Option informatique : la deuxième année

Laurent Chéno

été 1996

Lycée Louis-le-Grand, Paris

Table des matières

I Arbres **13**

1 Arbres binaires **15**

1.1 Définitions et notations . . . 15
1.1.1 Définition formelle d'un arbre binaire . . . 15
1.1.2 Définition des arbres binaires en Caml . . . 16
1.1.3 Une indexation des éléments constitutifs d'un arbre . . . 16
1.2 Notion de profondeur dans un arbre . . . 17
1.2.1 Définitions . . . 17
1.2.2 Calcul de la profondeur en Caml . . . 18
1.3 Squelette d'un arbre binaire . . . 18
1.3.1 Un exemple . . . 18
1.3.2 Définition du squelette . . . 18
1.3.3 Le squelette comme une classe d'équivalence . . . 19
1.3.4 Écriture en Caml . . . 19
1.4 Combinatoire des arbres et squelettes binaires . . . 19
1.4.1 Nœuds et feuilles . . . 19
1.4.2 Profondeur et taille . . . 19
1.4.3 Dénombrement des squelettes binaires . . . 22
1.5 Exercices pour le chapitre 1 . . . 24

2 Parcours d'un arbre **27**

2.1 Parcours en largeur d'abord . . . 27
2.1.1 Description de l'ordre militaire . . . 27
2.1.2 Programmation Caml . . . 28
2.2 Parcours en profondeur d'abord . . . 28
2.2.1 Parcours préfixe, infixe, suffixe . . . 28
2.2.2 Programmation en Caml . . . 29
2.2.3 Problème inverse . . . 29
2.3 Exercices pour le chapitre 2 . . . 30

3 Arbres de recherche **31**

3.1 Définition d'un arbre binaire de recherche . . . 31
3.1.1 Définition générale . . . 31
3.1.2 Cas particulier . . . 31
3.2 Recherche d'un élément . . . 31
3.2.1 Position du problème . . . 31
3.2.2 Recherche dans un arbre binaire de recherche . . . 32
3.2.3 Évaluation . . . 32
3.3 Structure dynamique de recherche . . . 33
3.3.1 Ajout d'un élément . . . 33
3.3.2 Suppression d'un élément . . . 33
3.3.3 Application au tri . . . 34

3.3.4 Le problème de l'équilibrage . . . 35
3.4 Exercices pour le chapitre 3 . . . 35

4 Tri et tas **37**
4.1 Généralités . . . 37
4.1.1 Files de priorité . . . 37
4.1.2 Tas . . . 37
4.1.3 Implémentation des tas à l'aide de tableaux . . . 38
4.2 Percolation . . . 39
4.2.1 Description de l'algorithme . . . 39
4.2.2 Programmation . . . 41
4.2.3 Évaluation . . . 42
4.2.4 Application : création d'un tas . . . 42
4.3 Le tri par les tas . . . 43
4.3.1 Programmation . . . 43
4.3.2 Évaluation . . . 43

5 Arbres n-aires et expressions arithmétiques **45**
5.1 Arbres n-aires . . . 45
5.1.1 Définition . . . 45
5.1.2 Implémentation . . . 46
5.1.3 Propriétés . . . 46
5.1.4 Parcours d'un arbre n-aire . . . 46
5.2 Expressions arithmétiques . . . 47
5.2.1 Une définition . . . 47
5.2.2 Syntaxe concrète et syntaxe abstraite . . . 48
5.2.3 Expressions et arbres . . . 49
5.3 Dérivation formelle . . . 50
5.3.1 Dérivation guidée par la structure d'arbre . . . 50
5.3.2 Simplification : une première approche . . . 53
5.4 Exercices pour le chapitre 5 . . . 55

II Automates **57**

6 Automates finis déterministes ou non déterministes **59**
6.1 Automates finis déterministes . . . 59
6.1.1 Présentation informelle . . . 59
6.1.2 Présentation mathématique . . . 60
6.1.3 Transitions et calculs . . . 60
6.1.4 Langage reconnu par un automate fini déterministe . . . 61
6.2 Implémentation en Caml d'un *afd* . . . 61
6.3 Automates finis non déterministes . . . 62
6.3.1 Présentation informelle . . . 62
6.3.2 Définition mathématique . . . 63
6.4 Implémentation en Caml d'un *afnd* . . . 64
6.5 Déterminisation d'un *afnd* . . . 67
6.5.1 Algorithme de déterminisation . . . 67
6.5.2 Programmation en Caml . . . 69
6.6 Exercices pour le chapitre 6 . . . 71

7 Langages rationnels et automates **75**
7.1 Langages rationnels et expressions régulières 75
7.1.1 Définition 75
7.1.2 Expressions régulières 76
7.2 Le théorème de Kleene 77
7.2.1 Des langages rationnels aux automates 78
7.2.2 Des automates aux langages rationnels 79
7.3 Langages non rationnels 83
7.3.1 Exemples 83
7.3.2 Le lemme de l'étoile 83
7.4 Exercices pour le chapitre 7 84

III Corrigé de tous les exercices **87**

1 Exercices sur *Arbres binaires* **89**

2 Exercices sur *Parcours d'un arbre* **93**

3 Exercices sur *Arbres de recherche* **97**

5 Exercices sur *Arbres n-aires* **101**

6 Exercices sur *Automates finis* **103**

7 Exercices sur *Langages rationnels et automates* **107**

Table des figures

1.1 Premiers exemples d'arbres . . . 15
1.2 Indexation d'un arbre binaire . . . 17
1.3 Ambiguïté de l'indexation . . . 17
1.4 Construction du squelette . . . 18
1.5 Taille et profondeur . . . 20
1.6 Squelettes d'arbres complets . . . 22

2.1 Un arbre pour l'exemple . . . 27
2.2 Ambigüité de la description infixe . . . 30

3.1 Un arbre binaire de recherche sur $\mathbb{N}$ . . . 32
3.2 Arbre obtenu par suppression du 7 . . . 34
3.3 Arbre obtenu par suppression du 7 puis du 5 . . . 34

4.1 Une file de priorité . . . 37
4.2 Un tas . . . 38
4.3 Un tas avec sa numérotation des nœuds . . . 38
4.4 Un arbre à percoler . . . 39
4.5 L'étape intermédiaire de la percolation . . . 40
4.6 Le tas obtenu par percolation . . . 40

5.1 Un exemple d'arbre n-aire . . . 45
5.2 Un autre arbre n-aire de même parcours préfixe . . . 47

6.1 Un premier exemple d'automate fini déterministe . . . 59
6.2 Deux automates pour un même langage . . . 61
6.3 Un automate fini non déterministe . . . 63
6.4 Un *afnd* pour les mots qui finissent par ab . . . 64
6.5 Un *afnd* pour les mots finissant par $abab$ . . . 67
6.6 L'automate déterminisé . . . 68
6.7 Un *afnd* à la déterminisation coûteuse . . . 69

7.1 Automates pour les expressions régulières atomiques . . . 78
7.2 Automate pour la somme de deux expressions régulières . . . 78
7.3 Automate pour le produit de deux expressions régulières . . . 79
7.4 Automate pour l'étoile d'une expression régulière . . . 79
7.5 Avant la suppression de l'état x . . . 80
7.6 Après la suppression de l'état x . . . 80
7.7 Automate à 2 états . . . 80
7.8 Quel est le langage reconnu par cet automate ? . . . 81
7.9 On a supprimé l'état 2 . . . 81
7.10 Après suppression des états 2 et 4 . . . 82
7.11 On recommence en supprimant l'état 2 . . . 82
7.12 Après suppression des états 2 et 3 . . . 82

Liste des programmes

1.1 Définition du type `arbre_binaire` . . . 16
1.2 Calcul de la profondeur d'un arbre . . . 18
1.3 Nombre de nœuds, de feuilles d'un arbre, taille d'un squelette . . . 20

2.1 Parcours d'un arbre binaire en ordre militaire . . . 28
2.2 Parcours d'un arbre binaire en ordres préfixe, infixe et suffixe . . . 29
2.3 Reconstitution d'un arbre binaire à partir de sa description en ordre préfixe . . . 29
2.4 Reconstitution d'un arbre binaire à partir de sa description en ordre suffixe . . . 30

3.1 Recherche séquentielle dans une liste . . . 32
3.2 Recherche dans un arbre binaire de recherche . . . 32
3.3 Recherche dans un arbre binaire de recherche sur $\mathbb{N}$ . . . 33
3.4 Ajout d'un élément dans un arbre binaire de recherche . . . 33
3.5 Suppression d'un élément dans un arbre binaire de recherche . . . 35
3.6 Tri à l'aide d'arbres binaires de recherche . . . 35

4.1 Percolation récursive . . . 41
4.2 Percolation itérative . . . 41
4.3 Réorganisation d'un arbre en tas . . . 42
4.4 Le tri par tas *(heap sort)* . . . 43

5.1 Nombre de nœuds et feuilles d'un arbre n-aire . . . 46
5.2 Profondeur d'un arbre n-aire . . . 46
5.3 Parcours préfixe et suffixe d'un arbre n-aire . . . 47
5.4 Reconstitution d'un arbre n-aire à partir de son parcours préfixe . . . 48
5.5 Conversions arbres d'expression/expressions arithmétiques . . . 49
5.6 Évaluation des expressions arithmétiques . . . 50
5.7 Impressions préfixe et suffixe des expressions . . . 51
5.8 Impression infixe des expressions . . . 52
5.9 Dérivation formelle des expressions arithmétiques . . . 52
5.10 Simplification des expressions algébriques . . . 54

6.1 Reconnaissance d'une chaîne par un *afd* . . . 62
6.2 Quelques fonctions utiles . . . 65
6.3 Reconnaissance d'une chaîne par un *afnd* . . . 66
6.4 Fonctions utiles sur les ensembles . . . 71
6.5 La déterminisation des automates . . . 72

Liste des exercices

Exercice 1.1 *Indexation d'un arbre binaire* 24
Exercice 1.2 *Sous-arbres de profondeur donnée* 24
Exercice 1.3 *Calcul du squelette* 24
Exercice 1.4 *Génération des squelettes d'arbres binaires* 24
Exercice 1.5 *Squelette d'arbre complet de taille donnée* 24
Exercice 1.6 *Test de complétude* 25
Exercice 1.7 *Test d'équilibrage* 25

Exercice 2.1 *Reconstitution à partir de l'ordre militaire* 30
Exercice 2.2 *Conversions préfixe/suffixe* 30

Exercice 3.1 *Une autre structure d'arbre de recherche* 35
Exercice 3.2 *Balance et équilibrage* 35
Exercice 3.3 *Taille d'un arbre AVL* 36
Exercice 3.4 *Arbres 2–3* 36

Exercice 5.1 *Reconstitution d'un arbre n-aire à partir du parcours suffixe* 55
Exercice 5.2 *Impression infixe des expressions* 55

Exercice 6.1 *Quelques automates simples* 71
Exercice 6.2 *Déterminisation d'un automate simple* 71
Exercice 6.3 *Preuve de la déterminisation* 71
Exercice 6.4 *Ajout d'un état mort* 71
Exercice 6.5 *Déterminisation d'un afd!* 73
Exercice 6.6 *Minimisation d'un afd* 73

Exercice 7.1 *Langages rationnels, intersection et complémentaire* 84
Exercice 7.2 *Équivalence des expressions régulières* 84
Exercice 7.3 *Le langage des facteurs des mots d'un langage* 84
Exercice 7.4 *Reconnaissance d'un même langage* 84
Exercice 7.5 *Exemples de langages non rationnels* 84
Exercice 7.6 *Expressions régulières décrivant des langages donnés* 85

Première partie

Arbres

Chapitre 1

Arbres binaires

1.1 Définitions et notations

1.1.1 Définition formelle d'un arbre binaire

Commençons par une définition très mathématique.

Définition 1.1 (Arbres binaires) On considère deux ensembles de valeurs F (valeurs des feuilles) et N (valeurs des nœuds). Un arbre binaire sur ces ensembles est défini de façon récursive comme suit. Toute feuille, élément de F, est un arbre. Étant donnés une valeur n de nœud ($n \in N$), et deux arbres g et d, (n, g, d) est un nouvel arbre, de racine n, de fils gauche g, de fils droit d.

La définition que nous avons donnée ici permet de distinguer le type des informations portées respectivement par les nœuds et les feuilles d'un arbre.
On utilise d'habitude, au lieu d'écritures comme f pour une simple feuille, ou encore, pour donner un exemple plus complexe, $(n_1, (n_2, f_{21}, f_{22}), (n_3, f_{31}, (n_4, f_{41}, f_{42})))$, la représentation graphique qu'on trouvera dans la figure suivante.

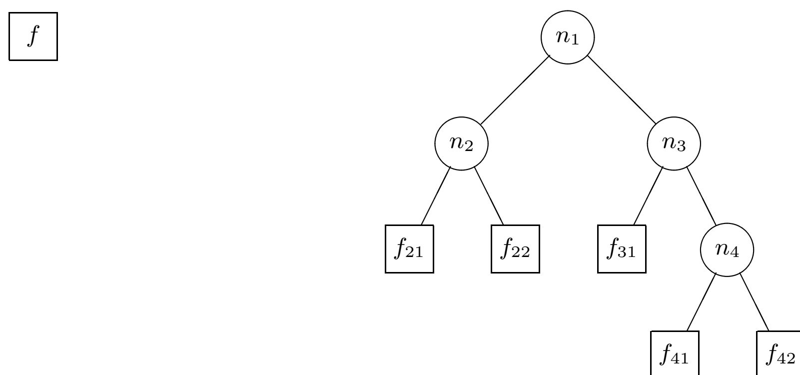


Fig. 1.1: Premiers exemples d'arbres

Dans cette représentation, les feuilles sont dessinées à l'aide de carrés, les nœuds par des cercles. Notons que certains auteurs utilisent les expressions *nœuds externes* pour les feuilles et *nœuds internes* pour ce que nous avons appelé les nœuds.
La représentation graphique introduit la notion de haut et de bas, et bien sûr les informaticiens ayant la tête un peu à l'envers, tous nos arbres auront leurs feuilles en bas et leur racine en haut...
Pour illustrer nos définitions, tentons de décrire avec notre nouveau vocabulaire le deuxième arbre de la figure ci-dessus. Sa racine est le nœud n_1, qui a deux fils, les arbres de racines n_2 et n_3. Le sous-arbre

gauche de racine n_2 a pour fils gauche et droit deux feuilles, tandis que le sous-arbre droit de racine n_3 a respectivement pour fils gauche et droit une feuille et un nouveau sous-arbre de racine n_4 qui à son tour a pour fils gauche et droit deux feuilles.
Le plus souvent, on se permettra l'abus de langage qui consiste à citer un nœud en pensant à l'arbre (le sous-arbre) dont il est racine. Ainsi n_2 désignera-t-il tout aussi bien le nœud précis qui porte ce nom que tout le sous-arbre qui l'admet pour racine.
Remarquons pour terminer que toute feuille figure nécessairement en bas de l'arbre, puisqu'elle ne peut avoir de fils. Enfin, on dit que a est père de b aussi bien quand b est une feuille qui est un fils (droit ou gauche) de a et quand b est la racine du sous-arbre fils droit ou gauche de a.

1.1.2 Définition des arbres binaires en Caml

En Caml, on définit un type à deux paramètres qui sont les types respectifs des feuilles et des nœuds, à l'aide de l'instruction suivante :

Programme 1.1 Définition du type `arbre_binaire`

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

On définira les deux arbres de l'exemple ci-dessus à l'aide d'instructions comme

```
let f = Feuille("f")
and a = Nœud("n1", Nœud("n2",Feuille("f21"),Feuille("f22")),
                   Nœud("n3",Feuille("f31"),Nœud("n4",Feuille("f41"),Feuille("f42")))) ;;
```

1.1.3 Une indexation des éléments constitutifs d'un arbre

Nous allons maintenant définir une indexation des éléments constitutifs d'un arbre (feuilles et nœuds) à l'aide de mots sur l'alphabet $\{0, 1\}$.

Définition 1.2 (Mots binaires) On appelle mot binaire toute suite éventuellement vide de 0 ou de 1. Autrement dit, un mot binaire est ou bien le mot vide, noté ϵ, ou bien le résultat de la concaténation d'un mot binaire et d'un 0 ou d'un 1.

À tout mot binaire est naturellement associé un entier naturel, dont le mot choisi est une écriture en base 2. Par exemple, le mot binaire *10010* est associé à l'entier 18. Par convention, le mot vide ϵ sera associé à $-\infty$, et on notera $\mathbb{N}' = \mathbb{N} \cup \{-\infty\}$. Nous noterons par un dièse ($\sharp$) cette application des mots binaires vers $\mathbb{N}'$: ainsi écrirons-nous $\sharp 10010 = 18$.
On peut alors numéroter les éléments d'un arbre en leur associant un mot binaire. La règle est fort simple, et peut s'énoncer ainsi.

Définition 1.3 (Indexation) On considère un arbre binaire. S'il s'agit d'une feuille, on l'indexe par le mot vide ϵ. Si en revanche il s'agit d'un nœud (n, g, d), on commence par indexer sa racine n par le mot vide ϵ, et par effectuer l'indexation de tous les éléments de l'arbre g et de l'arbre d. Enfin, on ajoute devant l'index de chaque élément de l'arbre g (*resp.* d) un 0 (*resp.* un 1).

On retrouvera dans la figure suivante le deuxième arbre de la figure 1.1 page précédente où chaque élément est indexé à sa gauche par un mot binaire.
L'indexation ainsi effectuée discrimine effectivement les feuilles de l'arbre, ce qui fait l'objet du théorème suivant.

Théorème 1.1
Dans l'indexation précédente, si f_1 et f_2 sont deux feuilles distinctes d'un même arbre et m_1, m_2 les mots binaires associés, on a $m_1 \neq m_2$.

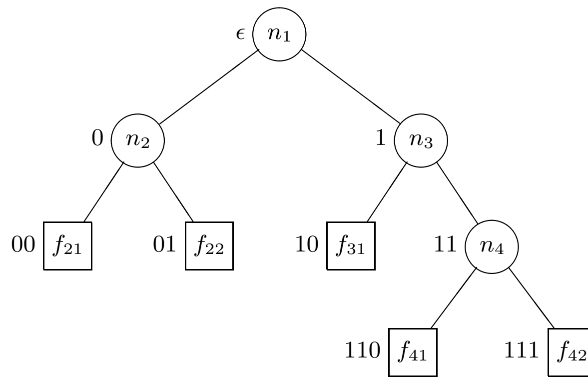


Fig. 1.2: Indexation d'un arbre binaire

✧ La démonstration se fait par récurrence structurelle.
Pour un arbre réduit à une feuille, le résultat est clair.
Soit $a = (n, g, d)$ un arbre. Soit f_1 et f_2 deux feuilles de cet arbre.
Ou bien f_1 et f_2 sont des feuilles du même sous-arbre g de a. Dans ce cas leurs index m_1 et m_2 sont obtenus à partir des index m'_1 et m'_2 qu'elles portent dans l'arbre g grâce aux relations $m_1 = 0m'_1$ et $m_2 = 0m'_2$. Comme par hypothèse de récurrence $m'_1 \neq m'_2$, on a bien $m_1 \neq m_2$.
Ou bien f_1 et f_2 sont des feuilles du même sous-arbre d de a. Dans ce cas on obtient de même $m_1 = 1m'_1 \neq 1m'_2 = m_2$.
Ou bien f_1 est une feuille de g et f_2 une feuille de d. Mais alors m_1 commence par un 0 et m_2 par un 1, et donc $m_1 \neq m_2$. ❖

En revanche, il faut faire attention à ce qu'on peut avoir $\sharp m_1 = \sharp m_2$, comme dans le cas de la figure 1.3, où les feuilles f_{41} et f_{31} ont pour index respectifs 010 et 10 qui sont bien différents, certes, mais pourtant $\sharp 010 = \sharp 10 = 2$.

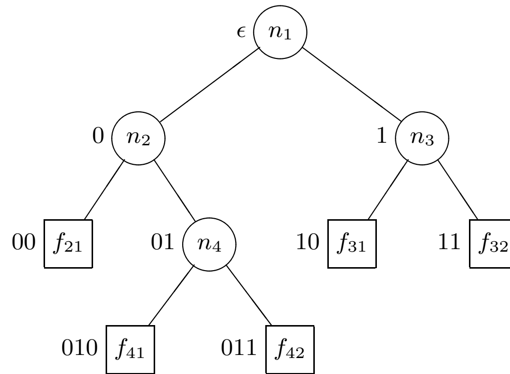


Fig. 1.3: Ambiguïté de l'indexation

1.2 Notion de profondeur dans un arbre

1.2.1 Définitions

On appelle profondeur d'un nœud ou d'une feuille d'un arbre le nombre d'arêtes qu'il faut traverser pour descendre de la racine de l'arbre au nœud ou la feuille visé(e).

D'une façon plus mathématique, on peut dire que la profondeur d'un élément d'un arbre est la longueur du mot binaire qui l'indexe dans l'indexation décrite plus haut.
La profondeur de l'arbre est définie comme étant le maximum des profondeurs de ses éléments, ou encore, puisque les feuilles sont sous les nœuds, comme le maximum des profondeurs de ses feuilles.

1.2.2 Calcul de la profondeur en Caml

Bien sûr, le calcul de la profondeur est aisé à l'aide d'un programme récursif :

Programme 1.2 Calcul de la profondeur d'un arbre

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

On aura noté que l'information portée par les nœuds et feuilles de l'arbre ne joue aucun rôle dans le calcul de la profondeur, évidemment.

1.3 Squelette d'un arbre binaire

1.3.1 Un exemple

Considérons à nouveau l'arbre de la figure 1.1 page 15. On peut commencer par *effacer* toute l'information portée par ses nœuds et par ses feuilles. On obtient ainsi un arbre dessiné à l'aide de cercles et de carrés. Si on supprime purement et simplement les feuilles, on obtient un *squelette* où figurent les seuls nœuds, comme dans la figure 1.4.

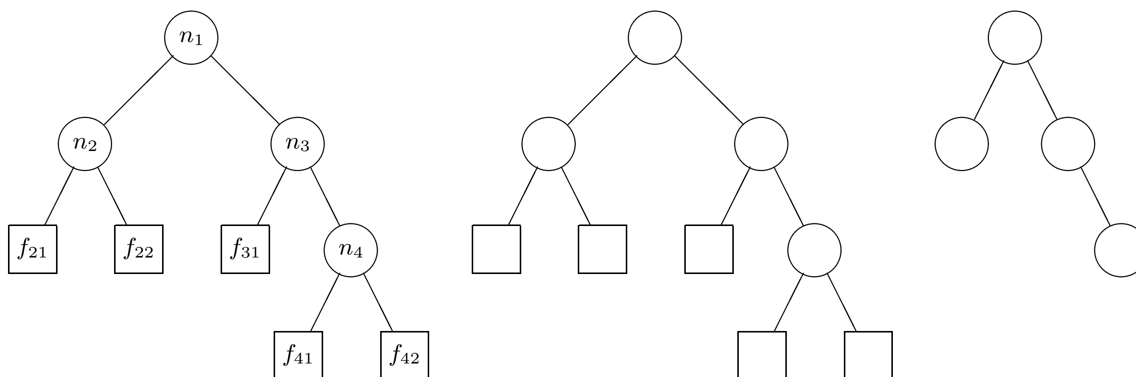


Fig. 1.4: Construction du squelette

Si bien entendu le passage du premier au second stade est destructif (on a effacé l'information portée), en revanche le passage du second au dernier est réversible, comme diraient les thermodynamiciens : il suffit en effet d'ajouter des feuilles à tous les endroits possibles, c'est-à-dire sous tous les nœuds qui ont 0 ou 1 fils.

1.3.2 Définition du squelette

Définition 1.4 (Squelettes binaires) Un squelette (d'arbre) binaire est défini de façon récursive comme suit. Le mot vide ϵ est un squelette binaire. Étant donnés deux squelettes g et d, (g, d) est un nouveau squelette, de fils gauche g, de fils droit d.

Avec cette définition — assez formelle —, tout squelette est un mot sur l'alphabet des trois caractères (,) et la virgule ,. Par exemple, le squelette de la figure précédente est $((,),(,(,)))$. Mais bien sûr tout mot ne représente pas un squelette, comme par exemple le mot $((,,))$.
On préfèrera néanmoins évidemment utiliser la représentation géométrique des squelettes d'arbres.

1.3.3 Le squelette comme une classe d'équivalence

Le passage d'un arbre binaire sur des ensembles N et F à son squelette se définit sans problème à l'aide d'une induction structurelle (voir l'exercice 1.3 page 24). L'égalité des squelettes définit sur l'ensemble des arbres binaires une relation d'équivalence, qui définit ce qu'on appelle souvent la *géométrie* d'un arbre binaire.

1.3.4 Écriture en Caml

Il suffit pour travailler sur les squelettes en Caml de définir un nouveau type de la façon suivante :

```
type squelette = Vide | Jointure of squelette * squelette ;;
```

1.4 Combinatoire des arbres et squelettes binaires

1.4.1 Nœuds et feuilles

Nous commençons par un résultat assez simple :

Théorème 1.2
Soit a un arbre binaire, et s son squelette. Soit respectivement n, p, n' le nombre de nœuds de a, le nombre de feuilles de a, et le nombre de nœuds de s. Alors $n = n' = p - 1$.

✧ L'égalité $n = n'$ est évidente.
Considérons l'arbre intermédiaire entre a et s, c'est-à-dire oublions l'information portée par les nœuds et les feuilles. Par définition même d'un arbre binaire, tout nœud de a a exactement 2 fils qui sont soit une feuille soit un nouveau nœud. Inversement, toute feuille et tout nœud sauf la racine admet un nœud-père. Ainsi $2n$ est le nombre de nœuds et feuilles qui ne sont pas à la racine de l'arbre, ou encore $2n = n + p - 1$, ce qui fournit l'égalité demandée. ✦

On aurait pu aussi démontrer ce théorème à l'aide d'une induction structurelle :

✧ Si a est une feuille, le résultat est clair car $n = 0$ et $p = 1$.
Sinon, $a = (x, g, d)$. Par récurrence, on sait que les nombres n_g et p_g (*resp.* n_d et p_d) de nœuds et feuilles de g (*resp.* de d) vérifient $n_g = p_g - 1$ et $n_d = p_d - 1$. Or a a pour feuilles les feuilles de g et celles de d, donc $p = p_g + p_d$ et pour nœuds les nœuds de g et ceux de d à quoi il faut ajouter le nœud x lui-même, donc $n = 1 + n_g + n_d$. Là encore on a bien $n = 1 + p_g - 1 + p_d - 1 = p_g + p_d - 1 = p - 1$. ✦

Personnellement, je préfère la première démonstration.

1.4.2 Profondeur et taille

Taille d'un arbre ou d'un squelette binaire

La notion de *taille* d'un arbre varie selon les auteurs, hélas : d'aucuns comptent les nœuds et les feuilles, d'autres seulement les nœuds, d'autres encore seulement les feuilles. Mais grâce au théorème 1.2 on passe aisément de l'une à l'autre.
Nous choisirons ici la définition suivante.

Définition 1.5 (Taille d'un arbre) On appelle taille d'un arbre binaire a et on note $|a|$ la taille de son squelette, c'est-à-dire le nombre de ses nœuds.

On calcule aisément le nombre de nœuds et/ou de feuilles d'un arbre binaire et la taille d'un squelette à l'aide de programmes récursifs très simples.

Programme 1.3 Nombre de nœuds, de feuilles d'un arbre, taille d'un squelette

```
(* calculs sur les arbres binaires *)
let rec nb_nœuds = function
    | Feuille(_) -> 0
    | Nœud(_,g,d) -> 1 + (nb_nœuds g) + (nb_nœuds d) ;;

let rec nb_feuilles = function
    | Feuille(_) -> 1
    | Nœud(_,g,d) -> (nb_feuilles g) + (nb_feuilles d) ;;

(* calcul sur les squelettes binaires *)
type squelette = Vide | Jointure of squelette * squelette ;;

let rec taille = function
    | Vide -> 0
    | Jointure(g,d) -> 1 + (taille g) + (taille d) ;;
```

Encadrement de la profondeur d'un arbre

On se rappelle que la profondeur d'un arbre est la profondeur maximale de ses nœuds et feuilles. Il est donc clair qu'on dispose du

Lemme 1.1 Si k est la profondeur d'un arbre binaire, son squelette est de profondeur $k-1$.

On se doute bien qu'un arbre binaire peut être très profond, il suffit de descendre toujours à gauche à partir de la racine, par exemple. Il est plus intéressant de prouver que pour une taille fixée, sa profondeur est minorée. La figure 1.5 montre deux squelettes de même taille et de profondeurs minimale et maximale. C'est ce que précise le théorème suivant :

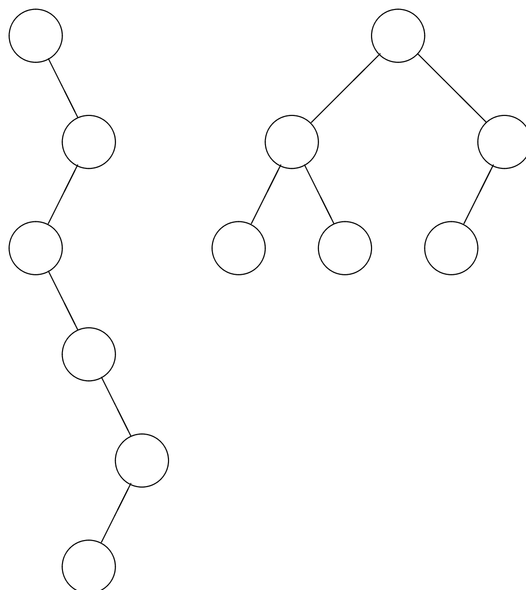


Fig. 1.5: Taille et profondeur

Théorème 1.3
Soit s un squelette binaire non vide, de taille n et de profondeur k. Alors $\lfloor \lg n \rfloor \leq k \leq n-1$.

qui a pour corollaire le

Théorème 1.4
Soit a un arbre binaire non réduit à une feuille, de taille n (il a donc n nœuds) et de profondeur k. Alors $1 + \lfloor \lg n \rfloor \leq k \leq n$.

Rappelons qu'en informatique lg désigne le logarithme en base 2, c'est-à-dire que $\lg n = \ln n / \ln 2$, et que $\lfloor x \rfloor$ désigne la partie entière de x.

✧ Nous démontrons le théorème 1.3 page précédente par induction structurelle.
Soit donc $s = (g, d)$ un squelette. Notons n sa taille, k sa profondeur, et n_g et k_g (*resp.* n_d et k_d) la taille et la profondeur de g (*resp.* de d).
Si g et d sont vides, $n = 1$ et $k = 0$: l'encadrement est correct.
Si g est vide, mais pas d, $n = 1 + n_d$ et $k = 1 + k_d$. Or on sait par hypothèse de récurrence que $\lg n_d \leq k_d \leq n_d - 1$. On en déduit que $k \leq n_d = n - 1$. La majoration est bien prouvée. En outre, puisque $n_d \geq 1$, on a $1 + \lfloor \lg n_d \rfloor = \lfloor 1 + \lg n_d \rfloor = \lfloor \lg(2n_d) \rfloor \geq \lfloor \lg(n_d + 1) \rfloor = \lfloor \lg n \rfloor$, ce qui fournit la minoration souhaitée.
Si d est vide, mais pas g, on raisonne de façon analogue.
Si enfin ni d ni g n'est vide, on a $n = 1 + n_d + n_g$ et $k = 1 + \max(k_d, k_g)$. La récurrence permet alors d'écrire les majorations : $k \leq 1 + \max(n_d, n_g) \leq 1 + \max(n - 1, n - 1) = n$. Pour ce qui est de la minoration, on écrit d'abord que $k_d \geq \lfloor \lg n_d \rfloor$ et $k_g \geq \lfloor \lg n_g \rfloor$.
Mais l'égalité entre entiers naturels $n = 1 + n_d + n_g$ montre que n_d ou n_g est plus grand que $\lceil (n-1)/2 \rceil$. Alors d'après l'hypothèse de récurrence, $\max(k_d, k_g) \geq \lfloor \lg \lceil \frac{n-1}{2} \rceil \rfloor$. Ainsi a-t-on $k \geq 1 + \lfloor \lg \lceil \frac{n-1}{2} \rceil \rfloor$.
Dans le cas où n est pair, $n = 2p + 2$, on a écrit $k \geq 1 + \lfloor \lg(p + 1) \rfloor = \lfloor \lg(2(p + 1)) \rfloor = \lfloor \lg n \rfloor$.
Dans le cas où n est impair, $n = 2p + 1$, on a écrit $k \geq 1 + \lfloor \lg p \rfloor = \lfloor \lg(2p) \rfloor = \lfloor \lg(2p + 1) \rfloor = \lfloor \lg n \rfloor$.
Cette dernière égalité fait l'objet du lemme suivant. ✦

Lemme 1.2 Pour tout entier naturel $p \geq 1$, on a $\lfloor \lg(2p + 1) \rfloor = \lfloor \lg(2p) \rfloor$.

✧ On a bien sûr l'inégalité $\lfloor \lg(2p + 1) \rfloor \geq \lfloor \lg(2p) \rfloor$.
Posons $k = \lfloor \lg p \rfloor$. On a $2^k \leq p \leq 2^{k+1} - 1$, d'où $2p + 1 \leq 2^{k+2} - 1 < 2^{k+2}$. Ainsi $\lg(2p + 1) < k + 2$, et $\lfloor \lg(2p + 1) \rfloor \leq k + 1 = \lfloor \lg(2p) \rfloor$, ce qui conclut. ✦

Arbres complets

On appelle *arbre complet* un arbre de profondeur k et de taille $n = 2^k - 1$. On trouvera dans la figure 1.6 page suivante les premiers arbres complets, ou plutôt leurs squelettes.
Donnons une caractérisation des arbres complets.

Théorème 1.5
Un arbre binaire est complet si et seulement si toutes ses feuilles sont à la même profondeur.

✧ Soit en effet a un arbre complet, de taille $n = 2^k - 1$ et de profondeur k. On sait qu'il y a au moins une feuille à la profondeur k. Supposons un moment qu'une autre feuille soit à une profondeur strictement plus petite que k. On pourrait la remplacer par un nœud d'où pendraient deux feuilles, de profondeur au plus égale à k. L'arbre ainsi étendu serait de taille $n + 1$ (on ajoute un seul nœud, on retranche une feuille et en ajoute deux nouvelles), et toujours de profondeur k. Mais le théorème 1.4 affirme $k \geq 1 + \lg(n + 1) = 1 + k$, ce qui fournit la contradiction souhaitée.
Montrons maintenant la réciproque : il suffit pour cela de compter le nombre de nœuds d'un arbre dont toutes les feuilles sont à profondeur k. Ceci ne pose pas de problème particulier, il suffit de remarquer que chaque nœud interne a deux fils, et on trouve effectivement $2^k - 1$ nœuds. ✦

Remarquons qu'un arbre complet de taille $n = 2^k - 1$ possède $n + 1 = 2^k$ feuilles (de profondeur k).

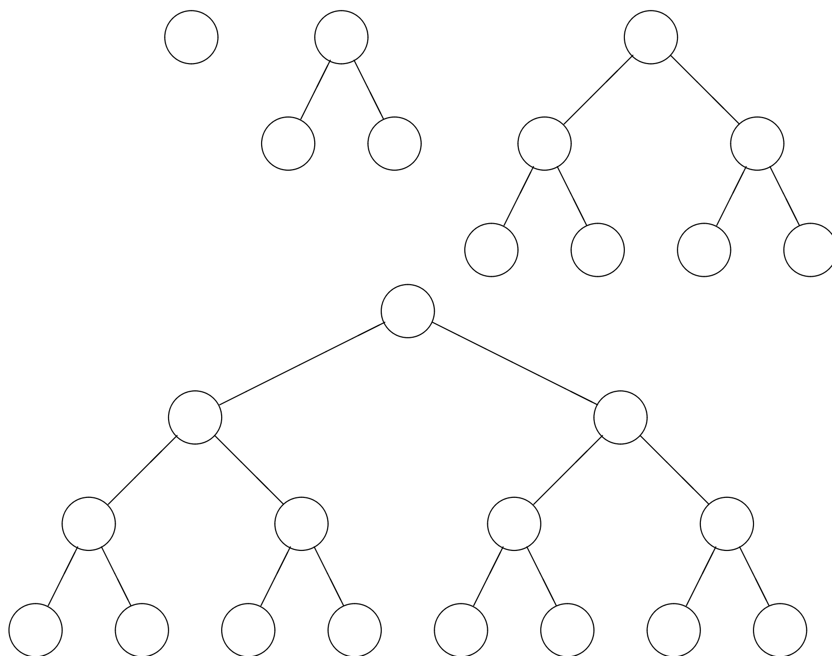


Fig. 1.6: Squelettes d'arbres complets

Arbres équilibrés

Un arbre sera dit *équilibré* quand ses feuilles se répartissent sur au plus deux niveaux seulement.
On verra plus loin dans ce cours l'intérêt que peut avoir pour un arbre cet équilibrage : ce sera le critère essentiel d'efficacité des algorithmes de recherche à l'aide de structures d'arbres.

Théorème 1.6 (Caractérisation des arbres équilibrés)
Un arbre binaire a de taille n et de profondeur k est équilibré si l'une ou l'autre des conditions équivalentes suivantes est vérifiée :

(1) toute feuille de a est de profondeur k ou $k-1$;

(2) l'arbre obtenu à partir de a en supprimant toutes ses feuilles de profondeur k est complet.

Un arbre équilibré vérifie la condition $k = 1 + \lfloor \lg n \rfloor$.

Bien entendu, quand on dit qu'on supprime des feuilles d'un arbre, on entend par là que l'on remplace leurs nœuds-pères par des feuilles.

✧ L'équivalence entre (1) et (2) est à peu près évidente.
Montrons simplement qu'on a bien, pour un arbre équilibré a, la relation $k = 1 + \lfloor \lg n \rfloor$.
Soit pour cela a' l'arbre complet obtenu par suppression dans a des feuilles de profondeur k. La taille de a' est n', il possède $n'+1$ feuilles, et donc $1 \leq n - n' \leq n' + 1$. La profondeur de a' est $k-1$, donc $n' = 2^{k-1} - 1$. Alors $1 \leq n - (2^{k-1} - 1) \leq 2^{k-1}$, et $2^{k-1} \leq n \leq 2^k - 1$. On a bien $\lfloor \lg n \rfloor = k - 1$. ✦

1.4.3 Dénombrement des squelettes binaires

Il va de soi que dénombrer les arbres binaires eux-mêmes n'a pas de sens, puisqu'aussi bien les ensembles des valeurs qui habillent nœuds et feuilles peuvent être infinis.
Il est donc naturel de s'intéresser ici aux squelettes d'arbres binaires.

Une récurrence naturelle

Pour compter les squelettes de taille n, on s'appuie sur le fait qu'un tel squelette est constitué d'une racine et de deux sous-arbres dont la somme des tailles vaut $n-1$, ce qui conduit à une formule du genre :

$$S_n = \sum S_i S_{n-1-i},$$

où S_k désigne le nombre de squelettes de taille k. Une convention s'impose donc : $S_0 = 1$. On en déduit immédiatement le théorème qui suit.

Théorème 1.7
Le cardinal S_n de l'ensemble des squelettes binaires de taille n vérifie la récurrence

$$\forall n \geq 1, S_n = \sum_{i=0}^{n-1} S_i S_{n-1-i}, \tag{1.1}$$

sous réserve de la convention $S_0 = 1$.

Le tableau suivant fournit les premiers termes de cette suite.

n	0	1	2	3	4	5	6	7	8	9	10
S_n	1	1	2	5	14	42	132	429	1430	4862	16796

Une formule plus *close*

Les informaticiens utilisent fréquemment ce qu'ils appellent des *fonctions génératrices*. Pour l'exemple qui nous intéresse, il s'agit de poser, pour tout (complexe) z (vérifiant certaines conditions dont nous nous occuperons plus tard)

$$S(z) = \sum_{k=0}^{+\infty} S_k z^k,$$

en espérant bien sûr qu'il n'y a pas de problème de convergence.
La récurrence qui fait l'objet de l'équation 1.1 se traduit, d'après la définition du produit de Cauchy de deux séries entières, et grâce à la condition $S_0 = 1$, par l'équation

$$S(z) = 1 + zS^2(z). \tag{1.2}$$

(Le décalage $n \to n-1$ de la somme de 1.1 est traduit par le facteur z devant $S^2(z)$.)
On s'intéresse donc naturellement à l'équation du second degré

$$X = 1 + zX^2 \tag{1.3}$$

qui a pour solution $X = \dfrac{1-\sqrt{1-4z}}{2z}$ (on choisit cette solution pour que faisant tendre z vers 0 on obtienne une limite égale à $S_0 = 1$).
Si on demande, pour se rassurrer (il est vrai que pour l'instant on s'est vraiment peu soucié de rigueur mathématique), à *Maple* de calculer le développement limité à l'ordre 8 de X, on obtient :

$$X = 1 + z + 2\,z^2 + 5\,z^3 + 14\,z^4 + 42\,z^5 + 132\,z^6 + 429\,z^7 + 1430\,z^8 + O\left(z^9\right). \tag{1.4}$$

Ces considérations conduisent à prendre en quelque sorte le problème à l'envers.
Posons

$$f(z) = \begin{cases} 1, & \text{si } z = 0\,; \\ \frac{1-\sqrt{1-4z}}{2z}, & \text{si } 0 \neq |z| < 1/4. \end{cases}$$

On montre alors facilement que f est de classe $\mathcal{C}^\infty$ sur $]-1/4, 1/4[$, et qu'elle vérifie à la fois l'équation 1.3 et la condition $f(0) = 1$.

Or il se trouve que l'on sait calculer son développement limité à l'origine à tout ordre. Les coefficients de ce développement limité vérifieront alors nécessairement l'équation 1.1. Ce seront bien les termes de la suite (S_n) !
On trouve de cette façon, au prix d'un calcul que tout hypo-taupin sait mener sans difficulté, le résultat qui s'énonce sous la forme du

Théorème 1.8 (Dénombrement des squelettes binaires)
Le nombre S_n de squelettes binaires de taille n, qui est aussi le nombre d'arbres binaires portant n nœuds (internes) vaut

$$S_n = \frac{1}{n+1}\binom{2n}{n}.$$

1.5 Exercices pour le chapitre 1

Exercice 1.1 Indexation d'un arbre binaire

Écrire une fonction Caml

```
indexation : ('f,'n) arbre_binaire -> (string,string) arbre_binaire
```

qui remplace toute l'information des nœuds et feuilles de l'arbre fourni en argument par le mot binaire correspondant à l'indexation décrite à la section 1.1.3 page 16.

Exercice 1.2 Sous-arbres de profondeur donnée

Écrire une fonction Caml

```
liste_à_profondeur : ('f,'n) arbre_binaire -> int -> ('f,'n) arbre_binaire list
```

qui prend en arguments un arbre binaire a et un entier n et qui renvoie la liste (éventuellement vide) de tous les sous-arbres de a dont la racine est à la profondeur n dans a.

Exercice 1.3 Calcul du squelette

Écrire une fonction Caml

```
déshabille : ('f,'n) arbre_binaire -> squelette
```

qui prend en argument un arbre binaire et qui renvoie son squelette.

Exercice 1.4 Génération des squelettes d'arbres binaires

Écrire une fonction Caml

```
engendre_à_profondeur : int -> int -> squelette list
```

qui prend en arguments deux entiers n et p et qui renvoie la liste des squelettes de taille n et de profondeur p. Dans le cas où $n = 0$, la valeur de p ne sera pas prise en compte.
En déduire une fonction

```
engendre : int -> squelette list
```

qui renvoie la liste de tous les squelettes dont la taille est fournie en argument.

Exercice 1.5 Squelette d'arbre complet de taille donnée

Écrire une fonction Caml

```
squelette_complet : int -> squelette
```

qui prend en argument une taille n et renvoie le squelette complet de taille n s'il existe, et déclenche une erreur sinon.

Exercice 1.6 Test de complétude

Écrire une fonction Caml

```
est_complet : ('f,'n) arbre_binaire -> bool
```

qui prend en argument un arbre binaire et qui dit si oui ou non il est complet.

Exercice 1.7 Test d'équilibrage

Écrire une fonction Caml

```
est_équilibré : ('f,'n) arbre_binaire -> bool
```

qui prend en argument un arbre binaire et qui dit si oui ou non il est équilibré.

Chapitre 2

Parcours d'un arbre

Ce court chapitre a pour but d'expliciter différentes méthodes utilisées pour lister les éléments d'un arbre de façon non ambigüe, de telle sorte qu'on puisse reconstituer sa structure à l'aide de cette seule liste.
On verra que la distinction faite dans les arbres binaires entre feuilles et nœuds permet de résoudre facilement ce problème, quand au contraire il faut davantage d'effort pour donner une description non ambigüe d'un squelette d'arbre binaire.
Dans ce qui suit, on utilisera comme exemple privilégié celui de l'arbre qui est dessiné dans la figure 2.1, et dont les feuilles sont des entiers, les nœuds des caractères.

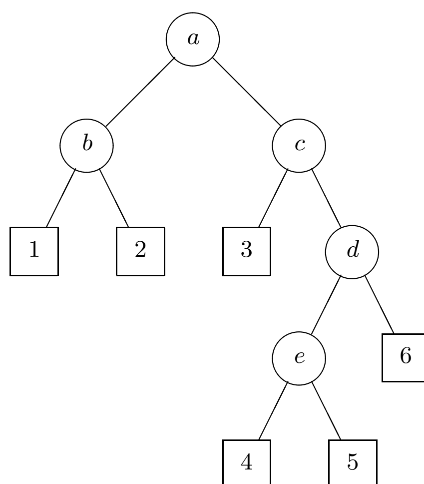


Fig. 2.1: Un arbre pour l'exemple

2.1 Parcours en largeur d'abord

2.1.1 Description de l'ordre militaire

Le premier type de parcours que nous allons écrire paraît sans doute le plus naturel, même si ce n'est pas le plus simple à programmer.
Certains l'appellent le *parcours militaire* d'un arbre, puisqu'il fait penser à l'axiome militaire bien connu qui donne la priorité *au plus ancien dans le grade le plus élevé*. Il suffit d'imaginer qu'en descendant dans l'arbre on descend dans la hiérarchie, et qu'on dispose les fils du plus ancien au plus jeune, pour faire le rapprochement.

On peut également décrire cet ordre de parcours en disant qu'on liste les nœuds et feuille par ordre croissant de profondeur, et de gauche à droite pour une profondeur donnée. C'est pourquoi d'autres l'appellent le parcours *en largeur d'abord*.
Dans le cas de notre exemple de la figure 2.1 page précédente, ce parcours s'écrit donc :

$$a\ b\ c\ 1\ 2\ 3\ d\ e\ 6\ 4\ 5.$$

La règle de reconstitution est simple : le premier élément écrit est la racine de l'arbre. À chaque fois qu'on dessine un nœud on dessine les deux arêtes pendantes qui en proviennent, et au fur et à mesure de la lecture, on remplit les trous ainsi constitués, de gauche à droite...
Reste à programmer ce parcours en Caml.

2.1.2 Programmation Caml

Nous aurons tout d'abord à définir le type qui correspond à la description de notre arbre. Il s'agira d'une liste d'éléments qui seront ou bien des feuilles ou bien des nœuds, ce qui s'écrit par exemple ainsi en Caml :

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

Le programme 2.1 permet alors de parcourir dans l'ordre militaire un arbre donné en argument.

Programme 2.1 Parcours d'un arbre binaire en ordre militaire

```
let parcours_militaire a =
    let rec parcours_rec nœuds_pendants = match nœuds_pendants with
        | [] -> []
        | Feuille(f) :: reste
            -> (F f) :: (parcours_rec reste)
        | Nœud(n,g,d) :: reste
            -> (N n) :: (parcours_rec (reste @ [ g ; d ]))
    in
    parcours_rec [a] ;;
```

On peut appliquer ce programme à notre arbre exemple, et voici le résultat :

```
#parcours_militaire exemple ;;
- : (int, char) listing_d'arbre list =
 [N `a`; N `b`; N `c`; F 1; F 2; F 3; N `d`; N `e`; F 6; F 4; F 5]
```

En revanche, la reconstruction de l'arbre à partir d'une telle liste est un problème beaucoup plus ardu...

2.2 Parcours en profondeur d'abord

Nous envisageons maintenant d'autres méthodes de parcours qui sont plus proches de la structure naturellement récursive des arbres, ce qui facilitera la programmation, et qui justifie qu'on les préfèrera au parcours militaire.
L'idée des trois parcours qui suivent est la même, et nous fera descendre tout en bas de l'arbre sur sa gauche avant de s'intéresser aux feuilles plus à droite : c'est ce qu'on appelle un parcours *en profondeur d'abord*.

2.2.1 Parcours préfixe, infixe, suffixe

Ces trois parcours s'appuient sur la structure récursive des arbres : on applique récursivement le parcours aux sous-arbres gauche et droit, et c'est le moment où on liste le nœud-père qui caractérise ces trois différents parcours.
Dans le parcours préfixe d'un nœud (n, g, d), on commence par lister n, puis on parcourt le sous-arbre g et enfin le sous-arbre d.

On obtient ainsi, dans le cas de notre arbre exemple, la séquence

$$a\ b\ 1\ 2\ c\ 3\ d\ e\ 4\ 5\ 6.$$

Dans le parcours infixe d'un nœud (n, g, d), on commence par parcourir le sous-arbre gauche, puis on liste n, et on termine par le sous-arbre d.
On obtient pour notre exemple la séquence

$$1\ b\ 2\ a\ 3\ c\ 4\ e\ 5\ d\ 6.$$

Dans le parcours suffixe (on dit aussi postfixe), enfin, on commence par parcourir les deux sous-arbres g et d et on termine en listant n, ce qui, pour notre exemple, fournit la séquence

$$1\ 2\ b\ 3\ 4\ 5\ e\ 5\ d\ c\ a.$$

2.2.2 Programmation en Caml

Ces trois parcours se prêtent tout à fait bien à une programmation récursive, et on obtient immédiatement le programme 2.2, qui met d'ailleurs en évidence la ressemblance de ces trois algorithmes de parcours.

Programme 2.2 Parcours d'un arbre binaire en ordres préfixe, infixe et suffixe

```
let rec parcours_préfixe = function
    | Feuille(f) -> [F f]
    | Nœud(n,g,d) -> [N n] @ (parcours_préfixe g) @ (parcours_préfixe d) ;;

let rec parcours_infixe = function
    | Feuille(f) -> [F f]
    | Nœud(n,g,d) -> (parcours_infixe g) @ [N n] @ (parcours_infixe d) ;;

let rec parcours_suffixe = function
    | Feuille(f) -> [F f]
    | Nœud(n,g,d) -> (parcours_suffixe g) @ (parcours_suffixe d) @ [N n] ;;
```

2.2.3 Problème inverse

À la différence du parcours en largeur d'abord, il n'est pas très difficile de procéder à la reconstitution de l'arbre initial à partir de sa description préfixe, comme le montre le programme 2.3.

Programme 2.3 Reconstitution d'un arbre binaire à partir de sa description en ordre préfixe

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

Contrairement à ce que certains s'imaginent parfois, le parcours suffixe d'un arbre ne fournit pas la description symétrique du parcours infixe, et il ne suffit pas d'appliquer `recompose_préfixe` au miroir d'une liste pour obtenir `recompose_suffixe`.
Mais une approche directe conduit à la solution, qui consiste à construire l'arbre final en faisant pousser petit à petit l'arbre à partir de ses feuilles les plus basses à gauche. C'est ce que fait le programme 2.4 page suivante.

Programme 2.4 Reconstitution d'un arbre binaire à partir de sa description en ordre suffixe

```
let recompose_suffixe l =
    let rec recompose ss_arbres liste = match ss_arbres,liste with
        | a,(F f) :: reste
            -> recompose (Feuille(f) :: a) reste
        | d :: g :: a,(N n) :: reste
            -> recompose (Nœud(n,g,d) :: a) reste
        | [ arbre ],[]
            -> arbre
        | _ -> failwith "Description suffixe incorrecte"
    in
    recompose [] l ;;
```

Il nous reste le problème de la reconstitution d'un arbre binaire à partir de sa description en ordre infixe. Et là, surprise ! on s'aperçoit que des trois méthodes de parcours d'arbre que nous venons de décrire, c'est la seule qui soit ambigüe ! C'est-à-dire qu'on peut très bien trouver un autre arbre que celui qui a été proposé dans la figure 2.1 page 27 avec pourtant le même parcours infixe, à savoir 1 b 2 a 3 c 4 e 5 d 6. Par exemple, on vérifiera que l'arbre de la figure 2.2 a aussi ce parcours infixe.

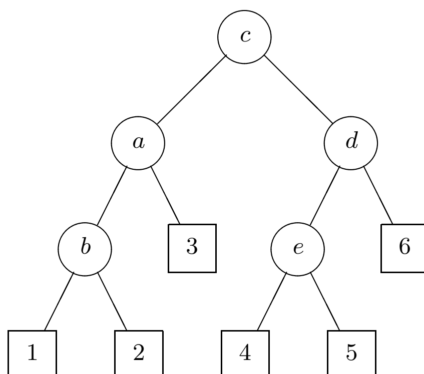


FIG. 2.2: Ambigüité de la description infixe

2.3 Exercices pour le chapitre 2

Exercice 2.1 Reconstitution à partir de l'ordre militaire

Écrire une fonction Caml

```
recompose_militaire : ('f,'n) listing_d'arbre list -> ('f, 'n) arbre_binaire
```

qui reconstitue un arbre binaire à partir de sa description en ordre militaire.

Exercice 2.2 Conversions préfixe/suffixe

Écrire une fonction Caml qui à partir de la description préfixe d'un arbre produit la description suffixe (sans reconstruire l'arbre, bien sûr). Que pensez-vous de la fonction inverse ?

Chapitre 3

Arbres de recherche

3.1 Définition d'un arbre binaire de recherche

3.1.1 Définition générale

On considère un ensemble ordonné $(F, \preceq)$ de feuilles et une application $\phi : F \to \mathbb{N}$ strictement croissante. C'est-à-dire que si $f_1 \prec f_2$ alors $\phi(f_1) < \phi(f_2)$.
Un arbre de recherche pour F est un arbre binaire sur les ensembles de feuilles et de nœuds F et $\mathbb{N}$ qui est ou bien une feuille ou bien un arbre (n, g, d) tel que pour toute feuille f du sous-arbre gauche g on a $\phi(f) \leq n$ et pour toute feuille f du sous-arbre droit d on a $n < \phi(f)$.
Il est clair qu'on peut choisir pour valeur de n tout entier de l'intervalle $[M, m[$, où M est le maximum des $\phi(f)$ sur les feuilles f de g et m le minimum des $\phi(f)$ sur les feuilles f de d. En général on choisit $n = M$, mais nous verrons qu'il faut savoir s'affranchir de cette contrainte, et on vérifiera que tous les résultats suivants n'utilisent pas cette propriété.
Si on se rappelle la définition des parcours préfixe, infixe et suffixe d'un arbre, on en déduit immédiatement par récurrence structurelle que les feuilles sont listées dans l'ordre croissant dans chacun de ces parcours.

3.1.2 Cas particulier

On utilise le plus généralement $F = \mathbb{N}$ avec l'identité pour ϕ, et tous nos exemples seront construits sur ce modèle.
On trouvera dans la figure 3.1 page suivante un premier exemple d'arbre de recherche sur $\mathbb{N}$, qui nous reservira dans la suite.

3.2 Recherche d'un élément

3.2.1 Position du problème

Comme leur nom l'indique, les arbres binaires de recherche servent à la recherche d'un élément d'une famille : plus précisément, on se donne un ensemble de valeurs et on veut écrire une fonction d'appartenance à cet ensemble.
Une solution simple consiste à utiliser une liste simple des éléments de l'ensemble considéré, et à faire une recherche dans cette liste, ce qui s'écrit en Caml comme dans le programme 3.1 page suivante.
Pas besoin d'une réflexion très approfondie pour prouver que cet algorithme tourne en $O(n)$, où n est la taille de l'ensemble de valeurs-cibles.

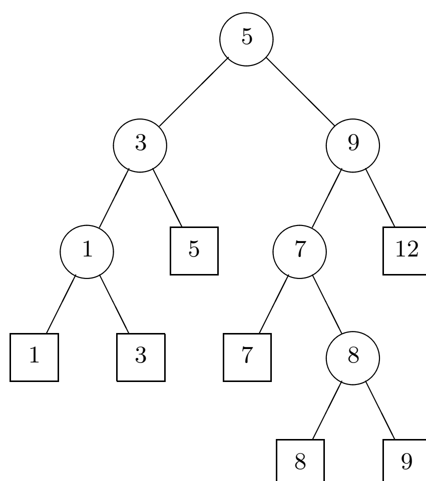


Fig. 3.1: Un arbre binaire de recherche sur $\mathbb{N}$

Programme 3.1 Recherche séquentielle dans une liste

```
let rec recherche l x =
    match l with
        | [] -> failwith "Élément absent de la liste"
        | t :: q -> t = x || recherche q x ;;
```

3.2.2 Recherche dans un arbre binaire de recherche

Les arbres binaires de recherche conduisent à une solution très simple du même problème, si on sait construire un arbre dont l'ensemble des feuilles est l'ensemble de nos valeurs-cibles.
C'est ce que propose l'algorithme du programme 3.2.

Programme 3.2 Recherche dans un arbre binaire de recherche

```
let rec recherche phi arbre x =
    match arbre with
        | Feuille(f) -> x = f || failwith "Élément absent"
        | Nœud(n,g,d) -> if phi x > n
                          then recherche phi d x
                          else recherche phi g x ;;
```

Bien sûr, ceci se simplifie dans le cas d'un arbre sur $\mathbb{N}$ et on obtient le programme plus simple 3.3 page suivante.

3.2.3 Évaluation

Si on suit sur l'arbre donné dans la figure 3.1 la recherche de l'élément 8, on se rend compte qu'on descend dans l'arbre en choisissant à chaque nœud de descendre soit à droite soit à gauche par une comparaison : ici, on descend successivement à droite, à gauche, à droite et à gauche.
Si on avait cherché la valeur 6 qui ne fait pas partie de l'ensemble des feuilles, on aurait utilisé la descente à droite, à gauche, à gauche, et on serait arrivé sur la feuille $7 \neq 6$, ce qui décide de l'échec.
On en déduit immédiatement par une récurrence structurelle le théorème qui suit.

Théorème 3.1 (Coût d'une recherche dans un arbre binaire)
La recherche dans un arbre binaire de recherche se réalise en un nombre de comparaisons au plus égal à $k+1$, où k est la profondeur de l'arbre.

Programme 3.3 Recherche dans un arbre binaire de recherche sur $\mathbb{N}$

```
let rec recherche arbre x =
    match arbre with
        | Feuille(f) -> x = f || failwith "Élément absent"
        | Nœud(n,g,d) -> if x > n then recherche d x
                                  else recherche g x ;;
```

En utilisant les théorèmes 1.2 page 19 et 1.4 page 21, on en déduit le corollaire suivant.

Théorème 3.2 (Encadrement du coût de la recherche dans un arbre binaire)
La recherche d'un élément dans un ensemble de n valeurs organisées en arbre binaire de recherche se réalise dans le cas le pire en un nombre $c(n)$ de comparaisons qui vérifie :

$$2 + \lfloor \lg(n-1) \rfloor \leq c(n) \leq n.$$

Dans le cas particulier où l'arbre de recherche utilisé est *équilibré*, on a l'égalité $c(n) = 2 + \lfloor \lg(n-1) \rfloor$, ce qui garantit un coût logarithmique pour notre recherche.
Toute la difficulté est donc de construire un arbre équilibré de recherche, ce que nous étudions dans la section suivante.

3.3 Structure dynamique de recherche

En pratique on veut maintenir une structure *dynamique* de l'ensemble des valeurs-cibles. C'est-à-dire qu'on veut pouvoir ajouter ou retrancher une valeur à cet ensemble. Il s'agit donc pour nous d'ajouter ou retrancher une feuille à un arbre binaire de recherche.

3.3.1 Ajout d'un élément

L'ajout d'un élément x se réalise un peu comme la recherche : on descend dans l'arbre jusqu'à arriver à une feuille. Ou bien c'est l'élément qu'on veut ajouter, il y est déjà, et il n'y a rien à faire. Ou bien c'est un élément y différent de x, ce qui signifie que x n'est pas encore dans l'arbre. On remplace alors la feuille y par un arbre $(\phi(x), x, y)$ si $\phi(x) < \phi(y)$ ou $(\phi(y), y, x)$ sinon.
Tout ceci se programme sans difficulté en Caml : c'est le programme 3.4.

Programme 3.4 Ajout d'un élément dans un arbre binaire de recherche

```
let rec ajout phi arbre x = match arbre with
    | Feuille(y)
        -> if x = y then Feuille(y)
           else if phi x < phi y then Nœud(phi x,Feuille(x),Feuille(y))
           else Nœud(phi y,Feuille(y),Feuille(x))
    | Nœud(n,g,d)
        -> if phi x > n then Nœud(n,g,ajout phi d x)
           else Nœud(n,ajout phi g x,d) ;;
```

La fonction `ajout` ainsi écrite renvoie le nouvel arbre résultat. Il est immédiat qu'elle tourne en $O(k)$ où k est la profondeur de l'arbre.

3.3.2 Suppression d'un élément

La suppression d'un élément dans un arbre binaire de recherche n'a bien sûr de sens que si cet élément figure parmi les feuilles, sans quoi il ne se passe rien.
Il suffit *a priori* de rechercher l'élément considéré, et de couper la feuille correspondante de l'arbre. Puis on remplace le nœud-père par le frère gauche ou droit qui subsiste.

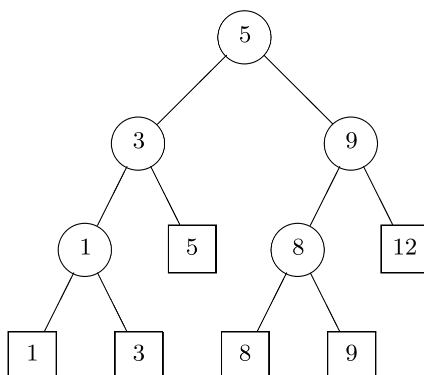


FIG. 3.2: Arbre obtenu par suppression du 7

Reprenant notre arbre de la figure 3.1 page 32, si on supprime l'élément 7, on remplace le père de la feuille 7 par son fils droit, et on obtient bien un arbre de recherche qui répond au problème qu'on s'était posé, comme le montre la figure 3.2.
Supprimons maintenant de ce nouvel arbre l'élément 5. On remplace donc son père par son frère gauche, et on obtient le nouvel arbre de la figure 3.3.

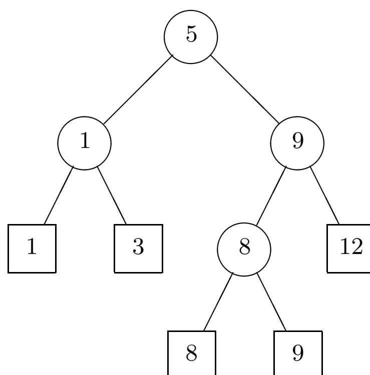


FIG. 3.3: Arbre obtenu par suppression du 7 puis du 5

L'arbre obtenu est bien un arbre de recherche, et la suppression a bien été effectuée. En revanche on s'aperçoit que la propriété *cosmétique* qui était jusqu'à présent conservée n'est plus vérifiée : les valeurs des nœuds ne sont plus nécessairement égales à la feuille maximale du sous-arbre gauche. Ainsi, la racine de notre arbre vaut-elle toujours 5, qui ne figure pourtant plus dans l'arbre. Mais on l'a déjà dit, ce n'est pas un problème.
On programme aisément cette suppression en Caml : c'est le programme 3.5 page suivante.
Ce programme tourne là encore en $O(k)$, où k est la profondeur de l'arbre.

3.3.3 Application au tri

Il suffit de combiner les programmes précédents pour obtenir un tri.
On construit pour commencer un arbre constitué d'une seule feuille contenant le premier élément de la liste à trier, auquel on ajoute successivement les autres éléments. Il suffit de lire les feuilles par un parcours récursif de l'arbre pour obtenir la liste triée. Le coût de cet algorithme de tri s'obtient en sommant les coûts des ajouts successifs. Si on peut s'assurer que l'arbre reste équilibré tout au long de l'algorithme, on aura un coût de l'ordre de

$$\lg 1 + \lg 2 + \cdots + \lg n = O(n \lg n);$$

Programme 3.5 Suppression d'un élément dans un arbre binaire de recherche

```
let rec suppression phi arbre x = match arbre with
    | Feuille(y)
        -> if x = y then failwith "Arbre vide"
           else Feuille(y)
    | Nœud(n,g,d)
        -> if phi x > n then
                if d = Feuille(x) then g
                else Nœud(n,g,suppression phi d x)
           else if g = Feuille(x) then d
                else Nœud(n,suppression phi g x,d) ;;
```

Programme 3.6 Tri à l'aide d'arbres binaires de recherche

```
let tri_par_arbre_de_recherche liste =
    let rec liste_feuilles = function
        | Feuille(f) -> [ f ]
        | Nœud(n,g,d) -> (liste_feuilles g) @ (liste_feuilles d)
    in
    let rec ajoute_feuilles arbre l = match l with
        | [] -> arbre
        | t :: q -> ajoute_feuilles (ajout (function x -> x) arbre t) q
    in
    match liste with
        | [] -> []
        | t :: q -> liste_feuilles (ajoute_feuilles (Feuille t) q) ;;
```

sinon, il se pourrait que chaque ajout se fasse pour un coût linéaire, et on retomberait alors sur un coût global quadratique, c'est-à-dire en $O(n^2)$.

3.3.4 Le problème de l'équilibrage

Tout le problème est donc bien d'assurer que l'arbre reste équilibré, ce qui n'est pas garanti par nos algorithmes trop rudimentaires. Il suffit pour s'en convaincre de considérer l'arbre obtenu par ajout successif des éléments 1, 2, ..., 8, qui est tout sauf équilibré !

3.4 Exercices pour le chapitre 3

Exercice 3.1 Une autre structure d'arbre de recherche

Une autre solution pour les arbres de recherche consiste à placer l'information à chercher aux nœuds. On écrit donc le type

```
type 'a arbre_de_recherche =
    | Vide
    | Nœud of 'a * 'a arbre_de_recherche * 'a arbre_de_recherche ;;
```

Ce ne sont plus les valeurs des feuilles du sous-arbre gauche qui seront plus petites que la racine, mais celles de ses nœuds.
Écrire pour cette nouvelle structure les fonctions de recherche, d'ajout et de suppression d'un élément.

Exercice 3.2 Balance et équilibrage

On reprend la représentation des arbres binaires de recherche définie dans l'exercice précédent.
Écrire la fonction `mesure_équilibrage` qui prend un arbre de type `'a arbre_de_recherche` et qui renvoie en résultat un arbre de type `('a * int) arbre_de_recherche` obtenu en ajoutant à chaque nœud l'entier $k_g - k_d$ où k_g est la profondeur de son fils gauche et k_d la profondeur de son fils droit. On utilisera par convention -1 comme profondeur de `Vide`.
Un arbre sera dit AVL si on a toujours $k_g - k_d \in \{-1, 0, +1\}$.

Exercice 3.3 Taille d'un arbre AVL

Écrire et démontrer les inégalités qui constituent l'encadrement de la profondeur k d'un arbre AVL en fonction de sa taille n.
On montre qu'on peut utiliser ces arbres comme arbres de recherches, ce qui assure des opérations élémentaires (ajout et suppression d'un élément) toutes de coût logarithmique, même dans le pire des cas. Il faut pour cela définir des fonctions d'ajout et de suppression d'un élément qui conserve la propriété des arbres AVL.

Exercice 3.4 Arbres 2–3

Un arbre de recherche 2–3 est un arbre où l'information recherchée est aux feuilles, dont tout nœud a 2 ou 3 fils, et dont toutes les feuilles sont à la même hauteur.
Proposer un type Caml pour ces arbres de recherche, et écrire la fonction de recherche correspondante.
Montrer que la profondeur k des feuilles et le nombre n de ces feuilles vérifient un encadrement que l'on précisera.
Là encore, on peut construire des algorithmes d'ajout et de suppression d'un élément sur un arbre 2–3, ce qui en fait une structure de recherche à coût toujours logarithmique.

Chapitre 4

Tri et tas

4.1 Généralités

4.1.1 Files de priorité

Une file de priorité est un arbre binaire dont les feuilles et les nœuds sont des entiers. C'est pourquoi, dans la suite de ce chapitre, on ne parlera plus de nœuds et de feuilles mais de nœuds internes et de nœuds externes. On dessinera tous ces nœuds de la même façon : avec un cercle. La taille de la file sera le nombre total de ses nœuds, internes ou externes.
Mais une file de priorité doit également vérifier la propriété suivante : tout père est plus grand que chacun de ses fils. Si l'on préfère, on peut dire que le long de tout chemin qui descend dans l'arbre de la racine à l'un de ses nœuds, on lit une suite décroissante d'entiers.
On notera qu'en revanche il n'y a pas de condition imposée sur des frères. Évidemment la racine d'une file de priorité est le plus grand entier présent dans tout l'arbre.
La figure 4.1 montre un exemple de file de priorité.

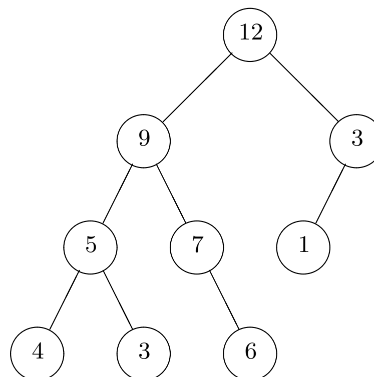


Fig. 4.1: Une file de priorité

4.1.2 Tas

Un tas est un cas particulier de file de priorité, qui est équilibré, et dont les nœuds externes de profondeur maximale sont tous le plus à gauche possible. On parle souvent dans cette situation d'arbre *parfait*.
On trouvera dans la figure 4.2 page suivante un exemple de tas.
En utilisant les résultats généraux étudiés dans les chapitres précédents, on peut démontrer le théorème 4.1 page suivante.

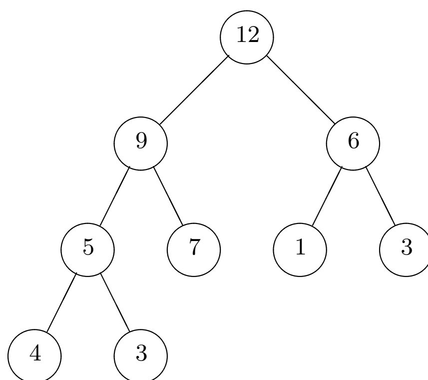


FIG. 4.2: Un tas

Théorème 4.1 (Propriétés des tas)
Soit $\mathcal{T}$ un tas de taille n. Sa profondeur k vérifie l'égalité $k = \lfloor \lg n \rfloor$. Parmi ses nœuds externes, il y a exactement $n + 1 - 2^k$ nœuds à profondeur égale à k, les autres sont de profondeur $k - 1$.

✧ La profondeur d'un arbre équilibré est connue depuis le théorème 1.6 page 22.
Comme l'arbre obtenu à partir de notre tas en supprimant les nœuds externes de profondeur k est complet et de profondeur $k-1$, il possède $2^k - 1$ nœuds, qui sont les nœuds internes du tas et ses nœuds externes de profondeur $k - 1$. C'est dire qu'il y a bien $n - (2^k - 1)$ nœuds externes de profondeur k. ✦

4.1.3 Implémentation des tas à l'aide de tableaux

On pourrait bien entendu utiliser une structure d'arbre pour représenter les tas, en définissant un type Caml de la façon suivante :

```
type tas = Vide | Nœud of int * tas * tas ;;
```

Il se trouve qu'il existe une implémentation beaucoup plus simple et efficace, à l'aide d'un bête tableau. Si en effet on veut décrire un tas, le plus simple est assurément de donner sa description dans un parcours militaire. Il n'y a pas d'ambigüité dans la mesure où seule la dernière *ligne* du tas peut être incomplète.
Précisons ceci, en définissant une numérotation des nœuds d'un tas de façon récursive : la racine est numérotée 0. Si i est le numéro d'un nœud interne, ses nœuds fils gauche et droit sont numérotés respectivement $2i + 1$ et $2i + 2$.
On trouvera dans la figure 4.3 le tas de l'exemple précédent avec les numéros de chacun des nœuds à sa gauche.

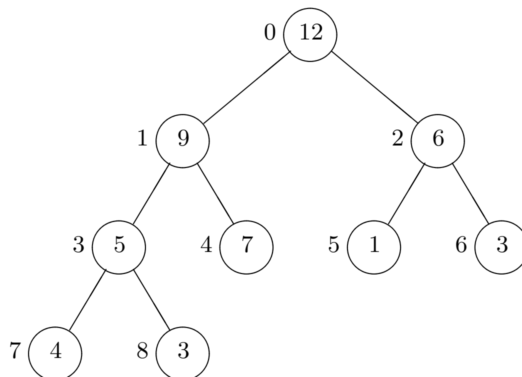


FIG. 4.3: Un tas avec sa numérotation des nœuds

Le théorème 4.2 montre que la numérotation que nous venons de définir correspond effectivement à un parcours militaire du tas.

Théorème 4.2 (Numérotation des nœuds d'un tas)
Pour la numérotation définie plus haut, on énumère les entiers de 0 à $n-1$ en numérotant les nœuds dans l'ordre du parcours militaire d'un tas de taille n.

✧ Nous procèdons par récurrence sur la profondeur k du tas.
Le résultat est clair pour un tas de profondeur nulle.
Supposons le acquis pour les tas de profondeur au plus égale à k ($k \geq 0$) et considérons un tas $\mathcal{T}$ de profondeur $k+1$.
La définition d'un tas montre qu'en supprimant dans $\mathcal{T}$ tous les nœuds (externes) de profondeur $k+1$, on obtient un arbre complet, de taille $2^{k+1}-1$. On a dit qu'il possédait 2^k nœuds externes, et d'après l'hypothèse de récurrence, elles portent les numéros 2^k-1 à $2^{k+1}-2$.
Posons $\lambda : i \mapsto 2i+1$ et $\mu : i \mapsto 2i+2$.
λ et μ sont bien sûr injectives, et $\lambda(\mathbb{N})$ et $\mu(\mathbb{N})$ sont clairement disjoints par raison de parité.
Enfin, pour tout entier i, $\lambda(i)$, $\mu(i)$, $\lambda(i+1)$ et $\mu(i+1)$ sont, dans cet ordre, des entiers consécutifs.
Il suffit pour conclure d'observer que $\lambda(2^k-1) = 2^{k+1}-1$. La $k+1$-ième *ligne* du tas porte donc des numéros consécutifs qui débutent effectivement à l'entier qui suit $2^{k+1}-2$, dernier numéro porté par la ligne de profondeur k. ✦

On va donc implémenter un tas de taille n sous la forme d'un vecteur v d'entiers, et on usera des analogies suivantes : v_{2i+1} est le fils gauche de v_i, v_{2i+2} est son fils droit, $v_{\lfloor (i-1)/2 \rfloor}$ est son père.

4.2 Percolation

4.2.1 Description de l'algorithme

On appelle percolation l'opération qui consiste à réorganiser un arbre sous forme d'un tas sachant que les deux sous-arbres de la racine étaient déjà des tas : il s'agit donc de mettre à sa place la racine.
Par exemple, on peut considérer l'arbre A de la figure 4.4, et, après percolation, on obtient le tas T de la figure 4.6 page suivante.

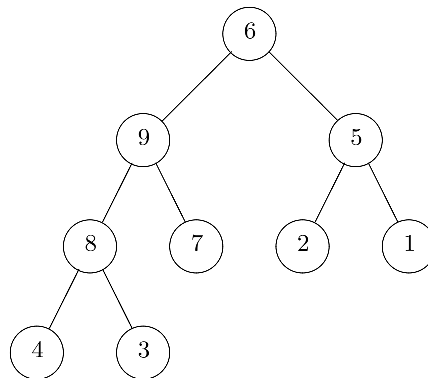


Fig. 4.4: Un arbre à percoler

L'algorithme de percolation est simple : on descend la valeur x inappropriée de la racine (dans notre exemple $x = 6$) en cherchant sa nouvelle place dans l'arbre. Si x est plus grand que ses deux fils, c'est qu'il a trouvé sa place, et on a fini, sinon, on échange x avec le plus grand de ses deux fils.
On trouvera dans la figure 4.5 page suivante l'étape intermédiaire de la percolation, où on a entouré d'un carré la position courante de x.
Évidemment l'arbre obtenu n'est un tas que dans la mesure où les deux fils gauche et droit de la racine de l'arbre initial étaient eux-mêmes des tas : on n'a fait que replacer au bon endroit la racine.

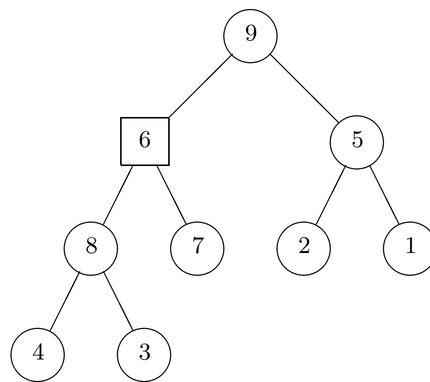


FIG. 4.5: L'étape intermédiaire de la percolation

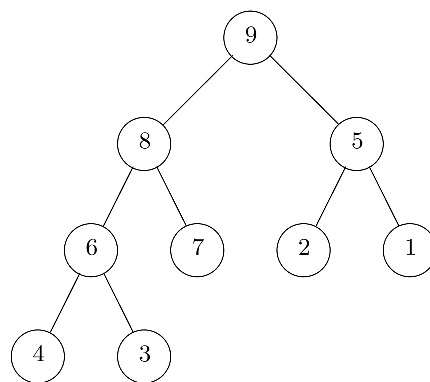


FIG. 4.6: Le tas obtenu par percolation

4.2.2 Programmation

Utilisons l'implémentation par vecteur d'un tas pour programmer cette percolation, ce qui fait l'objet des programmes 4.1 et 4.2, dans les deux versions récursive et itérative.

Programme 4.1 Percolation récursive

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

Programme 4.2 Percolation itérative

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

On aura noté comment dans les deux programmes on évite des comparaisons fastidieuses entre indices et n pour vérifier qu'on n'est pas sur un nœud externe avant de calculer les fils en comptant sur l'erreur qui serait déclenchée par un accès incorrect aux éléments du vecteur et en renvoyant alors une valeur, -1, inférieure à tous les autres éléments et qui termine donc la percolation. En outre, pour le programme itératif, on a choisi, plutôt que de procéder à des échanges dans le vecteur, de ne recopier la valeur de x que tout à la fin, quand sa position finale a été trouvée.

4.2.3 Évaluation

Il est bien évident que notre algorithme de percolation tourne en $O(k)$, où k est la profondeur de notre arbre, donc en $O(\lg n)$, puisqu'il s'agit d'un tas.

4.2.4 Application : création d'un tas

Réorganisation

Étant donné un vecteur v quelconque de n entiers naturels, on souhaite écrire une procédure `réorganise` qui permute ses éléments de telle sorte que le vecteur résultant soit la représentation d'un tas.
v représente *a priori* un arbre. On va, en commençant par les niveaux de plus grande profondeur, remonter en s'assurant au fur et à mesure que les sous-arbres dont la racine est à profondeur i sont des tas. Pour passer à la profondeur $i-1$, il suffira de percoler tous les nœuds de cette profondeur.
Observons toutefois qu'il n'est pas nécessaire de percoler les nœuds externes, qui ne posent évidemment pas de problème.
Or nous savons depuis le théorème 4.1 page 38 quels sont les nœuds internes et les nœuds externes, et il suffit donc de percoler à rebours tous les nœuds dont les indices varient de $\lfloor n/2 \rfloor - 1$ à 0. C'est ce que fait le programme 4.3.

Programme 4.3 Réorganisation d'un arbre en tas

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

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

On a réécrit ici la fonction de percolation, en demandant que `percolation tas i j` ne percole que le sous-arbre de `tas` dont la racine porte le numéro `i` et dont les nœuds sont de numéros strictement plus petit que `j`.

Évaluation

L'algorithme de réorganisation que nous avons écrit a un coût qu'il n'est pas difficile d'évaluer, puisqu'on applique la percolation à $n/2$ reprises pour des profondeurs qui ne dépassent pas $\lg n$. Il tourne donc en $O(n \lg n)$.

4.3 Le tri par les tas

4.3.1 Programmation

Nous avons ici une nouvelle façon de trier un vecteur v de n entiers naturels. Une fois réorganisé en tas, le maximum du tableau est simplement la racine v_0 de l'arbre. On l'échange avec le dernier élément v_{n-1}, et il n'y a plus qu'à recommencer avec le sous-tableau $v[0..n-2]$. En outre, il n'est pas nécessaire de réorganiser complètement ce sous-tableau : il suffit de procéder à une percolation.
C'est ce qui conduit à l'algorithme dit *tri par tas* ou, en anglais, *heap sort*, qui fait l'objet du programme 4.4.

Programme 4.4 Le tri par tas *(heap sort)*

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

4.3.2 Évaluation

Nous avons tout ce qu'il faut pour évaluer le coût de ce nouvel algorithme de tri. Nous avons dit que la réorganisation en tas avait un coût en $O(n \lg n)$; il reste ensuite $n-1$ percolations à réaliser, toutes de coût majoré par $\lg n$. L'algorithme proposé est donc bien un nouvel algorithme de tri en $O(n \lg n)$, ce que nous savons être optimal.

Chapitre 5

Arbres n-aires et expressions arithmétiques

5.1 Arbres n-aires

5.1.1 Définition

Les arbres n-aires sont une généralisation des arbres binaires : on autorise cette fois les nœuds à posséder un nombre quelconque de fils. On peut définir rigoureusement ces arbres de la façon suivante.

Définition 5.1 (Arbres n-aires) Soit F et N des ensembles de valeurs de feuilles et de nœuds. On définit récursivement un arbre n-aire sur ces ensembles en disant que toute feuille $f \in F$ est un arbre n-aire, et que si $n \in N$, si p est un entier, $p \geq 1$, et si enfin a_1, a_2, ..., a_p sont des arbres n-aires, alors $(n, a_1, \ldots, a_p)$ est aussi un arbre n-aire. Dans ce dernier cas on dit que n est le nœud père de chacun des a_i, que les a_i sont des sous-arbres frères, fils du nœud n.

On représente bien entendu graphiquement les arbres n-aires, d'une façon analogue à ce qu'on a fait pour les arbres binaires, comme sur l'exemple de la figure 5.1, qui représente l'arbre

$$(n_1, (n_2, f_1, f_2, f_3), (n_3, (n_4, f_4, f_5))).$$

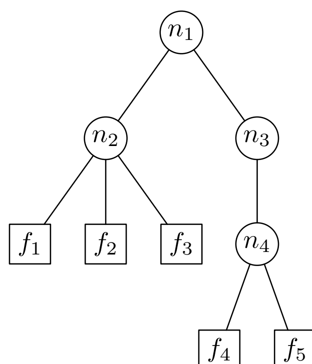


Fig. 5.1: Un exemple d'arbre n-aire

Notons qu'il n'est pas demandé aux nœuds d'un arbre n-aire d'avoir tous le même nombre de fils.

5.1.2 Implémentation

Une solution pour implémenter les arbres n-aires en Caml consiste à définir un nouveau type où les nœuds contiennent leur information propre et une liste d'arbres binaires :

```
type ('f,'n) arbre_n_aire =
    | Feuille of 'f
    | Nœud of 'n * ('f,'n) arbre_n_aire list ;;
```

L'arbre de l'exemple de la figure 5.1 page précédente s'écrit alors

```
Nœud("n1", [ Nœud("n2", [ Feuille("f1"); Feuille("f2"); Feuille("f3") ]) ;
             Nœud("n3", [ Nœud("n4" ,
                                [ Feuille("f4"); Feuille("f5") ] ) ] )
           ] )
```

5.1.3 Propriétés

On ne peut définir de la même façon que pour les arbres binaires la taille d'un arbre n-aire, car il n'y a plus de relation automatique entre nombre de feuilles et nombre de nœuds. Il conviendra donc de préciser ces deux nombres à chaque fois qu'on voudra parler de la *taille* d'un arbre n-aire.

Programme 5.1 Nombre de nœuds et feuilles d'un arbre n-aire

```
let rec nb_feuilles = function
    | Feuille(_) -> 1
    | Nœud(_,liste_fils)
        -> it_list (fun n a -> n + (nb_feuilles a))
                   0
                   liste_fils ;;

let rec nb_nœuds = function
    | Feuille(_) -> 0
    | Nœud(_,liste_fils)
        -> it_list (fun n a -> n + (nb_nœuds a))
                   1
                   liste_fils ;;
```

En revanche, on définit de façon analogue la profondeur d'un nœud ou d'une feuille de l'arbre, ce qui fait l'objet du programme 5.2.

Programme 5.2 Profondeur d'un arbre n-aire

```
let rec profondeur = function
    | Feuille(_) -> 0
    | Nœud(_,liste_fils)
        -> 1 + it_list (fun M a -> max M (profondeur a))
                       0
                       liste_fils ;;
```

Nous avons utilisé ici la fonction `it_list`, qui fait partie de la bibliothèque standard de Caml. Son type est : `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`.
L'appel `it_list f a [ b1 ; b2 ; ... ; bn ]` s'évalue en

$$f(\ldots(f(f(a, b_1), b_2)\ldots), b_n)$$

5.1.4 Parcours d'un arbre n-aire

On définit très naturellement les parcours préfixe et suffixe d'un arbre n-aire, et on peut écrire facilement les fonctions Caml correspondantes, ce qui fait l'objet du programme 5.3 page suivante.
En revanche, il est bien évident que cela n'aurait pas de sens de parler de parcours infixe : où placer la racine d'un arbre à 3 fils ?

Programme 5.3 Parcours préfixe et suffixe d'un arbre n-aire

```
let rec parcours_préfixe = function
    | Feuille(f) -> [ F f ]
    | Nœud(n,liste_fils)
        -> it_list (fun l a -> l @ (parcours_préfixe a))
                   [N n]
                   liste_fils ;;

let rec parcours_suffixe = function
    | Feuille(f) -> [ F f ]
    | Nœud(n,liste_fils)
        -> (it_list (fun l a -> l @ (parcours_suffixe a))
                    []
                    liste_fils) @ [ N n ] ;;
```

Un autre problème apparaît au moment de reconstituer un arbre à partir de son parcours préfixe (ou suffixe, c'est le même problème). En effet, si on reprend l'arbre exemple de la figure 5.1 page 45, son parcours préfixe s'écrit :

$$n_1, n_2, f_1, f_2, f_3, n_3, n_4, f_4, f_5.$$

Mais on vérifie sans peine que c'est aussi le parcours de l'arbre représenté dans la figure 5.2.

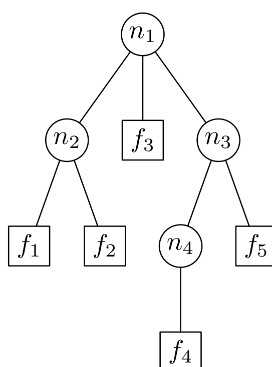


FIG. 5.2: Un autre arbre n-aire de même parcours préfixe

On peut néanmoins lever l'ambiguïté, à condition que tout nœud portant une même valeur n_k donnée ait le même nombre de fils. Ce nombre de fils s'appelle l'*arité* du nœud, et on suppose donc qu'il existe une fonction

```
arité : 'n -> int
```

Nous écrivons par exemple la reconstitution de l'arbre à partir de son parcours préfixe dans le programme 5.4 page suivante.

5.2 Expressions arithmétiques

5.2.1 Une définition

Nous allons maintenant donner une définition rigoureuse d'une expression arithmétique. Nous considèrerons des expressions construites à l'aide des opérations habituelles (addition, soustraction, multiplication, division et élévation à une puissance), des constantes réelles, des variables, et des fonctions (nous nous limiterons ici à l'exponentielle, son logarithme, et les sinus et cosinus).

Définition 5.2 (Expressions arithmétiques) On considère un ensemble $\mathcal{O}$ d'opérations à 2 arguments, par exemple $\mathcal{O} = \{+, -, \times, /, \uparrow\}$; un ensemble $\mathcal{C}$ de constantes, par exemple $\mathcal{C} = \mathbb{R}$; un ensemble $\mathcal{V}$ de variables,

Programme 5.4 Reconstitution d'un arbre n-aire à partir de son parcours préfixe

```
let recompose_préfixe_naire arité l =
    let rec recompose = function
        | (F f) :: reste -> Feuille(f), reste
        | (N n) :: reste
            -> let liste_fils,reste = cherche_fils (arité n) reste
               in
               Nœud(n,liste_fils),reste
        | [] -> failwith "Description préfixe incorrecte"
    and cherche_fils nb_fils reste =
        if nb_fils = 0 then [],reste
        else
        let fils,reste = recompose reste
        in
        let autres_fils,reste = cherche_fils (nb_fils-1) reste
        in
        fils :: autres_fils,reste
    in
    match recompose l with
        | a,[] -> a
        | _ -> failwith "Description préfixe incorrecte" ;;
```

par exemple $\mathcal{V} = \{A, B, \ldots, X, Y, Z\}$; et un ensemble $\mathcal{F}$ de fonctions à 1 argument, par exemple $\mathcal{F} = \{\sin, \cos, \ln, \exp\}$. On définit l'ensemble $\mathcal{A}$ des expressions arithmétiques correspondantes de façon récursive :

— toute constante est une expression arithmétique : $\mathcal{C} \subset \mathcal{A}$;

— toute variable est une expression arithmétique : $\mathcal{V} \subset \mathcal{A}$;

— si $c \in \mathcal{O}$ et si u et v sont des expressions arithmétiques, alors (ucv) est aussi une expression arithmétique ;

— si $f \in \mathcal{F}$ et si u est une expression arithmétique, alors $f(u)$ est aussi une expression arithmétique.

5.2.2 Syntaxe concrète et syntaxe abstraite

En Caml, on introduit ainsi très naturellement le type suivant :

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

Ainsi l'expression $2 + \sin^2(3 + X)$ est l'écriture mathématique habituelle de l'expression arithmétique $(2 + (\sin((3 + X)) \uparrow 2))$. Notons qu'on a l'habitude en mathématiques de se dispenser de certaines parenthèses qu'impose pourtant notre définition.
Notre expression correspond à une valeur Caml de type `(float,char,char,fonction) expression`, où le type `fonction` serait défini par

```
type fonction = Sin | Cos | Exp | Ln ;;
```

La valeur Caml s'écrirait :

```
Terme(Constante(2.0),
      `+`,
      Terme(Application(Sin,Terme(Constante(3.0),`+`,Variable(`X`))),
            `^`,
            Constante(2.0))) ;;
```

Comme on le constate aisément, passant de la dénotation mathématique habituelle (la syntaxe concrète), à la notation Caml (la syntaxe abstraite), on complique grandement l'écriture des expressions, mais on a réalisé l'essentiel de leur analyse, ce qui permet ensuite des manipulations bien plus faciles.

5.2.3 Expressions et arbres

Arbre d'une expression arithmétique

On peut naturellement associer à une expression arithmétique un arbre n-aire : l'ensemble des valeurs des feuilles est $\mathcal{C} \cup \mathcal{V}$, l'ensemble des valeurs des nœuds est $\mathcal{O} \cup \mathcal{F}$. Ceci s'exprime en Caml de la façon suivante :

```
type nœud = F of fonction | O of char ;;
type feuille = N of float | V of char ;;
```

Notons que nous sommes ici dans le cas favorable où il existe une fonction d'arité (voir la section précédente), ce qui permet de mettre bijectivement en relation les arbres d'expressions et leurs parcours préfixes ou suffixes.
On peut facilement écrire les fonctions de conversion d'une expression en arbre et inversement, ce qui fait l'objet du programme 5.5.

Programme 5.5 Conversions arbres d'expression/expressions arithmétiques

```
let rec arbre_d'expression = function
    | Constante x -> Feuille (N x)
    | Variable v  -> Feuille (V v)
    | Terme(a,o,b) ->   let g = arbre_d'expression a
                        and d = arbre_d'expression b
                        in
                        Nœud(O o, [ g ; d ])
    | Application(f,a) -> Nœud(F f, [ (arbre_d'expression a) ]) ;;

let rec expression_d'arbre = function
    | Feuille (N x) -> Constante(x)
    | Feuille (V x) -> Variable(x)
    | Nœud(O o, [ e ; f ])
        -> Terme(expression_d'arbre e,o,expression_d'arbre f)
    | Nœud(F f, [ e ]) -> Application(f,expression_d'arbre e)
    | _ -> failwith "Ce n'est pas un arbre d'expression" ;;
```

Évaluation et impression d'une expression arithmétique

On a déjà vu en première année comment évaluer une expression donnée par son écriture suffixe. On aura remarqué combien le programme écrit alors ressemble à celui qui permet la reconstitution d'un arbre à partir de son parcours suffixe, et on n'aura pas eu la naïveté de s'en étonner.
Plus généralement, l'arbre d'une expression permet facilement d'évaluer l'expression ou encore de l'imprimer en écriture préfixe, infixe ou suffixe.
Occupons nous pour commencer de l'évaluation. Il faut ici simplement ajouter à la sémantique décrite en première année ce qui concerne les variables. En bref, il faut expliquer ce qu'on veut obtenir dans l'évaluation d'une variable. Pour cela, il nous faut définir la notion d'environnement d'une expression.

Définition 5.3 (Environnement d'évaluation) On appelle environnement d'évaluation des variables d'une expression arithmétique sur les ensembles $\mathcal{C}$, $\mathcal{V}$, $\mathcal{F}$ et $\mathcal{O}$ toute application ψ de $\mathcal{V}$ dans $\mathcal{C}$.

En Caml, un environnement d'évaluation des variables sera simplement une liste associative, c'est-à-dire une liste de couples variables/valeurs de la forme (var_i, val_i). La bibliothèque Caml fournit la fonction `assoc` qui permet de chercher la valeur associée à une variable, et qu'on pourrait d'ailleurs écrire soi-même comme suit.

```
let rec assoc v env = match env with
    | [] -> failwith "Variable absente de l'environnement"
    | (var,val) :: q -> if v = var then val else assoc v q ;;
```

Définition 5.4 (Sémantique d'une expression arithmétique) Soit $\mathcal{A}$ l'ensemble des expressions arithmétiques sur les ensembles $\mathcal{C}$, $\mathcal{V}$, $\mathcal{F}$ et $\mathcal{O}$. Soit ψ un environnement d'évaluation des variables. On suppose qu'est

associée à tout $f \in \mathcal{F}$ une fonction $\hat{f}$ de $\mathcal{C}$ dans lui-même, et à tout $o \in \mathcal{O}$ une fonction $\hat{o}$ de $\mathcal{C}^2$ dans $\mathcal{C}$. On définit alors récursivement l'évaluation d'une expression :

- si $c \in \mathcal{C}$, alors $eval(c) = c$;
- si $v \in \mathcal{V}$, alors $eval(v) = \psi(v)$;
- si $f \in \mathcal{F}$, si u est une expression, alors $eval(f(u)) = \hat{f}(eval(u))$;
- si $o \in \mathcal{O}$, si u et v sont deux expressions, alors $eval((uov)) = \hat{o}(eval(u), eval(v))$.

L'évaluation est alors réalisée par le programme 5.6.

Programme 5.6 Évaluation des expressions arithmétiques

```
let rec évaluation environnement = function
    | Constante x -> x
    | Variable v -> assoc v environnement
    | Terme(a,o,b) -> let x = évaluation environnement a
                      and y = évaluation environnement b
                      in
                      ( match o with
                            | `+` -> x +. y
                            | `-` -> x -. y
                            | `/` -> x /. y
                            | `*` -> x *. y
                            | `^` -> power x y
                            | _ -> failwith "Opération inconnue" )
    | Application(f,a) -> let x = évaluation environnement a
                          in
                          ( match f with
                                | Sin -> sin x
                                | Cos -> cos x
                                | Exp -> exp x
                                | Ln  -> log x ) ;;
```

Les impressions préfixe et suffixe ne posent pas de problème particulier. Rappelons que l'existence d'une fonction d'arité permet de se dispenser de tout parenthésage. On trouvera ces fonctions dans le programme 5.7 page suivante.
Pour l'impression infixe, en revanche, il est indispensable d'écrire des parenthèses. La suppression des parenthèses qu'on n'écrit pas habituellement est un problème difficile, qui nécessite qu'on discute des priorités des opérateurs. C'est une difficulté analogue qu'on rencontre quand on veut programmer le passage de l'écriture infixe à la syntaxe abstraite des expressions. Nous nous contenterons ici d'écrire toutes les parenthèses, y compris celles qui sont superflues, ce qui est fait dans le programme 5.8 page 52.

5.3 Dérivation formelle

5.3.1 Dérivation guidée par la structure d'arbre

On peut également utiliser la structure qu'on a introduite ici des expressions arithmétiques pour écrire un programme de dérivation formelle, utilisant les formules classiques de dérivation :

$$\begin{array}{ccc} (u+v)' = u' + v' & (u-v)' = u' - v' & (uv)' = u'v + uv' \\ (u/v)' = u'/v - uv'/v^2 & (u^v)' = u^v v' \ln u + u^{(v-1)} u' v & (\ln u)' = u'/u \\ (\exp u)' = u' \exp u & (\sin u)' = u' \cos u & (\cos u)' = -u' \sin u \end{array}$$

Traduisant mot à mot ces règles, on obtient facilement le programme 5.9 page 52.
Reprenant l'expression $e = 2 + \sin^2(3 + X)$, on trouve comme dérivée par rapport à X l'expression $0 + \sin^2(3+X) \times 0 \times \ln \sin(3+X) + \sin^{(2-1)}(3+X) \times 2 \times (0+1) \times \cos(3+X)$. La dérivée par rapport à Y donne l'expression $0 + \sin^2(3+X) \times 0 \times \ln \sin(3+X) + \sin^{(2-1)}(3+X) \times 2 \times (0+0) \times \cos(3+X)$. Il va de soi que ces expressions ne correspondent pas tout à fait à notre attente, et il convient de les simplifier au moins un peu, ce qui fait l'objet de ce qui suit.

Programme 5.7 Impressions préfixe et suffixe des expressions

```
let impression_préfixe expr =
    let a = arbre_d'expression expr
    in
    let rec imprime = function
        | Feuille(N x) -> print_float x ; print_char ‘ ‘
        | Feuille(V x) -> print_char x ; print_char ‘ ‘
        | Nœud(O o, [ e ; f ])
            -> print_char o ; print_char ‘ ‘ ; imprime e ; imprime f
        | Nœud(F f, [ e ])
            ->  ( match f with
                    | Sin -> print_string "sin"
                    | Cos -> print_string "cos"
                    | Exp -> print_string "exp"
                    | Ln  -> print_string "ln" ) ;
                print_char ‘ ‘ ; imprime e
        | _ -> failwith "Erreur impossible"
    in
    imprime a ;;

let impression_suffixe expr =
    let a = arbre_d'expression expr
    in
    let rec imprime = function
        | Feuille(N x) -> print_float x ; print_char ‘ ‘
        | Feuille(V x) -> print_char x ; print_char ‘ ‘
        | Nœud(O o, [ e ; f ])
            -> imprime e ; imprime f ; print_char o ; print_char ‘ ‘
        | Nœud(F f, [ e ])
            ->  imprime e ;
                ( match f with
                    | Sin -> print_string "sin"
                    | Cos -> print_string "cos"
                    | Exp -> print_string "exp"
                    | Ln  -> print_string "ln" ) ;
                print_char ‘ ‘
        | _ -> failwith "Erreur impossible"
    in
    imprime a ;;
```

Programme 5.8 Impression infixe des expressions

```
let impression_infixe expr =
    let a = arbre_d'expression expr
    in
    let rec imprime = function
        | Feuille(N x) -> print_float x
        | Feuille(V x) -> print_char x
        | Nœud(O o, [ e ; f ])
            ->  print_char `(` ;
                imprime e ; print_char ` ` ;
                print_char o ;
                print_char ` ` ; imprime f ;
                print_char `)`
       | Nœud(O o, [ e ; f ])
            ->  print_char `(` ;
                imprime e ; print_char o ; imprime f ;
                print_char `)`
        | Nœud(F f, [ e ])
            ->  ( match f with
                    | Sin -> print_string "sin"
                    | Cos -> print_string "cos"
                    | Exp -> print_string "exp"
                    | Ln  -> print_string "ln" ) ;
                print_char `(` ; imprime e ; print_char `)`
        | _ -> failwith "Erreur impossible"
    in
    imprime a ;;
```

Programme 5.9 Dérivation formelle des expressions arithmétiques

```
let rec dérivation expr x =
    match expr with
        | Constante(_) -> Constante(0.0)
        | Variable(v) ->  Constante(if v = x then 1.0 else 0.0)
        | Terme(u,o,v)
            ->  let u' = dérivation u x
                and v' = dérivation v x
                in
                ( match o with
                    | `+` -> Terme(u',`+`,v')
                    | `-` -> Terme(u',`-`,v')
                    | `*` -> Terme(Terme(u',`*`,v),`+`,Terme(u,`*`,v'))
                    | `/` -> Terme( Terme(u',`/`,v),
                                    `-`,
                                    Terme(Terme(u,`*`,v'),
                                          `/`,
                                          Terme(v,`^`,Constante(2.0))))
                    | `^` -> Terme(expr,`*`,
                                Terme(Terme(v',`*`,Application(Ln,u)),
                                      `+`,
                                      Terme(v,`*`,Terme(u,`/`,u')))) )
        | Application(f,u)
            ->  let u' = dérivation u x
                in
                match f with
                    | Sin -> Terme(u',`*`,Application(Cos,u))
                    | Cos -> Terme(u',`*`,
                                Terme(Constante(-1.0),
                                      `*`,
                                      Application(Sin,u)))
                    | Exp -> Terme(u',`*`,expr)
                    | Ln ->  Terme(u',`/`,u) ;;
```

5.3.2 Simplification : une première approche

Nous nous contentons ici d'une simplification élémentaire des expressions. Une simplification plus complète des expressions poserait de redoutables problèmes de définition et nous n'en parlerons pas davantage. On obtient ainsi le programme 5.10 page suivante.
Et voilà le résultat, sur le même exemple que précédemment. On vérifiera sur cette session Caml que toutes les simplifications qu'on pouvait espérer ont bien été réalisées. En revanche, il ne fallait pas compter sur la simplification de $2 \sin a \cos a$ en $\sin(2a)$, bien sûr. On pourrait quand même essayer de reconnaître dans une différence l'égalité des deux termes, ou dans un quotient ou un produit l'égalité des facteurs, ce qui permettrait des simplifications supplémentaires.

Programme 5.10 Simplification des expressions algébriques

```
let rec simplification = function
    | Terme(u,`+`,v)
        -> let u,v = (simplification u), (simplification v)
           in
           ( match u,v with
                    | Constante(0.0),_ -> v
                    | _,Constante(0.0) -> u
                    | Constante(x),Constante(y) -> Constante(x +. y)
                    | _,_ -> Terme(u,`+`,v) )
    | Terme(u,`-`,v)
        -> let u,v = (simplification u), (simplification v)
           in
           ( match u,v with
                    | _,Constante(0.0) -> u
                    | Constante(x),Constante(y) -> Constante(x -. y)
                    | _,_ -> Terme(u,`-`,v) )
    | Terme(u,`*`,v)
        -> let u,v = (simplification u), (simplification v)
           in
           ( match u,v with
                    | Constante(0.0),_ -> Constante(0.0)
                    | _,Constante(0.0) -> Constante(0.0)
                    | Constante(1.0),_ -> v
                    | _,Constante(1.0) -> u
                    | Constante(x),Constante(y) -> Constante(x *. y)
                    | _,_ -> Terme(u,`*`,v) )
    | Terme(u,`/`,v)
        -> let u,v = (simplification u), (simplification v)
           in
           ( match u,v with
                    | Constante(0.0),_ -> Constante(0.0)
                    | _,Constante(1.0) -> u
                    | Constante(x),Constante(y) -> Constante(x /. y)
                    | _,_ -> Terme(u,`/`,v) )
    | Terme(u,`^`,v)
        -> let u,v = (simplification u), (simplification v)
           in
           ( match u,v with
                    | Constante(0.0),_ -> Constante(0.0)
                    | Constante(1.0),_ -> Constante(1.0)
                    | _,Constante(1.0) -> u
                    | Constante(x),Constante(y) -> Constante(power x y)
                    | _,Constante(-1.0) -> Terme(Constante(1.0),`/`,u)
                    | _,_ -> Terme(u,`^`,v) )
    | Application(Exp,u)
        -> let u = simplification u
           in
           ( match u with
                    | Constante(x) -> Constante(exp(x))
                    | _ -> Application(Exp,u) )
    | Application(Ln,u)
        -> let u = simplification u
           in
           ( match u with
                    | Constante(x) -> Constante(log(x))
                    | _ -> Application(Ln,u) )
    | Application(Sin,u)
        -> let u = simplification u
           in
           ( match u with
                    | Constante(x) -> Constante(sin(x))
                    | _ -> Application(Sin,u) )
    | Application(Cos,u)
        -> let u = simplification u
           in
           ( match u with
                    | Constante(x) -> Constante(cos(x))
                    | _ -> Application(Cos,u) )
    | expr -> expr ;;
```

Quoi qu'il en soit voici ce qu'on obtient :

```
#impression_infixe (simplification (dérivation expr `X`) ) ;;
(sin((3.0 + X)) * (2.0 * cos((3.0 + X))))- : unit = ()
#impression_infixe (simplification (dérivation expr `Y`) ) ;;
0.0- : unit = ()
```

5.4 Exercices pour le chapitre 5

Exercice 5.1 Reconstitution d'un arbre n-aire à partir du parcours suffixe

Écrire le programme qui permet de reconstituer un arbre n-aire à partir de son parcours suffixe, supposant connue la fonction d'arité.

Exercice 5.2 Impression infixe des expressions

Écrire un programme Caml qui gère une impression infixe des expressions sans parenthèses inutiles.
On rappelle que $\uparrow$ est prioritaire par rapport à $\times$ et $/$, eux-mêmes prioritaires par rapport à $+$ et $-$.

Deuxième partie

Automates

Chapitre 6

Automates finis déterministes ou non déterministes

6.1 Automates finis déterministes

6.1.1 Présentation informelle

Un automate est un système qui réagit à des *stimuli*. Ces stimuli peuvent être représentés de différentes manières. À chaque stimulus, l'automate peut se bloquer, ou bien passer de l'état où il était avant réception à un nouvel état, voire rester dans le même état.
Nous nous intéresserons dans ce cours tout particulièrement aux automates qui réagissent aux caractères d'un alphabet. Si donc on considère un automate dans un certain état initial, on lui fait *lire* une chaîne de caractères, c'est-à-dire qu'on le fait réagir successivement aux différents caractères qui composent la chaîne. Le processus peut s'interrompre avant la fin de la chaîne : ou bien l'automate s'est bloqué, ou bien tous les caractères sont lus, et on peut dire que l'automate est passé de son état initial à un nouvel état, qui est évidemment fonction de la chaîne lue.
En général, on numérote les états possibles de l'automate.
L'automate est dit fini s'il n'a qu'un nombre fini d'états et si l'alphabet (l'ensemble des stimuli) est lui aussi fini.
Les transitions d'un état à l'autre, sous l'effet des stimuli, sont représentées par des flèches entre les états étiquetées par le nom du stimulus.
La figure 6.1 représente un automate fini.

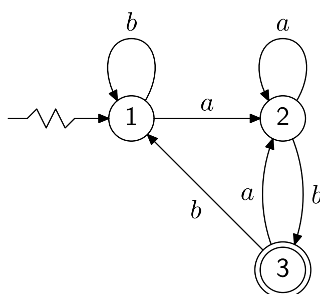


Fig. 6.1: Un premier exemple d'automate fini déterministe

Cet automate a trois états, notés 1, 2 et 3.
1 est l'état initial (désigné par le petit éclair).
3 est un état dit état d'acceptation ou état final (il peut y en avoir plusieurs), puisqu'il est entouré de deux cercles.

Partant de l'état 1, la lecture de la chaîne *abba* nous fait passer successivement aux états 2, 3, 1 et enfin 2. Comme ce n'est pas un état final, on dit que la chaîne n'est pas reconnue par l'automate.
Si on lisait la chaîne *abca*, l'automate se bloquerait à l'arrivée du c dans l'état 3. On dit que la chaîne n'est pas reconnue par l'automate à cause du blocage, bien qu'on soit dans un état final.
Enfin, la chaîne *bbaaab* est reconnue par cet automate, et on peut montrer que l'ensemble des chaînes reconnues est l'ensemble de toutes les chaînes sur les deux lettres a et b qui finissent exactement par ab : l'état 2 correspond au moment où on a lu un a et où on attend un b pour passer à l'état final 3 ; l'état 1 correspond à ce qui arrive quand après avoir bien lu ab, on tombe sur un b, et que tout est donc à refaire...

6.1.2 Présentation mathématique

De façon plus rigoureuse on peut donner la

Définition 6.1 (Automates finis déterministes) Soit $\mathcal{A}$ un alphabet fini. Un automate fini déterministe (en abrégé *afd*) est un quadruplet $(Q, q_0, \mathcal{F}, \varphi)$ où Q est un ensemble fini d'éléments appelés états, q_0 est un élément de Q, appelé état initial, $\mathcal{F}$ est une partie non vide de Q, dont les éléments sont appelés états finals, et où φ est une fonction de $Q \times \mathcal{A}$ dans Q.

On notera que φ est une fonction, et non une application : elle n'est pas nécessairement définie pour tout couple $(q, c) \in Q \times \mathcal{A}$.
L'automate étant fixé, si pour $c \in \mathcal{A}$ et pour deux états q et q' on a $\varphi(q, c) = q'$, on dira qu'il existe une transition de q vers q' sur c et on notera $q \xrightarrow{c} q'$.
Reprenant l'automate représenté dans la figure 6.1 page précédente, on peut choisir $\mathcal{A} = \{a, b\}$, $Q = \{1, 2, 3\}$, $q_0 = 1$, $\mathcal{F} = \{3\}$, et φ est définie par le tableau suivant :

q	c	$\varphi(q,c)$
1	a	2
1	b	1
2	a	2
2	b	3
3	a	2
3	b	1

6.1.3 Transitions et calculs

Définitions

On a dit qu'on notait $q \xrightarrow{c} q'$ si $\varphi(q, c) = q'$: il s'agit là d'une transition toute banale. On peut généraliser à ce qu'on appellera un calcul.

Définition 6.2 (Mots) Soit $\mathcal{A}$ un alphabet fini. Un mot est une suite finie éventuellement vide d'éléments de $\mathcal{A}$. Le mot vide est noté ε. Un mot w de taille n est constitué des éléments successifs w_0, w_1, ..., w_{n-1}. L'ensemble des mots sur $\mathcal{A}$ est noté $\mathcal{A}^*$.

Définition 6.3 (Calculs d'un afd) Soit $\mathcal{A}$ un alphabet fini et $(Q, q_0, \mathcal{F}, \varphi)$ un *afd* sur cet alphabet. Soit q et q' deux états de l'automate et w un mot non vide de longueur n. On dira qu'un calcul de l'automate fait passer de q à q' à la lecture de w si il existe $n-1$ états q_1, ..., q_{n-1} tels que

$$q \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots \xrightarrow{w_{n-2}} q_{n-1} \xrightarrow{w_{n-1}} q'.$$

On notera ce calcul : $q \xrightarrow{w} q'$. Par convention on a $q \xrightarrow{\varepsilon} q$ pour tout état $q \in Q$.

Propriétés

On démontre aisément les deux propriétés suivantes :

$$\forall (w_1, w_2) \in \mathcal{A}^{*2}, \forall (q_1, q_2, q_3) \in Q^3, (q_1 \xrightarrow{w_1} q_2 \text{ et } q_2 \xrightarrow{w_2} q_3) \Rightarrow q_1 \xrightarrow{w_1.w_2} q_3,$$

où on a noté $w_1.w_2$ le mot résultat de la concaténation des mots w_1 et w_2 ;

$$\forall w \in \mathcal{A}^*, \forall (q, q', q'') \in Q^3, (q \xrightarrow{w} q' \text{ et } q \xrightarrow{w} q'') \Rightarrow q' = q'',$$

ce qui traduit le déterminisme de l'automate.

6.1.4 Langage reconnu par un automate fini déterministe

On est en mesure de définir rigoureusement les mots reconnus par un *afd*.

Définition 6.4 (Mots acceptés par un afd) Étant donné un alphabet $\mathcal{A}$ et un *afd* $(Q, q_0, \mathcal{F}, \varphi)$, un mot $w \in \mathcal{A}^*$ est dit reconnu ou accepté par l'automate si il existe $q \in \mathcal{F}$ tel que $q_0 \xrightarrow{w} q$. Dans les autres cas, c'est-à-dire si l'automate se bloque pendant la lecture de w ou s'il s'arrête dans un état non final, on dit que le mot est rejeté par l'automate, ou encore qu'il n'est pas reconnu.

Le langage d'un *afd* α est l'ensemble des mots qu'il reconnaît : on le notera $L(\alpha)$.
Notons d'ores et déjà que deux automates distincts peuvent définir le même langage. On vérifiera par exemple que les automates de la figure 6.2 définissent le même langage que celui de la figure 6.1 page 59. Le premier d'entre eux comporte en effet un état (4) inaccessible, c'est-à-dire un état q tel que pour aucun mot w on n'ait $q_0 \xrightarrow{w} q$. Le second ne comporte pas d'état inaccessible, mais définit pourtant le même langage.

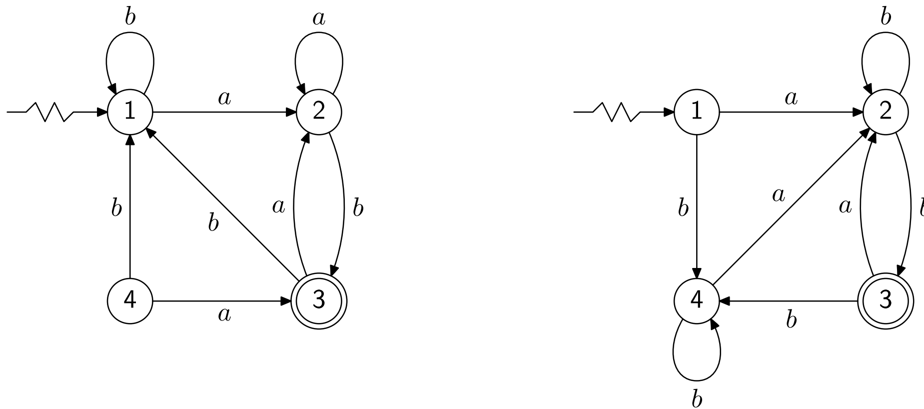


Fig. 6.2: Deux automates pour un même langage

Un *afd* est dit minimal si tout *afd* qui définit le même langage a au moins autant d'états. Il va de soi qu'on préfère en général manipuler des *afd* minimaux...

6.2 Implémentation en Caml d'un *afd*

Pour représenter les *afd* on pourra définir le type des états de la façon suivante :

```
type état_de_afd =
    { mutable est_terminal : bool ;
      transitions : arrivée vect }
and arrivée = Blocage | OK of état_de_afd ;;
```

Si `q` est un état, `q.est_terminal` dit si oui ou non $q \in \mathcal{F}$, et `q.transitions` est un vecteur de 256 valeurs, correspondant aux 256 caractères dans le codage ASCII. `q.transitions.(i)` vaudra `Blocage` s'il n'y a pas de transition à partir de q sur le caractère c de code ASCII i, ou bien `OK(q')` si $\varphi(q,c) = q'$.
On programme aisément la reconnaissance d'une chaîne, ce qui fait l'objet du programme 6.1.

Programme 6.1 Reconnaissance d'une chaîne par un *afd*

```
exception Échec ;;

type état_de_afd =
    { mutable est_terminal : bool ;
      transitions : arrivée vect }
and arrivée = Blocage | OK of état_de_afd ;;

let lit_transition_afd q1 c =
    match q1.transitions.(int_of_char c) with
        | Blocage -> raise Échec
        | OK(q2) -> q2 ;;

let écrit_transition_afd q1 c q2 =
    q1.transitions.(int_of_char c) <- OK(q2) ;;

let reconnaît état_de_départ chaîne =
    let rec transite état i n =
        if i = n then état.est_terminal
        else transite (lit_transition_afd état chaîne.[i]) (i+1) n
    in
    try
        transite état_de_départ 0 (string_length chaîne)
    with _ -> false ;;
```

Il est clair que cet algorithme de reconnaissance tourne en $O(n)$, où n est la longueur de la chaîne de caractères testée.
Indiquons au passage comment, par exemple, on entre la description de l'*afd* de la figure 6.1 page 59.

```
let q1 = { est_terminal = false ; transitions = make_vect 256 Blocage }
and q2 = { est_terminal = false ; transitions = make_vect 256 Blocage }
and q3 = { est_terminal = true ; transitions = make_vect 256 Blocage } ;;

écrit_transition_afd q1 `a` q2 ;;
écrit_transition_afd q1 `b` q1 ;;
écrit_transition_afd q2 `a` q2 ;;
écrit_transition_afd q2 `b` q3 ;;
écrit_transition_afd q3 `a` q2 ;;
écrit_transition_afd q3 `b` q1 ;;
```

L'appel `reconnaît q0 chaîne` renvoie `true` si et seulement si la chaîne est reconnue par l'automate dont l'état initial est `q0`.

6.3 Automates finis non déterministes

6.3.1 Présentation informelle

On trouvera dans la figure 6.3 page suivante un premier exemple d'automate fini non déterministe (*afnd* en abrégé).
On remarque quelques différences avec les *afd*, à savoir essentiellement :

- certaines flèches sont étiquetées par la lettre ε : il s'agit de ce qu'on appellera donc des ε-transitions, et l'automate passera tout seul en cas de besoin d'un état à un autre état par la voie d'une telle transition ;
- on peut trouver plusieurs flèches issues d'un même état et étiquetées de la même lettre : c'est ce qui fait toute l'ambiguïté d'un *afnd*, et explique qu'on parle de non-déterminisme.

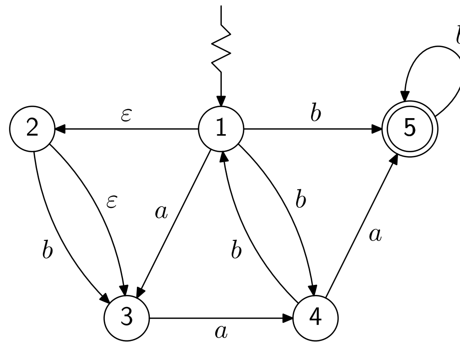


FIG. 6.3: Un automate fini non déterministe

On vérifiera sur cet exemple que les mots b, bbb ou encore baa sont reconnus par cet automate : les parcours d'états correspondants sont 15, 1415 ou 1555, 12345. On aura noté qu'il n'y a plus unicité...

6.3.2 Définition mathématique

Définition d'un *afnd*

Définition 6.5 (Automates finis non déterministes) Soit $\mathcal{A}$ un alphabet fini. Un automate fini non déterministe (en abrégé *afnd*) est un quintuplet $(Q, q_0, \mathcal{F}, \varphi, \psi)$ où Q est un ensemble fini d'éléments appelés états, q_0 est un élément de Q, appelé état initial, $\mathcal{F}$ est une partie non vide de Q, dont les éléments sont appelés états finals, et où φ (resp. ψ) est une application de $Q \times \mathcal{A}$ (resp. Q) dans $\mathcal{P}(Q)$, ensemble des parties de Q.

On aura noté qu'une transition à partir d'un état q sur le caractère c se calcule par φ : $\varphi(q, c)$ est l'ensemble des états auxquels on peut aboutir par cette transition, et $\emptyset$ dans le cas où il n'y a pas de transition à partir de q sur c.
D'une façon analogue, $\psi(q)$ désigne l'ensemble des états atteints à partir de q par une ε-transition.

Certains auteurs s'autorisent, pour les *afnd*, plusieurs états initiaux, q_1, q_2, ..., q_k. Il suffit, pour se ramener à notre définition, d'ajouter un unique état initial q_0 et des ϵ-transitions de q_0 vers chacun des q_i, $1 \leq i \leq k$. C'est pourquoi nous avons choisi de considérer systématiquement qu'il n'y a qu'un état initial pour chacun de nos automates.

Fermeture d'un état

Pour définir proprement les calculs qu'effectue un *afnd* il nous faut maintenant définir ce qu'on appelle la fermeture d'un état par ε-transition.

Définition 6.6 (Fermeture par ε-transition) Soit $\mathcal{A}$ un alphabet fini et $(Q, q_0, \mathcal{F}, \varphi, \psi)$ un *afnd* sur cet alphabet. Pour tout état $q \in Q$ on définit sa fermeture par ε-transition et on notera $\kappa(q)$ la plus petite partie de Q contenant q et vérifiant :

$$\bigcup_{q' \in \kappa(q)} \psi(q') \subset \kappa(q).$$

De façon évidente, on montre le

Théorème 6.1 (Calcul des fermetures)
Soit $\mathcal{A}$ un alphabet fini et $(Q, q_0, \mathcal{F}, \varphi, \psi)$ un *afnd* sur cet alphabet. Soit q un état fixé. On définit récursivement les parties (K_n) de Q en posant $K_0 = \{q\}$, et, pour $n \geq 1$, $K_n = K_{n-1} \cup \bigcup_{q' \in K_{n-1}} \psi(q')$. La suite (K_n) est croissante et stationnaire (pour l'inclusion), de limite égale à $\kappa(q)$.

✧ La suite (K_n) est évidemment croissante, et, comme Q est fini, elle est stationnaire. Soit K sa limite. Bien sûr $q \in K$.
Soit $q' \in K$. Il existe un indice $p \geq 0$ tel que $q' \in K_p$. Mais alors $\psi(q') \subset K_{p+1} \subset K$, et K vérifie donc bien la propriété requise par $\kappa(q)$, ce qui montre que $\kappa(q) \subset K$.
Réciproquement, on montre par récurrence sur n que les K_n sont inclus dans $\kappa(q)$. ✦

Intuitivement, $\kappa(q)$ désigne l'ensemble de tous les états (dont q) qui peuvent être atteints à partir de q après un nombre fini quelconque de ε-transitions.

Calculs d'un *afnd*

On considère ici un *afnd* $(Q, q_0, \mathcal{F}, \varphi, \psi)$ sur un alphabet $\mathcal{A}$.
Étant donnés deux états q_1 et q_2 et un caractère c, on notera $q_1 \xrightarrow{\varepsilon} q_2$ quand $q_2 \in \kappa(q_1)$, et on notera $q_1 \xrightarrow{c} q_2$ quand il existe $q_1' \in \kappa(q_1)$ et un état $q_2' \in \varphi(q_1', c)$ tels que $q_2 \in \kappa(q_2')$. C'est dire que dans les transitions, on passera gratuitement par autant de ε-transitions qu'il est nécessaire.
Avec ces définitions, on peut reprendre la même définition que pour les *afd* de la notation $q_1 \xrightarrow{w} q_2$. Mais cette fois l'automate n'étant plus déterministe, il n'y a pas unicité de l'état d'arrivée q_2.
Ici encore, un mot w est dit accepté par l'automate s'il existe un état final $q \in \mathcal{F}$ tel que $q_0 \xrightarrow{w} q$. Le langage de l'automate est toujours l'ensemble des mots acceptés par lui.

Comparaison des *afd* et des *afnd*

L'intérêt des automates non déterministes est qu'il est beaucoup plus facile en général de concevoir un *afnd* qu'un *afd* pour un langage donné, ce qu'on étudiera de plus près dans le chapitre suivant. L'exemple des mots sur $\{a, b\}$ qui se terminent par ab est explicite à cet égard. Autant il n'est pas évident que l'*afd* de la figure 6.1 page 59 convient, autant il est clair que l'*afnd* de la figure 6.4 est une solution.

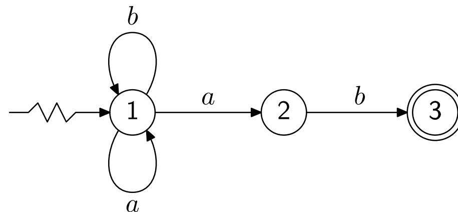


FIG. 6.4: Un *afnd* pour les mots qui finissent par *ab*

En revanche il est beaucoup moins simple de programmer le fonctionnement d'un *afnd*, et surtout, l'algorithme obtenu est beaucoup moins efficace. C'est ce qu'on va voir maintenant.

6.4 Implémentation en Caml d'un *afnd*

D'une façon analogue à ce qu'on a pu faire pour un *afd*, on est amené à définir le type suivant pour les états d'un *afnd* :

```
type état_de_afnd =
    { mutable est_final : bool ;
      mutable table : état_de_afnd list vect ;
      mutable table_epsilon : état_de_afnd list ;
      numéro : int } ;;
```

On a ajouté un numéro à chaque état, ce qui nous servira plus tard. En outre cette fois si transition il y a sur un caractère donné, on fournit la liste des états accessibles par cette transition. De même y a-t-il une liste des états accessibles par ε-transitions.
La programmation de la reconnaissance, comme nous l'avons déjà dit, est plus difficile. Nous utiliserons quelques fonctions auxiliaires simples qui font l'objet du programme 6.2 page ci-contre.

Programme 6.2 Quelques fonctions utiles

```
let rec it_list f a l = match l with
    | [] -> a
    | t :: q -> it_list f (f a t) q ;;

let rec list_it f l a = match l with
    | [] -> a
    | t :: q -> f t (list_it q a) ;;

let rec exists prédicat l = match l with
    | [] -> false
    | t :: q -> (prédicat t) || (exists prédicat q) ;;

let mem_état q ql =
    let n = q.numéro
    in
    exists (function x -> x.numéro = n) ql ;;

let rec union_états ql1 ql2 = match ql2 with
    | [] -> ql1
    | q :: r -> let ql = union_états ql1 r
                in
                if mem_état q ql then ql
                else q :: ql ;;

let union_listes_états ll = it_list union_états [] ll ;;
```

Nous avons écrit les fonctions `mem_état` et `union_états` qui testent l'appartenance d'un état à une liste d'états et calculent la réunion de deux listes d'états sans utiliser la fonction de comparaison standard de Caml (=) qui pourrait boucler sur les structures récursives éventuellement cycliques que sont les états d'un automate. C'est pour cela que nous avons utilisé la numérotation des états.

Notons toutefois que les fonctions `it_list`, `list_it`, `exists` et `map` sont prédéfinies en Caml.
Précisons que `it_list f a [ x0 ; x1 ; ... ; xk ]` s'évalue en $f(\ldots(f(fax_0)x_1)\ldots)x_k$, alors que `list_it f [ x0 ; x1 ; ... ; xk ] a` s'évalue en $fx_0(fx_1(\ldots(fx_ka)\ldots))$.
Maintenant, on peut écrire la reconnaissance, qui utilise bien sûr une fonction de calcul des fermetures par ε-transitions.

Nous avons écrit pour cela une fonction de recherche d'un point fixe. `pt_fixe` attend deux arguments : une fonction `f : 'a -> 'a list` et une liste `l : 'a list` et calcule la plus petite liste `l'` qui contient `l` et les images par `f` de chacun des éléments de `l'`.

Cet algorithme de reconnaissance prend en argument la liste des états de départ (le plus souvent ce sera `[ q0 ]`) et la chaîne à reconnaître.
Remarquons que sa complexité est sans commune mesure avec l'algorithme utilisé pour les automates déterministes.
Indiquons pour finir comment, par exemple, on entre l'automate de la figure 6.3 page 63 en Caml :

```
let q1 = { numéro = 1 ; est_final = false ;
           table = make_vect 256 [] ; table_epsilon = [] }
and q2 = { numéro = 2 ; est_final = false ;
           table = make_vect 256 [] ; table_epsilon = [] }
and q3 = { numéro = 3 ; est_final = false ;
           table = make_vect 256 [] ; table_epsilon = [] }
and q4 = { numéro = 4 ; est_final = false ;
           table = make_vect 256 [] ; table_epsilon = [] }
and q5 = { numéro = 5 ; est_final = true ;
           table = make_vect 256 [] ; table_epsilon = [] } ;;

q1.table_epsilon <- [ q2 ] ;
q2.table_epsilon <- [ q3 ] ;
```

Programme 6.3 Reconnaissance d'une chaîne par un *afnd*

```
type état_de_afnd =
    { mutable est_final : bool ;
      mutable table : état_de_afnd list vect ;
      mutable table_epsilon : état_de_afnd list ;
      numéro : int } ;;

let lit_transition_afnd q1 c = q1.table.(int_of_char c) ;;
let écrit_transition_afnd q1 c q2 =
    let i = int_of_char c
    in
    q1.table.(i) <- union q1.table.(i) [ q2 ] ;;

let pt_fixe f l =
    let rec étape res à_faire traités =
        match à_faire with
            | [] -> res
            | q :: r -> if mem_état q traités then
                            étape res r traités
                        else
                            let l = f q
                            in
                            étape (union_états l res)
                                  (union_états l à_faire)
                                  (q :: traités)
    in
    étape l l [] ;;

let calcule_fermeture ql =
    pt_fixe (function q -> q.table_epsilon) ql ;;

let reconnaissance_afnd états chaîne =
    let n = string_length chaîne
    in
    let rec avance ql i =
        if i = n then ql
        else
            let c = int_of_char chaîne.[i]
            in
            let ql' =
                union_listes_états
                     (map
                        (function q -> q.table.(c))
                        ql)
            in
            avance (calcule_fermeture ql') (i+1)
    in
    let ql = avance (calcule_fermeture états) 0
    in
    exists (function q -> q.est_final) ql ;;
```

```
écrit_transition_afnd q1 'a' q3 ;;
écrit_transition_afnd q3 'a' q4 ;;
écrit_transition_afnd q4 'a' q5 ;;
écrit_transition_afnd q1 'b' q5 ;;
écrit_transition_afnd q1 'b' q4 ;;
écrit_transition_afnd q2 'b' q3 ;;
écrit_transition_afnd q4 'b' q1 ;;
écrit_transition_afnd q5 'b' q5 ;;
```

6.5 Déterminisation d'un *afnd*

On s'intéresse ici au problème de la déterminisation d'un automate fini non déterministe α. Il s'agit de déterminer un automate fini déterministe β qui reconnaît le même langage : $L(\beta) = L(\alpha)$.

6.5.1 Algorithme de déterminisation

Un exemple

Considérons l'automate non déterministe α de la figure 6.5 qui reconnaît bien sûr tous les mots sur l'alphabet $\{a, b\}$ qui se terminent par la chaîne *abab*.

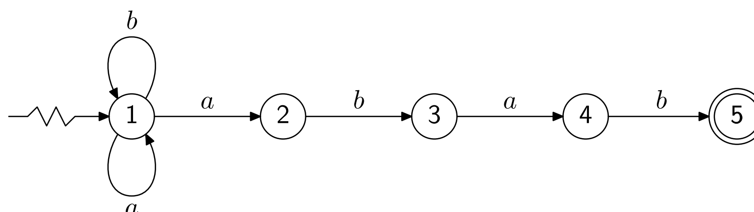


FIG. 6.5: Un *afnd* pour les mots finissant par *abab*

Nous allons construire pas à pas un automate déterministe β qui reconnaît le même langage. Ce nouvel automate aura un ensemble d'état égal à $\mathcal{P}(Q)$, ensemble des parties de l'ensemble des états de α.
Pour chaque partie $X = \{q_1, \ldots, q_k\}$ de Q, on définira les transitions de β à partir de X de la façon suivante : à la lecture d'un caractère c, on aboutira à un état qui est l'ensemble Y de tous les états de Q qu'on atteint à partir d'un quelconque des états qui composent X.
Pour notre exemple, le tableau suivant fournit quelques transitions de β :

état de départ	a est lu	b est lu
$\{1\}$	$\{1, 2\}$	$\{1\}$
$\{1, 2\}$	$\{1\}$	$\{1, 3\}$
$\{2, 3\}$	$\{4\}$	$\{3\}$
$\{1, 2, 3\}$	$\{1, 2, 4\}$	$\{1, 3\}$
$\{2, 3, 4, 5\}$	$\{4\}$	$\{3, 5\}$
...	...	...

Si on écrit le tableau complet, il y aura $2^5 = 32$ lignes à écrire... mais toutes ne sont pas utiles, car certaines parties de Q ne correspondent pas à des états de β accessibles à partir de l'état initial $\{1\}$.
On va donc reconstituer le tableau en écrivant les lignes nécessaires au fur et à mesure des besoins, c'est-à-dire de leur apparition dans les deux colonnes de droites.

état de départ	a est lu	b est lu
$\{1\}$	$\{1, 2\}$	$\{1\}$
$\{1, 2\}$	$\{1\}$	$\{1, 3\}$
$\{1, 3\}$	$\{1, 2, 4\}$	$\{1\}$
$\{1, 2, 4\}$	$\{1, 2\}$	$\{1, 3, 5\}$
$\{1, 3, 5\}$	$\{1, 2, 4\}$	$\{1\}$

Cette fois on n'a plus que 5 états ! Les états finals de β sont ceux qui contiennent un état final de α, ici il s'agit du seul état $\{1, 3, 5\}$.
On peut dessiner l'automate obtenu, ce qui est fait dans la figure 6.6.

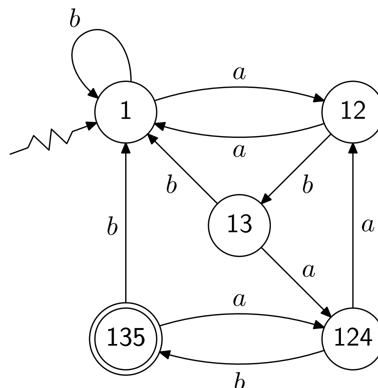


FIG. 6.6: L'automate déterminisé

Effectivement, il est beaucoup moins aisé de deviner sur cet *afd* le langage reconnu…

Explicitation de l'algorithme

Soit $\alpha = (Q, q_0, \mathcal{F}, \varphi, \psi)$ un *afnd* sur l'alphabet $\mathcal{A}$. On construit un *afd* β sur le même alphabet qui a pour ensemble d'états $\mathcal{P}(Q)$, pour état initial l'ensemble $\kappa(q_0)$. Ses états finals sont les parties de Q qui contiennent au moins un état final de α.
Voici comment on construit les transitions de β : soit $X = \{q_1, \ldots, q_k\}$ un état de l'automate β et un caractère c. Notons

$$Y = \bigcup_{1 \leq i \leq k} \varphi(q_i)$$

l'ensemble de tous les états auxquels on aboutit à partir d'un des q_i par une transition de α, puis

$$Z = \bigcup_{y \in Y} \kappa(y)$$

la fermeture par ε-transitions de Y. Alors Z est l'état de β auquel on aboutit depuis X par une transition sur le caractère c.
On montre alors le

Théorème 6.2 (Correction de l'algorithme de déterminisation)
L'automate déterministe β ainsi construit reconnaît le même langage que l'automate non déterministe α.

Complexité de la déterminisation

Hélas, la situation favorable de l'exemple ci-dessus, où l'automate déterminisé a le même nombre d'états que l'*afnd* de départ, est exceptionnelle.
On dispose en effet du

Théorème 6.3 (Complexité de la déterminisation)
Soit $n \geq 1$ un entier naturel. Il existe un *afnd* α à $n+1$ états, tel que tout *afd* β qui reconnaît le même langage ait au moins 2^n états. La déterminisation est donc un algorithme de coût exponentiel.

✧ Considérons, pour n fixé, l'automate non déterministe α de la figure 6.7 sur l'alphabet $\mathcal{A} = \{a, b\}$.

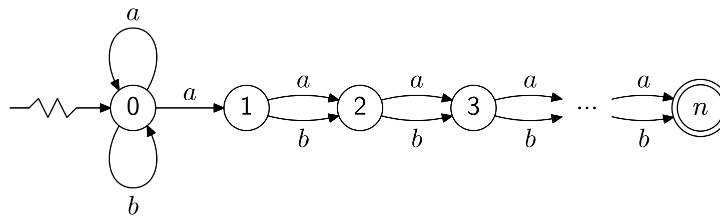


FIG. 6.7: Un *afnd* à la déterminisation coûteuse

α reconnaît tous les mots qui ont au moins n lettres et qui se terminent par un a suivi de $(n-1)$ caractères a ou b.
Montrons que si β est un *afd* qui reconnaît le même langage, alors β a au moins 2^n états.
Soit q_0 l'état initial de β. À tout mot m de n lettres dans $\{a, b\}$, on peut associer l'état q de β défini par $q_0 \xrightarrow{m} q$. On obtient ainsi une application de l'ensemble $\mathcal{A}^n$ des mots de n lettres dans l'ensemble Q des états de β.
Nous allons montrer que cette application est injective. Alors le cardinal de Q vaudra au moins celui de $\mathcal{A}^n$, qui est égal à 2^n, et la preuve sera terminée.
Supposons donc qu'on ait deux mots distincts m et m' de n lettres tels que $q_0 \xrightarrow{m} q$ et $q_0 \xrightarrow{m'} q$ pour un même état q de β.
Soit v le plus long suffixe commun à m et m'. Comme $m \neq m'$, c'est qu'on a $m = uav$ et $m' = u'bv$ (ou le contraire, bien sûr !). Notons que u et u' ont la même longueur (éventuellement nulle) et que v est de longueur au plus égale à $n-1$. Soit alors w un mot quelconque tel que vw soit de longueur exactement égale à $n-1$.
Alors $mw = uavw$ est reconnu par les automates α et donc β, alors que $m'w = u'bvw$ ne l'est pas.
Or $q_0 \xrightarrow{m} q$ et $q_0 \xrightarrow{m'} q$, donc $q \xrightarrow{w} q_f$ où q_f est un état final de l'automate β puisque $q_0 \xrightarrow{mw} q_f$ et mw est reconnu. Mais de même $q_0 \xrightarrow{m'w} q_f$ et $m'w$ n'est pas reconnu, donc q_f n'est pas un état final : on a trouvé la contradiction espérée. ✦

6.5.2 Programmation en Caml

La structure d'ensemble de la bibliothèque standard de Caml

Le module `set` de la bibliothèque standard de Caml permet la gestion des parties d'un ensemble muni d'une relation d'ordre. Voici une description des fonctions qui le composent :

typage le type `'a t` est le type des ensembles d'éléments du type `'a` ;

`set__empty` renvoie l'ensemble vide, cette fonction attend en argument une fonction de comparaison, de type `'a -> 'a -> int` qui renvoie 0 si les deux éléments sont égaux, un entier négatif (*resp.* positif) si le premier est plus petit (*resp.* grand) que le second, pour les entiers on pourra utiliser `set__empty (prefix -)`, pour les ensembles d'états d'un *afnd* (ce qui va nous intéresser ici spécialement, bien sûr), nous pourrons utiliser `set__empty compare_états`, où `compare_états q1 q2`

renvoie `q1.numéro - q2.numéro`, enfin, pour les ensembles d'ensembles, on pourra utiliser, ce qui va bientôt devenir plus clair, `set__empty set__compare` ;

`set__is_empty` renvoie `true` si et seulement si son argument est l'ensemble vide ;

`set__mem` est le test d'appartenance : `set__mem x E` dit si $x \in E$;

`set__add` ajoute un élément à un ensemble : `set__add x E` renvoie l'ensemble $\{x\} \cup E$;

`set__remove` réalise l'opération inverse : `set__remove x E` renvoie l'ensemble $E \setminus \{x\}$;

`set__union` réalise l'union de deux ensembles ;

`set__inter` réalise l'intersection de deux ensembles ;

`set__diff` réalise la différence ensembliste ;

`set__equal` teste l'égalité de deux ensembles ;

`set__compare` prend en arguments deux ensembles et renvoie un entier qui n'est nul que si les deux ensembles sont égaux, et définit dans les autres cas une relation d'ordre sur l'ensemble des ensembles de ce type, cette fonction est particulièrement utile pour créer le type des ensembles d'ensembles d'un type donné (voir la définition de `set__empty`) ;

`set__elements` renvoie la liste des éléments d'un ensemble ;

`set__iter` applique une même fonction successivement aux éléments d'un ensemble, renvoie toujours `()` ;

`set__fold` combine les éléments d'un ensemble : `set__fold f E a` renvoie $(f x_n \ldots (f x_2 (f x_1 a)) \ldots)$, où les x_i sont les éléments de l'ensemble E.

Quelques préliminaires

Nous commençons, dans le programme 6.4 page suivante par définir quelques fonctions utiles pour la suite.

`taille_ensemble` utilise la fonction `set__fold` pour calculer le cardinal d'un ensemble quelconque ;

`set_exists` dit si le prédicat est vérifié par au moins un des éléments de l'ensemble testé (on a utilisé le déclenchement d'une exception en cas de succès pour éviter de parcourir tout l'ensemble et sortir plus vite de la fonction `set__iter`) ;

`compare_états` permettra la comparaison de deux états par le biais de leurs numéros, et sera la fonction utilisée par `set__empty` pour nos ensembles d'états, ainsi que nous l'avons expliqué plus haut ;

`set_of_list` crée un ensemble à partir d'une liste d'états en supprimant les doublons éventuels au passage ;

`fermeture` agit comme `calcule_fermeture` mais prend un ensemble d'états et renvoie de même un ensemble d'états, au lieu de travailler sur des listes.

La déterminisation proprement dite

On suit très exactement l'algorithme qui a été développé plus haut dans notre programme 6.5 page 72. Cependant, à cause de la circularité des strutures utilisées, nous sommes contraints de construire dans un premier temps l'ensemble des états de l'*afd* en gestation, puis, dans un second temps, on met à jour les tables de transitions des états ainsi créés.
La fonction `ens_transition` (lignes 1–5) attend un ensemble d'états de départ de l'*afnd* et le code ASCII d'un caractère, et renvoie l'ensemble des états de l'*afnd* résultats de toutes les transitions correspondantes.
La fonction `liste_états_afd` (lignes 7–28) prend en argument l'état initial `q0` de l'*afnd*. On commence (ligne 24) par créer l'ensemble `e0` des états de sa fermeture, qui représentera l'état initial de l'*afd* final. On crée alors (ligne 26) un ensemble `eE0` formé du seul ensemble `e0` (on note systématiquement les ensembles d'ensembles par un identificateur commençant par `eE`). On appelle alors la fonction `itère` qui petit à petit construit l'ensemble de tous les états de l'*afd*, ou plutôt de leurs représentations en tant qu'ensembles d'états de l'*afnd*. C'est ce qu'on faisait à la main en remplissant les colonnes de notre tableau.
La fonction `créé_états_afd` (lignes 30–40) renvoie une liste de couples (e, q) où e est un ensemble d'états de l'*afnd* et q l'état correspondant de l'*afd*. Bien sûr, comme on l'a déjà annoncé, les tables de transition

Programme 6.4 Fonctions utiles sur les ensembles

```
let taille_ensemble e =
    set__fold (fun _ n -> n + 1) e 0 ;;

let set_exists prédicat e =
    try
        set__iter
            (function x
                -> if prédicat x then failwith "Gagné" else ())
            e ;
        false
    with Failure "Gagné" -> true ;;

let compare_états q1 q2 = q1.numéro - q2.numéro ;;

let set_of_list ql =
    list_it set__add ql (set__empty compare_états) ;;

let fermeture ens_états =
    set_of_list (calcule_fermeture (set__elements ens_états)) ;;
```

ne sont pas encore correctes : on ne peut écrire les transitions que quand on a à sa disposition tous les états.
La fonction `màj_transitions` (lignes 48–54) est justement chargée de mettre à jour ces tables de transition. Elle utilise la fonction `associe` (lignes 42–46) qui prend en argument un ensemble d'états de l'*afnd* et la liste de couples (e, q) décrite ci-dessus, et qui renvoie l'état q de l'*afd* correspondant.
La fonction `déterminise` (lignes 56–60) ne pose alors plus de problème.

6.6 Exercices pour le chapitre 6

Exercice 6.1 Quelques automates simples

Concevoir un automate sur l'alphabet $\{a, b\}$ qui accepte les chaînes ayant un nombre pair de a.
Concevoir un automate sur l'alphabet $\{a, b\}$ qui accepte les chaînes ayant au plus deux a consécutifs, c'est-à-dire qui rejette toute chaîne contenant le mot aaa.
Concevoir un automate sur l'alphabet $A = \{a, b, c, \ldots, z\}$ qui accepte les chaînes contenant le mot $info$.

Exercice 6.2 Déterminisation d'un automate simple

Reprendre le dernier des automates de l'exercice précédent et le déterminiser.

Exercice 6.3 Preuve de la déterminisation

Rédiger une démonstration du théorème 6.2 page 68.

Exercice 6.4 Ajout d'un état mort

Pour éviter les blocages d'un *afd*, on peut ajouter un nouvel état q_m dit *état mort* : pour tout état q et tout caractère c tel qu'il n'y ait pas de transition depuis q sur c, on ajoute la transition $q \xrightarrow{c} q_m$. En outre pour tout caractère c, on ajoute la transition $q_m \xrightarrow{c} q_m$. Enfin, q_m ne fait pas partie des états finals.
Il est bien clair qu'ainsi on ne change pas le langage reconnu par notre automate initial.
Soit α un *afd* et β le même automate auquel on a ajouté un état mort, comme on vient de l'expliquer. On sait que $L(\alpha) = L(\beta)$.
Soit α' (*resp.* β') l'automate obtenu à partir de α (*resp.* β) en rendant final tout état non final et réciproquement. Comparer $L(\alpha) = L(\beta)$, $L(\alpha')$ et $L(\beta')$.

Programme 6.5 La déterminisation des automates

```
1 let ens_transition e i =
2     set_of_list
3         (set__fold
4             (fun q l -> (calcule_fermeture q.table.(i)) @ l)
5             e []);;
6
7 let liste_états_afd q0 =
8     let rec itère eEproductifs eEdéfinitifs =
9         if set__is_empty eEproductifs then eEdéfinitifs
10        else
11        let eEnouveaux = ref (set__empty set__compare)
12        in
13        for i = 0 to 255 do
14            set__iter
15                (function e
16                    -> let e' = ens_transition e i
17                       in
18                       if not (set__is_empty e' || set__mem e' eEdéfinitifs)
19                       then eEnouveaux := set__add e' !eEnouveaux)
20                eEproductifs
21        done;
22        itère !eEnouveaux (set__union !eEnouveaux eEdéfinitifs)
23    in
24    let e0 = set_of_list (calcule_fermeture [ q0 ] )
25    in
26    let eE0 = set__add e0 (set__empty set__compare)
27    in
28    itère eE0 eE0;;
29
30 let crée_états_afd q0 =
31    let état_afd e =
32        {   est_terminal = set_exists (function q -> q.est_final) e;
33            transitions = make_vect 256 Blocage     }
34    in
35    let eE = liste_états_afd q0
36    in
37    set__fold
38        (fun e l
39            -> (e , (état_afd e)) :: l)
40        eE [];;
41
42 let rec associe e l = match l with
43    | [] -> raise Not_found
44    | (e',q_afd) :: r
45        -> if set__equal e e' then q_afd
46            else associe e r;;
47
48 let màj_transitions q_afd e les_états =
49    for i = 0 to 255 do
50        let e' = ens_transition e i
51        in
52        if not set__is_empty e' then
53            q_afd.transitions.(i) <- OK (associe e' les_états)
54    done;;
55
56 let déterminise q0 =
57    let les_états = crée_états_afd q0
58    in
59    map (function (e,q) -> màj_transitions q e les_états) les_états;
60    associe (set_of_list (calcule_fermeture [ q0 ])) les_états;;
```

Exercice 6.5 Déterminisation d'un *afd* !

Que se passe-t-il quand on applique notre algorithme de déterminisation à un automate déjà déterministe ?

Exercice 6.6 Minimisation d'un *afd*

Soit α un automate déterministe. Pour éviter tout problème, on le suppose muni d'un état mort (voir exercice 6.4).

On cherche à construire un automate déterministe β qui reconnaisse le même langage mais qui ait un nombre minimal d'états. L'algorithme que nous présentons ici s'appelle l'algorithme de Hopcroft.

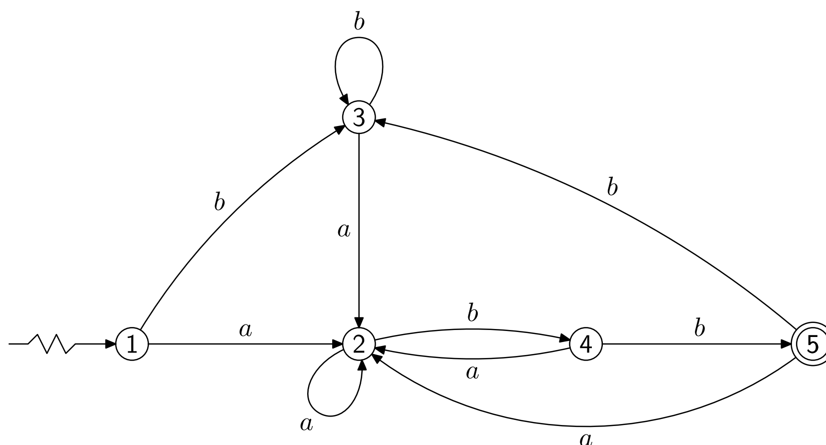
Pour cela, on introduit la notion d'états équivalents de α : deux états q_1 et q_2 sont dits équivalents si pour tout mot w, les deux calculs à partir de q_1 et q_2 bloquent tous les deux, ou si, quand $q_1 \xrightarrow{w} q$ et $q_2 \xrightarrow{w} q'$, q et q' sont tous les deux finals ou tous les deux non-finals.

Il est bien évident qu'on peut alors tout simplement identifier deux états équivalents sans changer le langage reconnu.

Plutôt que de chercher les états équivalents, on se propose de déterminer si deux états ne sont pas équivalents. Voici les deux règles que l'on appliquera :

1. deux états q_1 et q_2 dont l'un est final et l'autre pas ne sont pas équivalents ;
2. si q_1 et q_2 sont déjà connus non équivalents, si c est un caractère de l'alphabet, et si q'_1 et q'_2 vérifient $q'_1 \xrightarrow{c} q_1$ et $q'_2 \xrightarrow{c} q_2$, alors q'_1 et q'_2 sont non équivalents.

On va donc raffiner petit à petit la relation d'équivalence : on maintiendra une partition en classes, initialisée d'après la règle (1) à deux parties, les états finals d'un côté, et les autres. On applique ensuite la règle (2) jusqu'à ce que la partition ne soit plus modifiée : on cherchera si la lecture d'un caractère de l'alphabet partage une des parties en deux sous-parties. Exemple : pour l'automate ci-dessous, sur l'alphabet $\{a, b\}$, on obtient les partitions successives suivantes.



On commence par $\{\{1, 2, 3, 4\}, \{5\}\}$ (règle 1).

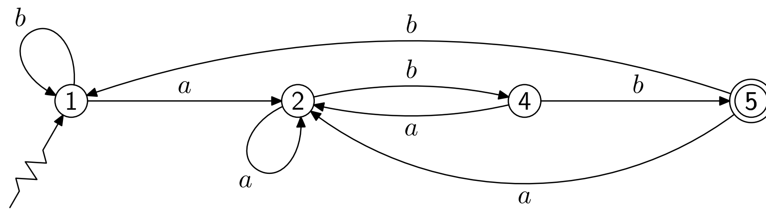
La lecture d'un a ne permet pas de séparer les états du premier groupe, mais la lecture d'un b isole l'état 4.

On obtient donc $\{\{1, 2, 3\}, \{4\}, \{5\}\}$.

Même chose ensuite, où l'on isole l'état 2.

La partition finale est $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.

Cela correspond à l'automate suivant :



Remarque : ces automates reconnaissent les mots sur $\{a, b\}$ qui se terminent par abb.
Écrire une fonction de minimisation en Caml.

Chapitre 7

Langages rationnels et automates

Dans tout ce chapitre, $\mathcal{A}$ désigne un ensemble fini, appelé alphabet.

7.1 Langages rationnels et expressions régulières

7.1.1 Définition

Langages, opérations sur les langages

Étant donné un alphabet $\mathcal{A}$, on note $\mathcal{A}^\star$ l'ensemble des mots sur $\mathcal{A}$: il s'agit, rappelons-le, des suites finies d'éléments de l'alphabet. Cet ensemble de mots est muni d'une opération interne, la concaténation, qu'on note parfois par un point, mais le plus souvent par simple juxtaposition : si w_1 et w_2 sont deux mots, on notera aussi bien $w_1.w_2$ que w_1w_2 le mot obtenu par concaténation. Le mot vide est noté ε.
Un langage sur l'alphabet $\mathcal{A}$ est une partie de $\mathcal{A}^\star$. Leur ensemble est noté $\mathcal{L}(\mathcal{A}) = \mathcal{P}(\mathcal{A}^\star)$. Notons que $\emptyset$ est un langage particulier.
La concaténation des mots fait de $\mathcal{A}^\star$ un monoïde : c'est-à-dire que la concaténation est une loi de composition interne associative.

Définition 7.1 (Opérations sur les langages) On définit sur $\mathcal{L}(\mathcal{A})$ trois opérations fondamentales :

l'union de deux langages L_1 et L_2 est tout simplement l'ensemble des mots qui appartiennent à L_1 ou à L_2, qui est bien sûr noté $L_1 \cup L_2$;

le produit noté L_1L_2 de deux langages L_1 et L_2 est l'ensemble des mots obtenus par concaténation d'un mot de L_1 et d'un mot de L_2 : $L_1L_2 = \{w_1w_2, w_1 \in L_1, w_2 \in L_2\}$;

l'étoile d'un langage L est l'ensemble noté $L^\star$ des mots obtenus par concaténation d'un nombre fini (éventuellement nul) de mots de L,

$$L^\star = \{\varepsilon\} \cup L \cup \{w_1w_2 \cdots w_n, n \geq 2, \forall i, w_i \in L\};$$

on peut dire aussi que $L^\star$ est le sous-monoïde de $\mathcal{A}^\star$ engendré par L.

Langages rationnels

Définition 7.2 (Langages rationnels) L'ensemble des langages rationnels sur un alphabet $\mathcal{A}$ est noté $R(\mathcal{A})$. Il s'agit de la plus petite partie de $\mathcal{L}(\mathcal{A})$ qui contienne le langage vide et les singletons, et qui soit stable pour les trois opérations listées ci-dessus, à savoir l'union, le produit et l'étoile.

Exemples sur l'alphabet $\mathcal{A} = \{a, b\}$: $\emptyset$ est un langage rationnel ; $\mathcal{A}^\star$ est un langage rationnel (c'est l'étoile d'une somme finie de singletons, puisque l'alphabet est fini) ; l'ensemble des mots qui se terminent par ab est rationnel car il est produit des trois langages rationnels $\mathcal{A}^\star$, $\{a\}$, $\{b\}$.

7.1.2 Expressions régulières

Les définitions très formelles précédentes ne permettent pas de vérifier rapidement qu'un langage est rationnel, et on préfère décrire les langages rationnels à l'aide de *formules* appelées expressions régulières : tout langage rationnel peut être décrit par une expression régulière (il n'y a pas du tout unicité), et réciproquement les expressions régulières ne décrivent de langages que rationnels.
On définit les expressions régulières de la même façon qu'on a pu définir des expressions arithmétiques : on se donnera des constantes, des variables, des opérations et des fonctions. En outre, la sémantique des expressions régulières consistera en une application de l'ensemble des expressions régulières sur $R(\mathcal{A})$.

Définition des expressions régulières

Les informaticiens utilisent le plus souvent le mot *motif* pour décrire une expression régulière (*pattern* en anglais). Nous allons ainsi donner ici la description des différents motifs sur un alphabet $\mathcal{A}$ fixé.
Pour écrire un motif non vide nous aurons à notre disposition des constantes (motifs constants) qui seront les éléments de l'alphabet et la constante ε. Nous aurons aussi des opérateurs binaires : la somme (encore appelée union), que nous noterons |, et le produit (encore appelé concaténation), qui sera notée par simple juxtaposition. Il existera aussi un opérateur unaire : l'étoile, notée de façon suffixe. Enfin, nous nous autoriserons l'usage de variables servant à représenter des sous-motifs.
Plus formellement :

Définition 7.3 (Expressions régulières) Soit $\mathcal{A}$ un alphabet fini. On définit récursivement l'ensemble correspondant $\mathcal{E}(\mathcal{A})$ des expressions régulières de la façon suivante :

- $\emptyset$ est une expression régulière ;
- ε et toute lettre de l'alphabet sont des expressions régulières, dites atomiques ;
- si e_1 et e_2 sont des expressions régulières, $(e_1|e_2)$ et (e_1e_2) sont des expressions régulières, appelées somme et produit des expressions régulières e_1 et e_2 ;
- si e est une expression régulière, $e\star$ est une expression régulière, appelée étoile (ou étoilée) de e.

L'usage est de ne pas écrire les parenthèses que rendent inutiles les priorités suivantes : l'étoile est prioritaire devant le produit, lui-même prioritaire devant la somme.
Exemples d'expressions régulières : $\emptyset$; $ab\star|(a|c)\star b = (a(b\star))|(((a|c)\star)b)$.
Pour éviter de trop longs motifs, on pourra ajouter aux expressions atomiques des variables représentant des sous-motifs.
On écrira ainsi par exemple : $\mathbf{A} = a|b|c|...|z|A|B|...|Z$, $\mathbf{B} = 0|1|2|3|4|5|6|7|8|9$ et $\mathbf{A}(\mathbf{A}|\mathbf{B})\star$ sera un motif qui représente les mots compsés de lettres et de chiffres commençant par une lettre.

Sémantique

Il est temps de donner la sémantique des expressions régulières :

Définition 7.4 (Sémantique des expressions régulières) On définit par induction structurelle une application de $\mathcal{E}(\mathcal{A})$ sur $R(\mathcal{A})$, qui définit la sémantique des expressions régulières en associant à chaque expression régulière un langage rationnel, de la façon suivante :

- le langage associé à $\emptyset$ est le langage vide ;
- à ε est associé le langage constitué du seul mot vide ;
- si c est une lettre de l'alphabet, le langage associé est le singleton $\{c\}$;
- si e_1 et e_2 sont des expressions régulières de langages associés L_1 et L_2, le langage associé à l'expression $(e_1|e_2)$ est la réunion $L_1 \cup L_2$;
- si e_1 et e_2 sont des expressions régulières de langages associés L_1 et L_2, le langage associé à l'expression (e_1e_2) est le produit L_1L_2 ;
- si e est une expression régulière de langage associé L, le langage associé à l'expression $e\star$ est l'étoile $L^\star$ du langage L.

✧ De par la définition même des langages rationnels, il est clair que tous les langages associés aux expressions régulières sont bien rationnels.
Pour montrer la réciproque, on remarque tout d'abord que vide et tout singleton sont représentés par des expressions régulières. L'ensemble des langages associés aux expressions régulières étant clairement stable par union, produit et étoile, on en déduit qu'on a bien décrit tous les langages rationnels. ✦

Comme les expressions régulières seront représentées par des chaînes de caractères, les informaticiens ont dû ajouter quelques notations particulières : ainsi le symbole . sert-il à désigner toute lettre de l'alphabet, c'est si l'on préfère l'expression somme de toutes les expressions atomiques sauf ε. En outre, on utilise le symbole + (en notation suffixe) pour représenter une répétition au moins une fois d'un motif : pour tout motif **A**, **A**+ désignera **AA**$*$. De même ? (toujours en notation suffixe) représentera une répétition zéro ou une fois d'un motif : pour tout motif **A**, **A**? désignera $(\varepsilon|\mathbf{A})$. Enfin, pour éviter toute ambiguïté, on utilise un caractère d'échappement, \ : $a\star$ représente une suite quelconque de a, mais $a\backslash\star$ représente un a suivi du caractère étoile.

Expressions régulières en Caml

Voici le type que l'on peut utiliser pour les expressions régulières non vides en Caml :

```
type expr_rat =
    | Epsilon
    | Mot of string
    | Étoile of expr_rat
    | Somme of expr_rat list
    | Produit of expr_rat list ;;
```

Pour éviter d'écrire abc comme `Produit(Lettre ‘a‘ , Produit(Lettre ‘b‘ , Lettre ‘c‘)`, on a préféré utiliser l'abréviation `Mot` pour le produit d'atomes. De même, on fait usage de l'associativité des opérateurs pour écrire `Produit [ X ; Y ; Z]` plutôt que `Produit(X , Produit(Y,Z))`.
L'expression régulière $ab(a|b)*ba$ qui représente les mots sur $\{a, b\}$ d'au moins quatre lettres, commençant par ab et se terminant par ba, sera représentée par l'objet Caml suivant :

```
Produit [ Mot "ab" ; Étoile( Somme [ Mot "a" ; Mot "b" ] ) ; Mot "ba" ] ;;
```

Nous n'aborderons pas ici le délicat problème du passage de la syntaxe concrète à la syntaxe abstraite des expressions régulières, c'est-à-dire de l'écriture d'une fonction `parseur : string -> expr_rat` qui serait utilisée par exemple ainsi :

```
#parseur "ab(a|b)*ba" ;;
- : expr_rat =
 Produit [Mot "ab"; Étoile (Somme [Mot "a"; Mot "b"]); Mot "ba"]
#
```

7.2 Le théorème de Kleene

Nous démontrons ici un théorème fondamental, le théorème de Kleene, qui s'énonce ainsi :

Théorème 7.1 (Kleene)
L'ensemble $R(\mathcal{A})$ des langages rationnels sur un alphabet fini $\mathcal{A}$ est exactement égal à l'ensemble des langages reconnus par des automates finis.

Ce théorème consiste en deux résultats différents, pour lesquels nous donnerons des preuves constructives :

1. tout langage rationnel est reconnu par un automate fini : dans la sous-section 7.2.1, nous prouverons ce résultat en donnant une méthode de construction d'un automate fini non déterministe qui reconnaît un langage rationnel représenté par une expression régulière fixée ;
2. le langage que reconnaît un automate fini est rationnel : dans la sous-section 7.2.2 page suivante, nous prouverons ce résultat en expliquant comment construire une expression régulière qui représente le langage reconnu par un automate fini déterministe fixé.

Notons toutefois que le premier de ces deux résultats est le plus intéressant, puisqu'on sait déterminiser un *afnd*, et qu'un automate fini déterministe permet de programmer facilement et efficacement la reconnaissance des mots d'un langage.

7.2.1 Des langages rationnels aux automates

Nous montrons ici que pour toute expression régulière on peut construire un automate fini non déterministe ayant un unique état initial et un unique état final qui reconnaisse le langage rationnel représenté par l'expression régulière proposée : pour ce faire, nous raisonnerons par induction structurelle.
On laisse au lecteur la construction d'un *afnd* pour le langage vide...
Pour le cas des expressions atomiques, on obtient très facilement les automates de la figure 7.1.

FIG. 7.1: Automates pour les expressions régulières atomiques

Soit e_1 et e_2 deux expressions régulières, et q_0, q_f (*resp.* q'_0, q'_f) les états initial et final de l'automate associé à e_1 (*resp.* e_2). On construira, conformément à la figure 7.2, l'automate de la somme $e_1|e_2$ en ajoutant un état initial duquel on mènera deux ε-transitions vers q_0 et q'_0 et un état final auquel arriveront deux ε-transitions depuis q_f et q'_f.

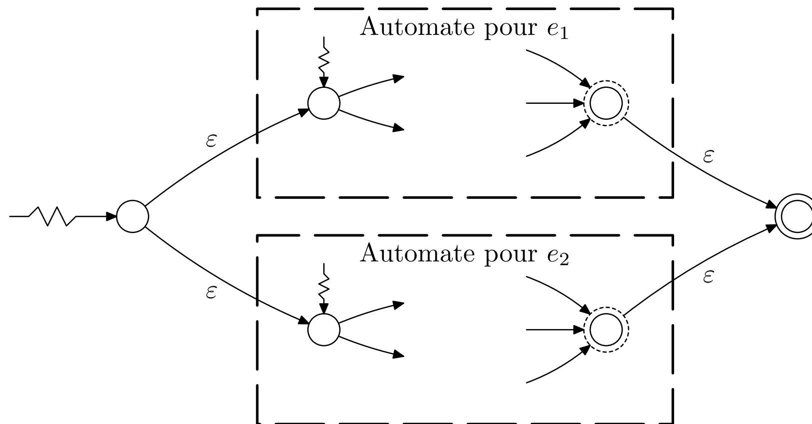


FIG. 7.2: Automate pour la somme de deux expressions régulières

Soit e_1 et e_2 deux expressions régulières, et q_0, q_f (*resp.* q'_0, q'_f) les états initial et final de l'automate associé à e_1 (*resp.* e_2). On construira, conformément à la figure 7.3 page ci-contre, l'automate du produit e_1e_2 en ajoutant un état initial duquel on mènera une ε-transition vers q_0 et un état final auquel arrivera une ε-transition depuis q'_f ; on ajoutera enfin une ε-transition depuis q_f vers q'_0.
Soit e une expression régulière, et q_0, q_f les états initial et final de l'automate associé. On construira, conformément à la figure 7.4 page suivante, l'automate de l'étoile $e\star$ en ajoutant un état initial Q_0 et un état final Q_f. De Q_0 on mènera deux ε-transitions vers q_0 et Q_f. De Q_f on mènera une ε-transition vers Q_0, et enfin une ε-transitions ira depuis q_f vers Q_f.

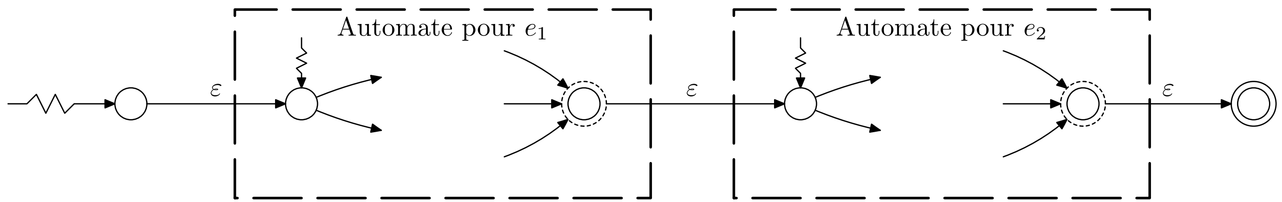


FIG. 7.3: Automate pour le produit de deux expressions régulières

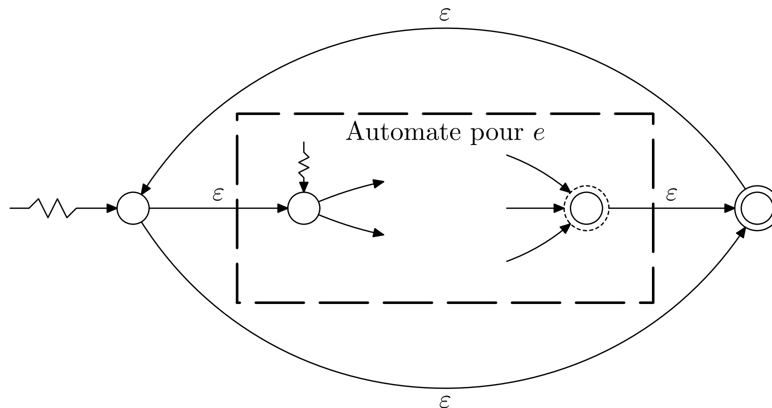


FIG. 7.4: Automate pour l'étoile d'une expression régulière

Bien sûr, les automates ainsi construits sont truffés de ε-transitions, mais elles disparaîtront dans la déterminisation qu'on ne manquera pas d'effectuer.

7.2.2 Des automates aux langages rationnels

Nous décrirons ici l'algorithme classique, dit *par suppression des états*, qui permet de construire une expression régulière qui représente le langage reconnu par un automate déterministe fini.
Soit α un tel automate, et $q_1, q_2, \ldots, q_k$ ses états finals. Notons α_i l'automate égal à α à ceci près que seul l'état q_i est considéré comme état final. D'après la définition des mots reconnus par α, il est bien évident que $L(\alpha) = \bigcup_{i=1}^{k} L(\alpha_i)$. C'est ce qui permet, dans la description de notre algorithme, de ne considérer que le cas d'automates finis déterministes n'ayant qu'un unique état initial q_0 et un unique état final q_f.
En effet, si e_i est une expression régulière qui décrit le langage reconnu par α_i, une expression régulière solution du problème posé est $e_1|e_2|\cdots|e_k$.

L'algorithme de suppression des états

L'idée est de supprimer petit à petit tous les états distincts de q_0 et q_f, jusqu'à n'avoir plus qu'un automate à 2 états. Il faudra cependant généraliser les transitions, et les étiqueter non plus avec des lettres de l'alphabet, mais avec des expressions régulières.
Supposons par exemple qu'on soit dans la situation de la figure 7.5 page suivante. On souhaite supprimer l'état x. Pour chaque couple d'états e, s tel qu'il existe une transition de e entrant dans x et une transition vers s sortant de x, on procède comme suit : si A est l'expression régulière qui étiquette la transition de e vers x, B de x à s, C de x vers x et D de e vers s, on écrira une transition de e vers s étiquetée par l'expression $D|AC\star B$. On obtient alors l'automate de la figure 7.6 page suivante.
Finalement, il ne restera plus qu'un état initial et un état final : on se retrouve dans la situation de la figure 7.7 page suivante. Le problème est alors résolu, puisqu'une expression régulière qui convient est la

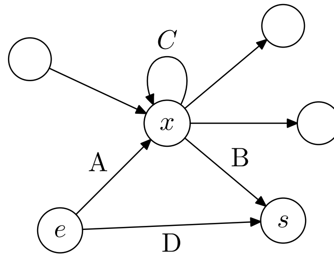


FIG. 7.5: Avant la suppression de l'état x

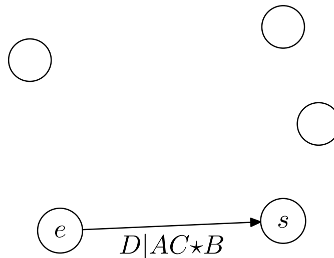


FIG. 7.6: Après la suppression de l'état x

suivante : $A\star B(D|CA\star B)\star$.

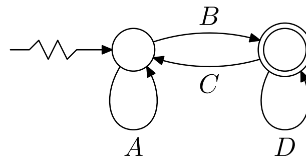


FIG. 7.7: Automate à 2 états

Un exemple

Considérons l'exemple de l'automate de la figure 7.8 page suivante.
On commence par s'intéresser à l'état final 3. On supprime l'état 2, obtenant l'automate de la figure 7.9 page ci-contre.
On aura noté que l'étiquette de la boucle sur l'état initial est devenue $a|ba$.
On supprime alors l'état 4, obtenant l'automate de la figure 7.10 page 82.
On peut maintenant appliquer la règle ci-dessus, et annoncer que le langage reconnu par ce dernier automate est décrit par le motif suivant :

$$(a|ba)\star aa(b|ab|aa(a|ba)\star bb)\star.$$

Reprenons maintenant l'automate initial, en nous intéressant à l'état final 4. Après suppression de l'état 2, nous obtenons l'automate de la figure 7.11 page 82.
On supprime encore l'état 3 obtenant l'automate de la figure 7.12 page 82.
On écrit l'expression régulière qui représente le langage reconnu :

$$(a|ba)\star bbb\star a(bb\star a|a(a|ba)\star bb\star a)\star.$$

Finalement voici une expression régulière qui représente le langage reconnu par l'automate initial :

$$(a|ba)\star aa(b|ab|aa(a|ba)\star bb)\star|(a|ba)\star bbb\star a(bb\star a|a(a|ba)\star bb\star a)\star.$$

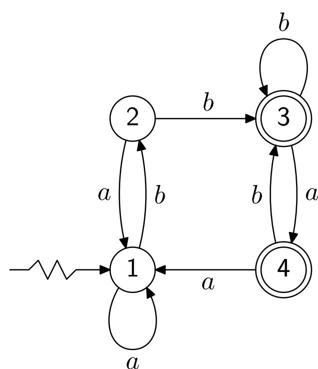


FIG. 7.8: Quel est le langage reconnu par cet automate ?

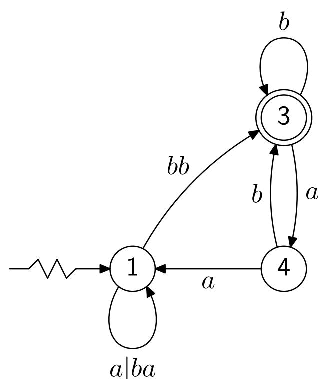


FIG. 7.9: On a supprimé l'état 2

Personne ne dit que l'expression régulière ci-dessus est la plus simple possible. Sur notre exemple, ce n'est d'ailleurs pas du tout le cas, car voici une expression régulière qui représente le même langage :

$$(a|b)\star bb(b|ab)\star(\varepsilon|a).$$

La simplification des expressions régulières n'est pas quelque chose de très aisé, même si les exercices qui se trouvent à la fin de ce chapitre peuvent donner quelques pistes d'inspiration.

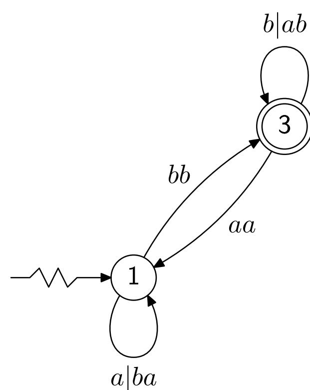


FIG. 7.10: Après suppression des états 2 et 4

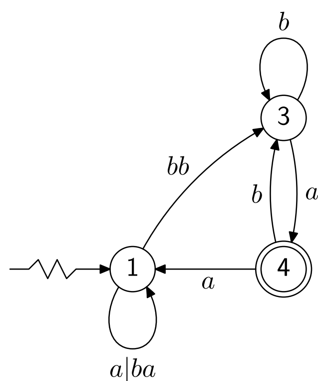


FIG. 7.11: On recommence en supprimant l'état 2

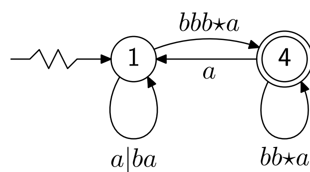


FIG. 7.12: Après suppression des états 2 et 3

7.3 Langages non rationnels

Nous montrons ici que tout langage n'est pas nécessairement rationnel. Nous commençons par donner deux exemples, pour lesquels nous démontrons *à la main* qu'il ne s'agit pas de langages rationnels, puis nous démontrons un lemme général qui permet de discriminer une large classe de langages non rationnels.

7.3.1 Exemples

Montrons à la main que les deux langages L_1 et L_2 suivants sur l'alphabet $\{a, b\}$ ne sont pas rationnels :

1. L_1 est le langage constitué des mots de la forme $a^n b^n$;
2. L_2 est le langage constitué des mots de la forme $a^n b^p$ avec $n \geq p$.

Le langage L_1 n'est pas rationnel

Si L_1 était rationnel, on pourrait trouver un automate fini déterministe α qui le reconnaisse. Soit m le nombre de ses états, et q_0 son état initial.
Soit w_ℓ la chaîne composée de ℓ fois le caractère a. On définit l'état q_ℓ par $q_0 \xrightarrow{w_\ell} q_\ell$, ce qui a bien du sens, puisque l'automate est déterministe, et qu'il ne peut pas se bloquer sur a^ℓ comme il accepte la chaîne $a^\ell b^\ell$. Comme l'automate est fini, il existe deux entiers i et j tels que $0 \leq i < j \leq m$ tels que $q_i = q_j$. C'est dire que $q_{j-1} \xrightarrow{a} q_i$: on a trouvé une boucle dans l'automate. On en déduit que $q_0 \xrightarrow{a^{i+k(j-i)}} q_i$ pour tout entier k.
Or on sait que $a^i b^i$ est reconnu par l'automate. Ainsi a-t-on $q_i \xrightarrow{b^i} q_f$ où q_f est un état final. Mais alors on aura pour tout entier k $q_0 \xrightarrow{a^{i+k(j-i)}b^i} q_f$, ce qui prouve que l'automate reconnaît des mots qui ne sont pas dans L_1 : c'est la contradiction recherchée.

Le langage L_2 n'est pas rationnel

Si L_2 était rationnel, on pourrait trouver un automate fini déterministe α qui le reconnaisse. Soit m le nombre de ses états, et q_0 son état initial.
Soit w_ℓ la chaîne composée de ℓ fois le caractère a. On définit l'état q_ℓ par $q_0 \xrightarrow{w_\ell} q_\ell$, ce qui a bien du sens, puisque l'automate est déterministe, et qu'il ne peut pas se bloquer sur a^ℓ comme il accepte la chaîne $a^\ell b^\ell$. Comme l'automate est fini, il existe deux entiers i et j tels que $0 \leq i < j \leq m$ tels que $q_i = q_j$. C'est dire que $q_{j-1} \xrightarrow{a} q_i$: on a trouvé une boucle dans l'automate. On en déduit que $q_0 \xrightarrow{a^{i+k(j-i)}} q_i$ pour tout entier k.
On sait que $a^j b^j$ est dans L_2 : c'est dire que $q_0 \xrightarrow{a^j b^j} q_f$ où q_f est un état final. On a donc $q_i \xrightarrow{b^j} q_f$. Mais alors on a aussi $q_0 \xrightarrow{a^i b^j} q_f$, puisque $q_0 \xrightarrow{a^i} q_i$. On en déduit que l'automate accepte le mot $a^i b^j$ qui n'appartient pourtant pas à L_2 puisque $i < j$: c'est la contradiction recherchée.

7.3.2 Le lemme de l'étoile

Lemme 7.1 (Lemme de l'étoile) Soit L un langage rationnel. Il existe un entier n tel que tout mot w de L de taille au moins égale à n s'écrit sous la forme $w = rst$ où s est un mot non vide de taille au plus égale à n et où le motif $rs{\star}t$ décrit des mots qui sont tous dans L.

✧ Soit en effet α un automate fini déterministe reconnaissant L, et soit n le nombre de ses états, q_0 son état initial et $\mathcal{F}$ l'ensemble de ses états finals.
Soit w un mot de L de taille supérieure ou égale à L. Notons w_ℓ les préfixes de w : w_ℓ est composé des ℓ premiers caractères de w. Définissons alors les états q_ℓ de l'automate en posant $q_0 \xrightarrow{w_\ell} q_\ell$. Ceci a bien du sens car l'automate est déterministe et ne peut pas se bloquer sur un préfixe d'un mot qu'il reconnaît.
Comme w a au moins autant de lettres que α d'états, on est certain de pouvoir trouver deux indices i et j tels que $0 \leq i < j \leq n$ avec $q_i = q_j$.

Soit alors r le préfixe (éventuellement vide) de w formé des i premiers caractères de w ; s le mot (non vide) formé des $j-i$ caractères suivants ; et t le suffixe (éventuellement vide) de w formé des caractères suivants.
On a bien $w = rst$. En outre $q_0 \xrightarrow{r} q_i \xrightarrow{s} q_j = q_i \xrightarrow{t} q_f$ où q_f est un état final.
On aura bien alors $q_0 \xrightarrow{rs^pt} q_f$ pour tout entier p, ce qui achève notre démonstration. ❖

On aura noté comment le lemme de l'étoile généralise les méthodes que nous avons utilisées pour les deux exemples ci-dessus.

7.4 Exercices pour le chapitre 7

Exercice 7.1 Langages rationnels, intersection et complémentaire

Soit $\mathcal{A}$ un alphabet fini. En utilisant certains exercices du chapitre précédent, montrer que $R(\mathcal{A})$ est stable par intersection et complémentaire, c'est-à-dire que le complémentaire (dans $\mathcal{A}^\star$) d'un langage rationnel est un langage rationnel, et que l'intersection de deux langages rationnels est un langage rationnel.

Exercice 7.2 Équivalence des expressions régulières

On dira de deux expressions régulières e_1 et e_2 qu'elles sont équivalentes, ce que nous noterons $e_1 \equiv e_2$ si elles décrivent le même langage.
Montrer les formules suivantes, valables pour toutes expressions régulières e_1 et e_2 :

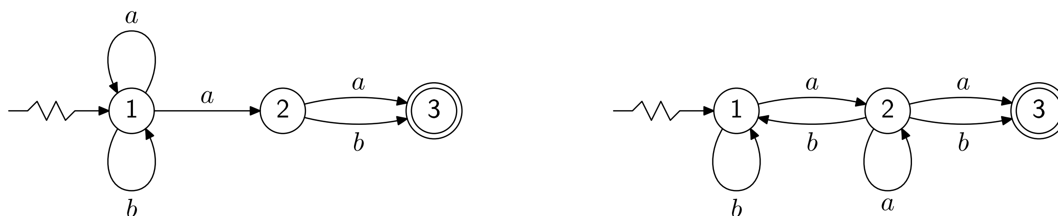
$$\begin{aligned} (e_1e_2)\star &\equiv \varepsilon|e_1(e_2e_1)\star e_2 \\ (e_1|e_2)\star &\equiv e_1\star(e_2e_1\star)\star \\ \forall p \geq 1, e_1\star &\equiv (\varepsilon|e_1|\cdots|e_1^p)(e_1^{p+1})\star \end{aligned}$$

Exercice 7.3 Le langage des facteurs des mots d'un langage

Soit L un langage rationnel et soit L' le langage constitué de tous les mots qui sont des sous-chaînes de mots de L. Montrer que L' est aussi rationnel.

Exercice 7.4 Reconnaissance d'un même langage

Montrer que les deux *afnd* de la figure ci-dessous reconnaissent le même langage.



Exercice 7.5 Exemples de langages non rationnels

Montrer que le langage sur $\{a, b\}$ formé des mots de la forme a^nba^n, où $n \geq 0$, n'est pas rationnel. Même question avec le langage sur le même alphabet constitué des mots de la forme ww' où w' est le miroir du mot w (les mêmes lettres, mais lues de droite à gauche).

Exercice 7.6 Expressions régulières décrivant des langages donnés

Écrire des expressions régulières qui décrivent les langages suivants :

1. les mots sur $\{a, b\}$ contenant au moins un a ;
2. les mots sur $\{a, b\}$ contenant au plus un a ;
3. les mots sur $\{a, b\}$ tels que toute série de a ait une longueur paire ($bbbaabaaaa$, b et $aaaab$ conviennent, mais pas $abbaa$ ni aaa) ;
4. les mots sur $\{a, b, c\}$ tels que deux lettres consécutives sont toujours distinctes.

Troisième partie

Corrigé de tous les exercices

Chapitre 1

Corrigé des exercices

Exercice 1.1

Voici une première solution :

```
let indexation arbre =
    let rec préfixe zéro_ou_un = function
        | Feuille(f)  -> Feuille(zéro_ou_un ^ f)
        | Nœud(n,g,d) -> let g',d' = (préfixe zéro_ou_un g),(préfixe zéro_ou_un d)
                         in
                         Nœud((zéro_ou_un ^ n),g',d')
    in
    let rec indexe = function
        | Feuille(_)  -> Feuille("")
        | Nœud(n,g,d) -> let g',d' = (indexe g),(indexe d)
                         in
                         Nœud("",(préfixe "0" g'),(préfixe "1" d'))
    in
    indexe arbre ;;
```

et en voici une seconde :

```
let indexation arbre =
    let rec indexe a préfixe = match a with
     | Feuille(_) -> Feuille(préfixe)
     | Nœud(_,g,d) -> let g' = indexe g (préfixe ^ "0")
      and d' = indexe d (préfixe ^ "1")
      in
      Nœud(préfixe,g',d')
    in
    indexe arbre "" ;;
```

Exercice 1.2

```
let liste_à_profondeur arbre n =
    let rec ajoute_à_profondeur a k déjà_trouvés =
        if k = 0 then a :: déjà_trouvés
        else match a with
            | Feuille(_)  -> déjà_trouvés
            | Nœud(_,g,d) -> ajoute_à_profondeur g (k - 1)
                                (ajoute_à_profondeur d (k - 1) déjà_trouvés)
    in
    ajoute_à_profondeur arbre n [] ;;
```

Exercice 1.3

```
let rec déshabille = function
    | Feuille(_) -> Vide
    | Nœud(_,g,d) -> Jointure((déshabille g),(déshabille d)) ;;
```

Exercice 1.4

On écrit les deux fonctions sur le même modèle. Il faudra simplement ajouter des tests supplémentaires pour le cas où l'on s'intéresse à la profondeur.
On écrit tout d'abord une fonction qui recombine des squelettes pour des listes fixées de sous-arbres gauches et de sous-arbres droits possibles, ce qui s'écrit ainsi :

```
(* construit les squelettes dont les sous-arbres gauche et droit *)
(* décrivent les listes l1 et l2 *)

let rec recompose l1 l2 =
    match l1 with
        | [] -> []
        | gauche :: q -> (map (function droit -> Jointure(gauche,droit)) l2)
                          @ (recompose q l2) ;;
```

La génération des squelettes de taille et profondeur fixées s'écrit alors :

```
let rec engendre_à_profondeur n p =
    if n = 0 then [ Vide ]
    else if p < 0 then []
    else if p = 0 then if n = 1 then [ Jointure(Vide,Vide) ]
                            else []
    else
        begin
            let résultat = ref []
            in
            for i = 0 to n-1 do
                let liste_gauche = engendre_à_profondeur i (p-1)
                and liste_droite = engendre_à_profondeur (n-i-1) (p-1)
                in
                résultat := !résultat @ (recompose liste_gauche liste_droite)
            done ;
            !résultat
        end ;;
```

On fait de même pour la génération de tous les squelettes de taille fixée :

```
let rec engendre n =
    if n = 0 then [ Vide ]
    else
        begin
            let résultat = ref []
            in
            for i = 0 to n-1 do
                let liste_gauche = engendre i
                and liste_droite = engendre (n-i-1)
                in
                résultat := !résultat @ (recompose liste_gauche liste_droite)
            done ;
            !résultat
        end ;;
```

Exercice 1.5

```
let rec squelette_complet n =
    let pair n = n = 2 * (n/2)
    in
    if n = 0 then Vide
    else if pair (n-1) then
        Jointure((squelette_complet ((n-1) / 2)),
                 (squelette_complet ((n-1) / 2)))
    else failwith "Taille incompatible avec la complétude du squelette" ;;
```

Exercice 1.6

```
let est_complet a =
    let rec feuilles_au_bon_niveau k = function
        | Feuille(_) -> k = 0
        | Nœud(_,g,d) -> feuilles_au_bon_niveau (k-1) g
                        && feuilles_au_bon_niveau (k-1) d
    in
    feuilles_au_bon_niveau (profondeur a) a ;;
```

Exercice 1.7

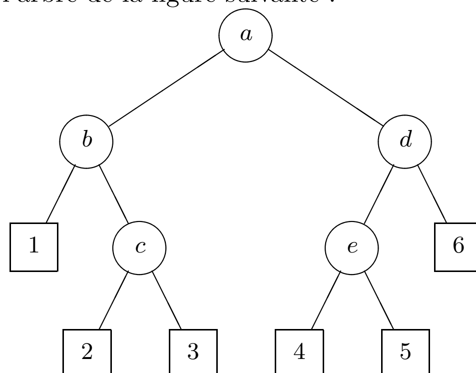
```
let est_équilibré a =
    let rec feuilles_au_bon_niveau k = function
        | Feuille(_) -> k = 0 || k = 1
        | Nœud(_,g,d) -> feuilles_au_bon_niveau (k-1) g
                        && feuilles_au_bon_niveau (k-1) d
    in
    feuilles_au_bon_niveau (profondeur a) a ;;
```

Chapitre 2

Corrigé des exercices

Exercice 2.1

Voilà sans doute l'exercice le plus difficile du chapitre.
Réfléchissons sur l'exemple de l'arbre de la figure suivante :



Son parcours militaire est : $P = abd1ce62345$.
Je propose l'algorithme suivant :

1. remplacer P par son miroir (ici $54326ec1dba$), qu'on va lire élément par élément par la suite ;
2. préparer une pile vide Π ;
3. si l'élément f lu est une feuille, l'empiler dans la pile Π, et reprendre à l'étape 3 ;
4. si l'élément n lu est un nœud, retirer de la pile Π ses deux derniers éléments, x et y, et empiler dans la pile Π l'arbre (n, x, y), puis reprendre à l'étape 3 ;
5. sinon, Π ne doit contenir qu'un élément qui est l'arbre résultat.

Dans notre exemple, on commence par empiler les feuilles 5, 4, 3, 2 , 6 : la pile contient donc dans cet ordre $6; 2; 3; 4; 5$. Vient alors le nœud e : on retire de la pile ses deux plus anciens éléments, c'est-à-dire 4 et 5. On construit l'arbre $(e, 5, 4)$ qu'on empile. La pile contient alors $(e, 5, 4); 6; 2; 3$.
On lit alors le nœud c, on dépile donc 2 et 3, construit l'arbre $(c, 2, 3)$ qu'on empile. La pile contient $(c, 2, 3); (e, 5, 4); 6$.
On lit encore la feuille 1, qu'on empile, puis le nœud d. On dépile $(e, 5, 4)$ et 6, et construit $(d, (e, 5, 4), 6)$ qu'on empile. La pile contient ici $(d, (e, 5, 4), 6); 1; (c, 2, 3)$.
On lit le nœud b, et construit $(b, 1, (c, 2, 3))$ qu'on empile.
La pile contient à ce moment $(b, 1, (c, 2, 3)); (d, (e, 5, 4), 6)$.
Lisant enfin le nœud a, on retrouve bien l'arbre souhaité.
Le problème majeur de programmation vient de ce que Π ne se comporte pas comme une pile : en effet ce sont les éléments les plus anciens qui doivent être retirés en priorité (*first in, first out*), au contraire d'une pile où les éléments retirés en priorité sont les plus récemment empilés (*last in, first out*).

C'est pourquoi je propose une nouvelle structure de données pour représenter Π : on parle de *queue* ou *file d'attente*.

Pour l'implémenter en Caml, nous utiliserons ici tout simplement un vecteur d'arbres, dans la mesure où nous savons qu'il y aura dans Π un nombre d'éléments plus petit que le nombre d'objets dans P.

Il faudra toutefois prendre garde à travailler sur les indices du vecteur modulo la taille du tableau, et garder toujours l'indice des éléments les plus récent et ancien.

On obtient cette gestion d'une queue.

```
type 'a queue =
    { empile : ('a -> unit) ;
      dépile : (unit -> 'a) ;
      est_vide : (unit -> bool) } ;;

exception Queue_Vide ;;
exception Queue_Pleine ;;

let crée_queue taille initialisateur =
    let n = taille + 1
    in
    let v = make_vect n initialisateur
    and début = ref 0
    and fin = ref 0
    in
    { est_vide =
        (function () -> !début = !fin) ;
      empile =
        (function x ->  if (!fin + n - !début) mod n = taille
                        then raise Queue_Pleine
                        else    v.(!début) <- x ;
                                début := (!début + taille) mod n ) ;
      dépile =
        (function () -> if !début = !fin then raise Queue_Vide
                        else    let x= v.(!fin)
                                in
                                fin := (!fin + taille) mod n ;
                                x ) } ;;
```

On l'applique comme expliqué ci-dessus à la reconstitution d'un arbre à partir de son parcours militaire, et on obtient le programme ci-dessous.

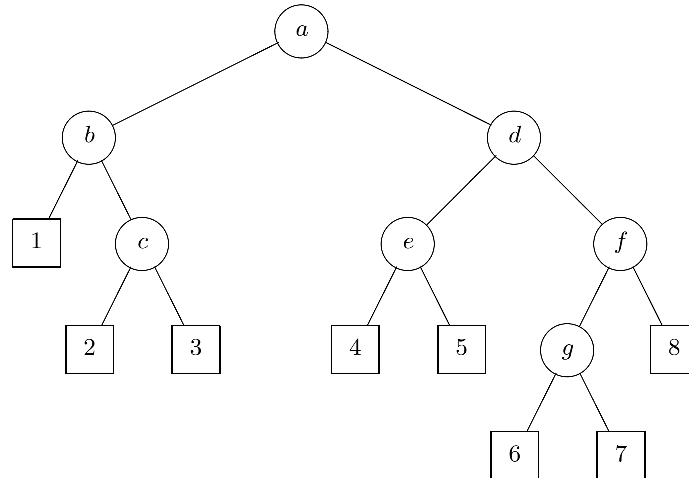
```
let recompose_militaire l feuille_bidon =
    let p = rev l
    and q = crée_queue (list_length l) (Feuille feuille_bidon)
    in
    let rec avance = function
        | [] -> let a = q.dépile ()
                in
                if q.est_vide () then a
                else failwith "Parcours militaire incorrect"
        | (F f) :: suite
            -> q.empile (Feuille f) ; avance suite
        | (N n) :: suite
            ->  let y = q.dépile ()
                in
                let x = q.dépile ()
                in
                q.empile (Nœud(n,x,y)) ; avance suite
    in
    try avance p
    with _ -> failwith "Parcours militaire incorrect" ;;
```

Le paramètre `feuille_bidon` ne sert qu'au typage de Caml qui doit initialiser ses tableaux avec quelque chose. . . C'est d'ailleurs aussi le rôle du paramètre `initialisateur` de `crée_queue`.

Exercice 2.2

Les deux conversions sont *a priori* très différentes. La conversion préfixe vers suffixe est certainement la plus simple, puisqu'il n'y a pas à revenir en arrière, quand on a trouvé un nœud, pour trouver ses fils. Pour l'arbre représenté dans la figure suivante, on va construire ce tableau :

lu	a	b	1	c	2	3	d	e	4	5	f	g	6	7	8
pile A	v	vv	vf	vfv	vff	f	fv	fvv	fvf	ff	ffv	$ffvv$	$ffvf$	fff	
écrit			1		2	$3cb$			4	$5e$			6	$7g$	$8fda$
pile B	a	ba	ba	cba	cba	a	da	eda	eda	da	fda	$gfda$	$gfda$	fda	



La pile A est une pile de v/f (qu'on pourra représenter par des booléens), la pile B est une pile de nœuds. Elles sont toujours de même taille, et d'ailleurs la pile A contient v(oid) à chaque fois qu'aucun fils n'a été complété pour le nœud correspondant de la pile B, et f(ull) quand le fils gauche est plein.
La règle de formation est alors simple : tout nœud nouvellement lu est empilé dans la pile B, alors qu'on empile un v dans la pile A ; à la lecture d'une feuille, on l'écrit aussitôt, et, si le sommet de la pile A est un v, on le remplace par un f, alors que s'il s'agit d'un f, on dépile tous les f de la pile A, et en même temps les nœuds correspondants de la pile B qu'on écrit, après quoi on transforme le v qui est au sommet de la pile A par un f, à moins qu'il n'y en ait plus, mais c'est qu'on a terminé.
Ceci correspond au programme suivant :

```
let suffixe_de_préfixe l =
    let rec avance pile_A pile_B = function
        | [] -> []
        | (N n) :: reste
           -> avance (true :: pile_A) (n :: pile_B) reste
        | (F f) :: reste
           -> match pile_A with
               | [] -> failwith "Description préfixe incorrecte"
               | true :: q_A
                   -> (F f) :: (avance (false :: q_A) pile_B reste)
               | false :: q_A
                   -> (F f) :: (vidage pile_A pile_B reste)
    and vidage pile_A pile_B reste = match pile_A,pile_B with
        | true :: q_A , n :: q_B -> avance (false :: q_A) pile_B reste
        | false :: q_A , n :: q_B -> (N n) :: (vidage q_A q_B reste)
        | [],[] -> if reste = [] then []
                   else failwith "Description préfixe incorrecte"
        | _ -> failwith "Description préfixe incorrecte"
    in
    avance [] [] l ;;
```

Pour ce qui est de la transformation inverse, le plus simple est — de très loin — de construire l'arbre à l'aide de `recompose_suffixe` puis de le parcourir en ordre préfixe. . .

Chapitre 3

Corrigé des exercices

Exercice 3.1

La seule réelle difficulté est dans la suppression de la racine d'un arbre binaire qui a deux fils non vides. Je propose dans ce cas de substituer à la racine le plus petit élément du fils droit, c'est-à-dire l'élément le plus profond à gauche de ce sous-arbre. On obtient le programme suivant :

```
let rec recherche phi arbre x = match arbre with
    | Vide -> failwith "Élément absent"
    | Nœud(a,g,d)
        -> a = x || recherche phi (if phi(x) < phi(a) then g else d) x ;;

let rec ajout phi arbre x = match arbre with
    | Vide -> Nœud(x,Vide,Vide)
    | Nœud(a,g,d)
        ->  if a = x then arbre
            else if phi(x) < phi(a)
                then Nœud(a,(ajout phi g x),d)
                else Nœud(a,g,(ajout phi d x)) ;;

let rec supprime phi arbre x =
    let rec cherche_min = function
        | Vide -> failwith "Erreur de programmation"
        | Nœud(a,Vide,d) -> a,d
        | Nœud(a,g,d) -> let mini,g' = cherche_min g
                         in
                         mini,Nœud(a,g',d)
    in
    let étête = function
        | Vide -> Vide
        | Nœud(_,Vide,Vide) -> Vide
        | Nœud(_,Vide,d) -> d
        | Nœud(_,g,Vide) -> g
        | Nœud(_,g,d) -> let min_droit,d' = cherche_min d
                         in
                         Nœud(min_droit,g,d')
    in
    match arbre with
        | Vide -> Vide
        | Nœud(a,g,d)
            ->  if a = x then étête arbre
                else if phi(x) < phi(a)
                        then Nœud(a,(supprime phi g x),d)
                        else Nœud(a,g,(supprime phi d x)) ;;
```

Exercice 3.2

```
let mesure_équilibrage arbre =
```

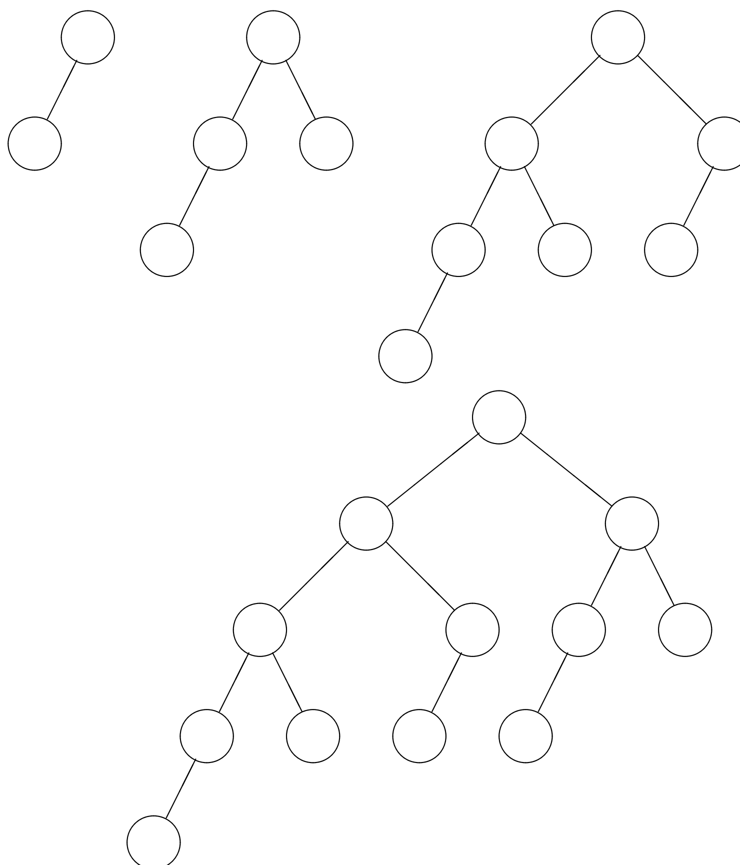
```
let rec mesure_profondeur = function
    | Vide -> Vide
    | Nœud(a,g,d)
        ->  match (mesure_profondeur g),(mesure_profondeur d) with
            | Vide,Vide -> Nœud((a,0,0),Vide,Vide)
            | Vide,(Nœud((_,kgd,kdd),gd,dd) as d)
                -> Nœud((a,0,max kgd kdd + 1),Vide,d)
            | (Nœud((_,kgg,kdg),gg,dg) as g),Vide
                -> Nœud((a,max kgg kdg + 1,0),g,Vide)
            | (Nœud((_,kgg,kdg),gg,dg) as g),
                    (Nœud((_,kgd,kdd),gd,dd) as d)
                -> Nœud((a,max kgg kdg + 1,max kgd kdd + 1),g,d)
in
let rec mesure = function
    | Vide -> Vide
    | Nœud((a,kg,kd),g,d)
        -> Nœud((a,kg-kd),(mesure g),(mesure d))
in
mesure (mesure_profondeur arbre) ;;
```

Exercice 3.3

On dispose toujours de l'inégalité $k \geq \lfloor \lg n \rfloor$, qui est valable pour tous les arbres. Les arbres complets réalisent l'égalité, et comme ils sont clairement AVL, cela prouve que cette borne est atteinte.

On cherche donc à déterminer maintenant les arbres AVL les plus profonds possible pour un n fixé, c'est-à-dire encore les plus *déséquilibrés*. À une symétrie près, il s'agit des arbres dont chaque nœud interne penche à gauche.

Voici les arbres correspondants, pour les profondeurs 1, 2, 3 et 4.



Notons F_k l'arbre de profondeur k ainsi construit. F_{k+2} se construit de la façon suivante : on crée un nouveau nœud, on lui accroche comme fils gauche F_{k+1} et F_k comme fils droit.
Ces arbres sont appelés *arbres de Fibonacci*. Si n_k est la taille de F_k, on obtient aussitôt $n_{k+2} = 1 + n_{k+1} + n_k$, avec comme conditions initiales $n_1 = 2$ et $n_2 = 4$ (on peut aussi poser $n_0 = 1$). Les $(1 + n_k)$ constituent donc la suite de Fibonacci…
On a donc $n_k = \left(1 + \frac{2\sqrt{5}}{5}\right)\varphi^k + \left(1 - \frac{2\sqrt{5}}{5}\right)(-1/\varphi)^k - 1$, ce qui fournit notre majoration asymptotique :

$$k \leq \log_\varphi n + O(1) \approx 1{,}44 \lg n.$$

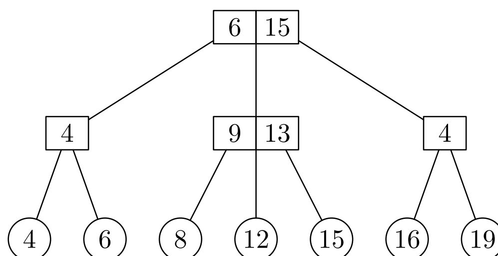
Voici, au passage, une fonction de construction des squelettes binaires de Fibonacci.

```
type squelette = Vide | Jointure of squelette * squelette ;;

let rec fibonacci = function
    | 0 -> Jointure(Vide,Vide)
    | 1 -> Jointure(Jointure(Vide,Vide),Vide)
    | n -> Jointure(fibonacci (n-1),fibonacci (n-2)) ;;
```

Exercice 3.4

Il faudra enregistrer aux feuilles qui ont trois fils deux valeurs pour discriminer ces fils. On trouvera dans la figure suivante un exemple d'arbre 2–3 de recherche.



Le type Caml correspondant est :

```
type ('f,'n) arbre23 =
    | Feuille 'f
    | Nœud2 of 'n * arbre23 * arbre23
    | Nœud3 of 'n * 'n * arbre23 * arbre23 * arbre23 ;;
```

La fonction de recherche s'écrit simplement :

```
let rec recherche phi arbre x = match arbre with
    | Feuille(f) -> x = f
    | Nœud2(a,g,d) -> recherche phi (if phi x > a then d else g) x
    | Nœud3(a,b,g,m,d)
       -> recherche phi
              (if phi x > b then d
               else if phi x > a then m
               else g) x ;;
```

Soit a un arbre 2–3 de profondeur k possédant n feuilles. Si tous ses nœuds sont à 2 fils, on sait que $n = 2^k$. De même si tous ses nœuds ont 3 fils, on aura $n = 3^k$. En général on a donc

$$0{,}63 \lg n \approx \log_3 n \leq k \leq \lg n.$$

Chapitre 5

Corrigé des exercices

Exercice 5.1

```
let recompose_suffixe_naire liste arité =
    let rec dépile k liste =
        if k = 0 then [],liste
        else match liste with
            | t :: q
                ->  let q1,q2 = dépile (k-1) q
                    in
                    t::q1 , q2
            | _ -> failwith "Description suffixe incorrecte"
    in
    let rec recompose ss_arbres liste = match ss_arbres,liste with
        | a,(F f) :: reste
            -> recompose (Feuille(f) :: a) reste
        | a,(N n) :: reste
            ->  let k = arité n
                in
                let fils,a = dépile k a
                in
                recompose (Nœud(n,fils) :: a) reste
        | [ arbre ],[]
            -> arbre
        | _ -> failwith "Description suffixe incorrecte"
    in
    recompose [] liste ;;
```

Exercice 5.2

On propose le programme suivant.

```
let rec imprime_expression expr =
    let parenthèse expr =
        print_char `(` ; imprime_expression expr ; print_char`)`
    in
    let est_atome = function
        | Constante(_) -> true
        | Variable(_) -> true
        | Application(_) -> true
        | _ -> false
    in
    match expr with
        | Constante(x) -> print_float x
        | Variable(v) -> print_char `v`
        | Application(f,u)
            ->  print_string (match f with
                                | Sin -> "sin"
```

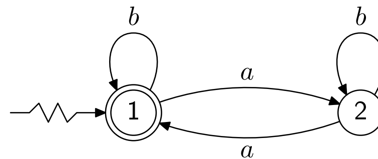
```
                        | Cos -> "cos"
                        | Exp -> "exp"
                        | Ln -> "ln") ;
        parenthèse u
| Terme(u,`^`,v)
    ->  if est_atome u then imprime_expression u
        else parenthèse u ;
        print_char `^` ;
        if est_atome v then imprime_expression v
        else parenthèse v
| Terme(u,`/`,v)
    ->  ( match u with
            | Terme(_,`+`,_) -> parenthèse u
            | Terme(_,`-`,_) -> parenthèse u
            | _ -> imprime_expression u ) ;
        print_char `/` ;
        ( match v with
            | Application(_) -> imprime_expression v
            | Constante(_) -> imprime_expression v
            | Variable(_) -> imprime_expression v
            | Terme(_,`^`,_) -> imprime_expression v
            | _ -> parenthèse v )
| Terme(u,`*`,v)
    ->  ( match u with
            | Terme(_,`+`,_) -> parenthèse u
            | Terme(_,`-`,_) -> parenthèse u
            | _ -> imprime_expression u ) ;
        print_char `*` ;
        ( match v with
            | Terme(_,`+`,_) -> parenthèse v
            | Terme(_,`-`,_) -> parenthèse v
            | _ -> imprime_expression v )
| Terme(u,`+`,v)
    ->  imprime_expression u ;
        print_char `+` ;
        imprime_expression v
| Terme(u,`-`,v)
    ->  imprime_expression u ;
        print_char `-` ;
        match v with
            | Terme(_,`-`,_) -> parenthèse v
            | _ -> imprime_expression v ;;
```

Chapitre 6

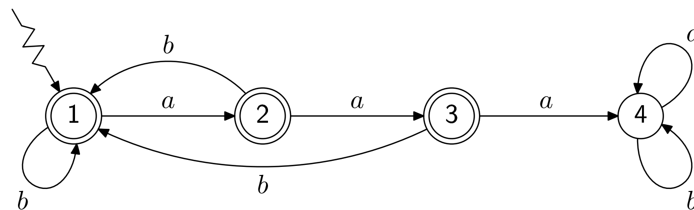
Corrigé des exercices

Exercice 6.1

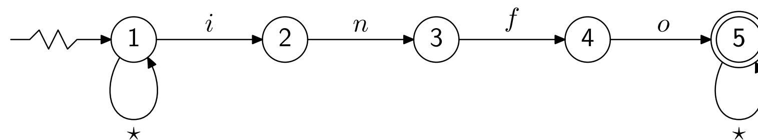
On trouve sans difficulté un automate déterministe pour le premier langage proposé :



De même, pour le second :

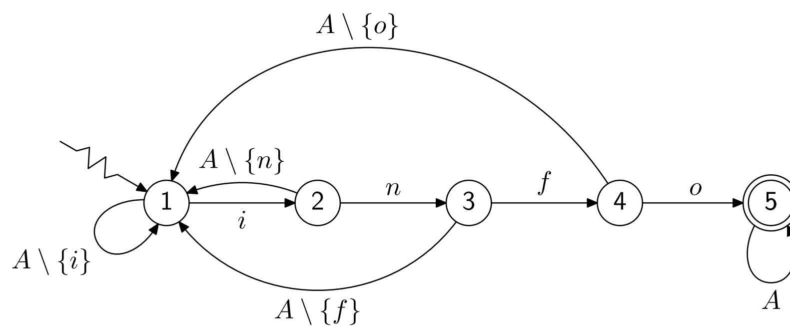


Pour le troisième, il est plus simple de fournir un automate non déterministe :

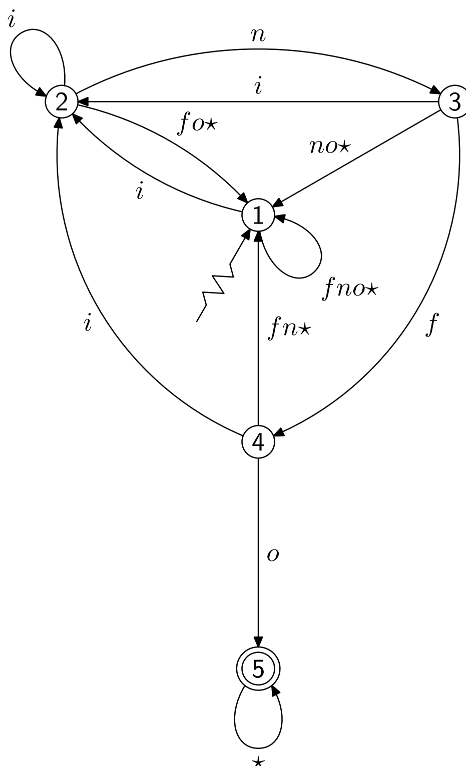


Exercice 6.2

On peut tenter une déterminisation *à la main*. Voilà ce qui fut ma première tentative :



Une étiquette comma $A \setminus \{o\}$ sur une transition signifie qu'elle a lieu sur tout caractère de l'alphabet sauf o.
Un petit peu de réflexion montre que cet automate est incorrect... On peut par exemple tester cet automate sur la chaîne $iinfo$ pour s'en convaincre.
En fait, il vaut mieux se fier aux méthodes du cours (évidemment !), et appliquer l'algorithme de déterminisation. On obtient, après simplification (regroupement des états finals), l'automate déterministe suivant :



Cette fois, l'étiquette $\star$ sur une transition signifie qu'elle a lieu sur tout caractère qui n'est pas dans $\{i, n, f, o\}$.

Exercice 6.3

Soit α l'*afnd* proposé, et β l'*afd* que construit l'algorithme. On notera $q_1 \xrightarrow{c,\alpha} q_2$ les transitions de α et $Q_1 \xrightarrow{c,\beta} Q_2$ celles de β.
La preuve de l'algorithme est alors aisée : pour chaque état Q de β il suffit de montrer par récurrence par récurrence sur la longueur d'un mot w que si $Q \xrightarrow{w,\beta} Q'$ alors $Q' = \{q' \,/\, \exists q \in Q, q \xrightarrow{w,\alpha} q'\}$.
La démonstration est claire pour $w = \varepsilon$, puisqu'on a *fermé* tous les états de β. La construction même des états de β permet de conclure quand $w = vc$: il suffit de découper w en chaîne-préfixe et caractère final.
On conclut en prenant $Q = \kappa(q_0)$, état initial de β : un mot w sera reconnu par β si et seulement si il est reconnu par α.

Exercice 6.4

$L(\beta')$ est simplement le complémentaire (dans l'ensemble de tous les mots sur l'alphabet qu'on s'est fixé) de $L(\alpha) = L(\beta)$.
En revanche, à cause de l'absence d'état mort dans α, il faut tenir compte des possibilités de blocage de la reconnaissance. Ainsi $L(\alpha')$ est-il la partie de $L(\beta')$ consistant en tous les mots qui ne provoquent pas

de blocage de α et qui ne sont pas acceptés par α. Bref il s'agit des mots w tels que $q_0 \xrightarrow{w} q$ avec q non final dans α.

Exercice 6.5

On obtient bien sûr le même automate, chaque état de l'état initial correspondant à son singleton, à ceci près qu'on a ajouté un état mort (qui correspond à un ensemble vide d'états de l'automate initial).

Exercice 6.6

Nous utiliserons ici une nouvelle représentation des *afd*.
Un automate sera représenté par un vecteur d'états, chaque état étant un objet Caml du type suivant :

```
type état_afd_vectoriel == bool * (int vect) ;;
```

Le booléen dit si l'état est terminal, et le vecteur décrit les transitions sur les 256 caractères : une transition est représentée par le numéro de l'état résultat.
La partition de l'ensemble des états sera représentée par un vecteur de listes d'entiers : chacune contient les numéros des états de la partie considérée de la partition.
La fonction suivante partage les états de l'automate en finals et non-finals.

```
let rec isole_finals qv i n =
    if i = n then [],[]
    else let finals,non_finals = isole_finals qv (i + 1) n
         in
         if fst qv.(i)  then (i :: finals),non_finals
                        else finals,(i :: non_finals) ;;
```

Elle prend en argument le vecteur des états, `i` qui doit valoir 0 pour tout balayer, et la taille du vecteur. Elle renvoie un couple de listes.
La fonction suivante renvoie le numéro du *paquet* de la partition qui contient un état donné.

```
exception Trouvé of int ;;

let numéro_partie q partition =
    let n = vect_length partition
    in
    try
        for i = 0 to n - 1 do
            if mem q partition.(i) then raise (Trouvé(i))
        done ;
        n
    with Trouvé(i) -> i ;;
```

Voici maintenant une fonction plus compliquée :

```
let répartit l =
    let rec ajoute (q,dest) = function
        | [] -> [ dest , [ q ] ]
        | (u,l) :: r when dest = u -> (u,q::l) :: r
        | (u,l) :: r -> (u,l) :: (ajoute (q,dest) r)
    in
    let rec rép l res =
        match l with
            | [] -> map snd res
            | (q,dest) :: r -> rép r (ajoute (q,dest) res)
    in
    rép l [] ;;
```

Elle prend en argument une liste de couples `(q,dest)` dont le premier élément est un état de l'automate et le second l'état auquel une transition le conduit. Elle renvoie une liste de couples `(dest,liste)` dont le premier élément est un état-cible et le second est la liste des éléments `q` de départ dans la liste fournie en argument.
On est alors en mesure d'écrire le cœur de l'algorithme.

```
let calcule_minimal qv =
    let n = vect_length qv
    in
    let partition = make_vect n []
    and taille_partition = ref 2
    and encore = ref true
    in
    let partage i ll =
        let nll = list_length ll
        in
        let rec rangement ll k = match ll with
            | [] -> ()
            | l :: ql -> partition.(k) <- l ; rangement ql (k + 1)
        in
        partition.(i) <- hd ll ;
        rangement (tl ll) !taille_partition ;
        taille_partition := !taille_partition + nll - 1 ;
        if nll > 1 then raise ÀSuivre else ()
    in
    ( match isole_finals qv 0 n with
        | [],l -> ( partition.(0) <- l ; taille_partition := 1 )
        | l,[] -> ( partition.(0) <- l ; taille_partition := 1 )
        | l1,l2 ->( partition.(0) <- l1 ; partition.(1) <- l2  ) ) ;
    while !encore do
        try
            encore := false ;
            for i = 0 to !taille_partition - 1 do
            if list_length partition.(i) > 1 then
                for c = 0 to 255 do
                    let l = map
                            (function q
                                -> q,
                                (numéro_partie (snd qv.(q)).(c) partition))
                            partition.(i)
                    in
                    partage i (répartit l) ;
                done
            done
        with ÀSuivre -> encore := true
    done ;
    ( partition,!taille_partition ) ;;
```

La fonction `partage` prend la liste créée par la fonction `répartit` décrite ci-dessus, et modifie en conséquence la partition. Il n'y a presque plus rien à faire pour terminer :

```
let minimise automate =
    let n = vect_length automate
    and partition,k = calcule_minimal automate
    in
    let mini = make_vect k (true,[| |])
    in
    for i = 0 to k - 1  do
        let x = automate.(hd partition.(i))
        in
        mini.(i) <- (fst x),(make_vect 256 0) ;
        match mini.(i) with _,v ->  for c = 0 to 255 do
                                         v.(c) <- numéro_partie (snd x).(c) partition
                                    done
    done ;
    mini ;;
```

Chapitre 7

Corrigé des exercices

Exercice 7.1

On a déjà répondu à la question pour le complémentaire dans l'exercice 6.4 : on a su construire un *afd* qui reconnaît le complémentaire du langage d'un *afd* fixé.
On sait que l'union de deux langages rationnels est un langage rationnel, passant aux complémentaires, on en déduit le résultat sur l'intersection (même s'il n'est pas évident de construire l'automate de l'intersection...)

Exercice 7.2

Il s'agit à chaque fois de prouver l'égalité de deux ensembles, donc deux inclusions. On pourra procéder par récurrence sur la taille des mots. On montre de façon analogue : $(e_1|e_2)\star \equiv (e_2\star e_1)\star e_2\star$.

Exercice 7.3

Il suffit de considérer un automate fini déterministe qui reconnaît le langage, et de considérer tous les états à la fois comme initiaux et finals.

Exercice 7.4

Les deux automates ayant la même transition de l'état 2 vers l'état 3, on peut se contenter de montrer l'équivalence des automates déduits de ceux proposés par suppression du dernier état.
On écrit les expressions régulières correspondantes à ces automates. On trouve facilement $(a|b)\star a$ pour le premier : ce sont là les mots qui se terminent par un a. Pour le second, on trouve une expression plus compliquée :

$$b\star a(a|bb\star a)\star.$$

En utilisant les règles exposées dans l'exercice 7.2, on écrit successivement :

$$\begin{aligned}
b\star a(a|bb\star a)\star &\equiv b\star a|b\star a(a|bb\star a)\star(a|bb\star a) \\
&\equiv b\star(\varepsilon|a(a|bb\star a)\star(\varepsilon|bb\star))a \\
&\equiv b\star(\varepsilon|a(a|bb\star a)\star b\star)a \\
&\equiv b\star(\varepsilon|a((\varepsilon|bb\star)a)\star b\star)a \\
&\equiv b\star(\varepsilon|a(b\star a)\star b\star)a \\
&\equiv b\star(\varepsilon|a(a|b)\star)a
\end{aligned}$$

On conclut en remarquant que :

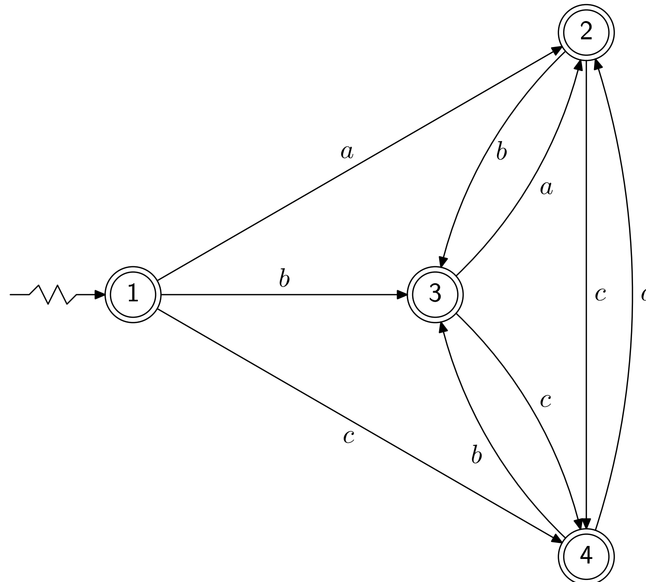
$$\begin{aligned}(a|b)\star a &\equiv b\star(ab\star)\star a\\ &\equiv b\star(\varepsilon|ab\star(ab\star)\star)a\\ &\equiv b\star(\varepsilon|a(a|b)\star)a\end{aligned}$$

Exercice 7.5

Utilisons le lemme de l'étoile : il existe un entier m qui vérifie les propriétés du lemme. Soit $n > m$, $a^n ba^n$ est reconnu, donc s'écrit sous la forme rst où s est de longueur au plus égale à m et où les mots $rs\star t$ sont reconnus. Si s contient un b, on trouverait des mots du langage avec plusieurs b. Sinon, $r = a^k$, $s = a^p$, et $t = a^{n-k-p}ba^n$. Mais alors $rsst = a^{n+p}ba^n$ serait dans le langage.
Pour le deuxième langage, procédons ainsi : soit m le nombre d'états d'un *afd* qui reconnaît notre langage, et soit q_0 son état initial. Appelons q_i l'état défini par $q_0 \xrightarrow{(ab)^i} q_i$ (pas de problème car $(ab)^i(ba)^i$ est bien reconnu). Nécessairement, on peut trouver $0 \leq i < j \leq m$ tels que $q_i = q_j$. Alors $q_0 \xrightarrow{(ab)^{i+k(j-i)}} q_i = q_j$ pour tout entier k. Alors tous les mots $(ab)^{i+k(j-i)}(ba)^i$ sont reconnus, puisque $q_i \xrightarrow{(ba)^i} q_f$ avec q_f final. Ces mots n'étant pas de la forme requise pour $k \geq 1$, notre langage n'est pas rationnel.

Exercice 7.6

Pour le premier langage : $(a|b)\star a(a|b)\star$.
Pour le deuxième : $b\star(a|\varepsilon)b\star$.
Pour le troisième : $(b|aa)\star$.
Pour le quatrième, on peut commencer par construire l'automate correspondant :



On utilisera alors la méthode du cours pour obtenir l'expression régulière cherchée. Mais il est tout aussi simple de tout faire *à la main*.
Appelons $\overline{\mathbf{A}}$ le motif qui décrit tous les mots de notre langages qui ne contiennent pas de a. Il est facile à expliciter :

$$\overline{\mathbf{A}} = (\varepsilon|c)(bc)\star(\varepsilon|b).$$

Il n'est alors pas difficile d'écrire une expression régulière solution de notre problème :

$$(\varepsilon|\overline{\mathbf{A}})(a\overline{\mathbf{A}})\star(\varepsilon|a).$$

On aura noté qu'il s'agit de la même expression à substitution près de c en $\overline{\mathbf{A}}$ et de b en a : on pourrait ainsi généraliser au langage des mots écrits sur n lettres sans répétition.