de taille n par adjonction aux feuilles est équivalent à choisir au hasard un arbre numéroté, chacun des $n!$ arbres numérotés ayant la probabilité $1/n!$ d'être choisi.
On note H_T la somme des hauteurs des nœuds d'un arbre binaire T de taille n de sorte que la hauteur moyenne d'un nœud de T est H_T/n et que la hauteur moyenne d'un nœud d'un arbre binaire de taille n construit aléatoirement est $h_n = H_n/(n.n!)$ où H_n est la somme des H_T quand T décrit l'ensemble A_n des arbres numérotés de taille n.

1. Des noms de G.M. ADELSON-VELSKY et E.M. LANDIS.
2. On pourra s'inspirer de la fonction `est_complet` du programme 3.2.

Notons de même K_T la somme des hauteurs des $n+1$ feuilles vides d'un arbre binaire T et K_n la somme des K_T quand T décrit A_n.

Si f est une feuille de $U \in A_{n-1}$, soit $T(U,f)$ l'arbre numéroté de taille n obtenu en remplaçant dans U la feuille f par un nœud d'étiquette n. $H_{T(U,f)} = H_U + p(f)$ où $p(f)$ désigne la hauteur de f; donc

$$(1)\quad H_n = \sum_{T \in A_n} H_T = \sum_{\substack{U \in A_{n-1} \\ f \text{ feuille de } U}} H_{T(U,f)} = \sum_{\substack{U \in A_{n-1} \\ f \text{ feuille de } U}} (H_U + p(f)) = nH_{n-1} + K_{n-1}$$

De même $(2)\quad K_n = \sum_{\substack{U \in A_{n-1} \\ f \text{ feuille de } U}} K_{T(U,f)} = \sum_{\substack{U \in A_{n-1} \\ f \text{ feuille de } U}} (K_U + p(f) + 2) = (n+1)K_{n-1} + 2n!$

De (2) et de $K_1 = 2$, on tire $\frac{K_n}{(n+1)!} = 2\sum_{i=2}^{n+1} \frac{1}{i}$. En reportant dans (1) et sachant que $H_1 = 0$, on trouve $\frac{H_n}{n!} = 2\sum_{k=2}^{n}\sum_{i=2}^{k} \frac{1}{i} = 2(n+1)\sum_{i=2}^{n} \frac{1}{i} - 2(n-1) \sim 2n \ln n$. ■

3.1.4 Parcours préfixe, infixe et postfixe d'un arbre binaire

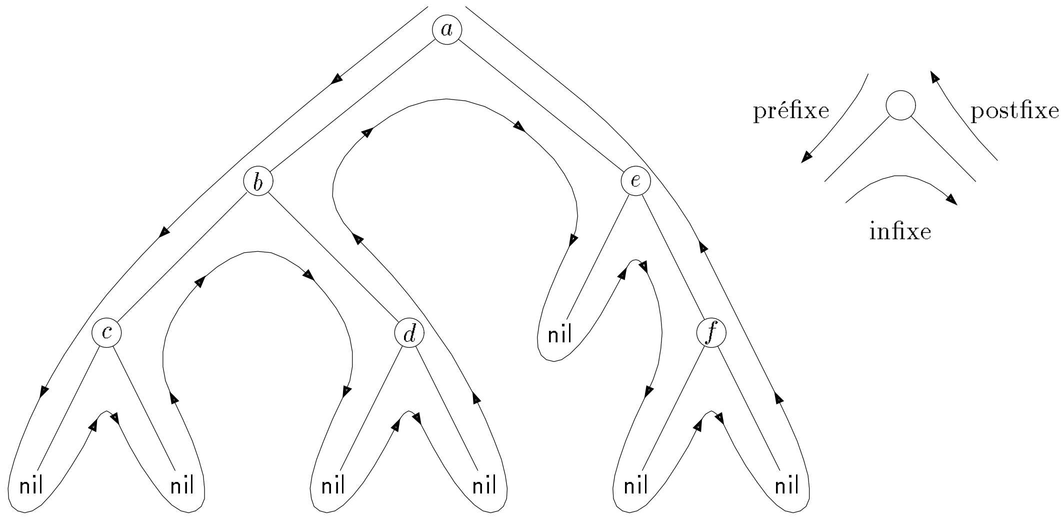


FIG. 3.6 – *Parcours d'un arbre binaire*

De manière informelle, effectuer un *parcours en profondeur* d'un arbre binaire T consiste à faire le tour de la représentation complète de T en partant à gauche de la racine tout en restant à tout moment au plus près de l'arbre. Le chemin fléché de la figure 3.6 illustre un tel parcours. Durant le parcours chaque nœud est rencontré trois fois et l'on peut décider d'éxécuter une action à chaque rencontre d'un nœud, l'action dépendant du nœud et aussi du type de la rencontre (première, deuxième ou troisième).

Plus formellement, si T est un arbre binaire sur un ensemble E et si $action\_préfixe$, $action\_infixe$ et $action\_postfixe$ sont trois actions éxécutables dépendant de la donnée d'un élément de E, le parcours en profondeur de T pour ces actions est défini récursivement par

- $parcours(\mathsf{nil})$ = ne rien faire;
- $parcours(U,x,V)$ =

  éxécuter $action\_préfixe(x)$

  $parcours(U)$ (parcourir le sous arbre gauche)

  éxécuter $action\_infixe(x)$

  $parcours(V)$ (parcourir le sous arbre droit)

  éxécuter $action\_postfixe(x)$

Si, dans un parcours en profondeur, on choisit une action donnée pour action préfixe (resp. infixe, postfixe) et des actions vides pour les deux autres, on obtient la notion de *parcours préfixe* (resp. *infixe*, *postfixe*) d'un arbre binaire. Le programme 3.3 fournit une fonction `parcours_préfixe` (resp. `parcours_infixe`, `parcours_postfixe`) de type
`('a -> 'b) -> 'a arbrebinaire -> unit`.
`parcours_`... *action t* effectue le parcours préfixe (resp. infixe, postfixe) de t pour l'action *action*.

Programme 3.3 Parcours préfixe, infixe et postfixe d'un arbre binaire

```
let parcours_préfixe action =
   let rec pref = function
      Nil -> () | Noeud(u,x,v) -> action x; pref u; pref v in
   pref
and parcours_infixe action =
   let rec inf = function
      Nil -> () | Noeud(u,x,v) -> inf u; action x; inf v in
   inf
and parcours_postfixe action =
   let rec post = function
      Nil -> () | Noeud(u,x,v) -> post u; post v; action x; () in
   post;;
```

D'après la proposition 3.1 p. 42, si la fonction `action` a une complexité O(1), chacune des trois fonctions de parcours a une complexité O(taille de l'arbre).

Si l'on considère les nœuds d'un arbre binaire dans l'ordre où ils sont rencontrés lors d'un parcours préfixe (resp. infixe, postfixe), on définit un ordre total sur les nœuds appelé l'*ordre préfixe* (resp. *infixe*, *postfixe*). Pour l'arbre binaire de la figure 3.6 :

- $a < b < c < d < e < f$ pour l'ordre préfixe;
- $c < b < d < a < e < f$ pour l'ordre infixe;
- $c < d < b < f < e < a$ pour l'ordre postfixe.

Si l'on concatène les étiquettes des nœuds rencontrés lors d'un parcours préfixe (resp. infixe, postfixe) d'un arbre binaire T, on obtient un mot $\mathrm{Pref}(T)$ (resp. $\mathrm{Inf}(T)$, $\mathrm{Post}(T)$) sur l'alphabet E appelé le *mot du parcours préfixe* (resp. *infixe*, *postfixe*) de T. Pour l'arbre de la figure 3.6, on obtient le mot $abcdef$ (resp. $cbdaef$, $cdbfea$). Ces mots admettent les définitions récursives :

- $\mathrm{Pref}(\mathsf{nil}) = \mu$, $\mathrm{Pref}(U,x,V) = x\mathrm{Pref}(U)\mathrm{Pref}(V)$;
- $\mathrm{Inf}(\mathsf{nil}) = \mu$, $\mathrm{Inf}(U,x,V) = \mathrm{Inf}(U)x\mathrm{Inf}(V)$;
- $\mathrm{Post}(\mathsf{nil}) = \mu$, $\mathrm{Post}(U,x,V) = \mathrm{Post}(U)\mathrm{Post}(V)x$;

Par exemple, si t est une valeur CAML de type `char arbrebinaire`,
`parcours_préfixe print_char` t affiche $\mathrm{Pref}(t)$ à l'écran.

3.1.5 Arbres binaires de recherche

Structures de données dynamiques de type dictionnaire

On s'intéresse ici à une famille importante de structures de données dynamiques[1] dont les arbres binaires de recherche sont un cas particulier.

Soit un ensemble E tel qu'à chaque élément x de E soit associé une *clé* qui appartient à un ensemble Z muni d'un ordre total. clé : $E \to Z$. Considérons une structure de données dynamique qui permette de représenter les sous ensembles finis X de E.[2]

1. Voir p. 104 pour une introduction à la notion de structure de données.
2. Dans certains cas, un même élément de E peut appartenir « plusieurs fois » à X. On dit alors que X est un *multiensemble*. De manière précise, un multiensemble d'éléments de E est une application X de E dans $\mathbf{N}$; on dit que $x \in X$ si $X(x) \neq 0$, auquel cas $X(x)$ est appelé la *multiplicité* de x dans X.

En général, l'ensemble des opérations que doit supporter la structure de données est contenu dans la liste suivante :

- *recherche*(X,c) recherche si X contient un élément de clé c et, éventuellement, le retourne;
- *insertion*(X,x) ajoute x à X;[1]
- *suppression*(X,c) élimine de X un élément de clé c s'il y en a;
- *minimum*(X) retourne un élément de X de clé minimum et, éventuellement, le supprime de X.
- *maximum*(X) idem;
- *successeur*(X,c) retourne un élément de X de clé la plus petite possible $\geqslant c$, avec diverses conventions possibles;
- *prédécesseur*(X,c) : idem;
- *construire*(ℓ) construit la structure de données représentant l'ensemble des éléments d'une liste donnée ℓ d'éléments de E.[2]

Si on n'impose qu'un nombre restreint d'opérations à une structure de données, il est clair que, en échange, ces quelques opérations, devront être performantes.

Une structure de données qui supporte les opérations de recherche, d'insertion et de suppression est appelée un *dictionnaire*.

Une structure de données qui supporte les opérations d'insertion et de maximum avec suppression est appelée une *file* (ou *queue*) *de priorité*.

Définition

L'ensemble des étiquettes d'un arbre binaire T sur E est une partie X de E. On dit que T est un *arbre binaire de recherche* ou un ABR (resp. un ABR *strict*) si pour tout nœud N de T d'étiquette x, les étiquettes des nœuds du sous arbre gauche de N ont une clé $\leqslant$ (resp. $<$) à la clé de x et les étiquettes des nœuds du sous arbre droit de N ont une clé $\geqslant$ à celle de x; voir la figure 3.7.

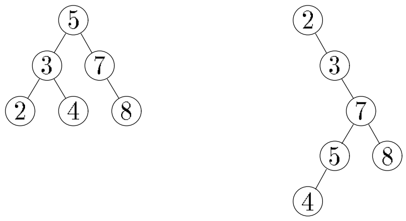


FIG. 3.7 – *Deux ABR représentant le même ensemble; on notera que le premier est mieux équilibré que le second.*

Exercice 3.4 Montrer qu'un parcours infixe d'un ABR décrit les nœuds dans l'ordre croissant de leurs clés.

IMPLÉMENTATION : Un ABR T sur E est représenté par le type `'a arbrebinaire` défini p. 41; la fonction clé est implémentée par le type `'a -> 'b`.[3]

Recherche dans un ABR

Pour rechercher un élément de clé $c \in Z$ donné dans un ABR $T = (U,x,V)$, on compare c à la clé de x :

- si clé$(x) = c$ on renvoie x;

1. Il y a des variantes: on peut par exemple choisir de n' ajouter réellement x que s'il ne figure pas déjà dans X ou encore s'il n'y a pas déjà dans X un élément de même clé que x.

2. En général *construire*(ℓ) s'implémente par autant d'appels à *insertion* qu'il y a d'éléments dans ℓ, mais, dans certains cas, la construction est plus rapide; par exemple si on représente X par une liste non triée, *construire*(ℓ) se contente de retourner ℓ.

3. Rappelons — voir p. 96 — que les opérateurs de comparaison CAML fonctionnent correctement sur chacun des types `int`, `float` ou `string`; on pourra donc se borner à l'un de ces types pour `'b`.

– si clé(x) $> c$ on continue la recherche dans U car s'il y a un élément de clé c dans T, il ne peut se trouver dans V;
– si clé(x) $< c$ on continue la recherche dans V.

Ces considérations permettent d'écrire la fonction `recherche` du programme 3.4.

Programme 3.4 Recherche dans un ABR

```
let recherche clé t c =
(* recherche : ('a -> 'b) -> 'a arbrebinaire -> 'b -> 'a *)
(* recherche clé t c = un élément de t de clé c ou déclenche Not_found *)
   let rec rech = function
   (* rech : 'a arbrebinaire -> 'a.  rech t = recherche clé t c *)
      Nil -> raise Not_found |
      Noeud(u,x,v) ->
         let cléx = clé x in
         if cléx = c then x else if cléx > c then rech u else rech v in
   rech t;;
```

Cette fonction parcourt l'arbre en partant de la racine et en descendant vers les feuilles jusqu'à trouver un élément de clé c. Si un tel élément ne se trouve pas dans l'arbre, le parcours se termine sur une feuille vide et une exception est déclenchée.

Complexité : O(hauteur); voir ex. 3.2 p. 42.

Insertion dans un ABR

On décrit deux techniques d'insertion.

La première, appelée *insertion aux feuilles*, est basée sur l'observation que, pour ajouter un élément y à un ABR $T = (U,x,V)$, il suffit d'ajouter y à U si la clé de y est $<$ à celle de x et à V dans le cas contraire; on obtient ainsi la fonction `insère_feuilles` du programme 3.5.

Programme 3.5 Insertion dans un ABR

```
let insère_feuilles clé t y =
(*  ('a -> 'b) -> 'a arbrebinaire -> 'a -> 'a arbrebinaire *)
(*  renvoie un ABR contenant les éléments de t et y        *)
   let c = clé y in
   let rec ins = function
   (* ins : 'a arbrebinaire -> 'a arbrebinaire. ins t = insère clé t y *)
      Nil -> Noeud(Nil,y,Nil) |
      Noeud(u,x,v) ->
         if c < clé x then Noeud(ins u,x,v) else Noeud(u,x,ins v) in
   ins t;;

let insère_racine clé t y =
   let c = clé y in
   let rec partitionne = function
   (* 'a arbrebinaire -> 'a arbrebinaire * 'a arbrebinaire           *)
   (* partitionne t = (u,v) : u et v sont des ABR; les éléments de u *)
   (* (resp. v) sont les éléments de t de clé < à c (resp. >= c).    *)
      Nil -> Nil,Nil |
      Noeud(u,x,v) ->
         if clé x < c then let v1,v2 = partitionne v in Noeud(u,x,v1),v2
                      else let u1,u2 = partitionne u in u1,Noeud(u2,x,v) in
   let u,v = partitionne t in
   Noeud(u,y,v);;
```

La fonction parcourt l'arbre en partant de la racine et en descendant jusqu'à une feuille vide qui est remplacée par y.

COMPLEXITÉ : O(hauteur).

La seconde technique d'insertion d'un élément y dans un ABR T, appelée *insertion à la racine*, consiste à partitionner T en deux ABR U et V tels que U contienne les éléments de T de clé $<$ à la clé de y et V les autres; il suffit alors de renvoyer l'ABR (U,y,V). Voir la fonction`insère_racine` du programme 3.5. La complexité est encore O(hauteur).

Suppression dans un ABR

Pour supprimer d'un ABR $T = (U,x,V)$ un élément de clé c donnée :

- si $c <$ clé(x), on supprime un élément de clé c dans U;
- si $c >$ clé(x), on supprime un élément de clé c dans V;
- si $c =$ clé(x) :
  - si U (resp. V) est nil, on renvoie V (resp. U);
  - si U et V sont $\neq$ nil, soit V' un ABR obtenu en supprimant de V un élément y de clé minimale (on construit V' en partant de la racine de V et en descendant à gauche tant que c'est possible); on renvoie l'ABR (U,y,V').

Voir le programme 3.6.

Programme 3.6 Suppression dans un ABR

```
let supprime clé t c =
(* ('a -> 'b) -> 'a arbrebinaire -> 'b -> 'a arbrebinaire                *)
(* supprime clé t c supprime de t un élt. de clé c ou déclenche Not_found *)
   let rec supprime_min = function
   (* 'a arbrebinaire -> 'a * 'a arbrebinaire *)
   (* supprime_min t = (x,t') où x est un élément de clé minimum de t et  *)
   (* t' est un ABR contenant les éléments de t à l'exception de x        *)
      Nil -> raise Not_found |
      Noeud(Nil,x,v) -> x,v |
      Noeud(u,x,v) -> let y,u' = supprime_min u in y,Noeud(u',x,v) in
   let rec supp = function
   (* supp : 'a arbrebinaire -> 'a arbrebinaire. supp t = supprime clé t c*)
      Nil -> raise Not_found |
      Noeud(u,x,v) ->
         let cléx = clé x in
         if cléx = c
            then if v = Nil then u
                 else if u = Nil then v
                 else let y,v'= supprime_min v in Noeud(u,y,v')
            else if cléx > c then Noeud(supp u,x,v)
                             else Noeud(u,x,supp v)  in
  supp t;;
```

COMPLEXITÉ : O(hauteur).

Preuve La suppression se compose d'une première phase de recherche de l'élément N à supprimer qui consiste à parcourir le chemin menant de la racine à N et d'une deuxième phase consistant à chercher l'élément de clé minimum dans la branche droite de N; dans cette deuxième phase, on descend encore dans l'arbre. Au total, le nombre d'opérations est dominé par la hauteur de l'arbre.
Plus formellement, si $t(T)$ est la complexité maximum d'une recherche dans T, en désignant par $h(T)$ la hauteur de T, il existe des constantes A, B et C telles que $t(\mathsf{nil}) \leqslant A$ et $t(U,x,V) \leqslant B + \max(t(U),t(V),Ch(V))$. On en déduit par induction que $t(T) \leqslant Dh(T) + A + D$ avec $D = \max(B,C)$. ∎

Exercice 3.5 Vérifier que les fonctions d'insertion et de suppression des programmes 3.5 et 3.6 produisent des ABR stricts quands ils sont appliqués à des ABR stricts.

Conclusion

La figure 3.8 illustre les opérations sur les ABR.

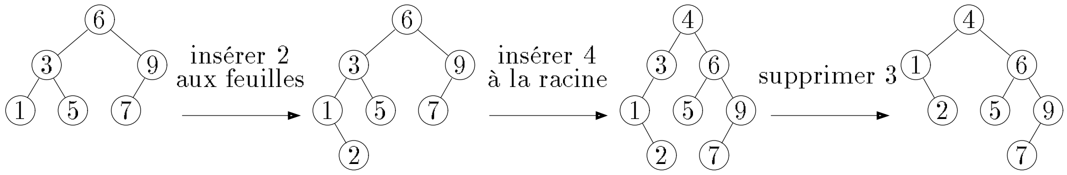


FIG. 3.8 – *Insertion aux feuilles, insertion à la racine et suppression dans un ABR.*

Les ABR forment une structure de données de type dictionnaire. Les trois algorithmes de base ont une complexité $O(hauteur)$ et sont donc plus performants quand ils sont appliqués à des arbres correctement équilibrés.

Si, dans un arbre initialement vide, on insère successivement n éléments dans un ordre aléatoire, la proposition 3.4 p. 44 montre que la complexité moyenne d'une recherche dans l'arbre obtenu est de l'ordre de $\log n$. Mais l'hypothèse de l'ordre d'insertion aléatoire est rarement vérifiée et si, à la limite, on insère les éléments dans l'ordre de leurs clés, on obtient un arbre peigne dont la hauteur est égale à la taille -1.[1]

3.2 Arbres ordonnés

3.2.1 Définitions

On définit récursivement l'ensemble des *arbres*[2] et l'ensemble des *forêts*[3] *étiquetés* par un ensemble E à l'aide des deux règles suivantes :

- si $x \in E$ et si F est une forêt alors le couple $T = (x,F)$ est un arbre;
- si $U_1,\dots,U_n$ sont n arbres ($n \geqslant 0$), le n-uplet $F = (U_1,\dots,U_n)$ est une forêt.

On notera l'existence de la forêt vide () constituée de 0 arbres. Par contre il n'y a pas de notion d'arbre vide; les plus petits arbres sont de la forme $T = (x,())$ avec $x \in E$.

Pour dessiner des arbres et des forêts on adopte des conventions analogues à celles des dessins d'arbres binaires; la figure 3.9 présente l'arbre

$$(a,((b,((c,((d,()),(e,())))),(f,()),(g,((h,()),(i,((j,()),(k,()))),(l,()),(m,((n,()))))))))$$

et la forêt non étiquetée[4] (U_1,U_2,U_3,U_4) avec

$U_1 = (\bullet,((\bullet,((\bullet,()))),(\bullet,((\bullet,((\bullet,()))),(\bullet,()))),(\bullet,()))),$
$U_2 = (\bullet,()),$
$U_3 = (\bullet,((\bullet,((\bullet,()),(\bullet,((\bullet,()))))),(\bullet,())))$ et
$U_4 = (\bullet,((\bullet,((\bullet,()),(\bullet,()),(\bullet,()),(\bullet,()),(\bullet,())))))$.

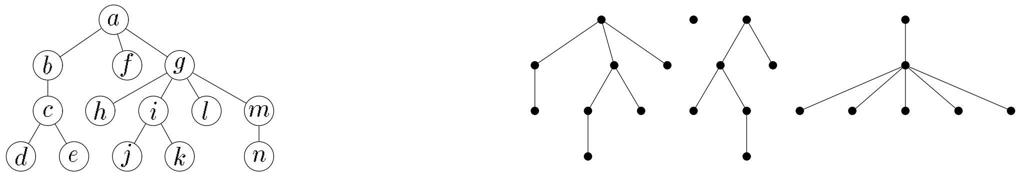


FIG. 3.9 – *Un arbre étiqueté et une forêt muette.*

Fonctions inductives et démonstrations par induction

Notons respectivement $\mathcal{A}$ et $\mathcal{F}$ l'ensemble des arbres et des forêts sur E. P et Q désignant deux ensembles quelconques, pour définir deux applications $f : \mathcal{A} \to P$ et $g : \mathcal{F} \to Q$, il

1. Il est possible de modifier les algorithmes d'insertion et de suppression de manière à ce qu'ils ne produisent que des arbres équilibrés (arbres AVL, arbres rouge et noir, etc.). Dans ces conditions, les complexités au pire des trois opérations deviennent O(log(taille)).
2. On dit aussi *arbres ordonnés* par opposition aux arbres non ordonnés définis p. 31.
3. Ou *forêts ordonnées*.
4. Quand l'ensemble E n'a qu'un élément, on parle d'arbres et de forêts *non étiquetés* ou *muets*.

suffit de savoir exprimer d'une part $f(x,F)$ en fonction de x et de $g(F)$ et d'autre part $g(U_1,\ldots,U_n)$ en fonction des $f(U_i)$. On dit alors que chacune des fonctions f et g est *inductive*.

Par exemple, on peut définir $h_{\mathcal{A}} : \mathcal{A} \to \mathbf{N}$ et $h_{\mathcal{F}} : \mathcal{F} \to \mathbf{Z}$ par $h_{\mathcal{A}}(x,F) = 1 + h_{\mathcal{F}}(F)$ et $h_{\mathcal{F}}(U_1,\ldots,U_n) = \max_{i=1,\ldots,n} h_{\mathcal{A}}(U_i)$ en convenant que $\max \emptyset = -1$. $h_{\mathcal{A}}(T)$ (resp. $h_{\mathcal{F}}(F)$) est appelé la *hauteur* de l'arbre T (resp. de la forêt F).

Pour démontrer que deux propriétés $R(T)$ et $S(F)$ sont vérifiées pour tout arbre T et toute forêt F, il suffit de démontrer (démonstration par *induction*) que

- $\forall F \in \mathcal{F},\ S(F) \Rightarrow R(x,F)$;
- $\forall U_1,\ldots,U_n \in \mathcal{A},\ R(U_1) \wedge \ldots \wedge R(U_n) \Rightarrow S(U_1,\ldots,U_n)$.

Terminologie

Il est clair que la donnée d'un arbre ordonné T permet de définir un arbre non ordonné T' au sens de la section 2.2.2 p. 31.[1] Les notions attachées à T' — sommet, racine, feuille, degré, taille, hauteur, père, fils, ascendant, descendant, sous arbre, branche — s'appliquent donc à T. Ajoutons à cela qu'un sommet de T' est plutôt appelé un *nœud* de T et que, dans un arbre $T = (x,(U_1,\ldots,U_n))$, U_i est appelé la i^{e} branche de la racine.[2]

3.2.2 Implémentation

Définition de deux types implémentant les arbres et les forêts

Pour définir deux types CAML `'a arbre` et `'a forêt` représentant les arbres et les forêts sur un ensemble E lui-même représenté par le type `'a`, la définition récursive conduit naturellement à poser

```
type 'a arbre = {étiquette : 'a; branches : 'a forêt}
and 'a forêt == 'a arbre list;;
```

Par exemple l'arbre $(1,((2,()),(3,())))$ sur $\mathbf{Z}$ est représenté par la valeur CAML

```
{étiquette = 1; branches = [{étiquette = 2; branches = []};
                            {étiquette = 3; branches = []}]}
```

Implémentation des fonctions inductives

Les fonctions de base sur les arbres et les forêts — taille, hauteur, nombre de feuilles — sont inductives et s'implémentent donc facilement; voir le programme 3.7. En ce qui concerne les fonctions `degré_arbre` et `degré_forêt`, elles ne sont pas inductives mais elles admettent les prolongements inductifs `da` = `degré_arbre` et `df` : $f \to$ (nombre d'arbres de f,degré de f).

Complexité des fonctions inductives

Chacune des huit fonctions du programme 3.7 a une complexité en $\mathrm{O}(n)$ où n est la taille de son argument.[3] De manière générale, on a la proposition suivante dont la preuve est laissée en exercice (voir la preuve de la proposition 3.1 p. 42):

Proposition 3.5 *La complexité de deux fonctions u et v de la forme*
`let rec` u `{étiquette = x; branches = f} =` $\varphi($`x`$,v($`f`$))$
`and` v `= function [] ->` c `| a :: f ->` $\psi(u($`a`$),v($`f`$))$
est O(taille), pourvu qu'un appel d'une fonction φ ou ψ ait une complexité O(1). ■

1. Indications pour une construction rigoureuse de T' : L'idée est de prendre pour sommets de T' les mots du langage P_T sur l'alphabet $\mathbf{N}^*$ défini inductivement par $P_{(x,(U_1,\ldots,U_n))} = \mu \cup \bigcup_{i=1,\ldots,n} i.P_{U_i}$. Un mot de P_T est appelé une *position* de T. On ordonne P_T par la relation $u \leqslant v \Leftrightarrow u$ est préfixe de v et on applique l'exercice 2.3 p. 32 pour définir l'arbre T'. Par exemple, si on prend pour T l'arbre de la figure 3.9, la position $3.2.2 \in P_T$ est le sommet de T' d'étiquette k; c'est le 2^{e} fils du 2^{e} fils du 3^{e} fils de la racine.

2. Les branches d'un nœud sont donc ordonnées et c'est pourquoi on parle d'arbre *ordonné*.

3. Si on programme les deux fonctions degrés par

```
let rec degré_arbre {branches = f} = max (degré_forêt f) (list_length f)
and degré_forêt = function [] -> 0 | a :: f -> max (degré_arbre a) (degré_forêt f);;
```

la complexité n'est plus O(taille).

Programme 3.7 Taille, hauteur, nombre de feuilles et degré d'un arbre et d'une forêt

```
let rec taille_arbre {branches = f} = 1 + taille_forêt f
and taille_forêt = function
   [] -> 0 | a :: f -> taille_arbre a + taille_forêt f;;

let rec hauteur_arbre {branches = f} = 1 + hauteur_forêt f
and hauteur_forêt = function
   [] -> -1 | a :: f -> max (hauteur_arbre a) (hauteur_forêt f);;

let rec feuilles_arbre {branches = f} = max 1 (feuilles_forêt f)
and feuilles_forêt = function
   [] -> 0 | a :: f -> feuilles_arbre a + feuilles_forêt f;;

let degré_arbre,degré_forêt =
    let rec da {branches = f} = (* da a = degré de l'arbre a *)
       let n,d = df f in max n d
    and df = function (* df f = (nbre d'arbres de la forêt f,degré de f) *)
       [] -> 0,0 | a :: f -> let n,d = df f in 1 + n,max (da a) d in
    da,function f -> snd (df f);;
```

3.2.3 Parcours préfixe et postfixe d'un arbre et d'une forêt

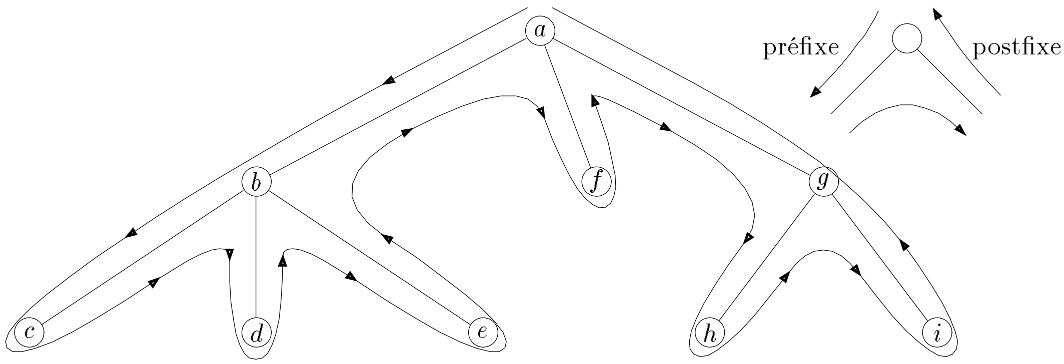


FIG. 3.10 – *Parcours d'un arbre*

Durant le parcours en profondeur d'un arbre ou d'une forêt, chaque nœud de degré d est rencontré $d+1$ fois. Effectuer un parcours *préfixe* (resp. *postfixe*) consiste, dans le parcours, à éxécuter une action à chaque première (resp. dernière) rencontre de chaque nœud. Si l'action a une complexité O(1), le parcours est en O de la taille de l'arbre ou de la forêt.

Si l'on considère les nœuds d'un arbre ou d'une forêt dans l'ordre où ils sont rencontrés lors d'un parcours préfixe (resp. postfixe), on définit un ordre total sur les nœuds appelé l'*ordre préfixe* (resp. *postfixe*). Pour l'arbre de la figure 3.10, $a < b < c < d < e < f < g < h < i$ pour l'ordre préfixe et $c < d < e < b < f < h < i < g < a$ pour l'ordre postfixe.

Si l'on concatène les étiquettes des nœuds rencontrés lors d'un parcours préfixe (resp. postfixe) d'un arbre ou d'une forêt A, on obtient un mot $\mathrm{Pref}(A)$ (resp. $\mathrm{Post}(A)$) sur l'alphabet E appelé le *mot du parcours préfixe* (resp. *postfixe*) de A. Pour l'arbre de la figure 3.10, on obtient le mot $abcdefghi$ (resp. $cdebfhiga$). Ces mots admettent les définitions récursives :

- $\mathrm{Pref}(x,F) = x\mathrm{Pref}(F)$, $\mathrm{Pref}(U_1,\dots,U_n) = \mathrm{Pref}(U_1)\dots\mathrm{Pref}(U_n)$;
- $\mathrm{Post}(x,F) = \mathrm{Post}(F)x$, $\mathrm{Post}(U_1,\dots,U_n) = \mathrm{Post}(U_1)\dots\mathrm{Post}(U_n)$.

3.3 Termes

Dans cette section on convient des notations suivantes pour représenter les arbres : l'arbre $(x,(U_1,\dots,U_n))$ est noté $x(U_1,\dots,U_n)$ et l'arbre $(x,())$ réduit à sa racine est noté x.

3.3.1 Syntaxe et sémantique : les termes et les algèbres

Termes (syntaxe)

Une *signature* est la donnée d'un ensemble Σ et d'une application $\alpha : \Sigma \to \mathbf{N}$. Un élément de Σ est appelé un *symbole fonctionnel* et, pour tout $f \in \Sigma$, $\alpha(f)$ est appelé l'*arité* du symbole f. Un symbole d'arité 0, 1, 2, etc. est dit *symbole constant*, *unaire*, *binaire*, etc. On notera Σ_n l'ensemble des symboles d'arité n.

Etant donnés une signature Σ et un ensemble V disjoint de Σ dont les éléments sont appelés des *variables*, un Σ-*terme* sur V ou une Σ-*expression abstraite* sur V est un arbre t étiqueté par $\Sigma \cup V$ vérifiant la propriété suivante :

> l'étiquette d'une feuille est soit un symbole constant, soit une variable et l'étiquette d'un nœud de degré $n > 0$ est un symbole d'arité n.

On peut donner une définition par induction de l'ensemble $T(\Sigma,V)$ des Σ-termes sur V :[1]

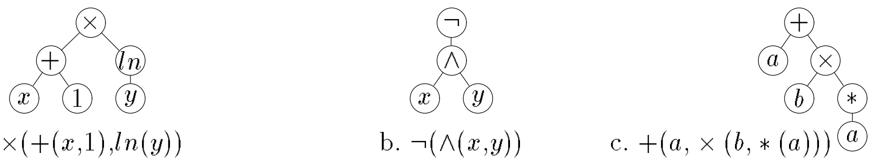
- $x \in V \Rightarrow x \in T(\Sigma,V)$;
- $f \in \Sigma_n \ \wedge \ t_1,\ldots,t_n \in T(\Sigma,V) \Rightarrow f(t_1,\ldots,t_n) \in T(\Sigma,V)$

Un Σ-terme est dit *clos* si les étiquettes de ses feuilles sont des symboles constants; autrement dit s'il ne contient pas de variable.

Exemple 3.3 expressions arithmétiques
On considère une signature Σ dont les symboles sont des mots :

- les symboles constants sont les mots comme 3,14 qui représentent en base 10 un nombre réel $\geqslant 0$;
- les symboles unaires sont donnés par $\Sigma_1 = \{ln,exp,sin,cos,arctan,moins\}$;
- les symboles binaires sont $\Sigma_2 = \{+,-,\times,/\}$;
- il n'y a pas de symbole d'arité > 2.

Pour cette signature particulière, un Σ-terme est appelé une *expression arithmétique*. La figure 3.11 a. montre l'expression arithmétique $\times(+(x,1),ln(y))$ sur $V = \{x,y\}$. A ce niveau il faut bien noter que le symbole *cos*, par exemple, n'est qu'un mot de trois lettres et ne désigne en aucun cas la fonction cosinus. De même, le symbole + est un mot d'une lettre qu'il faut se garder d'interpréter de quelque manière que ce soit.



a. $\times(+(x,1),ln(y))$ b. $\neg(\wedge(x,y))$ c. $+(a,\times(b,*(a)))$

FIG. 3.11 – *Une expression arithmétique, une expression logique et une expression rationnelle*

Exemple 3.4 expressions logiques
Si on définit Σ par $\Sigma_0 = \{0,1\}$, $\Sigma_1 = \{\neg\}$, $\Sigma_2 = \{\wedge,\vee,\Rightarrow,\Leftrightarrow\}$ et $\Sigma_n = \emptyset$ pour $n > 2$, un Σ-terme est appelé une *expression logique*. L'expression logique $\neg(\wedge(x,y))$ sur $V = \{x,y\}$ est représentée figure 3.11 b.

Exemple 3.5 expressions rationnelles
Un alphabet X étant fixé, si on définit Σ par $\Sigma_0 = \{\emptyset,\mu\} \cup X$, $\Sigma_1 = \{*\}$, $\Sigma_2 = \{+,\times\}$ et $\Sigma_n = \emptyset$ pour $n > 2$, un Σ-terme clos est appelé une *expression rationnelle*, en abrégé ER, sur l'alphabet X. L'ER $+(a,\times(b,*(a)))$ sur l'alphabet $X = \{a,b\}$ est représentée figure 3.11 c.

Algèbres (sémantique)

La notion de terme est fondamentalement syntaxique. Les deux expressions arithmétiques $+(\times(cos(x),cos(x)),\times(sin(x),sin(x)))$ et 1 désignent deux arbres différents de tailles respectives 11 et 1. On introduit une nouvelle notion qui permet d'interpréter les termes.

1. Voir les conventions de notation des arbres au début de cette section.

Soient une signature Σ et un ensemble A. Une *Σ-algèbre* $\mathcal{A}$ de *domaine* A est une application qui, à tout symbole $f \in \Sigma_n$ d'arité n, associe une application $f^{\mathcal{A}}$ de A^n dans A appelée la *$\mathcal{A}$-interprétation* du symbole f. Dans le cas où f est un symbole constant, $n = 0$, on peut identifier $f^{\mathcal{A}}$, application d'un ensemble à un élément dans A, à un élément de A.

Donnons nous une Σ-algèbre $\mathcal{A}$ de domaine A. $\mathcal{A}$ donne une interprétation des symboles et on se propose d'interpréter aussi les termes :

Etant donné un ensemble V de variables et un élément ℓ de l'ensemble A^V des applications de V dans A, on dit que $x \in V$ est *lié* à $a \in A$ par ℓ si $\ell(x) = a$; on dit aussi que le couple (x,a) est la *liaison* de x définie par ℓ de sorte que l'application ℓ est déterminée par l'ensemble des liaisons qu'elle définit.

On associe à tout Σ-terme t sur V sa *$\mathcal{A}$-interprétation* $t^{\mathcal{A}}$ qui est une application de A^V dans A. Pour tout $\ell \in A^V$, $t^{\mathcal{A}}(\ell) \in A$ est défini par induction :

- si $t = x \in V$ est une variable, on pose $t^{\mathcal{A}}(\ell) = \ell(x)$;
- si $t = f(t_1,\ldots,t_n)$ où $f \in \Sigma_n$ et $t_1,\ldots,t_n \in T(\Sigma,V)$ on pose $t^{\mathcal{A}}(\ell) = f^{\mathcal{A}}(t_1^{\mathcal{A}}(\ell),\ldots,t_n^{\mathcal{A}}(\ell))$.

$t^{\mathcal{A}}(\ell)$ est appelé la *$\mathcal{A}$-valeur* de t pour les liaisons des variables définies par ℓ. En particulier on peut parler de la $\mathcal{A}$-valeur $t^{\mathcal{A}}$ d'un terme clos ou sans variable t; c'est un élément de A défini par l'induction :

$$(f(t_1,\ldots,t_n))^{\mathcal{A}} = f^{\mathcal{A}}(t_1^{\mathcal{A}},\ldots,t_n^{\mathcal{A}}).$$

On peut d'ailleurs ramener le cas général au cas des termes clos en remarquant que si t est un terme non clos et $\ell \in A^V$, $t^{\mathcal{A}}(\ell) = t_\ell^{\mathcal{A}}$ où t_ℓ est le terme clos obtenu à partir de t en remplaçant chaque variable x par sa valeur $\ell(x)$.

Deux termes t et s sont dits *$\mathcal{A}$-équivalents* s'ils ont la même $\mathcal{A}$-interprétation : $t^{\mathcal{A}} = s^{\mathcal{A}}$.

Exemple 3.6 expressions arithmétiques (suite de l'exemple 3.3)
On choisit le domaine $A = \mathbf{R} \cup \{\mathrm{ERR}\}$ où ERR est une valeur particulière non réelle appelée *valeur d'erreur* et on définit une Σ-algèbre $\mathcal{A}$ qui interprète les symboles arithmétiques de manière naturelle :

- un symbole constant est interprété par la valeur réelle qu'il représente;
- le symbole ln est interprété par la fonction qui à $a \in \mathbf{R}_+^*$ associe $\ln a$ et à toute autre valeur associe ERR; les autres symboles unaires exp, sin, cos, $arctan$ et "moins" sont interprétés de manière analogue;
- on interprète $+$ par la fonction qui à $(a,b) \in A^2$ associe $a + b$ si $a,b \in \mathbf{R}$ et ERR si a ou $b = \mathrm{ERR}$; de même pour $-$, $\times$ et $/$ $(/^{\mathcal{A}}(a,0) = \mathrm{ERR})$.

Par exemple, l'expression arithmétique close $cos(0)$ a pour valeur $\cos 0 = 1$; l'expression $t = \times(+(x,1),ln(y))$ sur $\{x,y\}$ est interprétée par $t^{\mathcal{A}}(\ell) = (\ell(x)+1)\ln(\ell(y))$; les expressions $+(\times(cos(x),cos(x)), \times\,(sin(x),sin(x)))$ et 1 sur $\{x\}$ sont équivalentes.

Exemple 3.7 expressions logiques (suite de l'exemple 3.4)
On interprète une expression logique en utilisant la Σ-algèbre $\mathcal{A}$ de domaine $A = \{0,1\}$ — 0 pour « faux » et 1 pour « vrai » — définie naturellement par :

- $0^{\mathcal{A}} = 0$ et $1^{\mathcal{A}} = 1$;
- $\neg^{\mathcal{A}}(0) = 1$, $\neg^{\mathcal{A}}(1) = 0$;
- $\wedge^{\mathcal{A}}(x,y) = 1$ ssi $x = y = 1$, $\vee^{\mathcal{A}}(x,y) = 1$ ssi $x = 1$ ou $y = 1$,
  $\Rightarrow^{\mathcal{A}}(x,y) = 1$ ssi $x = 0$ ou $y = 1$, $\Leftrightarrow^{\mathcal{A}}(x,y) = 1$ ssi $x = y$.

Les expressions logiques $\neg(\wedge(x,y))$ et $\vee(\neg(x),\neg(y))$ sont équivalentes. Une *tautologie* est une expression logique dont l'interprétation est la fonction constante 1.

Exemple 3.8 expressions rationnelles (suite de l'exemple 3.5)
On prend pour domaine l'ensemble $A = \mathcal{P}(X^*)$ des parties du monoïde libre X^*, c.-à-d. des langages sur l'alphabet X et on définit une Σ-algèbre $\mathcal{A}$ en utilisant les opérations sur les langages définies p. 30 :

- $\emptyset^{\mathcal{A}} = \emptyset$ le langage vide, $\mu^{\mathcal{A}} = \{\mu\}$ le langage réduit au mot vide μ que l'on note simplement μ et, pour $x \in X$, $x^{\mathcal{A}} = \{x\}$ le langage réduit au seul mot x noté aussi x;
- si $L \subset X^*$, on pose $*^{\mathcal{A}}(L) = L^*$ l'étoile de L;
- pour $L,M \subset X^*$, on pose $+^{\mathcal{A}}(L,M) = L + M$ et $\times^{\mathcal{A}}(L,M) = LM$.

L'interprétation $L = t^{\mathcal{A}}$ d'une ER t est appelée le *langage reconnu* par t. L'ER $+(a,\times(b,*(a)))$ reconnaît le langage $a + ba^* = \{a\} \cup \{ba^n \mid n \in \mathbf{N}\}$. La proposition 2.1 p. 30 montre par exemple que si t et s sont deux ER, les deux ER $\times(t, *\,(\times(s,t)))$ et $\times(*(\times(t,s)),t)$ sont équivalentes.

Implémentation

Le programme 3.8 implémente les termes et les algèbres en supposant que les symboles sont d'arité $\leqslant 2$, ce qui est le cas des exemples étudiés.

Le type `terme` implémente les termes, les constructeurs `V`, `C`, `U` et `B` correspondant respectivement aux variables, constantes, symboles unaires et binaires. Par exemple, l'expression arithmétique $+(sin(3),ln(x))$ est représentée par la valeur
`B("+",U("sin",C "3."),U("ln",V "x"))` de type `terme`.[1]

Le type `'a algèbre` implémente les algèbres de domaine `'a`. En particulier, les valeurs `algèbre_arithmétique : float algèbre` et `algèbre_logique : bool algèbre` représentent les algèbres associées aux expressions respectivement arithmétiques et logiques.[2]

Un ensemble de liaisons de variables à des valeurs est implémenté par une valeur ℓ de type `(string * 'a) list`; la présence du couple (x,a) dans la liste ℓ signifiant que la variable x est liée à la valeur a.[3] La fonction `évalue` est de type
`'a algèbre -> terme -> (string * 'a) list -> 'a`; `évalue` $\mathcal{A}$ t ℓ renvoie $t^{\mathcal{A}}(\ell)$. Si chaque fonction définie par $\mathcal{A}$ a une complexité O(1), la complexité de l'évaluation d'un terme est un O de la taille du terme.

Tous les programmes de la suite de cette section utilisent les déclarations du programme 3.8.

Un exemple de calcul sur les termes : vérificateur de tautologie

Le programme 3.9 fournit une fonction `teste_tautologie : terme -> string list -> bool` telle que, si t est une expression logique dont les variables sont dans la liste v, `teste_tautologie` t v renvoie `true` ssi t est une tautologie, ce qui est le cas, par exemple, pour `teste_tautologie (B("ou",V"x",U("non",V"x"))) ["x"];;`.

La fonction contient une sous fonction `teste : (string * bool) list -> string list -> unit` spécifiée de la manière suivante : si ℓ lie une partie des variables de v et si w contient les autres variables de v, alors `teste` ℓ w déclenche l'exception `Faux` ssi il existe une façon de lier les variables de w qui donne la valeur `false` à t. Ainsi t est une tautologie ssi `teste []` v ne déclenche pas d'exception.

COMPLEXITÉ : $O(m2^n)$ où m est la taille de t et n la longueur de v.

> **Preuve** Si T_p est le temps maximum d'éxécution de `teste` ℓ w quand w est de longueur p, il existe des constantes A et B telles que $T_0 \leqslant Am$ et $T_p \leqslant B + 2T_{p-1}$; d'où, par récurrence, $T_p \leqslant A2^p m + B(2^p - 1)$. ∎

Pour tester l'équivalence de deux expressions logiques t et s, il suffit de faire le test de tautologie sur l'expression $\Leftrightarrow (t,s)$. On verra au chapitre 4 qu'il est aussi possible de tester l'équivalence de deux expressions rationnelles. Le cas des expressions arithmétiques est plus difficile.

Un exemple de réécriture de termes : la dérivation des expressions arithmétiques

Soient Σ et $\mathcal{A}$ la signature et l'algèbre des expressions arithmétiques. Une variable $x \in V$ étant fixée, on définit par induction une fonction $D_x : T(\Sigma,V) \to T(\Sigma,V)$ dont l'interpréta-

1. Pour représenter les termes les plus généraux on aurait pu introduire le type `terme = V of string | T of string * terme list`. Une autre possibilité est au contraire d'associer à chaque signature Σ un type particulier représentant les Σ-termes; par exemple on peut représenter les expressions logiques par le type `el = V of string | Faux | Vrai | Non of el | Et of el * el | Ou of el * el`.

2. On ne définit pas ici de valeur `algèbre_rationnelle` car il n'y a pas de moyen simple de représenter le langage reconnu par une ER; voir pour cela, le programme 4.3 p. 88 du chapitre 4.

3. Il serait plus efficace d'utiliser une table de hachage (voir p. 107).

Programme 3.8 Termes et algèbres

```
type terme =
   V of string |
   C of string |
   U of string * terme |
   B of string * terme * terme;;

type 'a algèbre = { eval_c : string -> 'a;
                    eval_u : string -> 'a -> 'a;
                    eval_b : string -> 'a -> 'a -> 'a };;

exception Erreur;;

let algèbre_arithmétique =
   {
   eval_c = (function s -> float_of_string s);
   eval_u = (function
     "ln"  -> log |
     "exp" -> exp |
     "sin" -> sin |
     "cos" -> cos |
     "arctan" -> atan |
     "moins" -> minus_float |
     _ -> raise Erreur);
   eval_b = (function
     "+" -> prefix +. |
     "-" -> prefix -. |
     "*" -> prefix *. |
     "/" -> prefix /. |
     _ -> raise Erreur)
   };;

let algèbre_logique =
    {
    eval_c = (function "0" -> false | "1" -> true | _ -> raise Erreur);
    eval_u = (function  "non" -> prefix not | _ -> raise Erreur);
    eval_b = (function
      "et" -> prefix & |
      "ou" -> prefix or |
      "implique" -> (fun x y -> not x or y) |
      "équivalent" -> (fun x y -> x = y) |
      _ -> raise Erreur)
    };;

let évalue {eval_c = evc; eval_u = evu; eval_b = evb} terme liaisons =
   let rec ev = function
      V x -> assoc x liaisons |
      C s -> evc s |
      U (s,u) -> evu s (ev u) |
      B (s,u,v) -> evb s (ev u) (ev v) in
   ev terme;;
```

tion est la dérivation par rapport à x : pour tout Σ-terme t sur V, $(D_x(t))^{\mathcal{A}} = \partial t^{\mathcal{A}}/\partial x$.

- $D_x(x) = 1$, $D_x(y) = 0$ si $y \in \mathbf{R} \cup V \setminus \{x\}$;
- $D_x(ln(t)) = /(D_x(t),t)$, etc.;
- $D_x(+(t,s)) = +(D_x(t),D_x(s))$, etc.

Programme 3.9 Test de tautologie

```
exception Faux;;

let teste_tautologie t v =
   let rec teste liaisons = function
      [] -> if not évalue algèbre_logique t liaisons then raise Faux |
      x :: w -> teste ((x,true) :: liaisons) w;
                teste ((x,false) :: liaisons) w in
   try teste [] v; true with Faux -> false;;
```

La fonction `dérive : string -> terme -> terme` du programme 3.10 impémente D.

Programme 3.10 Dérivation formelle

```
let dérive x =
   let der t = function
      "ln" -> B("/",C "1",t) |
      "exp" -> U("exp",t) |
      "sin" -> U("cos",t) |
      "cos" -> U("moins",U("sin",t)) |
      "arctan" -> B("/",C "1",B("+",C "1",B("*",t,t))) |
      "moins" -> U("moins",C "1") |
      f -> U(f ^ "'",t) (* cas d'un symbole unaire inconnu *) in
   let rec d = function
      V y -> if y = x then C "1" else C "0" |
      C _ -> C "0" |
      U(f,t) -> B("*",d t,der t f) |
      B(("+"|"-" as o),t,s)->B(o,d t,d s) |
      B("*",t,s) -> B("+",B("*",d t,s),B("*",t,d s)) |
      B("/",t,s) -> B("/",B("-",B("*",d t,s),B("*",t,d s)),B("*",s,s)) |
      _ -> failwith "dérive" in
   d;;
```

Par exemple, `dérive "x" (U("sin",B("*",V "x",V "y")))` renvoie
`B("*",B("+",B("*",C"1",V"y"),B("*",V"x",C"0")),U("cos",B("*",V"x",V"y")))`
ce qui est bien le résultat escompté, mis à part le fait que l'on obtient une expression non simplifiée. Le problème de la simplification des expressions arithmétiques est difficile. De plus, la structure d'arbre de degré 2 n'est pas adaptée aux algorithmes de simplification; les logiciels de calcul formel utilisent des arbres de degré non limité permettant de traiter des combinaisons linéaires ou des produits de puissances.

3.3.2 Syntaxe concrète

La notion de terme est adaptée au traitement par ordinateur mais l'utilisateur est plutôt habitué à représenter une expression abstraite telle que $\times(+(x,y),z)$ sous la forme concrète $(x+y)\times z$. Dans le dialogue utilisateur-machine, l'utilisateur rentre au clavier des expressions concrètes que la machine analyse pour en construire les formes abstraites. Réciproquement, si la machine doit afficher un terme calculé elle affiche la forme concrète du terme.

Il y a plusieurs manières de représenter un terme sous une forme concrète. La manière la plus commune est la forme infixe qui donne par exemple $(x+y)\times z$ mais on étudiera aussi les formes préfixes $\times\ +\ x\ y\ z$ et postfixes $x\ y\ +\ z\ \times$ qui présentent l'avantage de ne pas nécessiter de parenthèse.

Dans toute la suite, on fixe une signature Σ et un ensemble de variables V. Les formes concrètes des Σ-termes seront des mots sur l'alphabet $\Sigma \cup V$ éventuellement complété par une paire de parenthèses. Comme, dans les applications, les éléments de $\Sigma \cup V$ sont déja des mots, on pourra être amené à séparer les symboles et les variables par des espaces comme dans le mot *sin cos* $\times\ x\ y$.

Les programmes utilisent le type `terme` défini dans le programme 3.8 p. 56. Les mots sur $\Sigma \cup V$ seront implémentés par le type `string list`.

3.3.3 Forme concrète postfixée d'un terme

Définition

Le mot du parcours postfixe[1] $\mathrm{Post}(t)$ d'un terme t est appelé la *forme postfixe* de l'expression t. $\mathrm{Post}(t) \in (\Sigma \cup V)^*$ est donc défini récursivement par $\mathrm{Post}(f(t_1,\ldots,t_n)) = \mathrm{Post}(t_1)\ldots\mathrm{Post}(t_n)f$. Par exemple, $\mathrm{Post}(\times(+(x,y),z)) = x\ y\ +\ z\ \times$.

Programme 3.11 Forme postfixe d'un terme

```
let imprime_post =
   let p f = print_string f; print_char' ' in
   let rec imp = function
      V x -> p x |
      C f -> p f |
      U(f,u) -> imp u; p f |
      B(f,u,v) -> imp u; imp v; p f in
   imp;;

let post =
   let rec p l = function
      V x -> x :: l |
      C f -> f :: l |
      U(f,u) -> p (f :: l) u |
      B(f,u,v) -> p (p (f :: l) v) u in
   p [];;
```

Implémentation : La fonction `imprime_post : terme -> unit` du programme 3.11 imprime à l'écran la forme postfixe de son argument.
`imprime_post(B("*",B("+",V"x",V"y"),V"z"))` affiche `x y + z *`.

La fonction `post : terme -> string list` renvoie la forme posfixe de son argument sous la forme d'une liste de symboles et de variables;
`post(B("*",B("+",V"x",V"y"),V"z"))` renvoie `["x"; "y"; "+"; "z"; "*"]`.
Elle utilise une sous fonction récursive `p : string list -> terme -> string list` telle que `p` $\ell\ t$ renvoie la concaténée des listes $\mathrm{Post}(t)$ et ℓ.

Chacune de ces deux fonctions a une complexité en O de la taille de son argument.[2]

Calcul d'un terme à partir de sa forme postfixe

On se propose dans cette section de montrer que la connaissance de la forme postfixe u d'un terme permet de retrouver ce terme.
Autrement dit, la fonction $\mathrm{Post} : T(\Sigma,V) \to (\Sigma \cup V)^*$ est injective.

Partant par exemple du mot $u = x\ y\ ln\ +\ z\ \times$, on lit les lettres de gauche à droite et on utilise une pile de termes qui, à la fin de la lecture, contient un seul élément qui est le terme cherché; la figure 3.12 montre l'évolution de la pile au fur et à mesure de la lecture des symboles.
Soit f la lettre lue à une étape quelconque :

- si f est une constante ou une variable, on l'empile;
- si c'est un symbole unaire, on remplace l'élément t du haut de la pile par le terme $f(t)$;
- si c'est un symbole binaire, on supprime les deux termes s et t situés en haut de la pile et on empile $f(s,t)$.

1. Voir p. 52.
2. Ce qui n'aurait pas été le cas de la fonction `post` si elle avait été programmée directement (sans sous fonction) en utilisant l'opérateur de concaténation de liste `@`.

	y	$ln(y)$		z	
x	x	x	$+(x,ln(y))$	$+(x,ln(y))$	$\times(+(x,ln(y)),z)$
x	y	ln	$+$	z	$\times$

FIG. 3.12 – *Analyse de l'expression postfixée $x\ y\ ln\ +\ z\ \times$*

L'algorithme 3.1 formalise ces idées. [1]

Algorithme 3.1 Analyse syntaxique d'une expression postfixée

soit analyse_post u =
  soit *pile* = pile_vide **dans** (* pop et push sont relatifs à *pile* *)
  pour $k = 1$ **à** longueur(u) **faire**
    soit f = la k^e lettre de u **dans**
    soit n = l'arité de f **dans**
    soit t = tableau de dimension n **dans**
    pour $i = n$ **décroissant à** 1 **faire** $t_i \leftarrow$ pop
    push $f(t_1, \ldots ,t_n)$
  renvoyer pop

Proposition 3.6 *L'application* Post : $T(\Sigma,V) \to (\Sigma \cup V)^*$ *est injective et, pour tout terme t, la fonction* analyse_post *de l'algorithme 3.1 appliquée au mot $u = \mathrm{Post}(t)$, renvoie t.*

Preuve

On suppose, pour simplifier les notations, que $V = \emptyset$, ce qui ne nuit pas à la généralité car un Σ-terme sur V est aussi un $\Sigma \cup V$-terme sur $\emptyset$.

On note $L \subset \Sigma^*$ l'ensemble des mots de la forme $\mathrm{Post}(t)$ où $t \in T(\Sigma) = T(\Sigma,\emptyset)$.

Si $f \in \Sigma$ est un symbole d'arité n, on appelle *poids* de f l'entier $p(f) = 1 - n$. On prolonge la fonction p à Σ^* en définissant le poids d'un mot comme la somme des poids des lettres qui le composent, de sorte que $p : \Sigma^* \to \mathbf{Z}$ est un morphisme de monoïdes. [2]

On note $L' = \{u \in \Sigma^* \mid (p(u) = 1) \ \wedge \ (\forall v, \mu \prec v \prec u \Rightarrow p(v) \geqslant 1)\}$ où $v \preccurlyeq u$ (resp. $v \prec u$) signifie que v est un préfixe de u (resp. v est un préfixe de u différent de u).

La preuve procède en plusieurs étapes.

ETAPE 1. *Si $u_1, \ldots ,u_n$ sont n mots de L' et si $f \in \Sigma_n$ est un symbole d'arité n alors* le mot *$u = u_1 \ldots u_n f$ est dans L'.*

$p(u) = p(u_1) + \cdots + p(u_n) + p(f) = n \times 1 + (1 - n) = 1$ et si $\mu \prec v \prec u$ alors ou bien $v = u_1 \ldots u_i$ où $1 \leqslant i \leqslant n$ auquel cas $p(v) = p(u_1) + \cdots + p(u_i) = i \geqslant 1$, ou bien $v = u_1 \ldots u_i v'$ avec $0 \leqslant i < n$ et $\mu \prec v' \prec u_{i+1}$ auquel cas $p(v) \geqslant p(v') \geqslant 1$.

ETAPE 2. *$L \subset L'$.*

On procède par induction pour montrer que, pour tout terme t, $\mathrm{Post}(t) \in L'$: si $t_1, \ldots ,t_n$ sont n termes tels que $\forall i, u_i = \mathrm{Post}(t_i) \in L'$ et si $f \in \Sigma_n$ alors l'étape 1 montre que $\mathrm{Post}(f(t_1, \ldots ,t_n)) = u_1 \ldots u_n f \in L'$.

ETAPE 3. *Si $u \in L'$, l'éxécution de* analyse_post *u renvoie un terme t tel que* $\mathrm{Post}(t) = u$.

On montre que la boucle principale de analyse_post satisfait l'invariant suivant :

(I_k) Si $(t_1, \ldots ,t_m)$ est le contenu de la pile (« haut » de la pile à droite) alors le mot $\mathrm{Post}(t_1) \ldots \mathrm{Post}(t_m)$ est le préfixe de longueur k de u.

– (I_0) est bien sûr vérifié avant l'entrée dans la boucle.

– Supposons (I_{k-1}) vérifié : la pile contient $(t_1, \ldots ,t_m)$ et $u = vfw$ avec $f \in \Sigma$, $|v| = k - 1$ et $v = u_1 \ldots u_m$ où $u_i = \mathrm{Post}(t_i)$. Notons que l'arité n de f est $\leqslant m$ — en effet, comme les mots u_i et u sont dans L', on a $1 \leqslant p(vf) = p(v) + p(f) = m + (1 - n)$ —. Si l'on éxécute alors le corps de la boucle, la pile devient $(t_1, \ldots ,t_{m-n}, f(t_{m-n+1}, \ldots ,t_m))$ et $\mathrm{Post}(t_1) \ldots \mathrm{Post}(t_{m-n})\mathrm{Post}(f(t_{m-n+1}, \ldots ,t_m)) = vf$ est le préfixe de longueur k de u; donc (I_k) est vérifié.

Conclusion; à la fin de l'éxécution de la boucle (I_ℓ) est vérifié avec $\ell = |u|$ donc le contenu

1. Cet algorithme est une application d'une méthode générale d'analyse syntaxique appelée *méthode* LR. Cette méthode est utilisée par les calculatrices dites scientifiques et par la plupart des compilateurs.

2. Voir la section 2.1.2 p. 29.

de la pile est $(t_1,\dots,t_m)$ avec $\mathrm{Post}(t_1)\dots\mathrm{Post}(t_m) = u$. Mais, comme les $u_i = \mathrm{Post}(t_i)$ et u sont dans L', on a $m = p(u_1) + \cdots + p(u_m) = p(u) = 1$ donc la pile ne contient qu'un élément $t = t_1$ et $\mathrm{Post}(t) = u$.

ETAPE 4. $L = L'$.

C'est clair vues les deux étapes précédentes.

ETAPE 5. *Un suffixe strict d'un élément de L n'appartient pas à L.*

Si $u = vw$ avec $u \in L = L'$, $w \in L = L'$ et $v \neq \mu$, alors $\mu \prec v \prec u$ — car $w \neq \mu$ — et $p(v) = p(u) - p(w) = 1 - 1 = 0$, ce qui contredit $u \in L'$.

ETAPE 6. Post *est injective.*

On montre par récurrence sur $|u|$ que si $\mathrm{Post}(t) = \mathrm{Post}(t') = u$ alors $t = t'$:
Si $|u| = 1$, c'est clair.
Si la propriété est vérifiée pour les mots de longueur $< |u|$ et si $\mathrm{Post}(t) = \mathrm{Post}(t') = u$ alors $t = f(t_1,\dots,t_n)$ et $t' = f(t'_1,\dots,t'_m)$ où f est le dernier symbole de u. Si l'on note $u_i = \mathrm{Post}(t_i) \in L$ et $u'_i = \mathrm{Post}(t'_i) \in L$, on a $u_1\dots u_n f = \mathrm{Post}(t) = \mathrm{Post}(t') = u'_1\dots u'_m f$ donc $u_1\dots u_n = u'_1\dots u'_m$. Comme, d'aprés l'étape 5, aucun des deux mots u_n et u'_m de L n'est suffixe de l'autre, on a $u_n = u'_m$ et donc $u_1\dots u_{n-1} = u'_1\dots u'_{m-1}$. En poursuivant le même raisonnement, on trouve $n = m$ et $\forall i, u_i = u'_i$. D'après l'hypothèse de récurrence, $\forall i, t_i = t'_i$ et donc $t = t'$. ∎

Exercice 3.6 Montrer que si on applique la fonction analyse_post à un mot de $(\Sigma \cup V)^*$ qui n'est pas la forme postfixe d'un terme alors ou bien l'éxécution de l'algorithme produit une erreur — appel de pop sur une pile vide —, ou bien à la fin, la pile est non vide.

IMPLÉMENTATION : La fonction `analyse_post : (string -> int) -> string list -> terme` du programme 3.12[1] implémente la fonction réciproque de la fonction Post. Le premier argument, de type `string -> int` sert à spécifier l'arité de chaque symbole, en convenant que l'arité d'une variable est -1. Par exemple, `analyse_post arité_arithmétique ["x";"y";"+";"z";"*"]` renvoie `B("*",B("+",V"x",V"y"),V"z")`. Une exception — `Empty` ou `Erreur` — est déclenchée si l'argument n'est pas une forme postfixe.[2]

Programme 3.12 Analyse syntaxique de la forme postfixe d'un terme

```
#open"stack";;

let analyse_post arité l =
   let u = vect_of_list l
   and p = new() in
   for k = 0 to vect_length u - 1 do
      let f = u.(k) in
      match arité f with
         -1 -> push (V f) p |
         0  -> push (C f) p |
         1  -> let t = pop p in push (U (f,t)) p |
         2  -> let t = pop p and s = pop p in push (B (f,s,t)) p
   done;
   let t = pop p in
   if p <> new() then raise Erreur;
   t;;

let arité_arithmétique = function
   "ln" | "exp" | "sin" | "cos" | "arctan" | "moins" -> 1 |
   "+" | "-" | "*" | "/" -> 2 |
   s -> let c = int_of_char s.[0] - int_of_char `0` in
        if 0 <= c & c <= 9 then 0 else - 1;;
```

COMPLEXITÉ : O de la longueur du mot analysé, soit de la taille du terme renvoyé.

1. Voir p. 105 pour l'utilisation du module `stack` implémentant les piles.
2. Voir l'exercice 3.6.

3.3.4 Forme concrète préfixée d'un terme

Définition

Le mot du parcours préfixe $\text{Pref}(t)$ d'un terme t est appelé la *forme préfixe* de l'expression t. $\text{Pref}(t)$ est donc défini récursivement par $\text{Pref}(f(t_1,\ldots,t_n)) = f\text{Pref}(t_1)\ldots\text{Pref}(t_n)$. Par exemple, $\text{Pref}(\times(+(x,y),z)) = \times\ +\ x\ y\ z$.

IMPLÉMENTATION : La fonction `imprime_pref : terme -> unit` (resp. `pref : terme -> string list`) du programme 3.13 imprime (resp. renvoie) en O de la taille la forme préfixe de son argument.

Programme 3.13 Forme préfixe d'un terme

```
let imprime_pref =
   let p f = print_string f; print_char‘ ‘ in
   let rec imp = function
      V x -> p x |
      C f -> p f |
      U(f,u) -> p f; imp u |
      B(f,u,v) -> p f; imp u; imp v in
   imp;;

let pref =
   let rec p l = function
      V x -> x :: l |
      C f -> f :: l |
      U(f,u) -> f :: p l u |
      B(f,u,v) -> f :: p (p l v) u in
   p [];;
```

Calcul d'un terme à partir de sa forme préfixe

Dans cette section, on note L l'ensemble des formes préfixes de tous les termes.

Proposition 3.7 *La fonction* $\text{Pref} : T(\Sigma,V) \to (\Sigma\cup V)^*$ *est injective. De plus, un mot quelconque de* $(\Sigma\cup V)^*$ *admet au plus un préfixe élément de* L.

Preuve Si $u = u_1\ldots u_n$ est un mot sur un alphabet E, on note $m(u)$ le mot *miroir* de u obtenu en renversant l'ordre de ses lettres : $m(u) = u_n\ldots u_1$. Si t est un arbre sur E, on définit récursivement l'arbre *miroir* $m(t)$ de t : $m(x,(t_1,\ldots,t_n)) = (x,(m(t_n),\ldots,m(t_1)))$. On voit par induction que, pour tout arbre t, $\text{Pref}(t) = m(\text{Post}(m(t)))$, ce qui, dans le cas où $E = \Sigma\cup V$, donne l'injectivité de Pref sur l'ensemble des termes comme une conséquence de l'injectivité de Post démontrée dans la proposition 3.6 p. 59.

Cela montre aussi que le langage L des formes préfixes est l'ensemble des mots miroirs des formes postfixes et l'étape 5 de la preuve de la proposition 3.6 montre qu'un mot admet au plus un préfixe dans L. ■

Pour implémenter la fonction $\text{Pref}^{-1} : L \to T(\Sigma,V)$, on applique une méthode appelée *récursive descendante*.

Fixons un mot $u = u_1\ldots u_n \in (\Sigma\cup V)^*$ et un pointeur k sur une lettre de u; autrement dit, k est un entier variable $\geqslant 1$ qui représente, s'il est $\leqslant n$, la lettre u_k en cours de lecture dans le mot u. On lit le mot u de gauche à droite; plus précisément on n'autorise sur le pointeur k que l'unique instruction $k \leftarrow k+1$. On peut alors écrire une fonction *expr* sans argument à valeurs dans $T(\Sigma,V)$ spécifiée comme suit :

- si le mot $u_k u_{k+1}\ldots u_n$ admet un préfixe $v = u_k u_{k+1}\ldots u_h$ appartenant à L alors *expr*() effectue la lecture de v — précisément, k prend la valeur $h+1$ — et renvoie le terme $\text{Pref}^{-1}(v)$;
- si $k > n$ ou si $u_k u_{k+1}\ldots u_n$ n'admet aucun préfixe dans L, alors *expr*() échoue.

$expr()$ se programme de la manière suivante :

- Lire $f = u_k$ en faisant $k \leftarrow k+1$ (si $k > n$ faire échouer la fonction).
- Dans le cas où le mot $u_k \dots u_n$ admet un préfixe $v = u_k \dots u_h \in L$, le mot v s'écrit de manière unique sous la forme $v = fv_1 \dots v_d$ où d est l'arité de f et les v_i sont dans L. k pointe sur la première lettre de v_1;
  - poser $t_1 = expr()$, donc $t_1 = \mathrm{Pref}^{-1}(v_1)$ et k pointe sur la première lettre de v_2;
  - poser $t_2 = expr()$, donc $t_2 = \mathrm{Pref}^{-1}(v_2)$ et k pointe sur la première lettre de v_3;
  - ....
  - poser $t_d = expr()$, donc $t_d = \mathrm{Pref}^{-1}(v_d)$ et $k = h+1$ pointe après v;
  - renvoyer le terme $t = f(t_1, \dots, t_d)$; on a bien $t = \mathrm{Pref}^{-1}(v)$.
- Dans le cas où $u_k \dots u_n$ n'a pas de préfixe dans L, un raisonnement par l'absurede montre que le procédé précédent fait échouer la fonction.

La fonction $expr$ étant ainsi définie, pour tester si un mot u est dans L et, en cas de succés renvoyer $\mathrm{Pref}^{-1}(u)$, il suffit d'appeler $expr()$ après avoir fait pointer k sur la première lettre de u puis de tester que le mot u a été entièrement lu ($k = \text{longueur}(u)+1$).

Implémentation : Le programme 3.14 fournit la fonction

`analyse_pref : (string -> int) -> string list -> terme`

qui implémente la fonction réciproque de la fonction Pref.[1]

Par exemple, `analyse_pref arité_arithmétique ["*";"+";"x";"y";"z"]` renvoie `B("*",B("+",V"x",V"y"),V"z")`.[2]

Une exception est déclenchée si l'argument n'est pas une forme préfixe.

Programme 3.14 Analyse syntaxique de la forme préfixe d'un terme

```
let analyse_pref arité l =
   let u = vect_of_list l
   and k = ref 0 in
   let rec expr() =
      let f = u.(!k) in
      incr k;
      match arité f with
         -1 -> V f |
         0  -> C f |
         1  -> let t = expr() in U(f,t) |
         2  -> let s = expr() and t = expr() in B(f,s,t) in
   let t = expr() in
   if !k <> vect_length u then raise Erreur;
   t;;
```

Complexité : $O(n)$ où n est la taille du mot argument u. Il suffit, pour s'en convaincre, de remarquer que l'arbre des appels récursifs de la fonction $expr$ a précisément la même structure, et donc la même taille, que l'arbre $\mathrm{Pref}^{-1}(u)$.

3.3.5 Formes concrètes infixées d'un terme

Qualité d'un symbole unaire

On suppose à partir de maintenant que la signature Σ ne contient pas de symbole d'arité $>$ à 2 ($n > 2 \Rightarrow \Sigma_n = \emptyset$) et que l'ensemble Σ_1 des symboles unaires est partitionné en deux sous ensembles Σ_1^{g} et Σ_1^{d}. Les éléments de Σ_1^{g} (resp. Σ_1^{d}) sont appelés les symboles unaires *préfixes* (resp. *postfixes*).

On convient, pour les trois exemples de base que

- les symboles des expressions arithmétiques sont préfixes;

1. De même que pour la fonction `analyse_post` du programme 3.12 p. 60, le premier argument, de type `string -> int`, permet de spécifier l'arité de chaque symbole.
2. La valeur `arité_arithmétique` est définie dans le programme 3.12.

– le symbole *not* des expressions logiques est préfixe;

– le symbole $*$ des expressions rationnelles est postfixe.

On introduit deux nouveaux éléments; les parenthèses gauche « [» et droite «] ». Un élément de l'alphabet $X = \Sigma \cup V \cup \{[,]\}$ est appelé un *lexème*.[1]

Forme concrète infixée complètement parenthésée

On définit une version simplifiée de la notion de forme infixe avant d'aborder la version intéressante dans laquelle le nombre de parenthèses est minimisé.

La *forme infixe complètement parenthésée* d'un terme t est un mot $\mathrm{Infcp}(t)$ sur l'alphabet X des lexèmes défini par les règles :

si $t = f \in \Sigma_0 \cup V$, $\mathrm{Infcp}(t) = f$;

si $t = f(s)$ où f est unaire préfixe, $\mathrm{Infcp}(t) = f[\mathrm{Infcp}(s)]$;

si $t = f(s)$ où f est unaire postfixe, $\mathrm{Infcp}(t) = [\mathrm{Infcp}(s)]f$;

si $t = f(r,s)$ $\mathrm{Infcp}(t) = [\mathrm{Infcp}(r)]f[\mathrm{Infcp}(s)]$.

Par exemple, $\mathrm{Infcp}(\times(+(ln(x),y),z)) = [[ln[x]] + [y]] \times [z]$.

Associativité d'un symbole binaire et priorité d'un symbole

On suppose ici

– que, comme il a été vu, les symboles unaires sont partagés entre les préfixes et les postfixes;

– que les symboles binaires sont partagés en trois groupes, les *associatifs à gauche*, les *associatifs à droite* et les *non associatifs*;

– et qu'à chaque $f \in \Sigma \cup V$ est associé un entier $pr(f) > 0$ appelé la *priorité* de f vérifiant les propriétés suivantes :

1. deux symboles de même priorité sont de même nature (par exemple, tous deux binaires associatifs à gauche);
2. les éléments de $\Sigma_0 \cup V$ ont tous la priorité maximum.

Le tableau 3.1 fournit ces éléments pour les trois exemples de base.[2]

	expressions					
priorité	arithmétiques		logiques		rationnelles	
1	$+\ -$	gauche	$\Rightarrow\ \Leftrightarrow$	non associatif	$+$	gauche
2	*moins*	préfixe	$\vee$	gauche	$\times$	gauche
3	$\times\ /$	gauche	$\wedge$	gauche	$*$	postfixe
4	ln, etc.	préfixe	$\neg$	préfixe	Σ_0	
5	$\mathbf{R} \cup V$		$\{0,1\} \cup V$			

TAB. 3.1 – *Priorités et qualités des symboles*

Forme concrète infixée minimale et autres formes infixées

La *forme infixe minimale* $\mathrm{Inf}(t)$ d'un terme t est définie par les règles suivantes où on convient de noter $pr(t)$ la priorité de l'étiquette de la racine de t :

– si $t = f \in \Sigma_0 \cup V$, $\mathrm{Inf}(t) = f$;

– si $t = f(s)$, f préfixe, $\mathrm{Inf}(t) = \begin{cases} f[\mathrm{Inf}(s)] & \text{si } pr(s) < pr(t) \\ f\ \mathrm{Inf}(s) & \text{si } pr(s) \geqslant pr(t); \end{cases}$

– si $t = f(s)$, f postfixe, $\mathrm{Inf}(t) = \begin{cases} [\mathrm{Inf}(s)]f & \text{si } pr(s) < pr(t) \\ \mathrm{Inf}(s)\ f & \text{si } pr(s) \geqslant pr(t); \end{cases}$

1. En général les parenthèses sont notées « (» et «) »; c'est pour des raisons typographiques qu'elles sont notées ici « [» et «] ».

2. Voir aussi le tableau A.1 p. 105.

– si $t = f(r,s)$, f associatif à gauche, $\mathrm{Inf}(t) = u\,f\,v$ où
$$u = \begin{cases} [\mathrm{Inf}(r)] & \text{si } pr(r) < pr(t) \\ \mathrm{Inf}(r) & \text{si } pr(r) \geqslant pr(t) \end{cases} \text{ et } v = \begin{cases} [\mathrm{Inf}(s)] & \text{si } pr(s) \leqslant pr(t) \\ \mathrm{Inf}(s) & \text{si } pr(s) > pr(t); \end{cases}$$
– si $t = f(r,s)$, f associatif à droite, $\mathrm{Inf}(t) = u\,f\,v$ où
$$u = \begin{cases} [\mathrm{Inf}(r)] & \text{si } pr(r) \leqslant pr(t) \\ \mathrm{Inf}(r) & \text{si } pr(r) > pr(t) \end{cases} \text{ et } v = \begin{cases} [\mathrm{Inf}(s)] & \text{si } pr(s) < pr(t) \\ \mathrm{Inf}(s) & \text{si } pr(s) \geqslant pr(t); \end{cases}$$
– si $t = f(r,s)$, f non associatif, $\mathrm{Inf}(t) = u\,f\,v$ où
$$u = \begin{cases} [\mathrm{Inf}(r)] & \text{si } pr(r) \leqslant pr(t) \\ \mathrm{Inf}(r) & \text{si } pr(r) > pr(t) \end{cases} \text{ et } v = \begin{cases} [\mathrm{Inf}(s)] & \text{si } pr(s) \leqslant pr(t) \\ \mathrm{Inf}(s) & \text{si } pr(s) > pr(t). \end{cases}$$
Un mot de X^* qui s'obtient à partir du mot $\mathrm{Inf}(t)$ en y ajoutant des paires de parenthèses inutiles est appelé une *forme infixe* de t. La définition formelle de l'ensemble $\mathrm{INF}(t)$ des formes infixes de t s'obtient en paraphrasant celle de $\mathrm{Inf}(t)$:

– si $u \in \mathrm{INF}(t)$ alors $[u] \in \mathrm{INF}(t)$;
– si $t = f \in \Sigma_0 \cup V$, $f \in \mathrm{INF}(t)$;
– si $t = f(s)$, f préfixe et si $u \in \mathrm{INF}(s)$, alors $\begin{cases} f[u] \in \mathrm{INF}(t) & \text{si } pr(s) < pr(t) \\ f\,u \in \mathrm{INF}(t) & \text{si } pr(s) \geqslant pr(t); \end{cases}$
– etc.

La forme infixe minimale $\mathrm{Inf}(t)$ et la forme infixe complètement parenthésée $\mathrm{Infcp}(t)$ d'un terme t sont deux formes infixes particulières de t.

Etudions quelques exemples.

– Dans le cas des expressions arithmétiques, comme $\times$ a une priorité $>$ à celle de $+$, la forme infixe minimale de $\times(+(a,b),c)$ est $[a+b]\times c$ alors que celle de $+(a,\times(b,c))$ est $a+b\times c$.
– Comme les symboles arithmétiques $+$ et $-$ sont associatifs à gauche, l'expression arithmétique $-(+(-(a,b),c),d)$ admet les formes infixes $[[a-b]+c]-d$ et $a-b+c-d$, la deuxième forme étant minimale; mais la forme minimale de $-(a,+(b,-(c,d)))$ est $a-[b+[c-d]]$.
– Comme les symboles logique $\Rightarrow$ et $\Leftrightarrow$ sont non associatifs, les formes infixes minimales de $\Rightarrow(\Rightarrow(p,q),r)$ et $\Rightarrow(p,\Rightarrow(q,r))$ sont respectivement $[p\Rightarrow q]\Rightarrow r$ et $p\Rightarrow[q\Rightarrow r]$.

IMPLÉMENTATION : La fonction `imprime_inf : (string -> int) -> (int -> qualité) -> terme -> unit` du programme 3.15 imprime en O de la taille la forme infixe minimale de son dernier argument, les deux premiers arguments servant respectivement à définir les priorités des symboles [1] et les caractéristiques des symboles de priorité donnée. Par exemple, `imprime_inf priorité_arithmétique qualité_arithmétique (B("*",B("+",V"x",V"y"),V"z"))` imprime `(x+y)*z`.

Non ambiguïté d'une forme infixe

On note $L = \bigcup_{t\in T(\Sigma,V)} \mathrm{INF}(t)$ l'ensemble des mots de X^* qui sont des formes infixes de termes et $P = \{[u] \mid u \in L\} \subset L$.

Soit n la priorité maximum — celle des constantes et des variables —. On définit n langages $L_p = P \cup \left(\bigcup_{pr(t)\geqslant p} \mathrm{INF}(t)\right)$, $p = 1,\ldots,n$. On a donc $L = L_1 \supset L_2 \supset \cdots \supset L_n$. Les mots de $L_n = P \cup \Sigma_0 \cup V$ sont appelés des *atomes*

Un mot du langage L_p représente en quelque sorte un terme de priorité $\geqslant p$, à condition de convenir que si le mot est entouré de parenthèses, cela confère la priorité maximum au terme qu'il représente.

Les n langages L_p sont aussi définis par la *grammaire* [2] G suivante :

1. $L_n ::= \Sigma_0 \mid V \mid [L_1]$ et, pour tout $p < n$,

1. Pour distinguer les variables des symboles constants, on convient de leur donner la priorité $n+1$.
2. Une grammaire est une manière abrégée de définir des langages par induction. Par exemple, la deuxième règle de la grammaire, $L_p ::= L_{p+1} \mid fL_p$, doit s'interpréter comme l'ensemble suivant de règles de construction de L_p :
– si $u \in L_{p+1}$ alors $u \in L_p$;
– si $f \in \Sigma$ est de priorité p et si $u \in L_p$ alors $fu \in L_p$.

Programme 3.15 Forme infixe d'un terme

```
type qualité =
   Constante | Variable | Préfixe | Postfixe | Gassoc | Dassoc | Nassoc;;

let imprime_inf priorité qualité =
   let pr = function
      V x -> priorité x - 1 |
      C f -> priorité f |
      U(f,_) -> priorité f |
      B(f,_,_) -> priorité f in
   let rec imp =
      let i parenthèse t =
         if parenthèse then print_char `(`;
         imp t;
         if parenthèse then print_char `)` in
   function
      V x -> print_string x |
      C f -> print_string f |
      U(f,u) -> (match qualité (priorité f) with
                   Préfixe -> print_string f; i (pr u < priorité f) u |
                   Postfixe -> i (pr u < priorité f) u; print_string f) |
      B(f,u,v) ->
         let q = qualité (priorité f) in
         i (pr u < priorité f or pr u = priorité f & q <> Gassoc) u;
         print_string f;
         i (pr v < priorité f or pr v = priorité f & q <> Dassoc) v in
   imp;;

let  priorité_arithmétique = function
   "+" | "-" -> 1 |
   "moins" -> 2 |
   "*" | "/" -> 3 |
   "ln" | "exp" | "sin" | "cos" | "arctan" -> 4 |
   s -> let c = int_of_char s.[0] - int_of_char `0` in
        if 0 <= c & c <= 9 then 5 else 6
and qualité_arithmétique = function
   1 -> Gassoc |
   2 -> Préfixe |
   3 -> Gassoc |
   4 -> Préfixe |
   5 -> Constante |
   _ -> Variable;;
```

2. si les symboles f de priorité p sont unaires préfixes, $L_p ::= L_{p+1} \mid fL_p$;
3. si les symboles f de priorité p sont unaires postfixes, $L_p ::= L_{p+1} \mid L_p f$;
4. si les symboles f de priorité p sont binaires associatifs à gauche, $L_p ::= L_{p+1} \mid L_p f L_{p+1}$;
5. si les symboles f de priorité p sont binaires associatifs à droite, $L_p ::= L_{p+1} \mid L_{p+1} f L_p$;
6. si les symboles f de priorité p sont binaires non associatifs, $L_p ::= L_{p+1} \mid L_{p+1} f L_{p+1}$.

Une propriété importante des grammaires est la *non ambiguïté*. On dit que G est non ambiguë si, pour tout p et pour tout mot $u \in L_p$, il y a unicité de la règle qu'il faut appliquer pour montrer que u est dans L_p et de la manière d'appliquer cette règle. Par exemple, pour la règle 4, $L_{p+1} \cap L_p f L_{p+1} = \emptyset$ et, si $u \in L_p \setminus L_{p+1}$, il y a un unique f de priorité p et un unique $(v,w) \in L_p \times L_{p+1}$ tel que $u = vfw$.

Proposition 3.8 *La grammaire G est non ambiguë et, pour tout mot $u \in L$, il y a unicité du terme dont u est une forme infixe.*

Preuve

On appellera *poids* d'un mot $u \in X^*$ la différence entre le nombre d'occurrences de $[$ et le nombre d'occurrences de $]$ dans u; poids : $X^* \to \mathbf{Z}$ est le morphisme de monoïdes défini par poids($[$) $= 1$, poids($]$) $= -1$ et poids$(f) = 0$ pour tout $f \in \Sigma \cup V$.

LEMME 1. *Pour tout mot $u \in L$, u est de poids nul et tout préfixe de u a un poids $\geqslant 0$.*

Le lemme 1 se démontre sans difficulté par induction.

LEMME 2. *Si $u,v \in X^*$ et $f \in \Sigma \cup V$ vérifient $pr(f) < p$ et $ufv \in L_p$ alors u est de poids > 0 ou, de manière équivalente vu le lemme 1, v est de poids < 0.*

On démontre le lemme 2 par récurrence desendante sur p :

- Si $p = n$ c'est une conséquence du lemme 1.
- Si le lemme est vérifié pour $p+1$, montrons le pour p. On suppose que les symboles de priorité p sont binaires, les deux autres cas (unaires préfixes et unaires postfixes) se traitant de manière analogue. Si $ufv \in L_p$ avec $pr(f) < p$ alors il existe $k > 0$, $u_0, \ldots ,u_k \in L_{p+1}$ et $f_1, \ldots ,f_k$ symboles de priorité p tels que $ufv = u_0 f_1 u_1 f_2 \ldots u_{k-1} f_k u_k$. Comme f est différent de chaque f_i, il existe h,u',v' tels que $u = u_0 f_1 u_1 f_2 \ldots u_{h-1} f_h u'$ et $u_h = u' f v'$. D'après le lemme 1, chaque u_i a un poids nul; donc le poids de u est le même que celui de u' qui est > 0 d'après l'hypothèse de récurrence.

On est maintenant en mesure de montrer que, pour tout $p = 1,\ldots ,n$, les règles qui définissent L_p sont non ambiguës :

– Si $p = n$, c'est clair.
– Si les symboles de priorité p sont unaires préfixes, alors un mot fu appartenant à fL_p avec $pr(f) = p$ ne peut pas appartenir à L_{p+1}; en effet, si $fu = \mu f u \in L_{p+1}$, le lemme 2 donnerait $0 = \text{poids}(\mu) > 0$.
– Si les symboles f de priorité p sont binaires associatifs à gauche, le lemme 2 montre que $L_{p+1} \cap L_p f L_{p+1} = \emptyset$ et aussi que si $u = vfw$ avec $v \in L_p$ et $w \in L_{p+1}$, alors w est le plus long mot de L_{p+1} suffixe de u.
– Les autres cas sont analogues.

Montrons enfin par induction que, pour tout p et tout mot $u \in L_p$, il existe un seul terme dont u est une forme infixe :

– Si $p = n$, le cas où $u \in \Sigma_0 \cup V$ est clair; dans le cas contraire, $u = [v]$ avec $v \in L_1$, et on peut appliquer l'hypothèse d'induction à v : le seul terme dont v est une forme infixe est aussi le seul terme dont u est une forme infixe.
– Si les symboles de priorité p sont binaires associatifs à gauche, le cas où $u \in L_{p+1}$ se déduit de l'hypothèse d'induction; dans le cas contraire, soit t un terme dont u soit une forme infixe. Comme $u \in L_p \setminus L_{p+1}$, $t = f(r,s)$ où $pr(f) = p$ donc $u = vfw$ où $v \in L_p$ est une forme infixe de r et $w \in L_{p+1}$ est une forme infixe de s. Comme la grammaire est non ambiguë, v, f et w sont déterminés de manière unique par la donnée de u et, d'après l'hypothèse d'induction, r et s sont déterminés de manière unique par v et w. Cela prouve l'unicité de t.
– Les autres cas sont analogues. ∎

Calcul d'un terme à partir d'une de ses formes infixes

Pour calculer le terme dont un mot $u \in L$ est une forme infixe, on applique la méthode récursive descendante déja utilisée p. 61 pour l'analyse d'une forme préfixe.

Un mot $u = u_1 \ldots u_n \in X^*$ étant donné, on lui ajoute une dernière lettre \$ n'appartenant pas à X, on introduit un pointeur k sur une lettre de $u\$$ et on écrit une fonction *expr* qui prend un entier en argument et renvoie un terme. Un appel de *expr p*, pour $p = 1,\ldots ,n$, est spécifié comme suit :

> soit v le plus long préfixe de $u_k u_{k+1} \ldots u_n\$$ qui est aussi préfixe d'un mot de L_p. *expr p* effectue la lecture de v et, si $v \in L_p$, renvoie le terme correspondant à v; sinon échoue.

Le programme 3.16 fournit la fonction `analyse_inf : int -> (string -> int) -> (int -> qualité) -> string list -> terme`. Par exemple, si u = `["x";"-";"y";"+";"z"]`, `analyse_inf 5 priorité_arithmétique qualité_arithmétique` u renvoie `B("+",B("-",V"x",V"y"),V"z")`.[1]

COMPLEXITÉ : L'algorithme termine et a une complexité en O de la taille du mot analysé.

1. Voir le programme 3.15 p. 65.

Programme 3.16 Analyse syntaxique d'une forme infixe d'un terme

```
let analyse_inf prioritémax priorité qualité l =
   let u = vect_of_list (l @ ["$"])
   and k = ref 0 in
   let rec expr p =
      if p = prioritémax
        then
           match u.(!k) with
              "(" -> incr k;
                     let t = expr 1 in
                     if u.(!k) <> ")" then raise Erreur;
                     incr k;
                     t |
              f -> incr k;
                    if priorité f = prioritémax then C f else V f
        else
           match qualité p with
              Préfixe -> let f = u.(!k) in
                          if priorité f = p
                             then (incr k; U(f,expr p))
                             else expr (p + 1) |
              Postfixe -> let t = ref (expr (p + 1)) in
                           while priorité u.(!k) = p do
                              t := U(u.(!k),!t);
                              incr k
                           done;
                           !t |
              Gassoc -> let t = ref (expr (p + 1)) in
                         while priorité u.(!k) = p do
                            let f = u.(!k) in
                            incr k;
                            t := B(f,!t,expr (p + 1))
                         done;
                         !t |
              Dassoc -> let t = expr (p + 1) in
                         if priorité u.(!k) = p
                            then let f = u.(!k) in incr k; B(f,t,expr p)
                            else t |
              Nassoc -> let t = expr (p + 1) in
                         if priorité u.(!k) = p
                            then let f = u.(!k) in incr k; B(f,t,expr(p+1))
                            else t in
   let t = expr 1 in if u.(!k) = "$" then t else raise Erreur;;
```

Preuve Il termine car, au bout d'un maximum de n appels récursifs, k est augmenté d'au moins une unité. On montre alors par induction que *expr* a une complexité en O de la taille du mot lu. ∎

Exercice 3.7
1. Soit f_0 un symbole binaire fixé.[1] Montrer que l'application qui, à $u \in L$, associe le mot u' obtenu à partir de u en supprimant toutes les occurrences de f_0 est injective. Modifier la fonction `analyse_inf` pour qu'elle fonctionne aussi bien avec l'argument u' qu'avec u.
2. Dans le cas des expressions arithmétiques, montrer que l'application qui, à $u \in L$, associe le mot obtenu en remplaçant chaque occurence de *moins* par $-$ est injective.

1. Par exemple le symbole $\times$ des expressions arithmétiques ou des expressions rationnelles.

Chapitre 4

Automates finis

Dans ce chapitre, X désigne un alphabet fini.

4.1 Automates finis non déterministes

4.1.1 Définition

Un *automate fini* — en abrégé AF — sur X est un quadruplet $\mathcal{A} = (Q,I,F,\Delta)$ tel que :

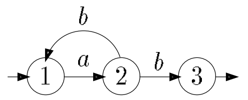
- Q est un ensemble fini dont les éléments sont appelés les *états* de l'automate $\mathcal{A}$;
- I est une partie de Q dont les éléments sont appelés les *états initiaux* ou *états de départ*;
- F est une partie de Q dont les éléments sont appelés les *états finaux* ou encore *états d'arrivée* ou *états d'acceptation*;
- Δ est une partie de $Q \times X \times Q$ dont les éléments sont appelés les *transitions* de $\mathcal{A}$.

Une transition $t = (q,x,q') \in \Delta$ étant donnée, q est l'*origine*, q' l'*extrémité* et x l'*étiquette* ou la *trace* de la transition t.

Le prédicat $(q,x,q') \in \Delta$ est noté $q \xrightarrow{x}_{\mathcal{A}} q'$ ou $q \xrightarrow{x} q'$ si aucune confusion n'est possible. On dit aussi que q' est un état *atteint* par *lecture* de la lettre x à partir de l'état q.

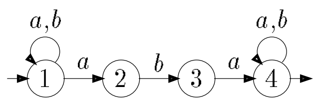
4.1.2 Représentations d'un automate

La figure 4.1 illustre les deux principales manières de représenter un AF.



	$\downarrow$ 1	2	$\uparrow$ 3
a	$\{2\}$	$\emptyset$	$\emptyset$
b	$\emptyset$	$\{1,3\}$	$\emptyset$

a. $(\{1,2,3\},\{1\},\{3\},\{(1,a,2),(2,b,3),(2,b,1)\})$



	$\downarrow$ 1	2	3	$\uparrow$ 4
a	$\{1,2\}$	$\emptyset$	$\{4\}$	$\{4\}$
b	$\{1\}$	$\{3\}$	$\emptyset$	$\{4\}$

b. $(\{1,2,3,4\},\{1\},\{4\},\{(1,a,2),(2,b,3),(3,a,4),(1,a,1),(1,b,1),(4,a,4),(4,b,4)\})$

FIG. 4.1 – *Représentations sagittale et fonctionelle de deux automates sur l'alphabet* $\{a,b\}$

La *représentation sagittale* utilise des conventions semblables à celles des graphes (voir p. 31). Chaque état est représenté par un cercle à l'intérieur duquel on indique éventuellement son nom; les états initiaux (resp. finaux) étant munis d'une flèche rentrante (resp. sortante). On représente une transition $q \xrightarrow{x} q'$ par une flèche étiquetée par x joignant q à q'.[1]

1. Si plusieurs transitions joignent q à q', on ne dessine qu'une flèche de q à q' étiquetée par l'ensemble des lettres x telles que $(q,x,q') \in \Delta$.

La *représentation fonctionelle* est un tableau à double entrée qui, pour tout couple (q,x) formé d'un état et d'une lettre, fournit l'ensemble $\delta(q,x)$ des états q' atteints par lecture de la lettre x à partir de q. Pour spécifier les états initiaux et finaux, on place des flèches rentrantes et sortantes au dessus des états.

4.1.3 Langage reconnu par un automate

Un *calcul* ou *chemin* d'un automate $\mathcal{A} = (Q,I,F,\Delta)$ est une suite finie
$$C = (q_0,x_1,q_1,x_2,q_2,\dots,q_{n-1},x_n,q_n)$$
avec $n \geqslant 0$, $q_0,\dots,q_n \in Q$, $x_1,\dots,x_n \in X$ et $\forall i \in [|1,n|], (q_{i-1},x_i,q_i) \in \Delta$; donc

$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} \cdots \xrightarrow{x_n} q_n.$$

Les états q_0 et q_n sont appelés respectivement l'*origine* et l'*extrémité* du calcul C. Le mot $u = x_1 \dots x_n \in X^*$ est appelé l'*étiquette* ou la *trace* du calcul. La longueur $n = |u|$ du mot u est aussi appelée la *longueur* du calcul.

En particulier, pour tout état q, il existe un unique calcul de longueur nulle et d'origine q; ce calcul $C = (q)$ admet aussi q pour extrémité et a pour étiquette le mot vide $\mu \in X^*$.

Deux états q et q' et un mot u étant donnés, le prédicat « il existe un calcul d'origine q, d'extrémité q' et d'étiquette u » est noté

$$q \xrightarrow{u}_{\mathcal{A}} q' \quad \text{ou} \quad q \xrightarrow{u} q'.$$

On dit aussi que q' est un état *atteint* par *lecture* du mot u à partir de l'état q.

Un calcul est dit *réussi* si son origine est un état initial et son extrémité un état final. Un mot $u \in X^*$ est dit *reconnu* ou *accepté* par l'automate $\mathcal{A}$ s'il existe un calcul réussi d'étiquette u. Par exemple, le mot vide μ est reconnu ssi $I \cap F \neq \emptyset$. L'ensemble des mots reconnus par $\mathcal{A}$ est appelé le *langage reconnu* par $\mathcal{A}$ et noté $|\mathcal{A}|$ ou $L(\mathcal{A})$. On a donc

$$u \in |\mathcal{A}| \Leftrightarrow \exists d,f \ : \ I \ni d \xrightarrow{u}_{\mathcal{A}} f \in F.$$

Un langage reconnu par un AF est dit *reconnaissable par automate fini* ou, simplement *reconnaissable*. L'ensemble des langages reconnaissables sur X est une partie de $\mathcal{P}(X^*)$ notée $\mathrm{Rec}(X^*)$.

Exemple 4.1 *Langage reconnu par l'automate de la figure* 4.1 a.

La première transition d'un calcul réussi de cet automate est nécessairement $1 \xrightarrow{a} 2$ et la dernière transition est $2 \xrightarrow{b} 3$. Les autres transitions forment un calcul menant de 2 à 2. Or les calculs de 2 à 2 sont $\underbrace{2 \xrightarrow{b} 1 \xrightarrow{a} 2 \cdots 2 \xrightarrow{b} 1 \xrightarrow{a} 2}_{2n \text{ transitions}}$ pour $n \geqslant 0$. Les calculs réussis sont donc $C_n = \underbrace{1 \xrightarrow{a} 2 \xrightarrow{b} 1 \xrightarrow{a} 2 \cdots 2 \xrightarrow{b} 1 \xrightarrow{a} 2 \xrightarrow{b} 3}_{2n+2 \text{ transitions}}$, $n \geqslant 0$. Comme l'étiquette de C_n est $\underbrace{ab \dots ab}_{n+1 \text{fois}} = (ab)^{n+1}$, le langage reconnu est $\{(ab)^{n+1} \mid n \geqslant 0\} = (ab)^+$

Exemple 4.2 *Langage reconnu par l'automate de la figure* 4.1 b.

Il s'agit de l'ensemble $L = (a+b)^* aba (a+b)^*$ des mots qui admettent le facteur aba. En effet, un mot $u = vabaw \in L$ est reconnu car $1 \xrightarrow{v} 1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{a} 4 \xrightarrow{w} 4$ et réciproquement, tous les chemins réussis sont de cette forme.

Noter qu'il peut exister plusieurs chemins réussis admettant pour étiquette un même mot de L. Par exemple, si $u = ababa$, on a $1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{a} 4 \xrightarrow{b} 4 \xrightarrow{a} 4$ et $1 \xrightarrow{a} 1 \xrightarrow{b} 1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{a} 4$.

Exercice 4.1 Le *miroir* d'un mot $u = x_1 \dots x_n \in X^*$ est le mot $m(u) = x_n \dots x_1$ et le *miroir* d'un langage $L \subset X^*$ est $m(L) = \{m(u) \mid u \in L\}$. Définir, pour tout AF $\mathcal{A}$, un AF $m(\mathcal{A})$ tel que $|m(\mathcal{A})| = m(|\mathcal{A}|)$ et $m(m(\mathcal{A})) = \mathcal{A}$.

4.1.4 Fonction de transition d'un automate

La *fonction de transition* d'un automate $\mathcal{A} = (Q,I,F,\Delta)$ est l'application qui, à un couple $(q,x) \in Q \times X$ associe l'ensemble d'états $q \bullet x = \{q' \in Q \mid q \xrightarrow{x} q'\} \in \mathcal{P}(Q)$. Autres notations possibles : $q \bullet_{\mathcal{A}} x$ (pour spécifier l'automate) ou $\delta(q,x)$.

Il est clair que la donnée de la fonction de transition $\bullet$ permet de retrouver l'ensemble des transitions : $\Delta = \{(q,x,q') \mid q' \in q \bullet x\}$.[1] C'est pourquoi il arrive que l'on spécifie un automate par sa fonction de transition plutôt que par Δ et que l'on utilise l'abus de notation $\mathcal{A} = (Q,I,F,\bullet)$.

Plus généralement, soient $P \subset Q$ un ensemble d'états et $u \in X^*$ un mot. On note encore $P \bullet u$ l'ensemble des états que l'on peut atteindre par lecture du mot u à partir d'un état de P : $P \bullet u = \{q' \in Q \mid \exists q \in P, q \xrightarrow{u} q'\}$. L'application $(P,u) \mapsto P \bullet u$ de $\mathcal{P}(Q) \times X^*$ dans $\mathcal{P}(Q)$ ainsi définie est appelée la *fonction de transition étendue* de l'automate $\mathcal{A}$.

En particulier, si $P = \{q\}$ est réduit à un singleton, on peut assimiler $P \in \mathcal{P}(Q)$ à $q \in Q$ et noter $q \bullet u = \{q' \in Q \mid q \xrightarrow{u} q'\}$.

Le langage reconnu peut être défini en termes de la fonction de transition étendue :

$$u \in |\mathcal{A}| \Leftrightarrow (I \bullet u) \cap F \neq \emptyset.$$

De plus, on dispose de la proposition suivante qui permet de faire des calculs.

Proposition 4.1 $\forall P \subset Q, \forall u,v \in X^*$,

$$\begin{array}{ll} (1) & P \bullet \mu = P \\ (2) & (P \bullet u) \bullet v = P \bullet uv \end{array}$$

Preuve (1) étant évident, montrons (2).
Pour tout $q \in Q$, on a, en notant $u = x_1 \dots x_n$ et $v = y_1 \dots y_m$:
$q \in (P \bullet u) \bullet v$
$\Leftrightarrow \exists p_0, \dots, p_m : P \bullet u \ni p_0 \xrightarrow{y_1} p_1 \cdots \xrightarrow{y_m} p_m = q$
$\Leftrightarrow \exists p_0, \dots, p_m, \exists q_0, \dots, q_n : P \ni q_0 \xrightarrow{x_1} q_1 \cdots \xrightarrow{x_n} q_n = p_0 \xrightarrow{y_1} p_1 \cdots \xrightarrow{y_m} p_m = q$
$\Leftrightarrow q \in P \bullet uv$ ■

L'algorithme 4.1 teste si un mot $u = u_1 \dots u_n$ est reconnu par un AF.

Algorithme 4.1 Reconnaissance d'un mot par un AF

soit AF_reconnaît $(Q,I,F,\bullet)$ $u_1 \dots u_n$ =
  soit $P \leftarrow I$ **dans**
  pour $i = 1$ **à** n **faire** $P \leftarrow \bigcup_{q \in P} q \bullet u_i$ $\{P = I \bullet u_1 \dots u_i\}$
  renvoyer $P \cap F \neq \emptyset$

Exemple 4.3 Soit $\mathcal{A}$ l'automate de la figure 4.1 a. Donnons une deuxième démonstration de $|\mathcal{A}| = (ab)^+$.

Comme $1 \bullet ab = \{1,3\}$ et $\{1,3\} \bullet ab = \{1,3\}$, on voit par récurrence que $\forall n > 0, 1 \bullet (ab)^n = \{1,3\}$. Comme 3 est final, cela montre déja que $(ab)^+ \subset |\mathcal{A}|$.

Réciproquement, soit $u \notin (ab)^+$. En considérant le plus long préfixe de u de la forme $(ab)^n$ où $n \geqslant 0$, u s'écrit sous l'une des trois formes :

- $u = (ab)^n a$, auquel cas $1 \bullet u = 1 \bullet (ab)^n \bullet a \subset \{1,3\} \bullet a = \{2\}$;
- $u = (ab)^n a^2 v$, auquel cas $1 \bullet u \subset \{1,3\} \bullet a^2 \bullet v = \emptyset \bullet v = \emptyset$;
- $u = (ab)^n bv$, auquel cas $1 \bullet u \subset \{1,3\} \bullet b \bullet v = \emptyset \bullet v = \emptyset$.

On peut encore prolonger la fonction de transition étendue. Si $P \in \mathcal{P}(Q)$ est un ensemble d'états et si $L \in \mathcal{P}(X^*)$ est un langage, on note encore $P \bullet L$ l'ensemble des états que l'on peut atteindre en lisant un mot de L à partir d'un état de P;

$$P \bullet L = \{q' \in Q \mid \exists q \in P, \exists u \in L, q \xrightarrow{u} q'\}.$$

1. C'est d'ailleurs ce fait qui permet de spécifier un automate par sa représentation fonctionelle; en effet, cette représentation fournit précisément la fonction de transition.

Introduisons une dernière notation utile. Si P et P' sont deux ensembles d'états, on note $(P^{-1}P')_{\mathcal{A}}$ ou simplement $P^{-1}P'$ l'ensemble des étiquettes des calculs d'origine dans P et d'extrémité dans P'; $P^{-1}P' = \{u \in X^* \mid \exists(p,p') \in P \times P', p \xrightarrow{u} p'\}$. En particulier,

$$|\mathcal{A}| = I^{-1}F.$$

Exercice 4.2 Soient $L,L',L'' \subset X^*$ et $P,P' \subset Q$. On note :
$L^{-1}L' = \{u \in X^* \mid Lu \cap L' \neq \emptyset\}$,
$LL'^{-1} = \{u \in X^* \mid L \cap uL' \neq \emptyset\}$ et
$PL^{-1} = \{q \in Q \mid P \cap (q \bullet L) \neq \emptyset\}$. Montrer que

$$\begin{array}{l|l}
(LL')L'' = L(L'L'') & (P \bullet L) \bullet L' = P \bullet (LL') \\
(LL'^{-1})L''^{-1} = L(L''L')^{-1} & (PL^{-1})L'^{-1} = P(L'L)^{-1} \\
(L^{-1}L')L''^{-1} = L^{-1}(L'L''^{-1}) & (P^{-1}P')L^{-1} = P^{-1}(P'L^{-1}) \\
L^{-1}(L'^{-1}L'') = (L'L)^{-1}L'' & L^{-1}(P^{-1}P') = (P \bullet L)^{-1}P'.
\end{array}$$

4.1.5 Suppression d'états inutiles

Soit un AF $\mathcal{A} = (Q,I,F,\Delta)$ et $P \subset Q$ un ensemble d'états. On note $\mathcal{A} \setminus P$ l'automate obtenu à partir de $\mathcal{A}$ en supprimant les états de P et les transitions dont l'origine ou l'extrémité est dans P:

$$\mathcal{A} \setminus P = ((Q \setminus P),(I \setminus P),(F \setminus P),\Delta \setminus ((P \times X \times Q) \cup (Q \times X \times P))).$$

Comme un calcul réussi de $\mathcal{A} \setminus P$ est aussi un calcul réussi de $\mathcal{A}$, on a

$$|\mathcal{A} \setminus P| \subset |\mathcal{A}|.$$

Un état de $\mathcal{A}$ est dit *accessible* s'il est l'extrémité d'un calcul dont l'origine est un état initial. L'ensemble des états accessibles est donc

$$acc(\mathcal{A}) = I \bullet X^*.$$

Un état est dit *coaccessible* s'il est l'origine d'un calcul dont l'extrémité est un état final. L'ensemble des états coaccessibles est donc

$$coacc(\mathcal{A}) = \{q \in Q \mid q^{-1}F \neq \emptyset\}.$$

Un *état puits* est un état q tel que toute transition d'origine q a aussi q pour extrémité. Un état puits qui n'est pas final est appelé un *état poubelle*. Un état poubelle n'est pas coaccessible.

Un état est dit *biaccessible* s'il est à la fois accessible et coaccessible, autrement dit si c'est l'un des états d'un calcul réussi.

L'automate $\mathcal{A}$ est dit *accessible* (resp. *coaccessible*, *biaccessible*) si tous ses états sont accessibles (resp. coaccessibles, biaccessibles).

Deux automates sont dits *équivalents* s'ils reconnaissent le même langage.

On ne change pas le langage reconnu en supprimant des — ou tous les — états non biaccessibles. Précisément, on a :

Proposition 4.2

1. *Soit $P \subset Q$ un ensemble d'états non accessibles d'un automate $\mathcal{A} = (Q,I,F,\Delta)$:*
   a. *les états accessibles de $\mathcal{A} \setminus P$ sont les états accessibles de $\mathcal{A}$;*
   b. *$|\mathcal{A} \setminus P| = |\mathcal{A}|$;*
   c. *Si P est l'ensemble de tous les états non accessibles de $\mathcal{A}$, $\mathcal{A} \setminus P$ est un automate accessible équivalent à $\mathcal{A}$.*
2. idem *en remplaçant « accessible » par « coaccessible ».*
3. idem *en remplaçant « accessible » par « biaccessible ».*

Preuve
1. Les calculs ayant pour origine un état initial sont les mêmes dans les deux automates $\mathcal{A}$ et $\mathcal{A} \setminus P$. En effet, si $I \ni q_0 \xrightarrow{x_1}_{\mathcal{A}} q_1 \cdots \xrightarrow{x_n}_{\mathcal{A}} q_n$ est un tel calcul dans $\mathcal{A}$, alors chacun des états q_i est accessible dans $\mathcal{A}$ donc $q_i \in Q \setminus P$ et chacune des transitions $q_{i-1} \xrightarrow{x_i} q_i$ est encore une transition de $\mathcal{A} \setminus P$; le calcul donné est donc encore un calcul de $\mathcal{A} \setminus P$.
   a. Comme les états accessibles d'un automate sont les extrémités de ces calculs, ce sont

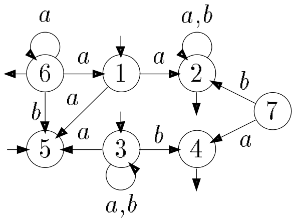
les mêmes dans $\mathcal{A}$ et dans $\mathcal{A} \setminus P$.

b.Comme, de plus, un état de $Q \setminus P$ est final dans $\mathcal{A}$ ssi il est final dans $\mathcal{A} \setminus P$, les calculs réussis sont les mêmes dans $\mathcal{A}$ et dans $\mathcal{A} \setminus P$; donc ces deux automates reconnaissent le même langage.

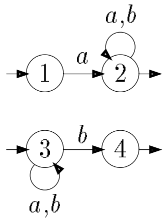
c. se déduit de a. et b.

2. et 3. Démonstrations analogues à 1. ■

L'automate, biaccessible équivalent à $\mathcal{A}$, obtenu en supprimant tous les états non biaccessibles de $\mathcal{A}$ est appelé l'*émondé* de $\mathcal{A}$. La figure 4.2 illustre cette notion.



un automate $\mathcal{A}$



l'émondé de $\mathcal{A}$

FIG. 4.2 – *Un automate $\mathcal{A}$ et son émondé. Les états accessibles de $\mathcal{A}$ sont* 1, 2, 3, 4 *et* 5*; tous les états sont coaccessibles sauf* 5 *qui est un état poubelle. On ne conserve que les états biaccessibles* 1, 2, 3 *et* 4. *L'émondé de $\mathcal{A}$, et donc aussi $\mathcal{A}$, reconnaît le langage $a(a+b)^* + (a+b)^*b$ des mots dont la première lettre est a ou la dernière lettre est b.*

Pour calculer les états accessibles ou coaccessibles d'un automate $\mathcal{A} = (Q,I,F,\Delta)$, on peut utiliser un parcours[1] du graphe $G_{\mathcal{A}} = (Q,\{(q,q') \mid \exists x : (q,x,q') \in \Delta\})$ induit par $\mathcal{A}$.

4.1.6 Une condition nécessaire de reconnaissabilité, le lemme de l'étoile

Soit L un langage reconnu par un AF $\mathcal{A} = (Q,I,F,\Delta)$ à $|Q| = n$ états. Si $z = x_1 \dots x_m$ est un mot de L de longueur $m \geqslant n$, soit $I \ni d = q_0 \xrightarrow{x_1} q_1 \cdots q_{m-1} \xrightarrow{x_m} q_m = f \in F$ un calcul réussi d'étiquette z. Comme le nombre $m+1$ d'états de ce calcul est $>$ au nombre total n d'états, il existe ℓ et h tels que

$$0 \leqslant \ell < h \leqslant n \text{ et } q_\ell = q_h.$$

Posons alors $u = x_1 \dots x_l$, $v = x_{\ell+1} \dots x_h$ et $w = x_{h+1} \dots x_m$, de sorte que $z = uvw$ et que, en notant $p = q_\ell = q_h$, $d \xrightarrow{u} p \xrightarrow{v} p \xrightarrow{w} f$. On en déduit que, pour $k \geqslant 0$, $d \xrightarrow{u} p \xrightarrow{v^k} p \xrightarrow{w} f$ donc $uv^k w \in L$. On a donc démontré :

Proposition 4.3 *(lemme de l'étoile)*

Si L est un langage reconnaissable, il existe $n \geqslant 0$ tel que tout mot $z \in L$ de longueur $|z| \geqslant n$ admet une factorisation $z = uvw$ telle que $|uv| \leqslant n$, $|v| \geqslant 1$ et $uv^*w \subset L$. ■

Pour montrer qu'un langage L n'est pas reconnaissable, il suffit donc d'exhiber une suite $(z_n)_{n \geqslant 0}$ de mots de L telle que

$$(\forall n, |z_n| \geqslant n) \ \wedge \ (\forall u,v,w, (z_n = uvw \wedge |uv| \leqslant n \wedge |v| \geqslant 1) \Rightarrow uv^*w \not\subset L)$$

Exemple 4.4 Le langage $L = \{a^n b^n \mid n \geqslant 0\}$ n'est pas reconnaissable.
En effet, soit $z_n = a^n b^n$. On a $|z_n| = 2n \geqslant n$ et si $z_n = uvw$ avec $|uv| \leqslant n$ et $|v| \geqslant 1$, alors $u = a^\alpha$, $v = a^\beta$ et $w = a^\gamma b^n$ avec $\alpha + \beta + \gamma = n$ et $\beta > 0$. Donc $uv^0w = a^{\alpha+\gamma}b^n \notin L$.

4.1.7 Fermeture de la classe des langages reconnaissables par les opérations rationnelles

Partant des langages de base $\emptyset$, μ et x, où $x \in X$ qui sont reconnaissables — voir figure 4.3 — on peut construire des langages plus élaborés en utilisant les *opérations rationnelles* qui sont l'addition, la concaténation et l'itération.[2] On se propose, dans cette section de

1. Voir la section 2.2.4 p. 33.
2. Voir p. 30.

$(\emptyset,\emptyset,\emptyset,\emptyset)$ $(\{1\},\{1\},\{1\},\emptyset)$ $(\{1,2\},\{1\},\{2\},\{(1,x,2)\})$

FIG. 4.3 – *Trois automates reconnaissant respectivement $\emptyset$, μ et x*

démontrer que tout langage construit de cette façon est reconnaissable. Plus précisément, on montre la fermeture de la classe $\mathrm{Rec}(X^*)$ par les opérations rationnelles; autrement dit que, si L et L' sont des langages reconnaissables, il en est de même de $L+L'$, LL' et L^*.

On suppose donnés deux automates $\mathcal{A} = (Q,I,F,\Delta)$ et $\mathcal{A}' = (Q',I',F',\Delta')$ reconnaissant respectivement L et L'. On peut supposer, quitte à renommer les états de Q', que $Q \cap Q' = \emptyset$.

Proposition 4.4 *(Fermeture de Rec(X^*) par addition)*

$$\mathcal{A}+\mathcal{A}' = (Q \cup Q', I \cup I', F \cup F', \Delta \cup \Delta') \text{ reconnaît } L+L'.$$

Preuve Comme un calcul réussi de $\mathcal{A}$ ou de $\mathcal{A}'$ est aussi un calcul réussi de $\mathcal{A}+\mathcal{A}'$, il est clair que $L+L' \subset |\mathcal{A}+\mathcal{A}'|$. Réciproquement, soit C un calcul réussi de $\mathcal{A}+\mathcal{A}'$: $I \cup I' \ni q_0 \xrightarrow{x_1}_{\mathcal{A}+\mathcal{A}'} q_1 \xrightarrow{x_2}_{\mathcal{A}+\mathcal{A}'} \cdots \xrightarrow{x_n}_{\mathcal{A}+\mathcal{A}'} q_n \in F \cup F'$. Supposons, par exemple, que $q_0 \in I$. Alors, comme $q_0 \notin Q'$, la transition $(q_0,x_1,q_1) \in \Delta \cup \Delta'$ ne peut appartenir à Δ', donc elle appartient à Δ. de même, $(q_1,x_2,q_2) \in \Delta$, et ainsi de suite jusqu'à $(q_{n-1},x_n,q_n) \in \Delta$. En particulier, $q_n \in Q \cap (F \cup F') = F$ et C est un calcul réussi de $\mathcal{A}$. ∎

Par exemple le deuxième automate de la figure 4.2 reconnaît $L+L'$ où L (resp. L') est le langage des mots qui commencent par a (resp. se terminent par b).

Proposition 4.5 *(Fermeture de Rec(X^*) par concaténation)*

LL' est reconnu par l'automate $\mathcal{AA}' = (Q \cup Q', I, F'', \Delta'')$ où

$$F'' = \begin{cases} F' & \text{si } I' \cap F' = \emptyset \\ F \cup F' & \text{si } I' \cap F' \neq \emptyset \end{cases} \text{ et } \Delta'' = \Delta \cup \Delta' \cup \{(f,x,q') \mid f \in F \wedge \exists d' \in I' : (d',x,q') \in \Delta'\}.$$

Preuve

Soit $u'' \in LL'$. $u'' = uu'$ avec $u = x_1 \ldots x_n \in L$ et $u' = x'_1 \ldots x'_m \in L'$.

- Si $u' = \mu$ alors $\mu \in L'$ et $I' \cap F' \neq \emptyset$; donc $F \subset F'' = F \cup F'$ et un calcul réussi de $\mathcal{A}$ de trace u est aussi un calcul réussi de $\mathcal{AA}'$, ce qui prouve que $u'' = u \in |\mathcal{AA}'|$.
- Si $u' \neq \mu$, considérons un calcul réussi de $\mathcal{A}$ d'étiquette u : $I \ni q_0 \xrightarrow{x_1}_{\mathcal{A}} q_1 \cdots q_{n-1} \xrightarrow{x_n}_{\mathcal{A}} q_n \in F$ et un calcul réussi de $\mathcal{A}'$ d'étiquette u' : $I' \ni q'_0 \xrightarrow{x'_1}_{\mathcal{A}'} q'_1 \cdots q'_{m-1} \xrightarrow{x'_m}_{\mathcal{A}'} q'_m \in F'$. Comme $(q_n,x'_1,q'_1) \in \Delta''$ (voir la définition de Δ''), on peut « coller » les deux calculs pour obtenir un calcul réussi de $\mathcal{AA}'$: $I \ni q_0 \xrightarrow{x_1} q_1 \cdots q_{n-1} \xrightarrow{x_n} q_n \xrightarrow{x'_1} q'_1 \cdots q'_{m-1} \xrightarrow{x'_m} q'_m \in F''$, ce qui montre que $u'' = uu' \in |\mathcal{AA}'|$.

Réciproquement, soit $C : I \ni q_0 \xrightarrow{x_1}_{\mathcal{AA}'} q_1 \cdots q_{p-1} \xrightarrow{x_p}_{\mathcal{AA}'} q_p \in F''$ un calcul réussi de $\mathcal{AA}'$ d'étiquette $v = x_1 \ldots x_p \in |\mathcal{AA}'|$. Soit n le plus grand des indices i tels que $q_i \in Q$. Alors $j \leqslant i \Rightarrow q_j \in Q$ (car il n'existe pas, dans $\mathcal{AA}'$ de transition d'un état de Q à un état de Q') et $I \ni q_0 \xrightarrow{x_1}_{\mathcal{A}} q_1 \cdots q_{n-1} \xrightarrow{x_n}_{\mathcal{A}} q_n$.

- Si $n = p$, comme $q_p \in F'' \cap Q$, on a $F'' \neq F'$ donc $F'' = F \cup F'$ et $q_p \in F$. C est donc un calcul réussi de $\mathcal{A}$ et $v \in L$. De plus on est dans le cas où $I' \cap F' \neq \emptyset$; donc $\mu \in L'$ et $v = v\mu \in LL'$.
- Si $n < p$, vu que la transition $Q \ni q_n \xrightarrow{x_{n+1}}_{\mathcal{AA}'} q_{n+1} \in Q'$ ne peut pas appartenir à $\Delta \cup \Delta'$, on a d'une part, $q_n \in F$ — ce qui prouve que $u = x_1 \ldots x_n \in L$ — et d'autre part il existe $d' \in I'$ tel que $(d',x_{n+1},q_{n+1}) \in \Delta'$. On en déduit le calcul de $\mathcal{A}'$: $I' \ni d' \xrightarrow{x_{n+1}}_{\mathcal{A}'} q_{n+1} \cdots q_{p-1} \xrightarrow{x_p} q_p$. Ce calcul est réussi car $q_p \in F'' \cap Q' = F'$; donc $u' = x_{n+1} \ldots x_p \in L'$ et $v = uu' \in LL'$. ∎

Exercice 4.3 Montrer que LL' est également reconnu par $(Q \cup Q', I''', F', \Delta''')$ où

$$I''' = \begin{cases} I & \text{si } I \cap F = \emptyset \\ I \cup I' & \text{si } I \cap F \neq \emptyset \end{cases} \text{ et } \Delta'' = \Delta \cup \Delta' \cup \{(q,x,d') \mid d' \in I' \wedge \exists f \in F : (q,x,f) \in \Delta\}.$$

On pourra procéder directement ou utiliser l'exercice 4.1 p. 70.

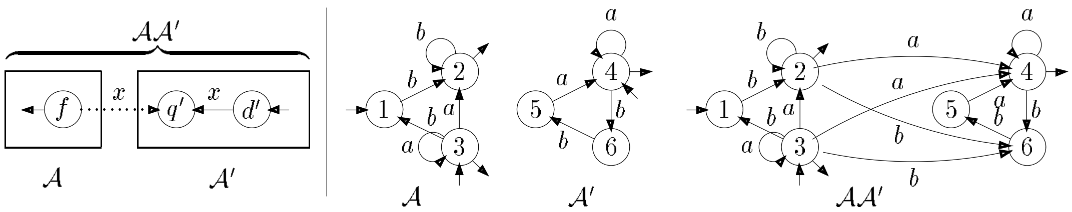


FIG. 4.4 – *Produit de deux automates. $\mathcal{A}$ contient deux états finaux f : 2 et 3. $\mathcal{A}'$ contient deux couples (x,q') tels qu'il existe un état initial d' avec $(d',x,q') \in \Delta'$: $(a,4)$ et $(b,6)$. Il y a donc $2 \times 2 = 4$ transitions (f,x,q') à rajouter. Les états initiaux de $\mathcal{A}\mathcal{A}'$ sont ceux de $\mathcal{A}$ et, comme $\mathcal{A}'$ contient un état à la fois initial et final, les états finaux de $\mathcal{A}\mathcal{A}'$ sont ceux de $\mathcal{A}$ et de $\mathcal{A}'$.*

La figure 4.4 illustre le procédé de construction de $\mathcal{A}\mathcal{A}'$.

Proposition 4.6 *(Fermeture de Rec(X^*) par itération stricte)*

$\mathcal{A}^+ = (Q,I,F,\Delta \cup \{(f,x,q) \mid f \in F \ \wedge\ \exists d \in I : (d,x,q) \in \Delta\})$ *reconnaît* L^+.

Preuve On a déjà $\mu \in L^+ \Leftrightarrow \mu \in L \Leftrightarrow I \cap F \neq \emptyset \Leftrightarrow \mu \in |\mathcal{A}^+|$.

Soit $u \in X^* \setminus \{\mu\}$.

Si $u \in L^+$, $u = u_1 \ldots u_n$ avec $\forall i, u_i \in L$. En supprimant les facteurs u_i égaux à μ, on peut se ramener au cas où $\forall i, u_i \neq \mu$. Donc, pour tout $i \in [|1,n|]$, on peut écrire $u_i = x_i v_i$ avec $x_i \in X$ et $v_i \in X^*$ et, comme $u_i \in L$, il existe des états d_i, q_i et f_i tels que $I \ni d_i \xrightarrow{x_i}_{\mathcal{A}} q_i \xrightarrow{v_i}_{\mathcal{A}} f_i \in F$. Pour $i \in [|2,n|]$, comme $f_{i-1} \in F$, $d_i \in I$ et $(d_i,x_i,q_i) \in \Delta$, on a $f_{i-1} \xrightarrow{x_i}_{\mathcal{A}^+} q_i$; donc $f_{i-1} \xrightarrow{u_i}_{\mathcal{A}^+} f_i$ car $q_i \xrightarrow{v_i}_{\mathcal{A}^+} f_i$. Finalement, $I \ni d_1 \xrightarrow{u_1}_{\mathcal{A}^+} f_1 \xrightarrow{u_2}_{\mathcal{A}^+} f_2 \cdots f_{n-1} \xrightarrow{u_n}_{\mathcal{A}^+} f_n \in F$ et $u = u_1 u_2 \ldots u_n \in |\mathcal{A}^+|$.

Réciproquement, supposons que $u \in |\mathcal{A}^+|$. Soit C un calcul réussi de $\mathcal{A}^+$ d'étiquette u. En mettant en évidence les transitions (f_i,x_i,q_i) de C qui n'appartiennent pas à Δ, on trouve une factorisation $u = w_0 x_1 w_1 \ldots x_n w_n$ de u (avec $x_i \in X$ et $w_i \in X^*$) telle que $I \ni d \xrightarrow{w_0}_{\mathcal{A}} f_1 \xrightarrow{x_1}_{\mathcal{A}^+} q_1 \xrightarrow{w_1}_{\mathcal{A}} f_2 \cdots f_n \xrightarrow{x_n}_{\mathcal{A}^+} q_n \xrightarrow{w_n}_{\mathcal{A}} f_{n+1} \in F$. Soit $i \in [|1,n|]$. Comme $(f_i,x_i,q_i) \notin \Delta$, $f_i \in F$ et il existe $d_i \in I$ tel que $(d_i,x_i,q_i) \in \Delta$. Donc $I \ni d_i \xrightarrow{x_i}_{\mathcal{A}} q_i \xrightarrow{w_i}_{\mathcal{A}} f_{i+1} \in F$, ce qui prouve que $x_i w_i \in L$. Comme, de plus, $I \ni d \xrightarrow{w_0}_{\mathcal{A}} f_1 \in F$, $u = (w_0)(x_1 w_1) \ldots (x_n w_n)$ est un produit de mots de L; soit $u \in L^+$. ■

Exercice 4.4 Montrer que $(Q,I,F,\Delta \cup \{(q,x,d) \mid d \in I \ \wedge\ \exists f \in F : (q,x,f) \in \Delta\})$ reconnaît également L^+.

Proposition 4.7 *(Fermeture de Rec(X^*) par itération)*

Soit $\mathcal{V} = (\{p\},\{p\},\{p\},\emptyset)$ *où* $p \notin Q$ *est un nouvel état.* $\mathcal{A}^* = \mathcal{A}^+ + \mathcal{V}$ *reconnaît* L^*.

Preuve $|\mathcal{A}^*| = |\mathcal{A}^+| + |\mathcal{V}| = L^+ + \mu = L^*$ (propositions 4.4 et 4.6). ■

La figure 4.5 illustre les constructions de $\mathcal{A}^+$ et de $\mathcal{A}^*$.

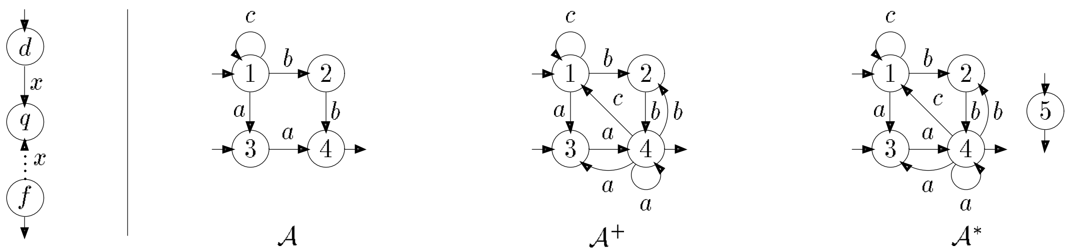


FIG. 4.5 – *Itéré strict et itéré d'un automate. $\mathcal{A}$ contient un état final $f = 4$ et quatre couples (x,q) tels qu'il existe un état initial d avec $(d,x,q) \in \Delta$: $(c,1)$, $(b,2)$, $(a,3)$ et $(a,4)$. Il y a donc $1 \times 4 = 4$ transitions (f,x,q) à rajouter pour obtenir $\mathcal{A}^+$.*

Remarque 4.1 La construction de $\mathcal{A}^*$ est basée sur le fait — appliqué à $\mathcal{A}^+$ au lieu de $\mathcal{A}$ — que $|\mathcal{A} + \mathcal{V}| = |\mathcal{A}| + \mu = L + \mu$. Dans les trois cas suivants on peut trouver un automate reconnaissant $L + \mu$ sans qu'il soit nécessaire de rajouter un état à $\mathcal{A}$:

S'il existe un état de $\mathcal{A}$ à la fois initial et final, alors $\mu \in L$ donc $L + \mu = L = |\mathcal{A}|$.

S'il y a un état initial d qui n'est l'extrémité d'aucune transition, $L+\mu = |(Q,I,F\cup\{d\},\Delta)|$.

S'il y a un état final f qui n'est l'origine d'aucune transition, $L + \mu = |(Q,I \cup \{f\},F,\Delta)|$.

Rappelons — voir les exemples 3.5 p. 53 et 3.8 p. 54 — qu'une expression rationnelle t est un Σ-terme clos où $\Sigma = X \cup \{\emptyset,\mu, *\,, +\,,\times\}$ ($\emptyset$, μ et $x \in X$ sont des symboles constants, $*$ est unaire et $+$ et $\times$ sont binaires) et que le langage reconnu par t est l'interprétation de t quand on interprète les symboles constants par les langages $\emptyset$, μ et x respectivement et les symboles $*$, $+$ et $\times$ par les opérations rationnelles correspondantes.

On dit qu'un langage est *rationnel* s'il est reconnu par une expression rationnelle. De manière équivalente, les langages rationnels admettent la définition par induction suivante :

- $\emptyset$, μ et x (où $x \in X$) sont des langages rationnels;
- si L et L' sont rationnels, $L + L'$, LL' et L^* sont rationnels.

Ou encore,

> l'ensemble $\mathrm{Rat}(X^*)$ des langages rationnels sur X est la plus petite partie de $\mathcal{P}(X^*)$ qui contient $\emptyset$, μ et x pour tout $x \in X$ et qui est fermée par les opérations rationnelles.

Les propriétés de fermeture de $\mathrm{Rec}(X^*)$ permettent donc d'affirmer que $\mathrm{Rat}(X^*) \subset \mathrm{Rec}(X^*)$ [1]:

Théorème 4.8 (Kleene) *Tout langage rationnel est reconnaissable.* ■

De plus à partir des automates reconnaissant $\emptyset$, μ et x (figure 4.3) et en utilisant les opérations addition, concaténation et itération des automates (propositions 4.4, 4.5 et 4.7), on peut associer un automate $\mathcal{A}_t$ à chaque expression rationnelle t de manière que les langages reconnus par t et $\mathcal{A}_t$ soient identiques. [2]

La figure 4.6 montre les étapes du calcul d'un automate $\mathcal{A}$ reconnaissant le langage rationnel $L = (a^*b + c)^*a$. L est reconnu par l'expression rationnelle $t = \times(*(+(\times(*(a),b),c)),a)$ et $\mathcal{A}$ est un automate équivalent à $\mathcal{A}_t$ et un peu plus simple que $\mathcal{A}_t$ car on a opéré des simplifications à certaines étapes du calcul.

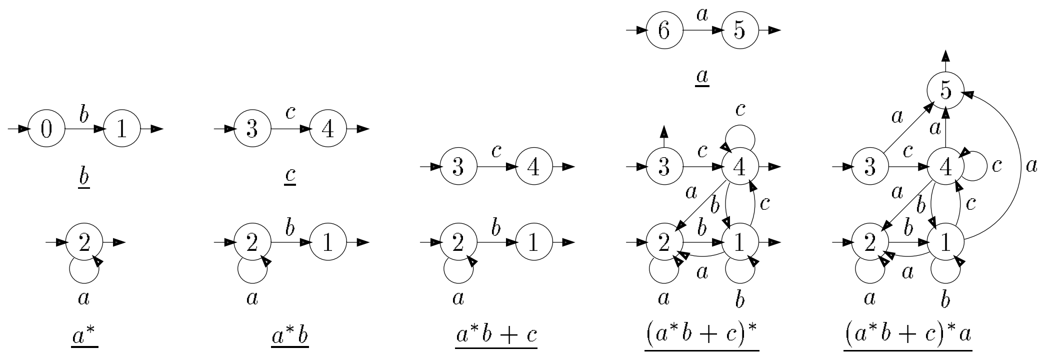


Fig. 4.6 – *Calcul d'un AF reconnaissant $(a^*b+c)^*a$. Les états non accessibles, 0 pour a^*b et 6 pour $(a^*b+c)^*a$, ont été supprimés. Noter que, dans la construction de l'AF reconnaissant $(a^*b + c)^*$ on a pu appliquer la remarque 4.1.*

Proposition 4.9 *Pour toute expression rationnelle t, le nombre d'états de l'automate associé $\mathcal{A}_t$ est un O de la taille de t.*

Preuve On peut vérifier par induction que $\mathcal{A}_t$ comporte $m + 2\ell + i \leqslant 2 \times \mathrm{taille}(t)$ états où m (resp. ℓ, i) désigne le nombre d'occurrences de μ (resp. $x \in X$, $*$) dans t. ■

4.2 Automates finis déterministes

4.2.1 Définition

Un automate fini $\mathcal{A} = (Q,I,F,\bullet)$ est dit *déterministe* — en abrégé AFD — si :

- il y a un unique état de départ : $|I| = 1$;

1. En fait $\mathrm{Rat}(X^*) = \mathrm{Rec}(X^*)$, voir p. 4.5.
2. $\mathcal{A}_t$ est l'interprétation de t pour la Σ-algèbre dont le domaine est l'ensemble des automates, les symboles constants étant interprétés par les automates de la figure 4.3 et les symboles $+$, $\times$ et $*$ par les opérations addition, concaténation et itération sur les automates.

– pour tout état $q \in Q$ et toute lettre $x \in X$, il existe au plus un état q' tel que $q \xrightarrow{x} q'$: $|q \bullet x| \leqslant 1$.

Dans ces conditions, si $(q,x) \in Q \times X$, $q \bullet x$ est soit un singleton, soit $\emptyset$. Si on identifie, comme on l'a déjà fait, un singleton $\{q'\}$ et son élément q', on peut considérer que la fonction de transition est une fonction partielle définie sur une partie de $Q \times X$ et à valeurs dans Q. Donc si $q \bullet x = \{q'\}$ on dira plutôt que $q \bullet x = q'$ et si $q \bullet x = \emptyset$ que $q \bullet x$ est non défini.

Ces conventions s'appliquent aussi à la fonction de transition étendue. Si $q \in Q$ et $u \in X^*$ est un mot, $q \bullet u$ est soit un singleton — un état —, soit vide — non défini — et la proposition 4.1 p. 71 devient

Proposition 4.10 $\forall q \in Q, \forall u,v \in X^*$,

$$(1) \quad q \bullet \mu = q$$
$$(2) \quad (q \bullet u) \bullet v = q \bullet uv$$
■

où (2) doit se comprendre ainsi : $q \bullet uv$ est défini ssi $q \bullet u$ et $(q \bullet u) \bullet v$ sont définis, auquel cas $(q \bullet u) \bullet v = q \bullet uv$.

Si $u = x_1 \ldots x_n$, le calcul de $q \bullet u = (\cdots((q \bullet x_1) \bullet x_2) \cdots \bullet x_{n-1}) \bullet x_n$ est déterministe dans le sens où il renvoie l'extrémité de l'unique chemin d'origine q étiqueté par u (s'il existe).

En particulier un mot u est reconnu par $\mathcal{A}$ ssi $d \bullet u$ (est défini et) est un état final, d désignant l'unique état initial.

Un AFD $(Q,\{d\},F,\bullet)$ est dit *complet* — en abrégé AFDC — si $q \bullet x \in Q$ est défini pour tout $(q,x) \in Q \times X$, auquel cas $q \bullet u \in Q$ est défini pour tout $(q,u) \in Q \times X^*$.

Suppression et ajout d'états dans un AFD

La preuve de la proposition suivante est laissée en exercice.

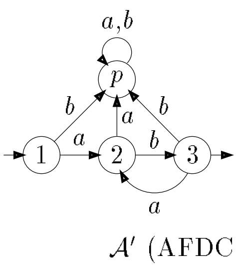
Proposition 4.11
- *Si on supprime des états différents de l'état de départ dans un AFD, l'AF obtenu est encore un AFD.*
- *Si le langage reconnu par un AFD est non vide, son émondé est un AFD.*
- *Si on supprime tous les états non accessibles d'un AFDC on obtient un AFDC accessible équivalent.* ■

On complète un AFD $\mathcal{A} = (Q,\{d\},F,\bullet)$ en lui ajoutant un état poubelle p; voir la figure 4.7. Le *complété* de $\mathcal{A}$ est l'AFDC $\mathcal{A}' = (Q \cup \{p\},\{d\},F,\bullet')$ où $p \notin Q$ et, pour tout $(q,x) \in (Q \cup \{p\}) \times X$, $q \bullet' x = \begin{cases} q \bullet x & \text{si } q \in Q \text{ et } q \bullet x \text{ est défini} \\ p & \text{si } q = p \text{ ou si } q \in Q \text{ et } q \bullet x \text{ non défini.} \end{cases}$

Proposition 4.12 *Le complété d'un AFD est un AFDC équivalent.* ■

$\mathcal{A}$	$\downarrow$ 1	2	$\uparrow$ 3
a	2		2
b		3	

$\mathcal{A}$ (AFD émondé de $\mathcal{A}'$)



$\mathcal{A}'$	$\downarrow$ 1	2	$\uparrow$ 3	p
a	2	p	2	p
b	p	3	p	p

$\mathcal{A}'$ (AFDC complété de $\mathcal{A}$)

FIG. 4.7 – *Un AFD et son complété*

Etats accessibles d'un AFD

Pour calculer les états accessibles d'un AFD $\mathcal{A} = (Q,\{d\},F,\bullet)$ et, dans le même temps, construire la représentation sagittale ou fonctionnelle de l'AFD acc$(\mathcal{A})$ obtenu en supprimant les états non accessibles, on procède de la manière suivante.

placer l'état de départ d;
tant que il y a des états placés et non traités, **faire**
  soit q le premier état placé et non traité **dans**
  traiter q, c.-à-d. **pour** toute lettre x **faire** placer $q \bullet x$ s'il ne l'est pas déjà. [1]

La figure 4.8 montre le déroulement de la méthode sur un exemple.

$$
\begin{array}{c|c}
 & \downarrow \\
 & 0 \\
\hline
2 & \\
5 &
\end{array}
\Longrightarrow
\begin{array}{c|ccc}
 & \downarrow & & \\
 & 0 & 2 & 5 \\
\hline
2 & 2 & & \\
5 & 5 & &
\end{array}
\Longrightarrow
\begin{array}{c|ccccc}
 & \downarrow & & & & \uparrow \\
 & 0 & 2 & 5 & 4 & 7 \\
\hline
2 & 2 & 4 & & & \\
5 & 5 & 7 & & &
\end{array}
\Longrightarrow
\begin{array}{c|ccccc}
 & \downarrow & & & & \uparrow \\
 & 0 & 2 & 5 & 4 & 7 \\
\hline
2 & 2 & 4 & 7 & & \\
5 & 5 & 7 & 7 & &
\end{array}
\Longrightarrow
$$

$$
\begin{array}{c|cccccc}
 & \downarrow & & & & \uparrow & \\
 & 0 & 2 & 5 & 4 & 7 & 6 \\
\hline
2 & 2 & 4 & 7 & 6 & & \\
5 & 5 & 7 & 7 & 7 & &
\end{array}
\Longrightarrow
\begin{array}{c|cccccc}
 & \downarrow & & & & \uparrow & \\
 & 0 & 2 & 5 & 4 & 7 & 6 \\
\hline
2 & 2 & 4 & 7 & 6 & 7 & \\
5 & 5 & 7 & 7 & 7 & 7 &
\end{array}
\Longrightarrow
\begin{array}{c|cccccc}
 & \downarrow & & & & \uparrow & \\
 & 0 & 2 & 5 & 4 & 7 & 6 \\
\hline
2 & 2 & 4 & 7 & 6 & 7 & 7 \\
5 & 5 & 7 & 7 & 7 & 7 & 7
\end{array}
$$

FIG. 4.8 – *Calcul de* $acc(\mathcal{A})$ *pour* $X = \{2,5\}$ *et* $\mathcal{A} = ([|0,7|],\{0\},\{7\},\bullet)$ *où* $q \bullet x = \min(q+x,7)$. $\mathcal{A}$ *modélise un distributeur de boissons proposant chaque boisson à 7 F et acceptant des pièces de 2 F et 5 F : un état représente la somme déjà payée et un mot la suite des pièces introduites.*

4.2.2 Déterminisation d'un automate

Le *déterminisé* ou *automate des parties* d'un automate fini $\mathcal{A} = (Q,I,F,\bullet)$ est l'AFDC $\text{dét}(\mathcal{A}) = (\mathcal{P}(Q),\{I\},F',\bullet')$ où $F' = \{P \subset Q \mid P \cap F \neq \emptyset\}$ et $\forall (P,x) \in \mathcal{P}(Q) \times X$, $P \bullet' x = P \bullet x$.

Précisons le sens de la définition de la fonction de transition $\bullet'$. Si $P \in \mathcal{P}(Q)$ est un état de $\text{dét}(\mathcal{A})$, P est un ensemble d'états de $\mathcal{A}$ donc, par définition de la fonction de transition étendue de $\mathcal{A}$, $P \bullet x$ est l'ensemble des extrémités des transitions de $\mathcal{A}$ d'origine dans P et d'étiquette x. En particulier, $P \bullet x$ est un ensemble d'états de $\mathcal{A}$ et c'est donc un état de $\text{dét}(\mathcal{A})$ de sorte que $\bullet'$ définit bien une application de $Q' \times X$ dans Q' où $Q' = \mathcal{P}(Q)$ est l'ensemble des états de $\text{dét}(\mathcal{A})$ et que $\text{dét}(\mathcal{A})$ est bien un AFDC.

Le théorème suivant montre que tout langage reconnaissable est reconnu par un automate déterministe.

Théorème 4.13 *Le déterminisé d'un AF est un AFDC équivalent.*

Preuve
Afin d'étudier $|\text{dét}(\mathcal{A})|$, on calcule d'abord l'état atteint par lecture d'un mot à partir d'un état donné : on montre par récurrence sur la longueur du mot u que

$$\forall (P,u) \in \mathcal{P}(Q) \times X^*, P \bullet' u = P \bullet u :$$

Si $u = \mu$, $P \bullet' \mu = P = P \bullet \mu$ (props 4.1 (1) p. 71 et 4.10 (1) p. 77).
Si $P \bullet' u = P \bullet u$ alors $P \bullet' ux = P \bullet' u \bullet' x = P \bullet u \bullet x = P \bullet ux$ (props 4.1 (2) et 4.10 (2)).
Alors $u \in |\text{dét}(\mathcal{A})| \Leftrightarrow I \bullet' u \in F' \Leftrightarrow I \bullet u \in F' \Leftrightarrow (I \bullet u) \cap F \neq \emptyset \Leftrightarrow u \in |\mathcal{A}|$. ■

Il est important de noter que la taille du déterminisé est exponentielle en la taille de l'automate : $|\mathcal{P}(Q)| = 2^{|Q|}$. Néanmoins, dans certains cas, $\text{dét}(\mathcal{A})$ admet un grand nombre d'états non accessibles et l'AFDC $\text{acc}(\text{dét}(\mathcal{A}))$ a une taille raisonnable.

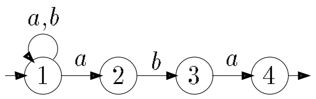
En pratique, pour calculer $\text{acc}(\text{dét}(\mathcal{A}))$, on utilise la méthode de la section précédente mais, pour ne pas alourdir les notations, on donne des noms — A, B, C, etc. — aux états accessibles du déterminisé. Voir les figures 4.9 et 4.10.

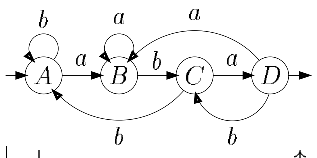
4.2.3 Fermeture de la classe des langages reconnaissables par les opérations ensemblistes

Fermeture de $\text{Rec}(X^*)$ par complémentaire

Proposition 4.14 *Le complémentaire* $L^c = X^* \setminus L$ *d'un langage reconnaissable* L *est reconnaissable.*

1. Il s'agit simplement d'un parcours en largeur du graphe induit par $\mathcal{A}$; voir p. 34.





$\mathcal{A}$	$\downarrow$ 1	2	3	$\uparrow$ 4
a	$\{1,2\}$	$\emptyset$	$\{4\}$	$\emptyset$
b	$\{1\}$	$\{3\}$	$\emptyset$	$\emptyset$

$\mathcal{A}$

$\mathcal{A}'$	$\downarrow$ $\{1\}$ A	$\{1,2\}$ B	$\{1,3\}$ C	$\uparrow$ $\{1,2,4\}$ D
a	B	B	D	B
b	A	C	A	C

$\mathcal{A}' = \mathrm{acc}(\mathrm{d\acute{e}t}(\mathcal{A}))$

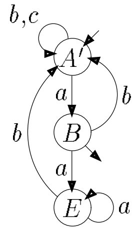
FIG. 4.9 – *Déterminisation d'un AF. $\mathcal{A}$ reconnaît le langage $L = (a+b)^*aba$ des mots qui se terminent par aba. L'AFDC $\mathcal{A}'$ peut être utilisé pour trouver toutes les occurrences du motif aba dans un mot u. Il suffit de faire lire u par $\mathcal{A}'$ en partant de l'état initial A: chaque rencontre avec l'état final D correspond à un préfixe de u élément de L et donc à une occurrence de aba dans u.*

	1	$\downarrow$ 2	$\downarrow$ 3	4	$\uparrow$ 5
a	$\{2,5\}$	$\{2\}$	$\{5\}$	$\{2,5\}$	$\emptyset$
b	$\{1\}$	$\{1\}$	$\emptyset$	$\{1\}$	$\emptyset$
c	$\{4\}$	$\emptyset$	$\{4\}$	$\{4\}$	$\emptyset$

$\mathcal{A}$

	$\downarrow$ $\{2,3\}$ A	$\uparrow$ $\{2,5\}$ B	$\{1\}$ C	$\{4\}$ D	$\{2\}$ E	$\emptyset$ F
a	B	E	B	B	E	F
b	C	C	C	C	C	F
c	D	F	D	D	F	F

$\mathcal{A}' = \mathrm{acc}(\mathrm{d\acute{e}t}(\mathcal{A}))$



$\mathcal{A}''$

FIG. 4.10 – *Déterminisation d'un AF. $\mathcal{A}$ reconnaît $L = (a^*b + c)^*a$ (v. fig. 4.6 p. 76). Dans $\mathcal{A}' = acc(dét(\mathcal{A}))$ on peut déjà supprimer l'état poubelle F. De plus, les trois états A, C et D sont non finaux et ont les mêmes transitions (B,C,D). Cela permet, sans changer le langage reconnu d'identifier ces trois états en un seul A', ce qui donne un AFD $\mathcal{A}''$ équivalent.*

Preuve Si $\mathcal{A} = (Q,\{d\},F,\bullet)$ est un AFDC qui reconnaît L alors $\mathcal{A}^c = (Q,\{d\},F^c,\bullet)$ reconnaît L^c. En effet $\forall u \in X^*, u \in |\mathcal{A}^c| \Leftrightarrow d \bullet u \in F^c \Leftrightarrow d \bullet u \notin F \Leftrightarrow u \notin |\mathcal{A}| = L \Leftrightarrow u \in L^c$. ■

Corollaire 4.15 *L'intersection de deux langages reconnaissables est reconnaissable.*

Preuve $L \cap L' = (L^c + L'^c)^c$. ■

Partant d'un AF $\mathcal{A}$ de taille (nombre d'états) n qui reconnaît un langage L, pour construire un AF $\mathcal{A}'$ reconnaissant L^c, il faut déterminiser $\mathcal{A}$ pour pouvoir appliquer la preuve de la proposition 4.14. L'AF résultant $\mathcal{A}'$ risque donc d'avoir une taille exponentielle en n.

La preuve du corollaire suggère donc une méthode inefficace de calcul d'un AF reconnaissant l'intersection de deux langages reconnus par deux AF. L'exercice suivant fournit un meilleur procédé.

Exercice 4.5 Montrer que si (Q,I,F,Δ) et (Q',I',F',Δ') sont deux AF qui reconnaissent respectivement L et L', alors
$(Q \times Q', I \times I', F \times F', \{((q,q'),x,(p,p')) \mid (q,x,p) \in \Delta \wedge (q',x,p') \in \Delta'\})$ reconnaît $L \cap L'$.

Automates produits

Soit ω un opérateur booléen binaire qui associe à deux valeurs booléennes $b,b' \in \mathcal{B} = \{\text{vrai,faux}\}$ une troisième valeur $b\,\omega\,b' \in \mathcal{B}$. ω permet de définir un opérateur binaire Ω sur les langages : Si $L,L' \subset X^*$, $L\,\Omega\,L' = \{u \in X^* \mid (u \in L)\,\omega\,(u \in L')\}$. Par exemple

ω	$\vee$	$\wedge$	$\vee$ exclusif	$(b,b') \mapsto b \wedge \neg b'$
Ω	$\cup$	$\cap$	$\triangle$ (différence symétrique)	$\setminus$ (différence)

On définit le *ω-produit* $\mathcal{A}'' = \mathcal{A} \times_\omega \mathcal{A}'$ de deux AFDC $\mathcal{A} = (Q,\{d\},F,\bullet)$ et $\mathcal{A}' = (Q',\{d'\},F',\bullet')$ par $\mathcal{A}'' = (Q \times Q', \{(d,d')\}, F'', \bullet'')$ où $F'' = \{(q,q') \in Q \times Q' \mid (q \in F)\,\omega\,(q' \in F')\}$ et $(q,q') \bullet'' x = (q \bullet x, q' \bullet' x)$.

La proposition suivante montre la fermeture de $\mathrm{Rec}(X^*)$ par Ω.

Proposition 4.16 $|\mathcal{A} \times_\omega \mathcal{A}'| = |\mathcal{A}| \,\Omega\, |\mathcal{A}'|$

Preuve On peut vérifier, par récurrence sur $|u|$, que, pour tout $u \in X^*$, $(q,q') \bullet'' u = (q \bullet u, q' \bullet' u)$. Donc $u \in |\mathcal{A}''| \Leftrightarrow (d,d') \bullet'' u \in F'' \Leftrightarrow (d \bullet u, d' \bullet' u) \in F'' \Leftrightarrow (d \bullet u \in F) \,\omega\, (d' \bullet' u \in F') \Leftrightarrow (u \in |\mathcal{A}|) \,\omega\, (u \in |\mathcal{A}'|) \Leftrightarrow u \in |\mathcal{A}| \,\Omega\, |\mathcal{A}'|$. ■

4.3 Automate minimal

4.3.1 Résiduels d'un langage

Si $L \subset X^*$ est un langage et $u \in X^*$ est un mot, le langage $u^{-1}L = \{v \in X^* \mid uv \in L\}$ est appelé un *résiduel* de L. On note $\mathrm{Res}(L) = \{u^{-1}L \mid u \in X^*\} \subset \mathcal{P}(X^*)$ l'ensemble des résiduels de L.

Proposition 4.17 *Si $\mathcal{A} = (Q,\{d\},F,\bullet)$ est un AFDC accessible reconnaissant L, l'application $\varphi_{\mathcal{A}} : q \mapsto q^{-1}F$ est une surjection de Q sur $\mathrm{Res}(L)$.*

Preuve On vérifie d'abord que $\forall (q,u,P) \in Q \times X^* \times \mathcal{P}(Q), (q \bullet u)^{-1}P = u^{-1}(q^{-1}P)$ — $v \in (q \bullet u)^{-1}P \Leftrightarrow (q \bullet u) \bullet v \in P \Leftrightarrow q \bullet uv \in P \Leftrightarrow uv \in q^{-1}P \Leftrightarrow v \in u^{-1}(q^{-1}P)$ —[1]

Soit $q \in Q$. Comme $\mathcal{A}$ est accessible, il existe un mot u tel que $q = d \bullet u$. Alors $q^{-1}F = (d \bullet u)^{-1}F = u^{-1}(d^{-1}F) = u^{-1}L$ donc $\varphi_{\mathcal{A}} : q \mapsto q^{-1}F$ est bien une application de Q dans $\mathrm{Res}(L)$. Elle est surjective car $\forall u \in X^*, u^{-1}L = \varphi_{\mathcal{A}}(d \bullet u)$. ■

Corollaire 4.18 *L'ensemble des résiduels d'un langage reconnaissable est fini.* ■

Exemple 4.5 Si $L = \{a^n b^n \mid n \geqslant 0\}$, on a $((a^n)^{-1}L) \cap X^n = \{b^n\}$ donc les résiduels $(a^n)^{-1}L$ sont distincts et $\mathrm{Res}(L)$ est infini. On retrouve ainsi la non reconnaissabilité de L déjà obtenue dans l'exemple 4.4 p. 73.

4.3.2 Définition

Proposition 4.19 $\forall L \subset X^*, \forall u,v \in X^*, (uv)^{-1}L = v^{-1}(u^{-1}L)$.

Preuve $w \in (uv)^{-1}L \Leftrightarrow uvw \in L \Leftrightarrow vw \in u^{-1}L \Leftrightarrow w \in v^{-1}(u^{-1}L)$. ■

Soit L un langage pour lequel $\mathrm{Res}(L)$ est fini. On définit un AFDC $\mathcal{A}_L$ appelé l'*automate minimal* ou l'*automate des résiduels* de L de la manière suivante : $\mathcal{A}_L = (Q_L,\{d_L\},F_L,\bullet_L)$ où

- $Q_L = \mathrm{Res}(L)$;
- $d_L = L$ (qui est bien un résiduel : $L = \mu^{-1}\mathrm{L}$);
- $F = \{R \in \mathrm{Res}(L) \mid \mu \in R\}$;
- $\forall (R,x) \in \mathrm{Res}(L) \times X, R \bullet_L x = x^{-1}R$ (si $R = u^{-1}L, x^{-1}R = (ux)^{-1}L \in \mathrm{Res}(L)$).

Proposition 4.20 *Si $\mathrm{Res}(L)$ est fini, l'automate minimal de L est un AFDC accessible qui reconnaît L.*

Preuve Une récurrence sur $|u|$ et la proposition 4.19 montrent que, pour tout $(R,u) \in \mathrm{Res}(L) \times X^*, R \bullet_L u = u^{-1}R$; en particulier, $d_L \bullet_L u = u^{-1}L$, ce qui prouve, d'une part que $\mathcal{A}_L$ est accessible et, d'autre part, que $u \in |\mathcal{A}_L| \Leftrightarrow u^{-1}L \in F_L \Leftrightarrow \mu \in u^{-1}L \Leftrightarrow u \in L$. ■

Corollaire 4.21 *Un langage est reconnaissable ssi il n'a qu'un nombre fini de résiduels.* ■

La figure 4.11 montre un calcul d'automate minimal.

1. Voir aussi l'exercice 4.2 p. 72

$L = aL_1 + bL_2$ où
– $L_1 = (aa)^*(\mu + bX^*)$;
– $L_2 = (bb)^*(\mu + aX^*)$.
$a^{-1}L = L_1$, etc.

	$\downarrow$	$\uparrow$	$\uparrow$		$\uparrow$		
	L	L_1	L_2	aL_1	X^*	bL_2	$\emptyset$
a	L_1	aL_1	X^*	L_1	X^*	$\emptyset$	$\emptyset$
b	L_2	X^*	bL_2	$\emptyset$	X^*	L_2	$\emptyset$

FIG. 4.11 – *Automate minimal du langage L sur $\{a,b\}$ des mots qui commencent par un nombre impair de a ou de b.*

4.3.3 Equivalence de Nérode d'un AFDC accessible

La méthode de calcul de l'automate minimal utilisée figure 4.11 est difficilement praticable et ne constitue pas un algorithme car il n'existe pas de méthode générale permettant de « calculer » un résiduel d'un langage ou même seulement de décider si deux résiduels sont égaux.

On dira que deux AFDC accessibles $\mathcal{A} = (Q,\{d\},F,\bullet)$ et $\mathcal{A}' = (Q',\{d'\},F',\bullet')$ sont *isomorphes* s'ils sont identiques aux noms des états près ou, plus précisément, s'il existe une bijection $f : Q \to Q'$ telle que $f(d) = d'$, $f(F) = F'$ et $\forall(q,x) \in Q \times X, f(q \bullet x) = f(q) \bullet' x$. Alors, par récurrence sur $|u|$, $\forall(q,u) \in Q \times X^*, f(q \bullet u) = f(q) \bullet' u$ et, en particulier, $f(d \bullet u) = f(d) \bullet' u = d' \bullet' u$, ce qui montre, vu que $\mathcal{A}$ est accessible, que f est unique. f est appelé l'*isomorphisme* de $\mathcal{A}$ sur $\mathcal{A}'$.

Soit un AFDC accessible $\mathcal{A} = (Q,\{d\},F,\bullet)$ reconnaissant L.[1] On appelle *équivalence de* NÉRODE de $\mathcal{A}$ la relation d'équivalence $\sim$ sur Q définie par

$$\forall q,q' \in Q, q \sim q' \Leftrightarrow q^{-1}F = q'^{-1}F.$$

On note $Q/\sim$ l'ensemble des classes et $\widetilde{q} \in Q/\sim$ la classe d'un état $q \in Q$. $\sim$ est la relation d'équivalence définie par la surjection $\varphi_{\mathcal{A}} : q \mapsto q^{-1}F$ de Q sur $\mathrm{Res}(L)$ introduite dans la proposition 4.17. $\varphi_{\mathcal{A}}$ induit donc une bijection $\widetilde{\varphi_{\mathcal{A}}} : Q/\sim \to \mathrm{Res}(L)$ telle que $\forall q \in Q, \widetilde{\varphi_{\mathcal{A}}}(\widetilde{q}) = q^{-1}F$.[2] On peut alors définir un AFDC accessible $\widetilde{\mathcal{A}}$ d'ensemble d'états $Q/\sim$ en imposant à la bijection $\widetilde{\varphi_{\mathcal{A}}}$ d'être un isomorphisme de $\widetilde{\mathcal{A}}$ sur l'automate minimal $\mathcal{A}_L$; ce qui donne :

Proposition 4.22 *Si $\sim$ est l'équivalence de* NÉRODE *d'un AFDC accessible $\mathcal{A} = (Q,\{d\},F,\bullet)$, alors on a un automate isomorphe à l'automate minimal de L : $\widetilde{\mathcal{A}} = (Q/\sim, \{\widetilde{d}\}, F/\sim, \widetilde{\bullet})$ avec $F/\sim = \{\widetilde{q} \mid q \in F\}$ et $\forall(q,x) \in Q \times X, \widetilde{q}\,\widetilde{\bullet}\,x = \widetilde{q \bullet x}$.*

Preuve Soit $\widetilde{\mathcal{A}} = (Q/\sim, \{d'\}, F', \bullet')$ l'automate défini par la condition que $\widetilde{\varphi_{\mathcal{A}}}$ soit un isomorphisme de $\widetilde{\mathcal{A}}$ sur $\mathcal{A}_L$.
$\widetilde{\varphi_{\mathcal{A}}}(\widetilde{d}) = d^{-1}F = L = d_L$ donc $d' = \widetilde{d}$.
$\widetilde{q} \in F' \Leftrightarrow q^{-1}F = \widetilde{\varphi_{\mathcal{A}}}(\widetilde{q}) \in F_L \Leftrightarrow \mu \in q^{-1}F \Leftrightarrow q = q \bullet \mu \in F$ donc $F' = F/\sim$.
Pour $(q,x) \in Q \times X$, $\widetilde{\varphi_{\mathcal{A}}}(\widetilde{q} \bullet' x) = \widetilde{\varphi_{\mathcal{A}}}(\widetilde{q}) \bullet_L x = (q^{-1}F) \bullet_L x = x^{-1}(q^{-1}F) = (q \bullet x)^{-1}F = \widetilde{\varphi_{\mathcal{A}}}(\widetilde{q \bullet x})$ donc $\widetilde{q} \bullet' x = \widetilde{q \bullet x}$: $\widetilde{q \bullet x}$ ne dépend que de $\widetilde{q}$ et de x et $\bullet' = \widetilde{\bullet}$.[3] ■

Corollaire 4.23 *Soit n le nombre d'états de l'automate minimal $\mathcal{A}_L$ d'un langage reconnaissable L et soit $\mathcal{A}$ un AFDC reconnaissant L.*

(1) *Le nombre d'états de $\mathcal{A}$ est $\geqslant n$.*

(2) *Si $\mathcal{A}$ a n états alors $\mathcal{A}$ est isomorphe à $\mathcal{A}_L$.*

Preuve Soit Q (resp. Q') l'ensemble des états (resp. des états accessibles) de $\mathcal{A}$. La proposition 4.17 montre que $\mathrm{acc}(\mathcal{A})$ a plus de $n = |\mathrm{Res}(L)|$ états; donc $|Q| \geqslant |Q'| \geqslant n$. Si $\mathcal{A}$ a n états alors $|Q| = |Q'| = n$ donc $\mathcal{A}$ est accessible ($Q = Q'$) et $\varphi_{\mathcal{A}}$ est bijective. Dans ce cas, $\mathcal{A}$ est isomorphe à $\widetilde{\mathcal{A}}$ — $q \mapsto \widetilde{q} = \{q\}$ est un isomorphisme — donc à $\mathcal{A}_L$. ■

1. Rappelons que l'algorithme de déterminisation d'un AF fournit directement un AFDC accessible.
2. De manière générale, à toute application $f : E \to F$, on associe la relation d'équivalence sur E définie par $x \sim x' \Leftrightarrow f(x) = f(x')$ ($\sim$ est encore appelée le *noyau* de f) et f induit une bijection $\widetilde{f} : E/\sim \to f(E)$ telle que $\forall x \in E, \widetilde{f}(\widetilde{x}) = f(x)$.
3. Pour la formule $x^{-1}(q^{-1}F) = (q \bullet x)^{-1}F$, voir la preuve de la proposition 4.17.

4.3.4 Calcul de l'automate minimal

Pour déterminer l'automate minimal d'un langage L reconnu par un AFDC accessible $\mathcal{A} = (Q,\{d\},F,\bullet)$, il suffit de calculer l'équivalence de NÉRODE $\sim$ de $\mathcal{A}$; l'automate $\widetilde{\mathcal{A}}$ s'en déduisant immédiatement.

Soit une relation d'équivalence $\mathcal{R}$ sur Q, ce qui équivaut à la donnée d'une partition[1] $\Pi = Q/\mathcal{R}$ de Q.

On dira qu'une lettre $x \in X$ *sépare* une classe $C \in \Pi$ s'il existe deux états $q,q' \in C$ tels que les deux états $q \bullet x$ et $q' \bullet x$ ne soient pas Π-équivalents, c.-à-d. appartiennent à deux classes différentes de Π. Dans ces conditions, la relation d'équivalence $\mathcal{R}_{C,x}$ sur C définie par $q \mathcal{R}_{C,x} q' \Leftrightarrow q \bullet x = q' \bullet x$ définit donc au moins deux classes et on note $\Pi_{C,x}$ la partition de Q obtenue en remplaçant, dans Π, la classe C par les classes définies par $\mathcal{R}_{C,x}$: $\Pi_{C,x} = (C/\mathcal{R}_{C,x}) \cup (\Pi \setminus \{C\})$.

Par exemple, pour l'automate $\mathcal{A}$ de la figure 4.12 et $\Pi = \{\{2,3,5,7\},\{1,4\},\{6\}\}$, la lettre a sépare la classe $C = \{2,3,5,7\}$ car, par exemple, $2 \bullet a = 4$ et $3 \bullet a = 6$ ne sont pas Π-équivalents. On calcule $C/\mathcal{R}_{C,a} = \{\{2\},\{3\},\{5,7\}\}$; d'où $\Pi_{C,a} = \{\{2\},\{3\},\{5,7\},\{1,4\},\{6\}\}$.

Proposition 4.24 *L'algorithme suivant renvoie $Q/\sim$*

soit $\Pi \leftarrow \{Q \setminus F,F\}$ ***dans***
tant que *il existe $x \in X$ et $C \in \Pi$ tel que x sépare C* ***faire*** $\Pi \leftarrow \Pi_{C,x}$
renvoyer Π

Preuve Précisons d'abord l'équivalence de NÉRODE $\sim$. Deux états q et q' ne sont pas $\sim$-équivalents ($q \not\sim q'$) s'il existe un mot u qui appartient à l'un et pas à l'autre des deux ensembles $q^{-1}F$ et $q'^{-1}F$ c.-à-d. $q \bullet u \in F$ et $q' \bullet u \notin F$ ou le contraire. On a $q \sim q' \Rightarrow \forall v \in X^*, q \bullet v \sim q' \bullet v$ car si $q \bullet v \not\sim q' \bullet v$ il existe u tel que, par exemple, $q \bullet vu \in F$ et $q' \bullet vu \notin F$.

La boucle termine car le nombre de classes de Π reste $\leqslant |Q|$ et augmente strictement dans le corps de cette boucle.

La propriété — Π est plus fine[2] que $\{Q \setminus F,F\}$ et moins fine que $Q/\sim$ — est un invariant de la boucle. En effet si elle est vérifiée et si $\Pi' = \Pi_{C,x}$ alors Π' est plus fine que Π donc que $\{Q \setminus F,F\}$ et, pour vérifier que Π' est moins fine que $Q/\sim$, soient deux états $q \sim q'$. Alors q et q' sont Π-équivalents donc s'ils ne sont pas dans C ils sont Π'-équivalents et s'ils sont dans C, comme $q \bullet x \sim q' \bullet x$, $q \bullet x$ et $q' \bullet x$ sont Π-équivalents c.-à-d. q et q' sont $\mathcal{R}_{C,x}$-équivalents donc Π'-équivalents.

L'invariant est vérifié à l'entrée dans la boucle car si $q \sim q'$, alors $q = q \bullet \mu$ et $q' = q' \bullet \mu$ sont tous deux dans F ou tous deux dans $Q \setminus F$ donc sont $\{Q \setminus F,F\}$-équivalents. Il l'est donc aussi à la sortie.

La partition Π renvoyée à la sortie est donc moins fine que $Q/\sim$. Elle est aussi plus fine car si q et q' sont Π-équivalents alors, pour toute lettre x, comme x ne sépare pas de classe de Π, $q \bullet x$ et $q' \bullet x$ sont Π-équivalents et, par récurrence sur $|u|$, pour tout mot u, $q \bullet u$ et $q' \bullet u$ sont Π-équivalents donc $\{Q \setminus F,F\}$-équivalents; soit $q \sim q'$. ■

La figure 4.12 fournit un exemple de calcul d'automate minimal. Un autre exemple est donné par la figure 4.10 p. 79 où on peut vérifier que $\mathcal{A}'' = \widetilde{\mathcal{A}'}$.

4.4 Transitions instantanées

4.4.1 Définition

Soit $\varepsilon \notin X$ une nouvelle lettre dite *lettre vide*. Un AF $\mathcal{A} = (Q,I,F,\bullet)$ sur l'alphabet $X \cup \{\varepsilon\}$ est appelé un *automate à transitions instantanées* ou ε-AF sur l'alphabet X. Une transition de $\mathcal{A}$ d'étiquette ε est dite *instantanée* ou *ε-transition*.

Notons $L_\varepsilon = |\mathcal{A}|_\varepsilon \subset (X \cup \{\varepsilon\})^*$ le langage sur $X \cup \{\varepsilon\}$ reconnu au sens usuel par $\mathcal{A}$.

On définit une nouvelle notion de langage reconnu : un mot $u \in X^*$ sera dit *reconnu par le ε-AF $\mathcal{A}$* s'il s'obtient à partir d'un mot $u' \in L_\varepsilon$ en supprimant toutes les occurrences de

1. Pour faire plus court, on appelle ici *partition* de Q une partition dont chaque classe est supposée non vide.

2. On dit qu'une partition Π_1 est *plus fine* qu'une autre Π_2 si toute classe de Π_2 est une réunion de classes de Π_1 c.-à-d. si deux éléments Π_1-équivalents sont Π_2-équivalents.

	↓↑				↑		↑
	1	2	3	4	5	6	7
a	2	4	6	4	5	4	7
b	3	5	3	7	7	7	5

$\mathcal{A}$

	{2,3,4,6},{1,5,7}
a	{2,3,4,6},{1},{5,7}
b	{3},{2,4,6},{1},{5,7}

calcul de $Q/\!\sim$

	↓↑			↑
	A	B	C	D
a	B	B	B	D
b	C	D	C	D

$\widetilde{\mathcal{A}}$

FIG. 4.12 – *Calcul de l'automate minimal d'un AFDC $\mathcal{A}$. On part de la partition $\Pi \leftarrow \{2{,}3{,}4{,}6\},\{1{,}5{,}7\}$. a sépare la classe $C = \{1{,}5{,}7\}$ donc on fait $\Pi \leftarrow \Pi_{C,a} = \{2{,}3{,}4{,}6\},\{1\},\{5{,}7\}$. b sépare alors $C' = \{2{,}3{,}4{,}6\}$, d'où $\Pi \leftarrow \Pi_{C',b} = \{3\},\{2{,}4{,}6\},\{1\},\{5{,}7\}$. Ni a ni b ne sépare aucune classe donc $\Pi = Q/\!\sim$. Pour représenter l'automate minimal $\widetilde{\mathcal{A}}$, on donne des noms aux classes $A = \{1\}$, $B = \{2{,}4{,}6\}$, $C = \{3\}$, $D = \{5{,}7\}$.*

ε dans u'. Autrement dit, lors de la lecture de u par un ε-AF, on peut passer d'un état à un autre soit en lisant une lettre de u et en suivant une transition étiquetée par cette lettre, ce qui constitue le procédé habituel, soit en ne lisant aucune lettre et en suivant une transition instantanée. Le langage sur X des mots de X^* reconnus en ce sens est appelé le langage *reconnu* et noté $L = |\mathcal{A}|$.[1]

4.4.2 Suppression des transitions instantanées

On se propose de montrer que, partant d'un ε-AF $\mathcal{A} = (Q,I,F,\bullet)$, on peut construire un AF (sans transitions instantanées) équivalent à $\mathcal{A}$; ce qui montrera que la classe des langages reconnus par ε-AF n'est pas plus large que la classe des langages reconnaissables.

On appelle *clôture instantanée* d'un ensemble d'états $P \subset Q$ de $\mathcal{A}$ l'ensemble $P \bullet \varepsilon^*$ des états que l'on peut atteindre à partir d'un état de P en suivant uniquement des transitions instantanées. Il s'agit donc de l'ensemble des sommets accessibles depuis un sommet de P dans le graphe $G = (Q,\{(q,q') \in Q \times Q \mid q' \in q \bullet \varepsilon\})$ obtenu à partir de $\mathcal{A}$ en ne gardant que les transitions instantanées. Un parcours[2] de G permet le calcul de $P \bullet \varepsilon^*$.

Proposition 4.25 *Soit $\mathcal{A} = (Q,I,F,\bullet)$ un ε-AF.*
$\mathcal{A}' = (Q,I \bullet \varepsilon^*,F,\bullet')$ où $q \bullet' x = (q \bullet x) \bullet \varepsilon^*$ est un AF équivalent à $\mathcal{A}$.

Preuve Si $u = x_1 \ldots x_n \in |\mathcal{A}|$, il existe un calcul réussi de $\mathcal{A}$ dont l'étiquette est de la forme $u' = \varepsilon^{k_0} x_1 \varepsilon^{k_1} x_2 \ldots x_{n-1} \varepsilon^{k_{n-1}} x_n \varepsilon^{k_n}$ où $\forall i, k_i \in \mathbf{N}$. Considèrons, pour $i = 1,\ldots,n$, l'état q_i de ce calcul origine de la transition d'étiquette x_i.
Dans $\mathcal{A}$: $I \ni d \xrightarrow{\varepsilon^{k_0}}_{\mathcal{A}} q_1 \xrightarrow{x_1 \varepsilon^{k_1}}_{\mathcal{A}} q_2 \cdots q_n \xrightarrow{x_n \varepsilon^{k_n}}_{\mathcal{A}} f \in F$;
donc, dans $\mathcal{A}'$: $I \bullet \varepsilon^* \ni q_1 \xrightarrow{x_1}_{\mathcal{A}'} q_2 \cdots q_n \xrightarrow{x_n}_{\mathcal{A}'} f \in F$ et $u \in |\mathcal{A}'|$.
La réciproque est laissée en exercice. ■

4.4.3 Application

L'utilisation des transitions instantanées permet parfois de simplifier certaines constructions. Par exemple la proposition suivante, dont la preuve est laissée en exercice, permet de retrouver la fermeture de la classe des langages reconnaissables par les opérations rationnelles.

Proposition 4.26 *Soient $\mathcal{A} = (Q,I,F,\Delta)$ et $\mathcal{A}' = (Q',I',F',\Delta')$ deux ε-AF sans état commun reconnaissant respectivement les langages L et L'.*

(1) $|(Q \cup Q', I \cup I', F \cup F', \Delta \cup \Delta')| = L + L'$
(2) $|(Q \cup Q', I, F', \Delta \cup \Delta' \cup (F \times \{\varepsilon\} \times I'))| = LL'$
(3) $|(Q, I, F, \Delta \cup (F \times \{\varepsilon\} \times I))| = L^+$ ■

Exercice 4.6 Dans le cas où $\mathcal{A}$ et $\mathcal{A}'$ sont des AF, montrer que l'AF obtenu en supprimant les transitions instantanées de l'ε-AF (2) (resp. (3)) est précisément l'AF défini à l'exercice 4.3 p. 74 (resp. 4.4 p. 75).

1. L est l'image de L_ε par la projection de $(X \cup \{\varepsilon\})^*$ sur X^*; voir l'exemple 2.3 p. 30.
2. Voir la section 2.2.4 p. 33.

4.5 Rat(X^*)=Rec(X^*)

On se propose de démontrer que tout langage reconnaissable est rationnel, ce qui constitue la réciproque du théorème de KLEENE (th. 4.8 p. 76).

Soit $\mathcal{A} = (Q,I,F,\Delta)$ un AF. L'ojectif est de montrer que le langage $L = |\mathcal{A}|$ est rationnel.

Soit $n = |Q|$ le nombre d'états; on peut supposer que les états sont les entiers de 1 à n : $Q = [|1,n|]$.

A tout triplet $(p,i,j) \in [|0,n|] \times Q^2$ on associe le langage $e_{i,j}^{(p)}$ formé des étiquettes des calculs d'origine i, d'extrémité j et dont les états intermédiaires (non compris i et j) sont dans $[|1,p|]$.

Proposition 4.27 *Pour tout couple d'états $(i,j) \in Q^2$,*

$$(1) \quad e_{i,j}^{(0)} = \sum_{(i,x,j)\in\Delta} x + \delta_{i,j} \text{ où } \delta_{i,j} = \mu \text{ si } i = j \text{ et } \emptyset \text{ sinon.}$$

$$(2) \quad \forall p \in [|1,n|], e_{i,j}^{(p)} = e_{i,j}^{(p-1)} + e_{i,p}^{(p-1)} \left(e_{p,p}^{(p-1)}\right)^* e_{p,j}^{(p-1)}.$$

Preuve
(1) Un calcul de i à j sans état intermédiaire est soit constitué d'une unique transition $(i,x,j) \in \Delta$, soit, si $i = j$, réduit au calcul vide de i à $j = i$.
(2) Soit $u \in e_{i,j}^{(p)}$. Il existe un calcul C de i à j d'étiquette u et dont les états intermédiaires sont $\leqslant p$. Si les états intermédiaires de C sont tous $< $ à p alors $u \in e_{i,j}^{(p-1)}$. Sinon, en considérant toutes les occurrences de p dans C, on trouve une factorisation $u = u_1 \dots u_k$ de u telle que $i \xrightarrow{u_1} p \xrightarrow{u_2} p \cdots p \xrightarrow{u_{k-1}} p \xrightarrow{u_k} j$ et que les états intermédiaires des k sous calculs de C ainsi mis en évidence soient $< p$. Donc $u_0 \in e_{i,p}^{(p-1)}$, $\forall i = 2,\dots,k-1, u_i \in e_{p,p}^{(p-1)}$ et $e_k \in e_{p,j}^{(p-1)}$. Finalement, $u = u_1(u_2 \dots u_{k-1})u_k \in e_{i,p}^{(p-1)} \left(e_{p,p}^{(p-1)}\right)^* e_{p,j}^{(p-1)}$, ce qui prouve l'inclusion « $\subset$ » dans (2). L'inclusion « $\supset$ » est facile. ∎

Corollaire 4.28 *(réciproque du théorème de* KLEENE*) Tout langage reconnaissable est rationnel.*

Preuve La proposition permet de démontrer par récurrence sur p que tous les langages $e_{i,j}^{(p)}$ sont rationnels. $L = \sum_{(d,f)\in I\times F} e_{d,f}^{(n)}$ est donc rationnel. ∎

La proposition 4.27 fournit un moyen pratique de calculer une ER (expression rationnelle) de L.[1]

Soit $E^{(p)} = \left(e_{i,j}^{(p)}\right)_{i,j=1,\dots,n}$. $E^{(p)}$ est une matrice $n \times n$ à coefficients dans l'ensemble des ER. Pour calculer $E^{(n)}$ on introduit une matrice $n \times n$ $A = (a_{ij})_{i,j=1,\dots,n}$ que l'on initialise à $E^{(1)}$ en utilisant (1) et que l'on modifie en utilisant (2) pour finalement aboutir à $A = E^{(n)}$.

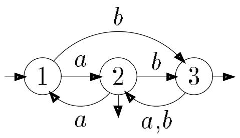
La formule (2) se simplifie si $i = p$ ou $j = p$:
$$e_{p,j}^{(p)} = \left(e_{p,p}^{(p-1)}\right)^* e_{p,j}^{(p-1)}, \qquad e_{i,p}^{(p)} = e_{i,p}^{(p-1)} \left(e_{p,p}^{(p-1)}\right)^* \text{ et } \qquad e_{p,p}^{(p)} = \left(e_{p,p}^{(p-1)}\right)^*.$$

L'algorithme 4.2 précise les opérations à appliquer à la matrice A. Un exemple est fourni par la figure 4.13.

1. On écrit ici les ER sous forme infixe de sorte que, par exemple, $a^*b + c$ désigne à la fois l'ER $t = +(\times(*(a),b),c)$ et le langage reconnu par t.

Algorithme 4.2 Calcul d'une ER du langage reconnu par un AF

soit calcule_ER($[|1,n|]$,I,F,Δ) =
    soit $A \leftarrow$ une matrice $n \times n$ **dans**
    pour $(i,j) \in [|1,n|]^2$ **faire** $a_{ij} \leftarrow \sum_{(i,x,j)\in\Delta} x + \delta_{i,j}$ $\quad \{A = E^{(0)}\}$
    pour $p = 1$ **à** n **faire**
        $\{A = E^{(p-1)}\}$
        soit $\alpha = a_{pp}^*$ **dans**
        $a_{pp} \leftarrow \alpha$ $\quad \{a_{pp} = e_{pp}^{(p)}\}$
        pour $j \in [|1,n|] \setminus \{p\}$ **faire** $a_{pj} \leftarrow \alpha a_{pj}$ $\quad \{a_{pj} = e_{p,j}^{(p)}\}$
        pour $(i,j) \in ([|1,n|] \setminus \{p\})^2$ **faire** $a_{ij} \leftarrow a_{ij} + a_{ip}a_{pj}$ $\quad \{a_{ij} = e_{i,j}^{(p)}\}$
        pour $i \in [|1,n|] \setminus \{p\}$ **faire** $a_{ip} \leftarrow a_{ip}\alpha$ $\quad \{a_{ip} = e_{i,p}^{(p)}\}$
        $\{A = E^{(p)}\}$
    renvoyer $\sum_{(i,j)\in I\times F} a_{ij}$



$E^{(0)}$	1	2	3
1	μ	a	b
2	a	μ	b
3	$\emptyset$	$a+b$	μ

$E^{(1)}$	1	2	3
1	μ	a	b
2	a	$\mu+a^2$	$b+ab$
3	$\emptyset$	$a+b$	μ

$E^{(2)}$	1	2	3
1	$(a^2)^*$	$a(a^2)^*$	a^*b
2	$(a^2)^*a$	$(a^2)^*$	a^*b
3	$(a+b)(a^2)^*a$	$(a+b)(a^2)^*$	$\mu+(a+b)a^*b$

$$L = \left(e_{1,2}^{(3)}\right) + \left(e_{1,3}^{(3)}\right) = \left(a(a^2)^* + a^*b((a+b)a^*b)^*(a+b)(a^2)^*\right) + \left(a^*b((a+b)a^*b)^*\right)$$
$$L = a(a^2)^* + a^*b((a+b)a^*b)^*(\mu + (a+b)(a^2)^*)$$

FIG. 4.13 – *Calcul d'une ER du langage reconnu par un AF. Il est inutile de calculer entièrement la matrice $E^{(3)}$ car seuls deux de ses coefficients sont nécessaires.*

4.6 Implémentation

Un langage reconnaissable peut être représenté par plusieurs types d'objets :
- une chaîne de caractères comme `a*b+c` définissant la forme infixée d'une ER;
- une expression rationnelle;
- un automate fini;
- un automate fini déterministe;
- l'automate minimal.

Le principal objectif de cette section est d'implémenter les opérations suivantes :
chaîne $\xrightarrow{\text{analyse syntaxique}}$ ER $\xrightarrow{\text{opérations}}$ AF $\xrightarrow{\text{déterminisation}}$ AFD $\xrightarrow{\text{minimisation}}$ AFD minimal.

4.6.1 Analyse syntaxique (prog. 4.1)

Une ER est représentée par le type `er` synonyme du type `terme` défini dans le programme 3.8 p. 56. Les symboles $+$, $\times$ et $*$ sont respectivement représentés par les chaînes `"+"`, `""` et `"*"`. Les propriétés de ces symboles définies par le tableau 3.1 p. 63 sont spécifiées par les valeurs `priorité_rationnelle : string -> int` et `qualité_rationnelle : int -> qualité` (le type `qualité` est défini dans le programme 3.15 p. 65).
`er_print : er -> unit` imprime à l'écran la forme infixe de son argument; elle appelle la fonction `imprime_inf` du programme 3.15.
`er_of_string : string -> er`. Si u `: string` représente une forme infixe d'une ER e, `er_of_string` u renvoie e en $O(|u|)$. Une première étape, l'*analyse lexicale*, consiste à calculer la liste `lexèmes` des lexèmes à analyser. Il s'agit de la liste des chaînes de longueur 1 constituées des caractères de u; mais une chaîne vide est intercalée chaque fois que le symbole $\times$ est attendu. Par exemple, si $u =$ `"a*b"`, `lexèmes = ["a";"*";"";"b"]`. La fonction `analyse_inf` du programme 3.16 p. 67 effectue ensuite l'*analyse syntaxique* proprement dite de la liste `lexèmes` pour calculer e.

Programme 4.1 Transformations : chaîne $\leftrightarrow$ ER

```
type er == terme;;

let  priorité_rationnelle = function
   "+"  -> 1 | "" -> 2 | "*" -> 3 | _ ->4
and qualité_rationnelle = function
   1 | 2 -> Gassoc | 3 -> Postfixe | 4 -> Constante;;

let (er_print : er -> unit) =
   imprime_inf priorité_rationnelle qualité_rationnelle;;

let er_of_string s =
   let n = string_length s in
   let lexèmes =
      let rec lexic i fin =
         if i = n  then []
                   else let x = make_string 1 s.[i] in
                        let l = x :: lexic (i + 1) (x <> "+" & x <> "(") in
                        if fin & x <> "*" & x <> "+" & x <> ")"
                           then "" :: l else l in
      lexic 0 false in
   (analyse_inf 4 priorité_rationnelle qualité_rationnelle lexèmes : er);;
```

4.6.2 Représentation d'un AF (prog. 4.2)

Une valeur de type `état` définit un état par son numéro — les numéros des différents états doivent être différents — et la liste des transitions dont il est l'origine.
`nouvel_état : unit -> état` renvoie un état isolé (origine d'aucune transition).
`ajoute_transition : état -> char -> état -> unit`. `ajoute_transition` $q\ x\ q'$ ajoute une transition étiquetée par x de l'état q à l'état q'.

Le type `état t` défini dans le module `set`[1] implémente les ensembles d'états.
`vide : état t` représente l'ensemble (d'états) vide.

Un AF est défini par une valeur de type `af` qui spécifie l'ensemble I des états initiaux, l'ensemble F des états finaux et un booléen dont la valeur est $I \cap F \neq \emptyset$. La donnée d'une valeur de type `af` permet d'accéder à tous les états accessibles en suivant les transitions.
`af_delta : état t -> char -> état t` implémente la fonction de transition $(P,x) \mapsto P \bullet x$. Complexité $O(|P|d)$ où d est le *degré* (nombre maximum de transitions issues d'un état).
`af_rec : af -> string -> bool`. `af_rec` $\mathcal{A}\ u$ renvoie `true` si le mot u est reconnu par l'AF $\mathcal{A}$. Complexité $O(nd|u|)$ où n est le nombre d'états.

Programme 4.2 Représentation d'un AF

```
type état = {mutable transitions : (char * état) list; numéro : int};;

let compteur_d'états = ref 0;;

let nouvel_état() =
   incr compteur_d'états;
   {transitions = []; numéro = !compteur_d'états}

and ajoute_transition e c e' = e.transitions <- (c,e') :: e.transitions;;

#open"set";;

let vide = empty (fun {numéro = n} {numéro = m} -> n - m);;

type af = {initial : état t; final : état t; contient_mu : bool};;

let af_delta q x = (* état t -> char -> état t *)
      let q' = ref vide in
      iter (function {transitions = l} ->
            do_list (function (y,e) -> if x = y then q' := add e !q') l) q;
      !q';;

let af_rec {initial = i; final = f} u =
   let q = ref i in
   for j = 0 to string_length u - 1 do
      q := af_delta !q u.[j]
   done;
   not inter !q f = vide;;
```

1. Voir p. 106.

4.6.3 Opérations sur les AF (prog. 4.3)

`af_vide` : `af`. AF reconnaissant $\emptyset$.
`af_motvide` : `unit -> af` renvoie un AF reconnaissant μ.
`af_car` : `char -> af`. `af_car` x renvoie un AF reconnaissant x.

`af_add` : `af -> af -> af` implémente la somme de deux AF.[1] Complexité $O(n + n')$ où n et n' sont les nombres d'états des deux AF arguments.
`af_mult` : `af -> af -> af` implémente le produit de deux AF sans état commun mais modifie le premier.[2] Complexité $O(nn'd')$ où n et n' sont les nombres d'états des deux AF arguments et d' est le degré du deuxième AF (voir la triple boucle de la fonction `accole` qui ajoute les transitions nécessaires à la construction).
`af_étoile` : `af -> af` implémente l'itéré d'un AF en le modifiant.[3] Complexité $O(n^2 d)$.

`af_of_er` : `er -> af` utilise les opérations précédentes et la fonction `évalue` du programme 3.8 p. 56 pour calculer un AF $\mathcal{A}$ reconnaissant le même langage qu'une ER t donnée. Le langage vide et le mot vide sont respectivement représentés par les caractères `V` et `M`. Complexité $O(n^4)$ où n est la taille de t. En effet, le nombre d'états de $\mathcal{A}$ est $\leqslant 2n$ et à chaque nœud de t correspond donc un temps d'éxécution dominé par n^3.

Programme 4.3 Opérations sur les AF

```
let af_vide = {initial = vide; final = vide; contient_mu = false}
and af_motvide() = let e = add (nouvel_état()) vide in
                   {initial = e; final = e; contient_mu = true}
and af_car c =
   let e = nouvel_état() and e' = nouvel_état() in
   ajoute_transition e c e';
   {initial = add e vide; final = add e' vide; contient_mu = false};;

let af_add t t'= {initial     = union t.initial t'.initial;
                  final       = union t.final t'.final;
                  contient_mu = t.contient_mu or t'.contient_mu};;

let accole t t' =
   iter (function f ->
      iter (function i' ->
         do_list (function (c,q) ->
            ajoute_transition f c q) i'.transitions) t'.initial) t.final;;

let af_mult t t' =
   accole t t';
   {initial = t.initial;
    final = if t'.contient_mu then union t.final t'.final else t'.final;
    contient_mu = t.contient_mu & t'.contient_mu };;
and af_étoile t =
   let e = nouvel_état() in
   accole t t;
   {initial = add e t.initial; final = add e t.final; contient_mu = true};;

let af_of_er (e : er) = évalue
 {eval_c = (function "V" ->af_vide | "M" ->af_motvide() | c ->af_car c.[0]);
  eval_u = (function "*" -> af_étoile | _ -> raise Erreur);
  eval_b = (function "+" -> af_add | _ -> af_mult)} e [];;
```

1. Voir la proposition 4.4 p. 74.
2. Voir la proposition 4.5 p. 74.
3. Voir la proposition 4.7 p. 75.

4.6.4 Représentation d'un AFD (prog. 4.4)

`définir_alphabet : char vect -> unit`. `définir_alphabet` t définit pour alphabet X l'ensemble des caractères de t. Les lettres sont alors numérotées de 0 à $p-1$, p désignant la longueur de t.
`nombre_de_lettres : unit -> int` renvoie p.
`lettre_of_numéro : int -> char`.
`numéro_of_lettre : char -> int`.

Une valeur $\mathcal{A}$ de type `afd` représente un AFDC dont les états sont les entiers de 0 à $n-1$. Si $q \in [|0,n-1|]$ est un état et $i \in [|0,p-1|]$ est le numéro de la lettre x, $\mathcal{A}$`.delta.(`q`).(`i`)`$= q \bullet x$ et $\mathcal{A}$`.accept.(`q`)` indique si q est final.
`afd_rec : afd -> string -> bool` permet de décider si un mot u est reconnu par un AFD. Complexité $O(|u|)$.

Programme 4.4 Représentation d'un AFD

```
type alphabet =
    {mutable nombre_lettres : int;
     mutable lettres        : char vect;
     mutable numero         : (char,int) hashtbl__t};;

let alphabet_global =
   {nombre_lettres = 0; lettres = [||]; numero = hashtbl__new 11};;

let définir_alphabet t =
  alphabet_global.nombre_lettres <- vect_length t;
  alphabet_global.lettres <- t;
  hashtbl__clear alphabet_global.numero;
  for i = 0 to vect_length t-1 do
     hashtbl__add alphabet_global.numero t.(i) i
  done;;

let nombre_de_lettres() = alphabet_global.nombre_lettres
and lettre_of_numéro i = alphabet_global.lettres.(i)
and numéro_of_lettre c = hashtbl__find alphabet_global.numero c;;

type afd = {delta : int vect vect; accept : bool vect};;

let afd_rec {delta = d; accept = a} u =
   let q = ref 0 in
   for j = 0 to string_length u - 1 do
      q := d.(!q).(numéro_of_lettre u.[j])
   done;
   a.(!q);;
```

4.6.5 Déterminisation d'un AF (prog. 4.5)

construire_afd :
`('a -> 'a -> bool) -> 'a -> ('a -> char -> 'a) -> ('a -> bool) -> afd`
implémente la méthode de construction d'un AFD décrite p. 77. **construire_afd** f d δ fin renvoie l'AFD dont l'état de départ est d, la fonction de transition est définie par δ et les états finaux par fin; la fonction f permet de tester l'égalité de deux états. Les états à traiter sont stockés dans une file d'attente.[1] Complexité $O(|X|n(nc_f + c_\delta))$ où n est le nombre d'états de l'AFD construit et c_f et c_δ sont les complexités des fonctions f et δ.[2]
afd_of_af : `af -> afd` déterminise l'AF argument. Complexité $O(|X|n(nm + |X|m^2))$ où m est le nombre d'états de l'AF et $n = O(2^m)$ celui de l'AFD construit.[3]

Programme 4.5 Déterminisation d'un AF

```
let construire_afd égal départ delta final =
   let t = ref []          (* liste (état placé,numéro de l'état).       *)
   and f = queue__new()    (* file des états à traiter.                  *)
   and delta' = ref []     (* liste des vecteurs delta'.(0), delta'.(1),..*)
   and accept = ref []     (* liste  accept(0),...                       *)
   and n = ref 0 in        (* nombre d'états.                            *)
   let numéro_of_état q =
      (* 'a -> int. (l'état q est rajouté à t et à f si nécessaire) *)
      let rec cherche = function
         [] -> raise Not_found |
         (q',p) :: l -> if égal q q' then p else cherche l in
      try cherche !t
      with Not_found ->
         t := (q,!n) :: !t;
         queue__add q f;
         accept := final q :: !accept;
         incr n;
         !n - 1 in
   numéro_of_état départ;
   while f <> queue__new() do
      let d = make_vect (nombre_de_lettres ()) 0
      and q = queue__take f in
      for i = 0 to nombre_de_lettres () - 1 do
         d.(i) <- numéro_of_état (delta q (lettre_of_numéro i))
      done;
      delta' := d :: !delta'
   done;
   {delta  = vect_of_list (rev !delta');
    accept = vect_of_list(rev !accept)};;

let afd_of_af {initial = i; final = f} =
   construire_afd equal i af_delta (function q -> not inter q f = vide);;
```

1. Voir p. 106.
2. Si la fonction f est est un simple test d'égalité on peut remplacer la file d'attente par une table de hachage ce qui ramène la complexité à $O(|X|n(c_f + c_\delta))$.
3. D'après la note précédente il aurait été plus efficace d'implémenter un ensemble d'états par une liste triée.

4.6.6 Minimisation d'un AFD (prog. 4.6)

`am_of_afd : afd -> afd` renvoie l'automate minimal.

Soit le graphe $G = (S,A)$ où S est l'ensemble des paires d'états de l'AFD $\mathcal{A}$ à minimiser et $A = \{(\{i,j\},\{i \bullet x, j \bullet x\}) \mid i,j \in S \wedge i \neq j \wedge x \in X \wedge i \bullet x \neq j \bullet x\}$. Soit T l'ensemble des sommets $\{i,j\}$ de G tels que un et un seul des états i ou j soit final. Deux états i et j sont non $\sim$-équivalents ssi le sommet $\{i,j\}$ de G est coaccessible à partir de T. On utilise l'algorithme 2.3 p. 36 pour construire une matrice de booléens `marqué` telle que, pour $i < j$, `marqué.(`i`).(`j`)` $\Leftrightarrow i \not\sim j$.

Complexité $O(|X|n^2)$ où n est le nombre d'états de $\mathcal{A}$.[1]

Programme 4.6 Minimisation d'un AFD

```
let am_of_afd {delta = delta; accept = accept} =
   let ne = vect_length delta
   and nl = nombre_de_lettres () in
   (* Marquage des paires d'états non équivalents {i,j}, i < j. *)
   let t = make_matrix (ne - 1) ne []
   and marqué = make_matrix (ne - 1) ne false in
   let rec marquer_rec (i,j) =
      marqué.(i).(j) <- true;
      do_list
         (function (i',j') ->
            if not marqué.(i').(j') then marquer_rec (i',j'))
         t.(i).(j) in
   for i = 0 to ne - 2 do for j = i + 1 to ne - 1 do
      if accept.(i) <> accept.(j) then marqué.(i).(j) <- true
   done done;
   for i = 0 to ne - 2 do for j = i + 1 to ne - 1 do
      let faire f =
      (* (int * int -> unit) -> unit.                    *)
      (* Applique f à tous les sommets adjacents à (i,j). *)
         for x = 0 to nl - 1 do
            let i' = delta.(i).(x)
            and j' = delta.(j).(x) in
            if i' < j' then f (i',j') else if i' > j' then f (j',i')
         done in
      if not marqué.(i).(j)
         then
            try
               faire
                  (function (i',j') -> if marqué.(i').(j') then raise Exit);
               faire
                  (function (i',j') ->
                     if t.(i').(j') = [] or hd t.(i').(j') <> (i,j)
                        then t.(i').(j') <- (i,j) :: t.(i').(j'))
            with Exit -> marquer_rec (i,j)
   done done;
   (* Construction de l'automate minimal à partir du tableau marqué. *)
   construire_afd
     (fun i j ->
        i = j or i < j & not marqué.(i).(j) or i > j & not marqué.(j).(i))
     0
     (fun q c -> delta.(q).(numéro_of_lettre c))
     (function q -> accept.(q));;
```

1. Il existe un algorithme en $O(|X| n \log n)$.

4.6.7 Essai

Les instructions

```
définir_alphabet [|`a`;`b`;`c`|];;
let exemple = am_of_afd(afd_of_af(af_of_er(er_of_string"(a*b+c)*a")));;
```

affectent à `exemple` la valeur

```
{delta = [|[|1; 0; 0|]; [|2; 0; 3|]; [|2; 0; 3|]; [|3; 3; 3|]|];
 accept = [|false; true; false; false|]}
```

qui est l'AFD $\mathcal{A}''$ de la figure 4.10 p. 79.

On peut aussi écrire des fonctions d'affichage des AFD. Par exemple, pour le lecteur connaissant LaTeX, la figure 4.14 a été obtenue en insérant la commande

```
\input{exemple.tex}
```

dans le fichier LaTeX qui a produit le présent document. Le fichier `exemple.tex` a lui-même été créé par l'instruction Caml

```
output_afd "exemple.tex" exemple;;
```

où `output_afd : string -> afd -> unit` produit un fichier LaTeX contenant la représentation fonctionnelle d'un AFD. [1]

	$\downarrow$	$\uparrow$		
	1	2	3	4
a	2	3	3	4
b	1	1	1	4
c	1	4	4	4

Fig. 4.14 – *AFD minimal reconnaissant* $(a^*b + c)^*a$

1. Toutes les figures de ce document ont ainsi été produites par des programmes Caml. En ce qui concerne les dessins d'arbres, il est relativement facile d'écrire une fonction qui prend un arbre en argument — et quelques autres données annexes — et qui effectue le dessin de l'arbre. Pour les représentations sagittales des automates, on utilise des fonctions de positionnement d'états et de tracé de transitions.
De plus les fichiers LaTeX produits par ces programmes Caml sont compatibles avec LaTeX-CAD, ce qui permet d'éditer les figures pour les retravailler à la souris. Mais cette possibilité n'a été utilisée que pour une seule figure.

Annexe A

Memento CAML

A.1 Généralités

A.1.1 Types et expressions

Un ensemble d'objets mathématiques de même nature est implémenté en CAML par un *type*. Certains types sont prédéfinis, comme le type `int` [1] qui implémente l'ensemble **Z** des entiers relatifs (en fait un intervalle borné de **Z**); d'autres sont à définir en utilisant des déclarations de types.

Un objet de type donné est définissable par une *expression* que CAML peut *évaluer* (en donner la *valeur*). C'est ainsi que `1 + 1` est une expression de type `int` dont la valeur est `2`, ce que l'on peut exprimer par la formule

`1 + 1 : int = 2`

ou encore, si l'on ne s'intéresse qu'au type de l'expression,

`1 + 1 : int`

A.1.2 Expressions fonctionnelles

CAML est un langage *fonctionnel* dans lequel les fonctions jouent donc un rôle primordial. Si t_1 et t_2 sont deux types qui implémentent respectivement les ensembles mathématiques E_1 et E_2, alors on dispose du type t_1 `->` t_2 qui implémente l'ensemble $\mathcal{F}(E_1,E_2)$ des applications de E_1 dans E_2. On verra plus loin les différentes manières de composer des expressions de type t_1 `->` t_2; on a par exemple

`(function n -> n + 1) : int -> int`

Si f est une expression de type t_1 `->` t_2 et si x est une expression de type t_1 alors l'expression de type t_2 qui s'évaluera en la valeur de la fonction f en x s'obtient simplement en juxtaposant les expressions f et x : $f\ x$ [2]. On a par exemple

`(function n -> n + 1) 2 : int = 3`

A.1.3 Fonctions curryfiées

Si, pour $i = 1,2,3$, t_i est un type qui implémente un ensemble E_i, le type t_1 `->` $(t_2$ `->` $t_3)$ [3] que l'on peut noter sans parenthèse t_1 `->` t_2 `->` t_3 [4] implémente l'ensemble $\mathcal{F}(E_1,\mathcal{F}(E_2,E_3))$ [5].

---

1. On convient qu'une chaine de caractères qui doit être reproduite telle quelle sera écrite en caractères machines comme `type` ou `1 + 1`.
2. On peut mettre des parenthèses comme dans $f(x)$, mais c'est le plus souvent inutile.
3. à ne pas confondre avec le type $(t_1$ `->` $t_2)$ `->` t_3 qui implémente $\mathcal{F}(\mathcal{F}(E_1,E_2),E_3)$.
4. On dit que l'opérateur `->` de construction de types est *associatif à droite*.
5. Cet ensemble $\mathcal{F}$ est en bijection naturelle avec l'ensemble $\mathcal{F}' = \mathcal{F}(E_1 \times E_2,E_3)$: à $f \in \mathcal{F}$, on fait correspondre $f' \in \mathcal{F}'$ défini par $f'(x,y) = (f(x))(y)$. On dit que f est la version *curryfiée* de la fonction non curryfiée f'.

A.1.4 Opérateurs binaires

Si $\top$ est un opérateur qui, à deux expressions e_1 et e_2 de types respectifs t_1 et t_2 associe $e_1 \top e_2$ de type t_3 [1] alors on dispose de la fonction `prefix` $\top$ de type t_1 `->` t_2 `->` t_3 telle que les deux expressions $e_1 \top e_2$ et `prefix` $\top$ e_1 e_2 sont équivalentes.

Ainsi, dire qu'en CAML est définie la fonction `prefix + : int -> int -> int` signifie qu'est défini un opérateur infixe `+` tel que, si e_1 et e_2 sont deux expressions de type `int`, alors e_1 `+` e_2 est une expression de type `int` [2].

A.1.5 Motifs

L'expression suivante est de type `int -> int` et implémente la fonction qui vaut 0 en 0 et en 1 et qui vaut $n+1$ en $n \geqslant 2$

```
function (0 | 1) -> 1 | n -> n + 1
```

Dans cette expression, les membres gauches des flèches (`(0 | 1)` et `n`) sont appelés des *motifs* et l'évaluation de la fonction sur un argument p s'effectue de la manière suivante : la fonction *filtre* d'abord p par le premier motif `(0 | 1)` (c.-à-d. teste si p est égal à 0 ou à 1). Si le filtrage réussit ($p \in \{0,1\}$) alors `1` est renvoyé. Si le filtrage échoue ($p \geqslant 2$) p est filtré par le second motif `n` qui est un identificateur. Ce filtrage réussit nécessairement, la valeur de p est liée à l'identificateur `n` et la valeur renvoyée est la valeur de l'expression `n + 1` quand `n` est lié à p, c.-à-d. $p+1$.

Quand un motif filtre avec succés une valeur, le filtrage peut opérer un certain nombre de liaisons de cette valeur ou d'autres valeurs à des identificateurs [3]. Un motif est défini par l'ensemble des valeurs qu'il filtre et par l'ensemble des liaisons qu'il définit. Voici les premières règles permettant de construire des motifs [4]:

- `_` est un motif qui filtre n'importe quelle valeur de tout type;
- si x est un identificateur, x est un motif qui filtre n'importe quelle valeur de tout type et lie cette valeur à l'identificateur x;
- si c est une constante de type `int`, `float`, `string`, `char` ou `bool`, c est un motif qui ne filtre que cette constante;
- si m_1 et m_2 sont deux motifs, m_1 `|` m_2 filtre toute valeur filtrée par l'un des deux motifs mais les liaisons sont interdites dans les deux motifs;
- si m est un motif, `(` m `)` est un motif équivalent à m;
- si m est un motif et x un identificateur, m `as` x est un motif qui filtre le motif m en effectuant les mêmes liaisons que m et qui, de plus, lie la valeur filtrée à x;

A.1.6 Conventions

Les éléments du langage sont définis par des règles de construction dont l'ensemble constitue la grammaire de CAML. Ces éléments sont principalement

- les *expressions*, par exemple `1 + 1`;
- les *expressions de type*, par exemple `int` ou `int -> int`;
- les *motifs* par exemple `(0|1)`.

Les différentes règles seront données au fur et à mesure
– de manière formelle comme dans

<expr> ::= `function` <motif> `->` <expr>

qui signifie que si m est un motif et si e est une expression alors `function` m `->` e est une expression [5];
– ou de manière implicite; par exemple, le fait de dire que `1.` et `-.2.6e-3` sont de type `float`

1. On dit alors que $\top$ est un opérateur binaire *infixe*.
2. Même s'il n'est pas dit que e_1 `+` e_2 s'évalue en la somme des valeurs de e_1 et de e_2, cela se devine.
3. Ces liaisons ne sont que provisoires: si des liaisons sont opérées lors d'un filtrage tel que <motif> `->` <expr>, ces liaisons ne sont effectives que pendant l'évaluation de <expr>.
4. Voir la section A.3 pour les autres règles.
5. Une telle règle n'explique pas la manière dont l'expression est évaluée mais cela est en général évident avec le contexte; ici, l'expression s'évalue en une fonction.

est une façon informelle de donner les règles de construction des expressions constantes de type `float`.

A.2 Sessions Caml

Quand on utilise CAML en mode interactif, on rentre au clavier des définitions ou des commandes qui sont analysées par CAML qui fournit éventuellement une réponse. L'ensemble constitué par ces entrées-sorties est appelé une *session*[1]. Il y a cinq sortes d'entrées :

A.2.1 Expressions

Si on rentre une expression CAML valide selon la syntaxe

<expr>`;;`

CAML évalue l'expression et renvoie son type et sa valeur; par exemple si on rentre

```
1 + 1;;
```

CAML répond

```
- : int = 2
```

A.2.2 Définitions de valeurs globales

Pour évaluer une expression <expr> et lier la valeur obtenue à un identificateur[2] <ident>, on utilise la syntaxe

`let` <ident> `=` <expr>`;;`

Si <ident> était déja lié à une autre valeur avant cette définition, l'ancienne liaison est détruite mais il est alors loisible à l'expression <expr> d'utiliser <ident> comme dans

```
let x = 1;;                     x : int = 1
let x = x + 1;;                 x : int = 2
```

Dans ces deux lignes, les définitions `let` ··· sont entrées au clavier et les messages `x : int =` ··· sont les réponses de CAML.

Si l'on veut que l'évaluation de <expr> utilise la liaison de <ident> à <expr> (définition récursive), il faut remplacer `let` par `let rec` comme dans

```
let rec factorielle = function 0 -> 1 | n -> n * factorielle (n - 1);;
```

Pour lier plusieurs valeurs à des identificateurs :

`let` <ident$_1$> `=` <expr$_1$> `and` <ident$_2$> `=` <expr$_2$> ...`;;`

(ou `let rec` pour des définitions mutuellement récursives). Par exemple

```
let x = 1 and y = 2;;
```

On peut aussi utiliser la syntaxe plus générale

`let` <motif> `=` <expr>`;;`

comme dans

`let x,y = 1,2;;`[3]

A.2.3 Définitions de types

La syntaxe est

`type` <ident> `=` <expression de type>`;;` [4]

On peut aussi définir plusieurs types (éventuellement mutuellement récursifs) dans la même déclaration :

`type` ··· `=` ··· `and` ··· `=` ··· `;;`

Une définition de type peut avoir un paramètre comme dans

```
type 'a suite == int -> 'a
```

1. Il est également possible de stocker les entrées dans un fichier et de faire compiler ce fichier par CAML.
2. Un *identificateur* est une chaine de caractères dont le premier caractère est une lettre (comme `a`, `A` ou `à`), les caractères suivants étant des lettres, des chiffres ou le caractère `_`; de plus, les mots réservés du langage (comme `function`, `for`, `do`, etc.) ne peuvent pas être des identificateurs.
3. Voir page 100 pour les motifs n-uplets.
4. Si <expression de type> est un type déja défini, il faut remplacer `=` par `==`.

A.2.4 Définitions d'exceptions

L'évaluation d'une expression peut délencher une *exception* (Voir la section A.5) qui, si elle n'est pas rattrapée, stoppe la compilation. Certaines de ces exceptions (`Not_found`, `Exit`, etc.) sont prédéfinies mais on peut en définir d'autres selon une syntaxe définie à la section A.5.

A.2.5 Directives

Ne parlons que de la plus importante :

`#open "<nom de module>";;`

qui permet d'utiliser les objets (définitions de valeurs, de types, etc.) contenus dans un module externe <nom de module>.

A.3 Les types Caml et leur utilisation

A.3.1 Ensemble à un élément : `unit`

Le type `unit` contient une seul valeur `()` qui implémente un objet mathématique que l'on pourrait nommer *rien*. Une fonction de type <type de l'argument> `-> unit` (ne) renvoie *rien* mais effectue en général une action [1]. Une telle fonction est aussi appelée une *procédure*.

A.3.2 Valeurs booléennes : `bool`

Le type `bool` contient deux valeurs `false` et `true`. Il implémente l'ensemble $\mathcal{B} = \{0{,}1\}$ que l'on peut assimiler à $\{faux{,}vrai\}$.

On dispose des opérateurs binaires `&` (ou `&&`), `or` (ou `||`) et de l'opérateur unaire `not`.

Opérateurs de comparaison :

`prefix =` et `prefix <> : 'a -> 'a -> bool` [2]

`prefix <` , `prefix >` , `prefix <=` , `prefix >= : 'a -> 'a -> bool`

> CAML munit chaque type non fonctionnel d'un ordre total (Dans le cas des entiers, des réels et des chaines de caractères, il s'agit de l'ordre naturel). C'est cet ordre qui est utilisé par les opérateurs `<`, `>`, `<=` et `>=`.

A.3.3 Entiers : `int`

Le type `int` implémente l'ensemble des entiers compris entre -2^{30} et $2^{30}-1 = 1073741823$.

Opérateurs et fonctions :

`prefix +` , `prefix -` , `prefix * : int -> int -> int`

`prefix /` , `prefix mod : int -> int -> int`

> a `/` b et a `mod` b renvoient respectivemnet le quotient et le reste de la division euclidienne de a par b.

`min` , `max : int -> int -> int`

`abs` , `succ` , `pred : int -> int`

> `abs` a, `succ` a et `pred` a renvoient respectivement $|a|$, $a+1$ et $a-1$.

On dispose aussi des opérations bit à bit suivants

`prefix land : int -> int -> int`

> `land` implémente le « et logique » défini comme suit : Soit $n = \overline{n_p \dots n_1 n_0}$ l'écriture d'un entier $n \geqslant 0$ en base 2 ($\forall i, n_i \in \{0{,}1\}$). Soit de même $m = \overline{m_q \dots m_1 m_0}$. On peut supposer que $p = q$ quitte à ajouter des 0 à gauche de l'écriture du plus petit des deux nombres n et m. Alors n `land` $m = \overline{\ell_p \dots \ell_1 \ell_0}$ où $\ell_i = 1$ si $n_i = m_i = 1$ et 0 sinon.

1. On dit aussi que la fonction produit un *effet de bord*.

2. Les opérateurs `=` et `<>` testent l'égalité *structurelle* de leurs arguments. Il existe aussi `==` et `!=` qui testent l'égalité *physique*.

De même, les opérateurs binaires `lor` et `lxor` et l'opérateur unaire `lnot : int -> int` implémentent respectivement le « ou logique », le « ou exclusif logique » et le « non logique ».
n `lsl` m multiplie n par 2^m (décalage à gauche).
n `lsr` m divise n par 2^m (décalage à droite).
`string_of_int : int -> string` et `int_of_string : string -> int`

`string_of_int (- 10)` renvoie la chaine `"-10"`
`int_of_string "-10"` renvoie `-10`

`print_int : int -> unit` imprime à l'écran l'entier argument.

A.3.4 Réels : `float`

Le type `float` implémente l'ensemble des nombres réels comme `1.` ou `-.2.6e-3`.
Opérateurs et fonctions :
`prefix +. : float -> float -> float` et aussi `-.`, `*.`, `/.` et `**`.
`exp`, `log`, `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `abs_float : float -> float`
`atan2 : float -> float -> float`

`atan2` $x\ y = \theta \in]-\pi,\pi]$ tel que $x + iy = e^{i\theta}$

`int_of_float : float -> int` et `float_of_int : int -> float`
`string_of_float : float -> string` et `float_of_string : string -> float`
`print_float : float -> unit` imprime à l'écran le réel argument.

A.3.5 Caractères : `char`

Le type `char` implémente l'ensemble des 256 caractères ASCII; par exemple `` `a` `` ou `` `;` ``.
Les deux fonctions suivantes s'appuient sur les codes ASCII des caractères (dans $[|0,256|]$)
`int_of_char : char -> int` et `char_of_int : int -> char`
`print_char : char -> unit` imprime à l'écran le caractère argument.

A.3.6 Chaines de caractères : `string`

Le type `string` implémente les chaines de caractères comme `"Caml est un langage"`. On peut modifier les caractères d'une chaine[1]. La longueur n d'une chaine s doit être $\leqslant 2^{16} - 1 = 65535$; les caractères de s sont numérotés de 0 à $n-1$.
s`.[`k`] : char` (k^{e} caractère de la chaine s)
s`.[`k`] <-` c `: unit` (affectation)
`prefix ^ : string -> string -> string` (concaténation)
`string_length : string -> int` (longueur de la chaine)
`create_string : int -> string`

`create_string` n renvoie une chaine de longueur n.

`sub_string : string -> int -> int -> string`

`sub_string` $s\ d\ \ell$ renvoie la sous chaine de s de longueur ℓ débutant à l'indice d.

`compare_strings : string -> string -> int`

`compare_strings` $s\ s'$ renvoie un entier < 0 ou > 0 suivant que s est avant ou après s' dans l'ordre lexicographique (0 si $s = s'$).

`print_string : string -> unit` imprime à l'écran la chaine argument.
`print_endline : string -> unit` imprime la chaine et passe à la ligne.
`print_newline : unit -> unit` passe à la ligne.

A.3.7 Variables ou références : `'a ref`

Une valeur de type `'a ref` est une *référence* à une valeur de type `'a`. Le type `'a ref` est mutable au sens où il est possible de modifier la valeur de type `'a` référencée par une valeur de type `'a ref`.

1. On dit que le type `string` est *mutable*.

`ref : 'a -> 'a ref`

`ref` x renvoie une référence (initialisée) à x

`prefix ! : 'a ref -> 'a`

`!`r renvoie la valeur contenue dans la référence r

`prefix := :'a ref -> 'a -> unit`

r `:=` x affecte la valeur x à la référence r

`incr` et `decr : int ref -> unit`

`incr` r équivaut à r `:= !`r `+ 1`

`decr` r équivaut à r `:= !`r `- 1`

A.3.8 Tableaux : `'a vect`

Le type `'a vect` Le type `'a vect` implémente les tableaux d'éléments de type `'a`. Les éléments d'un tableau de longueur n ($\leqslant 2^{14}-1 = 16383$) sont numérotés de 0 à $n-1$. Les éléments d'un tableau peuvent être modifiés.

`[|`x_0`;...;`x_{n-1}`|] : 'a vect` (tableau constitué des x_i `:'a`)

t`.(`k`) : 'a` (k^{e} caractère du tableau t `: 'a vect`)

t`.(`k`) <-` x `: unit` (affectation)

`concat_vect : 'a vect -> 'a vect -> 'a vect` (concaténation)

`vect_length : 'a vect -> int` (longueur du tableau)

`make_vect : int -> 'a -> 'a vect`

`make_vect` n x renvoie un tableau de n éléments égaux à x.

`make_matrix : int -> int -> 'a -> 'a vect vect`

`make_matrix` p q x renvoie une matrice de dimension $p \times q$ dont tous les éléments sont égaux à x.

`sub_vect : 'a vect -> int -> int -> 'a vect`

`sub_vect` t d ℓ renvoie le sous tableau de t de longueur ℓ débutant à l'indice d.

`do_vect : ('a -> 'b) -> 'a vect -> unit`

`do_vect` f t applique f à tous les éléments de t (en ignorant les résultats)

`map_vect : ('a -> 'b) -> 'a vect -> 'b vect`

`map_vect` f t applique f à tous les éléments de t et renvoie les résultats dans un autre tableau

`copy_vect : 'a vect -> 'a vect`

`copy_vect` t renvoie un autre tableau contenant les mêmes éléments que t

`list_of_vect : 'a vect -> 'a list`

`vect_of_list : 'a list -> 'a vect`

A.3.9 Listes : `'a list`

Si E est un ensemble, l'ensemble L des *listes* (d'éléments de E) est défini récursivement par les règles :

- nil est une liste (nil est un objet mathématique fixé quelconque; p. ex. $\mathsf{nil} = \emptyset$);
- si ℓ est une liste et $x \in E$ alors le couple (x,ℓ) est une liste (x est la *tête* et ℓ la *queue* de la liste (x,ℓ)).

ce qui peut s'exprimer de manière plus compacte par :

$$L ::= \mathsf{nil} \mid E \times L$$

Par exemple, (4,(3,(2,(1,nil)))) est une liste d'entiers de longueur 4.

Pour démontrer qu'une propriété $P(\ell)$ est vérifiée pour toute liste $\ell \in L$, on peut procéder par *induction* de la manière suivante :

- démontrer $P(\mathsf{nil})$;
- démontrer que $\forall \ell \in L, \forall x \in E, P(\ell) \Rightarrow P(x,\ell)$.

Soient un élément a d'un ensemble F et une application $\varphi : E \times F \to F$. On peut alors définir récursivement une application $f : L \to F$ par les règles :

- $f(\mathsf{nil}) = a$;

– $\forall \ell \in L, \forall x \in E, f(x,\ell) = \varphi(x,f(\ell))$.

Une fonction f qui peut être définie de cette manière est dite *inductive*..
Par exemple, la fonction s qui à une liste d'entiers associe la somme de ses éléments est inductive car $s(x,\ell) = x + s(\ell)$.
Si une fonction $f : L \to F$ n'est pas inductive, on peut trouver un *prolongement inductif* de f, c.-à-d. une fonction inductive $g : L \to G$ telle que $f = \psi \circ g$ où $\psi : G \to F$. Par exemple, si $E = \mathbf{R}$, la fonction $f : L \to \mathbf{R}$ qui à une liste de réels associe la moyenne de ses éléments admet le prolongement inductif $g : \ell \in L \mapsto$ (somme des éléments de ℓ, longueur de ℓ) $\in \mathbf{R} \times \mathbf{N}$.

Le type `'a list` implémente les listes d'éléments de type `'a`.

`[`x_1`;`…`;`x_n`] : 'a list` représente la liste $(x_1,(x_2,(\ldots,(x_n,\mathsf{nil})\ldots)))$ où x_i `: 'a`
x `::` ℓ

`::` est l'opérateur de *construction* de liste : x `::` ℓ représente la liste (x,ℓ); donc si x `: 'a` et $\ell =$ `[`x_1`;`…`;`x_n`] : 'a list` alors x `::` $\ell =$ `[`x`;`x_1`;`…`;`x_n`] : 'a list`

`prefix @: 'a list -> 'a list -> 'a list` (concaténation)
`list_length : 'a list -> int` (longueur de la liste)
`hd : 'a list -> 'a` (tête) et `tl : 'a list -> 'a list` (queue)

si $n > 0$, `hd [`x_1`;`…`;`x_n`]` $= x_1$ et `tl [`x_1`;`…`;`x_n`] = [`x_2`;`…`;`x_n`]`

`do_list : ('a -> 'b) -> 'a list -> unit`

`do_list` f ℓ applique f à tous les éléments de ℓ (en ignorant les résultats)

`map : ('a -> 'b) -> 'a list -> 'b list`

`map` f ℓ applique f à tous les éléments de ℓ et renvoie les résultats dans une autre liste

`rev : 'a list -> 'a list` (renverse la liste)
`for_all` et `exists : ('a -> bool) -> 'a list -> bool`

`for_all` $f\ \ell = \forall x \in \ell : fx$
`exists` $f\ \ell = \exists x \in \ell : fx$

`mem` et `memq : 'a -> 'a list -> bool`

`mem` x ℓ renvoie `true` si ℓ contient un élément égal à x (physiquement égal pour `memq`)

`except` et `exceptq : 'a -> 'a list -> 'a list`

`except` x ℓ supprime si possible de ℓ le premier élément égal à x (physiquement égal pour `exceptq`)

`subtract` , `union` , `intersect : 'a list -> 'a list -> 'a list`
`assoc` et `assq : 'a -> ('a * 'b) list -> 'b`

`assoc` x ℓ renvoie y tel qu'il existe x' égal (physiquement égal pour `assq`) à x avec x'`,`$y \in \ell$; si x' n'existe pas l'exception `Not_found` est déclenchée

Motifs : Le filtrage de motifs est un procédé particulièrement efficace quand on l'applique aux listes. Aux règles définissant les motifs, on peut ajouter :

– pour $n \geqslant 0$, si $m_1, \ldots m_n$ sont n motifs de même type `'a`, alors `[`m_1`;`…`;`m_n`]` est un motif de type `'a list`[1];
– si m_1 est un motif de type `'a` et m_2 un motif de type `'a list`, m_1 `::` m_2 est un motif de type `'a list`[2].

Les fonctions inductives sur les listes se programment de manière quasi automatique[3]. Par exemple, pour la somme des éléments d'une liste d'entiers :

```
let rec somme = function [] -> 0 | x :: l -> x + somme l;;
```

---

1. Il faut comprendre que `[`m_1`;`…`;`m_n`]` filtre toute liste `[`x_1`;`…`;`x_n`]` telle que, pour tout $i = 1,\ldots,n$, le motif m_i filtre la valeur x_i; et que les liaisons effectuées lors des filtrages des x_i par les m_i sont aussi effectuées lors du filtrage de `[`x_1`;`…`;`x_n`]` par `[`m_1`;`…`;`m_n`]`.
2. Remarque analogue à la note précédente.
3. Il existe d'ailleurs en Caml la fonction `list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` telle que `list_it` φ ℓ a renvoie l'image de ℓ par la fonction inductive définie par a et φ. Par exemple, si ℓ est une liste d'entiers, `list_it (prefix +)` ℓ `0` vaut la somme des éléments de ℓ.

A.3.10 Produit cartésien : `'a * 'b`

Les types `'a * 'b`, `'a * 'b * 'c`, etc. implémentent les couples, les triplets, etc.
$x_1, \ldots ,x_n$ représente le n-uplet $(x_1, \ldots ,x_n)$
`fst : 'a * 'b -> 'a` et `snd : 'a * 'b -> 'b` (projections)
MOTIFS :

- si $m_1, \ldots ,m_n$ sont n motifs, $m_1, \ldots , m_n$ est un motif.

A.3.11 Produit à champs nommés (enregistrements) : `{ ··· }`

Comme un type `*`, un type enregistrement implémente les n-uplets. La différence tient essentiellement à ce que chaque composante (ou *champ*) d'un n-uplet enregistrement possède un nom (une *étiquette*). De plus si un des champs d'un type enregistrement a été déclaré mutable, alors ce champ peut être modifié dans une valeur de ce type. Par exemple, pour implémenter les triplets formés de deux chaines de caractères et d'un entier modifiable, on peut définir le type
`type individu = {Nom : string; Prénom : string; mutable Age : int};;`
La syntaxe générale est

`type` <ident> `= {`[`mutable`] <étiquette> `:` <expression de type> `;... }`

où `mutable` est optionnel et les étiquettes peuvent être des identificateurs quelconques.

Pour définir une valeur de type enregistrement, on utilise le signe `=` comme dans
`let dupont = {Nom = "Dupont"; Prénom = "Jean"; Age = 19};;`
Si x est une valeur de type enregistrement et si e est l'étiquette d'un champ de ce type, on accède au champ d'étiquette e de x par $x.e$; au cas où le champ a été déclaré mutable, l'affectation de v au champ e de la valeur x se fait par l'expression de type `unit` : $x.e$ `<-` v. Par exemple, après l'instruction [1]
`dupont.Age <- 20;;` l'expression `dupont.Prénom ^ " " ^ dupont.Nom ^ " a " ^ (string_of_int dupont.Age) ^ " ans."` aura pour valeur la chaine `"Jean Dupont a 20 ans."`.
MOTIFS :

- si les m_i sont des motifs et si les e_i sont des étiquettes d'un même type enregistrement, alors `{`e_1 `=` m_1`;...}` est un motif.

Par exemple, les fonctions `function x -> x.Age` et `function {Age = n} -> n` de type `individu -> int` sont équivalentes.

A.3.12 Somme : `··· | ···`

Un type *somme* permet d'implémenter la réunion disjointe d'ensembles eux-mêmes implémentés par des types donnés. La syntaxe générale de définition d'un type somme est

`type` <ident> `=` <constructeur> `of` <expression de type> `| ... ;;`

où les <constructeurs> sont des identificateurs quelconques [2]. Dans une telle déclaration, chaque terme de la somme représente un type et la somme elle-même implémente la réunion de ces types. Au cas où l'un des termes représente un type réduit à un élément comme le type `unit`, il ne faut pas définir ce terme par <constructeur> `of unit` mais, plus simplement par <constructeur>; on parle alors de *constructeur constant*. Par exemple, pour implémenter la droite numérique achevée, $\overline{\mathbf{R}} = \mathbf{R} \cup \{+\infty\} \cup \{-\infty\}$ on peut définir le type
`type r_barre = Réel of float | Plusinfini | Moinsinfini;;`
Pour construire des expressions d'un type somme, on utilise la syntaxe

<constructeur non constant> <expr> ou <constructeur constant>

Par exemple, les éléments 3,14 et $+\infty$ de $\overline{\mathbf{R}}$ sont implémentés respectivement par les valeurs CAML `Réel 3.14` et `Plusinfini` de type `r_barre`.
MOTIFS :

- si *constr* est un constructeur constant, *constr* est un motif (qui ne filtre qu'une valeur);
- si *constr* est un constructeur non constant et si m est un motif du type associé à *constr* alors *constr* m est un motif de type somme.

1. On appelle ici *instruction* une expression de type `unit`.
2. Mais l'usage veut que la première lettre d'un constructeur soit une majuscule.

Par exemple, la valeur CAML suivante
`(function Réel x -> Réel (x *. x) | _ -> Plusinfini) : r_barre -> r_barre`
implémente l'application $x \in \overline{\mathbf{R}} \mapsto x^2 \in \overline{\mathbf{R}}$.

A.3.13 Fonctions : `'a -> 'b`

Le type `'a -> 'b` implémente l'ensemble des *fonctions* de `'a` dans `'b`[1].
Si f `: 'a -> 'b` et x `: 'a`, alors l'expression $f\ x$ s'évalue en la valeur de la fonction f pour l'argument x.
La syntaxe d'une expression de type `'a -> 'b` est

`function` <motif$_1$> `->` <expr$_1$> `| ... |` <motif$_n$> `->` <expr$_n$>

où les <motif$_i$> filtrent des valeurs de type `'a` et les <expr$_i$> sont des expressions de type `'b`.

Si f `: 'a -> 'b -> 'c` est une fonction curryfiée, x `: 'a` et y `: 'b` alors l'expression $f\ x\ y$ `: 'c` se lit $(f(x))(y)$.
La syntaxe d'une expression de type `'a -> 'b -> 'c` est

`fun` <motif$_1^1$> <motif$_1^2$> `->` <expr$_1$> `| ... |` <motif$_n^1$> <motif$_n^2$> `->` <expr$_n$>

A.4 Les structures de contrôle

A.4.1 Séquence

Quand une expression de la forme

<expr$_1$> `;` <expr$_2$>

est évaluée, <expr$_1$> puis <expr$_2$> sont évaluées et la valeur de l'expression entière est celle de <expr$_2$>. Par exemple, l'expression `false;3.14;1` s'évalue en `1`.

A.4.2 Filtrage d'une expression

Soient <expr> une expression de type t, <motif$_i$> des motifs de type t et <expr$_i$> des expressions de type t'. Alors l'expression <expr'> suivante

`match` <expr> `with` <motif$_1$> `->` <expr$_1$> `| ...`

est une expression de type t' évaluée comme suit :
<expr> est filtrée par <motif$_1$>. Si le filtrage réussit, <expr'> s'évalue en <expr$_1$> avec les liaisons effectuées par le filtrage. S'il échoue, <expr> est filtrée par <motif$_2$> et ainsi de suite[2].

A.4.3 Liaison locale

L'expression <expr> suivante

`let` <motif> `=` <expr$_1$> `in` <expr$_2$>

s'évalue ainsi : <expr$_1$> est filtrée par <motif> et si le filtrage réussit (ce qui sera toujours le cas si <motif> est un identificateur)[3], <expr> s'évalue en la valeur de <expr$_2$> avec les liaisons effectuées par le filtrage[4].

Pour lier un identificateur à une fonction, on peut remplacer l'expression

`let` <ident> `= fun` <motif$_1$> ... <motif$_n$> `->` <expr$_1$> `in` <expr$_2$>

par l'expression simplifiée équivalente

`let` <ident> <motif$_1$> ... <motif$_n$> `=` <expr$_1$> `in` <expr$_2$>

Dans le cas d'une fonction récursive (<expr$_1$> contient des appels de la fonction <ident>) il faut remplacer `let` par `let rec` comme dans

---

1. Voir page 93.
2. Une expression équivalente à <expr'> est donc (`function` <motif$_1$> `->` <expr$_1$> `| ... )` <expr>
3. Si le filtrage échoue, une exception `Match_failure` est déclenchée.
4. <expr> est donc équivalente à l'expression `match` <expr$_1$> `with` <motif> `->` <expr$_2$>

`let rec fact n = if n = 0 then 1 else n * fact (n - 1) in fact 5`
qui s'évalue en $5! = 120$[1].

Pour effectuer plusieurs liaisons, utiliser `and` :

`let` ... `=` ... `and` ... `and` ... `=` ... `in` $\langle\mathrm{expr}\rangle$

A.4.4 Sélection

Si $\langle\mathrm{expr}_1\rangle$ est une expression de type `bool` et si $\langle\mathrm{expr}_2\rangle$ et $\langle\mathrm{expr}_3\rangle$ sont deux expressions de même type,alors

`if` $\langle\mathrm{expr}_1\rangle$ `then` $\langle\mathrm{expr}_2\rangle$ `else` $\langle\mathrm{expr}_3\rangle$

est une expression qui s'évalue en la valeur de $\langle\mathrm{expr}_2\rangle$ si $\langle\mathrm{expr}_1\rangle$ s'évalue en `true` et en celle de $\langle\mathrm{expr}_3\rangle$ si $\langle\mathrm{expr}_1\rangle$ s'évalue en `false`.
L'expression

`if` $\langle\mathrm{expr}_1\rangle$ `then` $\langle\mathrm{expr}_2\rangle$ `else ()`

peut être simplifiée en

`if` $\langle\mathrm{expr}_1\rangle$ `then` $\langle\mathrm{expr}_2\rangle$ [2]

A.4.5 Répétition (boucle *for*)

Si $\langle\mathrm{expr}_1\rangle$ et $\langle\mathrm{expr}_2\rangle$ sont deux expressions de type `int` et si $\langle\mathrm{expr}_3\rangle$ est une expression de type quelconque, alors l'expression

`for` $\langle\mathrm{ident}\rangle$ `=` $\langle\mathrm{expr}_1\rangle$ `to` $\langle\mathrm{expr}_2\rangle$ `do` $\langle\mathrm{expr}_3\rangle$ `done`

évalue $\langle\mathrm{expr}_3\rangle$ pour les valeurs $\langle\mathrm{expr}_1\rangle$, $\langle\mathrm{expr}_1\rangle+1, \ldots, \langle\mathrm{expr}_2\rangle$ de $\langle\mathrm{ident}\rangle$ (ne fait rien si $\langle\mathrm{expr}_1\rangle > \langle\mathrm{expr}_2\rangle$); la valeur de l'expression entière est `()`.

L'indice $\langle\mathrm{ident}\rangle$ croît donc dans la boucle. Pour le faire décroître, remplacer `to` par `downto`.

A.4.6 Répétition (boucle *while*)

Si $\langle\mathrm{expr}_1\rangle$ est une expression de type `bool` et si $\langle\mathrm{expr}_2\rangle$ est une expression de type quelconque, alors l'expression

`while` $\langle\mathrm{expr}_1\rangle$ `do` $\langle\mathrm{expr}_2\rangle$ `done`

évalue $\langle\mathrm{expr}_2\rangle$ tant que $\langle\mathrm{expr}_1\rangle$ s'évalue en `true`; la valeur de l'expression entière est `()`.

A.5 Gestion des exceptions

A.5.1 Arrêt d'un programme par déclenchement d'une exception

Le type `exn` est un type somme prédéfini en CAML de la manière suivante :
`type exn = Exit | Not_found | Failure of string;;`[3]
Le type `exn` a la particularité d'être extensible. Pour ajouter un terme au type `exn`, il faut employer l'une des deux syntaxes

`exception` $\langle$constructeur constant$\rangle$ `;;`

`exception` $\langle$constructeur non constant$\rangle$ `of` $\langle$expression de type$\rangle$`;;`

Ainsi, si l'on définit
`exception Monexception of int;;`
le type `exn` deviendra
`Exit | Not_found | Failure of string | Monexception of int;;`

On dispose de la fonction fondamentale de déclenchement d'exception
`raise : exn -> 'a`
Cette fonction est à valeurs dans un type quelconque (`'a`), ce qui signifie que, si e est une expression de type `exn` (une *valeur exceptionnelle*), l'expression `raise` e peut être placée

1. Il est parfois possible d'utiliser `let rec` pour lier des identificateurs à des valeurs cycliques non fonctionnelles comme dans `let rec x = 1 :: x in x = tl x` qui s'évalue en `true`.
2. $\langle\mathrm{expr}_2\rangle$ doit donc être de type `unit`.
3. En fait, le type `exn` contient d'autres termes comme `Out_of_memory`, `Invalid_argument of string` et `Match_failure of string * int * int` (pour les échecs de filtrages).

dans un programme à tout endroit où on attend une expression (quelque soit le type de l'expression attendue). Si `raise` *e* est évaluée, l'éxécution du programme est immédiatement stoppée (le programme « plante ») et on voit s'afficher le message (d'erreur) :
`Uncaught exception :` *e*
qui signifie « exception non rattrapée : *e* ». On dit que l'exception *e* a été *déclenchée*.

Exemple A.1 La bibliothèque CAML fournit la fonction `failwith : string -> 'a` définie par `let faiwith s = raise (Failure s);;`. On convient d'utiliser la fonction `failwith` pour déclencher l'exception `Failure "`<ident>`"` quand la fonction <ident> a été appelée avec un argument invalide. Par exemple si on définit la fonction `fact` par le programme A.1 alors `fact 5` s'évalue en `120` et l'évaluation de `fact (-5)` déclenche l'exception `Failure "fact"`.

Programme A.1 Factorielle

```
let rec fact = function  (* fact : int -> int *)
   0 -> 1 | n -> if n < 0 then failwith "fact" else n * fact(n-1);;
```

Exemple A.2 Considérons la fonction `rech` du programme A.2.

Programme A.2 Recherche dans un tableau non trié avec déclenchement d'une exception en cas de succés

```
exception Indice of int;;

let rech t x =  (* rech : 'a vect -> 'a -> unit *)
   for i = 0 to vect_length t - 1 do
      if x = t.(i) then raise(Indice i)
   done;;

rech [|5;10;-8|] 10;; (* déclenche l'exception Indice 1 *)
rech [|5;10;-8|] 1;;  (* renvoie ()                     *)
```

Soit i le plus petit indice tel que $t.(i) = x$, s'il en existe, alors l'évaluation de `rech` t x déclenche l'exception `Indice` i; si i n'existe pas, `rech` t x s'évalue simplement en `()`.

A.5.2 Rattrapage d'une exception

En fait, planter un programme est toujours fâcheux. On s'arrange toujours pour que l'éxécution d'un programme ou bien ne déclenche jamais d'exception, ou bien rattrape chaque exception déclenchée. Cela se fait en utilisant la syntaxe

try <expr> with <motif$_1$> `->` <expr$_1$> `|` ... `|` <motif$_n$> `->` <expr$_n$>

où <expr> et les <expr$_i$> sont des expressions de même type t et les <motif$_i$> sont des motifs de type `exn`.
L'expression `try` ... est une expression <expr$'$> de type t évaluée de la manière suivante : En premier lieu, <expr> est évaluée;

- si l'évaluation de <expr> fournit le résultat v (sans déclencher d'exception) alors l'évaluation de <expr$'$> est terminée et le résultat est v;
- si l'évaluation de <expr> déclenche l'exception e, alors<expr$'$> s'évalue en `match` e `with` <motif$_1$> `->` <expr$_1$> `|` ... `|` <motif$_n$> `->` <expr$_n$>
  si le filtrage réussit pour <motif$_i$>, <expr$'$> s'évalue donc en <expr$_i$>; si le filtrage échoue, <expr$'$> déclenche l'exception e.

Exemple A.3 (suite de l'exemple A.1) Le programme A.3, qui utilise la fonction `fact` du programme A.1, fournit une fonction `c : int -> int -> int` telle que, si n et p sont $\geqslant 0$, `c` n p renvoie le nombre de parties à p éléments d'un ensemble à n éléments[1].

1. La méthode utilisée est particulièrement inefficace : il serait préférable d'utiliser le triangle de Pascal.

Programme A.3 Coefficient du binôme

```
let c n p =
   try fact n / (fact p * fact(n-p)) with Failure "fact" -> 0;;

c 4 2;; (* renvoie 6 *)
c 2 4;; (* renvoie 0 *)
```

Exemple A.4 (suite de l'exemple A.2) Le programme A.4 fournit une fonction `rech' : 'a vect -> 'a -> int` telle que `rech'` t x renvoie le plus petit indice i tel que t`.(`i`)` $= x$ s'il en existe et -1 sinon.

Programme A.4 Recherche dans un tableau non trié

```
let rech' t x =
   try
      for i = 0 to vect_length t - 1 do
         if x = t.(i) then raise(Indice i)
      done;
      -1
   with Indice i -> i;;

rech' [|5;10;-8|] 10;; (* renvoie 1  *)
rech' [|5;10;-8|] 1;;  (* renvoie -1 *)
```

A.6 Priorités et associativités des opérateurs

Le tableau A.1 donne les priorités des opérateurs de construction d'expressions, de motifs et d'expressions de type. Dans chacun des trois cas un opérateur est prioritaire sur tout autre opérateur situé plus bas. Les opérateurs situés sur une même ligne ont la même priorité.

De plus, la plupart des opérateurs binaires possèdent une *associativité gauche* ou *droite*. Ainsi l'opérateur de construction d'expressions `/` est associatif à gauche, ce qui signifie que l'expression a `/` b `/` c est équivalente à `(` a `/` b `)` `/` c. On a déja vu à la section A.1.3 que l'opérateur `->` de construction d'expressions de type était associatif à droite.

A.7 La bibliothèque Caml

Les éléments CAML décrits jusqu'à présent font partie du noyau de base du langage. Les éléments complémentaires décrits dans cette section constituent l'essentiel de la *bibliothèque d'utilitaires*. Cette bibliothèque est partagée en *modules*.

Pour accéder aux éléments d'un module, il suffit d'*ouvrir* ce module par la directive :

`#open "`<nom du module>`";;`

Soit e un élément d'un module (une valeur, un type, etc.); même si le module n'est pas ouvert, il est possible d'accéder à cet élément à condition d'employer la syntaxe

<nom du module>`__`e

Cette manière de faire est d'ailleurs la seule possible dans le cas où l'on désire utiliser deux éléments de deux modules différents qui ont le même nom [1].

Certains des modules contiennent de nouvelles fonctions (une fonction de tri, des fonctions permettant de générer des nombres aléatoires et des fonctions graphiques); d'autres implémentent de nouvelles *structures de données*.

Une structure de données *abstraite* est définie par un ensemble d'objets et par un ensemble d'opérations sur ces objets. par exemple, les objets de la structure de données *tableau*

1. Par exemple, les modules `stack` et `queue` contiennent tous les deux un type nommé `t` et une fonction nommée `new`. Pour utiliser les deux modules conjointement, il faut préciser le module comme dans `stack__new` ou `queue__new`.

expressions		**motifs**		**expressions de type**	
`!`		constructeur		constructeur [a]	
`. .(`		`::`	droite	`*`	
application de fonction	gauche	`,`		`->`	droite
application de constructeur		`|`	gauche		
`- -.` (unaires)		`as`			
`mod`	gauche				
`* *. / /.`	gauche				
`+ +. - -.`	gauche				
`::`	droite				
`@ ^`	droite				
`= == < <.` etc.	gauche				
`not`					
`&`	gauche				
`or`	gauche				
`,`					
`<- :=`	droite				
`if`					
`;`	droite				
`let match function try`					

TAB. A.1 – *Priorités et associativités des opérateurs*

[a] un *constructeur de type* est l'identificateur qui intervient dans un type paramétré comme `int vect`

sont les n-uplets d'éléments d'un ensemble fixé et les opérations sont :
- création d'un tableau de longueur donnée;
- accés à un élément d'indice donné d'un tableau;
- modification d'un élément d'un tableau donné par son indice.

Une structure de données est dite *dynamique* si les opérations peuvent modifier un objet de la structure (c'est le cas des tableaux).

Une structure de données (concrète) est une implémentation d'une structure de données abstraite. Deux structures de données concrètes implémentant une même structure de données abstraite peuvent avoir des performances différentes en espace pour le stockage des objets; et en temps pour les opérations.

A.7.1 Fonction de tri (module `sort`)

`sort : ('a -> 'a -> bool) -> 'a list -> 'a list`

`sort` f ℓ trie la liste ℓ en ordre croissant selon la relation d'ordre définie par f : $x \leqslant y \Leftrightarrow f\ x\ y$.

`merge : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list`

`merge` f ℓ_1 ℓ_2 fusionne les listes ℓ_1 et ℓ_2 supposées triées pour renvoyer une liste triée contenant les éléments de ℓ_1 et de ℓ_2.

A.7.2 Piles (module `stack`)

Les *piles* ou *files LIFO* (Last In First Out) forment une structure de données dynamique dont les objets sont les n-uplets d'éléments d'un ensemble E. Le dernier élément d'une pile non vide est appelé le *sommet*, ce qui suggère de représenter une pile verticalement. Les opérations de base sont
- création d'une pile vide;
- *empiler* (ou *push*) : ajouter un élément au sommet d'une pile (cet élément devient le nouveau sommet);
- *dépiler* (ou *pop*) : supprimer le sommet d'une pile non vide et le renvoyer.

Le module `stack` implémente les piles.

`type 'a t` le type des piles.

`new : unit -> 'a t`	`new()` retourne une pile vide.
`push : 'a -> 'a t -> unit`	`push` x p ajoute x au sommet de p.
`pop : 'a t -> 'a`	`pop` p supprime et retourne le sommet de p.
`clear : 'a t -> unit`	`clear` p vide la pile.
`length : 'a t -> int`	`length` p retourne le nombre d'éléments de p.
`iter : ('a -> 'b) -> 'a t -> unit`	`iter` f p applique f à chaque élément de p.

La fonction `pop` déclenche l'exception `Empty` quand elle est appelée sur une pile vide.
`iter` opère en partant du sommet.

A.7.3 Files d'attente (module `queue`)

Les *files d'attente* ou *files FIFO* (First In First Out) forment une structure de données dynamique dont les objets sont les n-uplets d'éléments d'un ensemble E. Les opérations de base sont

- création d'une file vide;
- ajouter un élément à la fin d'une file (cet élément devient le nouveau dernier élément);
- supprimer et renvoyer le premier élément (début) d'une file non vide.

Le module `queue` implémente les files d'attente.

`type 'a t`	le type des files d'attente.
`new : unit -> 'a t`	`new()` retourne une file vide.
`add : 'a -> 'a t -> unit`	`add` x ℓ ajoute x à la fin de ℓ.
`take : 'a t -> 'a`	supprime et retourne l'élément de début de la file.
`peek : 'a t -> 'a`	retourne l'élément de début de la file.
`clear : 'a t -> unit`	vide la file.
`length : 'a t -> int`	retourne le nombre d'éléments de la file.
`iter : ('a -> 'b) -> 'a t -> unit`	`iter` f ℓ applique f à chaque élément de ℓ.

Appelées sur une file vide, les fonctions `take` et `peek` déclenchent l'exception `Empty`.
`iter` opère du début à la fin de la file.

A.7.4 Générateur de nombres aléatoires (module `random`)

`init : int -> unit`	initialise le générateur de nombres aléatoires.
`int : int -> int`	`int` n renvoie un entier dans $[|0,n-1|]$.
`float : float -> float`	`float` x renvoie un réel dans $[0,x[$.

A.7.5 Ensembles (module `set`)

Les objets de la structure de données *ensembles* sont les sous ensembles d'un ensemble E. Les opérations supportées sont les opérations ensemblistes habituelles

- création d'un ensemble vide;
- tester si un ensemble est vide;
- tester si un élément appartient à un ensemble;
- ajouter un élément à un ensemble (sauf s'il s'y trouve déjà);
- supprimer un élément d'un ensemble (sauf s'il ne s'y trouve pas);
- tester si deux ensembles contiennent les mêmes éléments;
- réunion, intersection, différence.

Une implémentation possible consiste à représenter un ensemble par la liste de ses éléments mis dans un ordre quelconque[1].

Une méthode plus efficace pour des opérations telles que la réunion est de représenter un ensemble par la liste ordonnée de ses éléments, un ordre total étant donné sur E.

Le module `set` propose une implémentation utilisant des arbres binaires équilibrés (voir le chapitre 3). Pour représenter des ensembles d'éléments de type `'a`[2] il faut disposer d'une

1. On dispose d'ailleurs déjà des fonctions `mem`, `subtract`, `union` et `intersect` (voir le paragraphe A.3.9).
2. Le type `'a` ne doit pas être un type fonctionnel

relation d'ordre total $\leqslant$ sur `'a` et d'une *fonction d'ordre* f : `'a -> 'a -> int` qui définit cette relation : $f\ x\ y > 0 \Leftrightarrow x > y$ et $f\ x\ y < 0 \Leftrightarrow x < y$. Pour cela on dispose de la fonction f = `eq__compare` qui convient pour tous les types non fonctionnels et non cycliques. On peut aussi choisir une fonction d'ordre pour chaque type particulier; p. ex. f = `prefix -` pour le type `int`.

`type 'a t`	le type des ensembles d'élts de type `'a`
`empty : ('a -> 'a -> int) -> 'a t`	`empty` <fonction d'ordre> renvoie $\emptyset$
`is_empty : 'a t -> bool`	`is_empty` e teste si $e = \emptyset$
`mem : 'a -> 'a t -> bool`	`mem` $x\ e$ teste si $x \in e$
`add : 'a -> 'a t -> 'a t`	`add` $x\ e = \{x\} \cup e$
`remove : 'a -> 'a t -> 'a t`	`remove` $x\ e = e \setminus \{x\}$
`equal : 'a t -> 'a t -> bool`	`equal` $e\ e'$ teste l'égalité $e = e'$
`union , inter , diff : 'a t -> 'a t -> 'a t`	réunion, intersection, différence
`compare : 'a t -> 'a t -> int`	ordre total sur les ensembles
`elements : 'a t -> 'a list`	`elements` e = liste des éléments de e
`iter : ('a -> 'b) -> 'a t -> unit`	`iter` $f\ e$ applique f à ts les élts de e
`choose : 'a t -> 'a`	`choose` e renvoie un élément de e si $e \neq \emptyset$

A.7.6 Tables d'association (module `hashtbl`)

Une *table d'association* associant des éléments d'un ensemble F à des éléments d'un ensemble E est une application t qui à tout élément x de E associe une pile d'éléments de F. Si la pile $t(x)$ n'est pas vide, on dit que x est *lié* dans la table t et un élément de $t(x)$ est appelé une *liaison* de x dans t. Une liaison y de x est dite *plus récente* qu'une autre si elle est située plus près du sommet de $t(x)$; le sommet de la pile $t(x)$ (c.-à-d. la liaison la plus récente de x) est appelé la *liaison courante* de x.

Les principales opérations sont

- création d'une table vide (dans laquelle aucun élément de E n'est lié);
- ajouter une liaison (x,y) à une table t, c.-à-d. empiler y au sommet de la pile $t(x)$;
- trouver la liaison courante d'un élément x dans une table t, c.-à-d. si $t(x)$ n'est pas vide, retourner le sommet de $t(x)$;
- supprimer la liaison courante d'un élément x dans une table t, c.-à-d. dépiler $t(x)$.

Une possibilité d'implémentation est de représenter une table d'association par une liste des couples (x,y) tels que y soit une liaison de x en imposant la condition qu'une liaison y de x plus récente qu'une autre y' corresponde dans la liste à un couple (x,y) situé plus à gauche que (x,y'). Avec cette méthode, l'opération « ajouter » est en O(1) mais les opérations « trouver » et « supprimer » ont une complexité de l'ordre du nombre total des liaisons.

Le module `hashtbl` implémente les tables d'association en utilisant des *tables de hachage* (voir le problème Centrale 1998) [1].

`type ('a,'b) t`	le type des tables du type `'a` vers le type `'b`
`new : int -> ('a,'b) t`	`new` n crée une table vide de taille initiale n
`add : ('a,'b) t -> 'a -> 'b -> unit`	ajout d'une liaison à une table
`find : ('a,'b) t -> 'a -> 'b`	trouve la liaison courante (ou `Not_found`)
`remove : ('a,'b) t -> 'a -> unit`	supprime la liaison courante
`clear : ('a,'b) t -> unit`	vide une table

A.7.7 Graphisme (module `graphics`)

En *mode graphique* on effectue des tracés dans un plan discret $\mathbf{Z}^2$ mais il n'apparaîtra à l'écran que la partie du tracé contenue dans le rectangle $[|0,\ell-1|] \times [|0,h-1|]$ où ℓ est la largeur de l'écran (nombre de pixels sur une horizontale) et h la hauteur. Les coordonnées des points inférieur gauche et supérieur droit de l'écran sont donc respectivement (0,0) et

1. Il existe aussi le module `map` qui utilise des arbres binaires équilibrés.

$(\ell - 1, h - 1)$. A tout instant est défini un *point courant* $PC \in \mathbf{Z}^2$ qui désigne en quelque sorte l'endroit où se trouve le stylo traceur.

`open_graph : string -> unit`	`open_graph ""` passe en mode graphique
`clear_graph : unit -> unit`	`clear_graph()` efface l'écran graphique
`size_x` et `size_y : unit -> int`	`size_x()` $= \ell$ et `size_y()` $= h$
`plot : int -> int -> unit`	`plot` x y trace le point (x,y)
`moveto : int -> int -> unit`	`moveto` x $y \Leftrightarrow PC \leftarrow (x,y)$
`lineto : int -> int -> unit`	`lineto` x y trace le segment $[PC,(x,y)]$ et $PC \leftarrow (x,y)$
`current_point : unit -> int * int`	coordonnées du PC
`draw_string : string -> unit`	trace une chaine au PC
`close_graph : unit -> unit`	repasse en mode texte

Index