Informatique et algorithmique

JPV

25 juin 2004

Table des matières

1 Introduction **6**
1.1 Algorithmes . . . 6
1.2 Que signifie *Caml* ? . . . 7
1.3 Introduction à *Caml* . . . 7
1.3.1 Définition locale . . . 8
1.3.2 Fonctions . . . 8
1.3.3 Fonctions de plusieurs variables . . . 8
1.3.4 Fonctions sans variables . . . 9
1.3.5 Fonctions anonymes . . . 9
1.3.6 Types . . . 9
1.3.7 Fonctions anonymes à plusieurs variables . . . 10
1.3.8 Tests . . . 10
1.3.9 Valeurs de vérité . . . 10
1.3.10 Valeurs et programmes . . . 10
1.3.11 Impression . . . 10
1.3.12 Syntaxe . . . 11
1.3.13 Diagramme syntaxique . . . 12

2 Survol de Caml **13**
2.1 Phrases . . . 13
2.2 Références . . . 13
2.3 Vecteurs et tableaux . . . 14
2.4 Fonctions et procédures . . . 14
2.5 Fonctions . . . 15
2.6 Fonctionnelles . . . 15
2.7 Symboles, séparateurs, identificateurs . . . 16
2.8 Types de base . . . 16
2.9 Expressions . . . 17
2.10 Blocs et portée des variables . . . 18
2.11 Débogage . . . 19
2.12 Instructions . . . 19
2.13 Filtrage . . . 20

2.14 Boucles . . . 20
2.15 Exceptions . . . 20
2.16 Entrées – sorties . . . 21
2.16.1 Impressions simples . . . 21
2.16.2 Lecture et écriture sur fichiers . . . 21
2.17 Copie de fichiers . . . 21
2.18 Définitions de types . . . 21
2.19 Modules . . . 21
2.20 Bibliothèques . . . 21
2.21 Fonctions graphiques . . . 21

3 Itération **22**
3.1 Boucles . . . 22
3.2 Exemples . . . 22
3.3 Polynômes pleins . . . 23
3.3.1 Manipulation de polynômes . . . 23
3.4 Preuves de programmes . . . 25
3.4.1 La correction partielle . . . 25
3.4.2 Preuve d'arrêt . . . 25
3.4.3 Exemples . . . 25

4 Récursivité **27**
4.1 Exemple : exponentiation rapide . . . 27
4.2 Récursivité mutuelle . . . 28
4.3 Récursivité terminale . . . 29
4.4 Preuves de terminaison . . . 30

5 Diviser pour régner **32**
5.1 Puissance rapide . . . 32
5.2 Produits de polynômes . . . 32

6 Listes **34**
6.1 Suites ordonnées . . . 34
6.2 Listes chaînées . . . 34
6.2.1 Recherche dans une liste . . . 37
6.2.2 Insérer ou supprimer un élément dans une liste . . . 38
6.3 Tris . . . 38
6.3.1 Tri insertion . . . 39
6.3.2 Tri sélection . . . 39
6.3.3 Tri rapide . . . 39
6.3.4 Tri bulle . . . 40
6.3.5 Tri fusion . . . 41

7 Piles **44**
7.1 Évaluation des formules postfixes . . . 44

8 Logique **45**
8.1 Rudiments de logique booléenne . . . 45
8.1.1 Tables de vérité . . . 45
8.1.2 Tautologies . . . 46
8.1.3 Fonctions booléennes . . . 47
8.2 Circuits logiques . . . 48
8.2.1 Exemple de construction de circuit . . . 48

9 Arbres **52**
9.1 Généralités . . . 52
9.2 Arbres binaires . . . 52
9.2.1 Opérations élémentaires sur les arbres binaires hétérogènes . . 55
9.2.2 Parcours d'un arbre binaire hétérogène complet . . . 55
9.3 Arbres binaires homogènes . . . 56
9.3.1 Accès aux nœuds d'un arbre binaire homogène . . . 56
9.3.2 Opérations sur les arbres binaires homogènes . . . 59
9.4 Arbres binaires de recherche . . . 59
9.4.1 Test d'un arbre binaire de recherche . . . 59
9.4.2 Recherche dans un arbre binaire de recherche . . . 61
9.4.3 Insertion dans un arbre binaire de recherche . . . 61
9.4.4 Suppression dans un arbre binaire de recherche . . . 62
9.5 Files de priorité et tas . . . 65
9.5.1 Files de priorité . . . 65
9.5.2 Tas . . . 67
9.5.3 Transformation de tas en arbres . . . 68
9.5.4 Suppression de la racine dans un tas . . . 68
9.5.5 Insertion dans un tas . . . 69
9.6 Programme de représentation des arbres . . . 70
9.6.1 Transformations diverses . . . 70
9.6.2 Programme Caml d'affichage . . . 71

10 Étude de la complexité de quelques tris **79**
10.1 Méthodes par sélection . . . 79
10.1.1 Le tri sélection . . . 79
10.1.2 Le tri à bulles . . . 79
10.2 Méthodes par insertion . . . 79
10.2.1 Le tri insertion . . . 79
10.2.2 L'insertion dichotomique . . . 80
10.3 Le tri rapide . . . 80
10.4 Le tri fusion . . . 80
10.5 Tri en tas . . . 81
10.6 Arbres de décision . . . 81

11 Langages et automates **82**
11.1 Introduction 82
11.2 Automates finis 82
11.2.1 Définition 82
11.2.2 Interprétation 83
11.2.3 Diagramme d'un automate 84
11.2.4 Implémentation d'un automate en Caml 84
11.3 Automates non déterministes 85
11.3.1 Généralités 85
11.3.2 Exemple de déterminisation 87
11.3.3 Simulation d'automates non déterministes 88
11.3.4 Transitions instantanées 88
11.4 Automates et langages 89
11.4.1 Opérations sur les langages 89
11.4.2 Expressions rationnelles 90
11.4.3 Langages rationnels 90
11.4.4 Lemme de l'étoile 93
11.4.5 Limitation des automates finis 94
11.4.6 Calcul d'un automate à partir d'une expression rationnelle . . 94
11.4.7 Exemple de calcul d'une expression rationnelle 97
11.5 Compléments 98
11.5.1 Constructions d'automates 98
11.5.2 Minimisation 98

12 Analyses lexicale, syntaxique, sémantique **103**
12.1 Termes 103
12.2 Sémantique 103
12.3 Syntaxes 104
12.4 Exemple : dérivation formelle 104

Table des figures

1 Structures des tableaux et des listes 34
2 Représentation en mémoire d'un vecteur et d'une liste chaînée 35
3 Liste chaînée, en peigne 35
4 Circuits élémentaires 49
5 Additionneur 1 bits 51
6 Représentation d'un arbre (non binaire) 53
7 Deux arbres binaires à ne pas confondre 53
8 L'arbre binaire de recherche est à gauche 59
9 Suppression d'un nœud 63
10 Suppression de la racine 64
11 Un tas 68
12 Parcours en largeur 70
13 mesures dans l'arbre 72

14	Parcours en profondeur à gauche	75
15	Un nœud pair est un fils gauche	76
16	Représentation d'un automate	83
17	Diagramme d'un automate	84
18	Diagramme d'un automate	85
19	Automate non déterministe	86
20	Automate déterministe $M^\star$	87
21	ANFDε.	89
22	Automate déterminisé.	89
23	Reconnait le langage vide.	90
24	Reconnait $\{\varepsilon\}$.	90
25	Reconnait $\{a\}$.	91
26	Automate de Thompson.	91
27	Automate reconnaissant $L_1\|L_2$.	91
28	Automate reconnaissant $L_1.L_2$.	92
29	Automate reconnaissant $L_1^\star$.	92
30	Reconnaît a	94
31	Reconnaît (ba)	95
32	Reconnaît $(a\|ba)$	95
33	Reconnaît $(a\|ba)$ (forme simplifiée)	95
34	Reconnaît (bb)	95
35	Reconnaît (ab)	95
36	Reconnaît $(a\|ab)$	95
37	Reconnaît $(ba\|a)^\star$	96
38	Reconnaît $(a\|ab)^\star$	96
39	Reconnaît $(ba\|a)^\star bb(a\|ab)^\star$	96
40	Automate déterministe reconnaissant $(ba\|a)^\star bb(a\|ab)^\star$	97
41	Automate déterministe.	97
42	t	103
43	t	105

1 Introduction

1.1 Algorithmes

Le mot « algorithme » a pour origine Al Khorawizmi, un mathématicien...

Théorème 1.1 *Un* Algorithme *est la composée d'un nombre fini d'applications.*

Il faut entendre ces mots au sens mathématique. Cependant, il y a plusieurs niveaux, par exemple :
$ax^2 + bx + c$ est un polynôme à coefficients réels dont on veut calculer les racines.

si $b^2 - 4ac >= 0$ **alors**

$$x' = \frac{-b + \sqrt{\Delta}}{2a},$$

$$x'' = \frac{-b - \sqrt{\Delta}}{2a}$$

sinon

$$x' = \frac{-b + i\sqrt{-\Delta}}{2a},$$

$$x'' = \frac{-b - i\sqrt{-\Delta}}{2a}$$

fin si

Il est supposé que l'on sait calculer, par ailleurs, les racines réelles ou complexes. Mais que signifie « calculer » ? Est-ce caractériser le nombre que l'on cherche (résultat symbolique) ou en donner une approximation décimale à une précision donnée ? Dans le premier cas, cet algorithme nous conviendra, dans le second cas il faudra faire appel à un autre algorithme qui calculera cette approximation.

Il apparaît que dans la définition d'algorithme doit se trouver la notion d'*effectivité*. Autrement dit, on demande à ce que les calculs donnent un résultat et soient effectués en un temps fini raisonnable : une fraction de seconde, quelques jours. Le résultat doit arriver à une date à laquelle il sera encore considéré comme utile.

Exemple 1.1 Calcul de la valeur de la fonction polynôme $f : x \mapsto 2x^2 + 3x + 4$ en un r réel.

Algorithme 1

lire r
calculer $(2r + 3)r + 4$

Algorithme 2

$a_0 \leftarrow 4$, $a_1 \leftarrow 3$, $a_2 \leftarrow 2$, $x \leftarrow r$ « lire x et $a = [a_0, a_1, a_2]$ »
$p \leftarrow 0$
pour k de 2 à 0 **faire**
    $p \leftarrow (p + a_k)x$
fin pour

1.2 Que signifie *Caml* ?

Paraphrasons la FAQ (d'après Pierre Weiss), « Caml » est un acronyme en langue anglaise de *Categorical Abstract Machine Language*, c'est-à-dire *langage de la machine abstraite catégorique* (donc *Lmac* en français). La CAM est une machine abstraite capable de définir et d'exécuter les fonctions, et qui est issue de considérations théoriques sur les relations entre la théorie des *catégories* et le *lambda-calcul*. Le premier compilateur du langage générait du code pour cette machine abstraite (en 1984). La deuxième filiation de ce nom est ML (acronyme pour *Meta Language*) : *Caml* est aussi issu de ce langage de programmation crée par Robin Milner en 1978, et qui servait à programmer les tactiques de preuves dans le système de preuves LCF.

Caml est un langage *fonctionnel* fortement *typé*. Nous reviendrons plus loin sur ces mots.

1.3 Introduction à *Caml*

La version du langage *Caml* qui est au programme de l'option est *Caml Light*. C'est *Objective Caml* qui est installé sur le serveur Linux.

Si l'on travaille avec Linux, on lance ocamlbrowser puis le Shell dans le menu File. On peut également lancer `ocaml` dans un terminal.

L'invite est représentée par le symbole `#`. *Caml* est un langage *compilé*, autrement dit le fichier source que l'on écrit, en respectant un certain vocabulaire et une certaine syntaxe, est traduit en langage machine par un compilateur. Un compilateur est un programme qui effectue cette opération. Le fichier compilé est aussi appelé fichier *exécutable*. Cependant *Caml* a aussi un mode de fonctionnement qui le fait ressembler à un language *interprété* : une ligne de commandes, une fois validée par un « retour chariot », sera immédiatement compilée et le résultat sera affiché. Un langage interprété à un fonctionnement similaire en apparence (le fichier n'est pas traduit en langage machine).

Ainsi il est possible d'utiliser *Caml* de façon interactive. Pour obtenir la valeur de la somme $1 + 2$, il suffit d'écrire `1+2` en ligne de commande suivi de `;;` qui marque la fin de la commande.

```
# 1+2;;
- : int = 3
```

Le mot `int` signifie que le résultat est de type entier (*integer*). En effet *Caml* est fortement typé : chaque élément du langage appartient à un ensemble défini par le type. Dans l'exemple ci-dessus, 1 et 2 sont des entiers. Pour faire un calcul en nombres flottants, il faudrait écrire `1.+.2.`, en *Caml* le typage est traité par la machine.

Pour retenir le résultat d'un calcul il faut le nommer, on dit que l'on affecte le résultat à une variable. Dans l'exemple suivant la variable appelée s aura la valeur 6.

```
# let s = 1+2+3;;
  val s : int = 6
```

Ainsi

```
# let carre = 2*s;;
val carre : int = 12
```

1.3.1 Définition locale

La valeur d'une variable peut-être modifiée localement (dans une phrase) :

```
# let x = 3;;
val x : int = 3
# let x = 2 in x*x;;
- : int = 4
# x;;
- : int = 3
```

La valeur de x n'a pas été changée.

1.3.2 Fonctions

Il y a plusieurs façons de définir les fonctions.

```
# let carre (x) = x*x;;
val carre : int -> int = <fun>
```

Dans cette dernière phrase, **:** et = sont des délimiteurs : `carre` associe un entier à un entier et est du type fonction.

On peut supprimer les parenthèses :

```
# let carre x = x*x;;
```

Il est possible de définir une fonction *localement* :

```
# let inconnue x = 2*x;;
val inconnue : int -> int = <fun>
# let inconnue x = 3*x in 5*(inconnue 4);;
- : int = 60
```

1.3.3 Fonctions de plusieurs variables

Il est possible de définir des fonctions d'un nombre quelconque de variables.

```
# let moyenne x y = (x+y)/2;;
val moyenne : int -> int -> int = <fun>
# moyenne 3 8;;
- : int = 5
```

La valeur est entière approchée. En flotants :

```
# let moyenne2 x y = (x+.y) /. 2.;;
val moyenne2 : float -> float -> float = <fun>
# moyenne2 3. 8.;;
- : float = 5.5
```

Ces fonctions sont dites *curryfiées*, par opposition aux fonctions définies sur des n-uplets :

```
# let somme_carres (x,y) = x*x+y*y ;;
val carre_norme : int * int -> int = <fun>
# somme_carres(3,4);;
- : int = 25
```

1.3.4 Fonctions sans variables

On doit utiliser `()` :

```
# let salut () = print_string "Bonjour";;
val salut : unit -> unit = <fun>
# salut();;
Bonjour- : unit = ()
```

1.3.5 Fonctions anonymes

On peut utiliser des fonctions sans nom :

```
# (function x -> x+2);;
- : int -> int = <fun>
# (function x -> x+2) 3;;
- : int = 5
```

Mais rien n'empêche de les nommer :

```
# let translation = (function x -> x+2);;
val translation : int -> int = <fun>
# translation 3;;
- : int = 5
```

1.3.6 Types

On peut préciser le type :

```
# ("Caml" : string);;
- : string = "Caml"
```

```
# let ( successeur : int -> int ) = function x -> x+1 ;;
val successeur : int -> int = <fun>
```

équivaut à **let** successeur x = x+1;;. En flottants :

```
# let ( successeur2 : float -> float ) = function x -> x+.1. ;;
val successeur : float -> float = <fun>
```

1.3.7 Fonctions anonymes à plusieurs variables

Il faut déclarer chaque argument avec `function` :

```
# (function x -> function y -> (x+y)/2);;
- : int -> int -> int = <fun>
```

1.3.8 Tests

```
# let valeur_absolue x = if x >= 0 then x else -x ;;
val valeur_absolue : int -> int = <fun>
# valeur_absolue -9;;
Characters 0-14:
This expression has type int -> int but is here used with type int
# valeur_absolue(-9);;
- : int = 9
```

Comme quoi, parfois, les parenthèses sont désirées.

1.3.9 Valeurs de vérité

Ou expressions booléennes :

```
# 2 < 1;;
- : bool = false
# sqrt(3.) < 2.;;
- : bool = true
```

1.3.10 Valeurs et programmes

Un programme est une fonction.

```
# let carre x = x*x;;
val carre : int -> int = <fun>
# let somme_des_carres x y = carre x + carre y ;;
val somme_des_carres : int -> int -> int = <fun>
# somme_des_carres 3 4 ;;
- : int = 25
```

1.3.11 Impression

Un *effet* est une action sur le monde extérieur (au langage, au compilateur), par exemple la fonction *primitive* (ou prédéfinie) `print_string` :

```
  # print_string "Bonjour";;
Bonjour- : unit = ()
```

Le résultat est de type unit et vaut () c'est à dire « rien » car le résultat est « extérieur ». Le symbole () signale que le travail a été fait. On dit que le résultat est *ignoré* (sous entendu que *Caml* ne fait rien d'autre que le renvoyer).

Dans la même veine, si deux résultats (ou plus) se suivent, seul le dernier est pris en compte, le précédent est oublié (ignoré).

Table de 2 :

```
# 2*1; 2*2; 2*3;;
Characters 0-3:
Warning: this expression should have type unit.
Characters 5-8:
Warning: this expression should have type unit.
- : int = 6
```

De même :

```
# print_string "Bonjour␣"; print_string "tout␣le␣monde";;
Bonjour tout le monde- : unit = ()
```

Il n'y a ici aucun message d'erreur car tout est de type unit donc comme c'est « rien », « rien » ne peut faire d'erreur !

Remarque 1.1 *Pour délimiter une suite d'instructions, on met* `begin` *et* `end;;`.

1.3.12 Syntaxe

`f x`	⇔	`f(x)`
`let f x`	⇔	`let f = function x ->`
`let f x y`	⇔	`let f = function x -> function y ->`
`f x+ g y`	⇔	`f(x) + g(y)`
`f x y`	⇔	`(f x) y`

Donc f −1 n'est pas égal à f(−1).
De manière générale, la priorité part de la gauche.

```
# let moyenne x y = (x+y)/2;;
val moyenne : int -> int -> int = <fun>
# moyenne (2 4);;
Characters 9-10:
This expression is not a function, it cannot be applied
```

`moyenne (2 4)` est lu comme la composée de `moyenne` et de 2 or 2 n'est pas une fonction ! De plus, si f et g sont des fonctions, on vérifie que `(f g) x` n'est pas égal à `f (g x)`.

```
# let f x = x*x;;
val f : int -> int = <fun>
# let g x = 2*x;;
val g : int -> int = <fun>
# (f g) 3;;
Characters 3-4:
This expression has type int -> int but is here used with type int
# f g 3;;
Characters 0-1:
This function is applied to too many arguments
# f(g 3);;
- : int = 36
```

1.3.13 Diagramme syntaxique

Le diagramme

```
expression ::= entier
             | chaîne de caractères
             | booléen
```

signifie que `expression` est un entier ou une chaîne de caractères ou un booléen.

Une expression qui est une suite peut se définir par

```
séquence ::= expression1 ; expression2
```

ou

```
séquence ::= begin expression1 ; expression2 end
```

Cette méthode est appelée BNF pour John Backus - Peter Naur - Form.

2 Survol de Caml

2.1 Phrases

Une phrase en *Caml* est une suite de *définitions*, de *procédures* ou de *fonctions*.

Une *définition* est introduite par le mot clé (ou mot réservé) `let` suivi du nom :

```
# let salut = "Bonjour";;
val salut : string = "Bonjour"
# salut;;
- : string = "Bonjour"
# let salut = "Bonjourno";;
val salut : string = "Bonjourno"
# salut;;
- : string = "Bonjourno"
# salut := "Buenos␣dias";;
Characters 0-5:
This expression has type string but is here used with type string ref
```

On ne peut modifier la valeur d'une variable de *Caml* sans la redéfinir.

Le symbole `:=` est celui de l'affectation (modification de la valeur de la variable).

2.2 Références

Il existe des variables que l'on peut modifier, elles sont appelées *références*.

```
# let salut2 = ref "Bonjour";;
val salut2 : string ref = {contents="Bonjour"}
```

`salut2` est une référence (une « case mémoire ») dont le contenu est la chaîne « Bonjour ».

```
# salut2 := !salut2 ^ "␣vous";;
- : unit = ()
# salut2;;
- : string ref = {contents="Bonjour␣vous"}
```

Ici `salut2` est défini comme étant une *référence*. On peut modifier `salut2` avec `:=` en *déréférençant* `salut2` avec le symbole `!`. Ainsi on distingue le nom de la variable (case mémoire) et sa valeur (son contenu). Dans cet exemple, on construit une nouvelle chaîne de caractères en juxtaposant deux chaînes à l'aide de l'opérateur de *concaténation* : `^`.

La commande

```
# salut2 := "Buenos␣dias";;
```

est autorisée.

L'opérateur ^ est réservé aux références.

2.3 Vecteurs et tableaux

Un *vecteur* est une suite (indexée) de références dont les indices commencent à 0.

```
# let v = Array.make 3 1;;
val v : int array = [|1; 1; 1|]
# v.(0) <- 21;;
- : unit = ()
# v;;
- : int array = [|21; 1; 1|]
```

On accède à la j^{e}coordonnées du vecteur v par `v.(j)`, et on la modifie avec le symbole `<-`.

Remarque 2.1 *Dans certaines versions on trouve la primitive* `make_vect` *à la place de* `Array.make`*. L'usage de* `Array.` *permet d'utiliser la primitive* `make` *du module* `Array` *sans charger toutes les fonctions du module.*

On définit un tableau à deux indices avec

```
# let m = Array.make_matrix 3 2 0;;
val m : int array array = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]
```

(trois lignes, deux colonnes, les coefficients sont initialisés à 0).

On modifie un coefficient par

```
# m.(2).(1) <- 1;;
- : unit = ()
```

En *Caml* les tableaux sont des vecteurs de vecteurs.

La taille d'un tableau est définie lors de sa création et n'est pas modifiable.

2.4 Fonctions et procédures

Les auteurs de *Caml* font une distinction entre *fonction* et *procédure* qui me semble un peut obscure : une procédure serait une fonction qui prend en argument, ou retourne une valeur, sans intérêt. En fait sans intérêt signifie de type `unit`. Le compilateur ne fait pas de distinction.

Les fonctions sont des valeurs comme les autres, *Caml* ne fait pas de différence entre les objets *simples* et les objets *complexes*.

```
# let g x = 3* f x;;
val g : int -> int = <fun>
# g 7;;
- : int = 42
```

Avec des notations mathématiques : $f : x \mapsto 2x$ et $g(x) = 3f(x)$. Les parenthèses ne sont pas obligatoires, il est d'usage de ne pas écrire les parenthèses superflues.

Remarquons que le compilateur prend en charge la définition des types. L'usage est donc de ne pas préciser le typage :

```
# let pair x = x mod 2 =0;;
val pair : int -> bool = <fun>
# let impair (x : int) = (x mod 2 = 1 : bool);;
val impair : int -> bool = <fun>
```

On écrira les programmes comme `pair` plutôt que comme `impair`.

2.5 Fonctions

Nous avons déjà vu les fonctions anonymes, les fonctions *normales*, il y a aussi les fonctions récursives (exemple de la factorielle) :

```
# let rec factorielle x = if x < 2 then 1 else x*factorielle (x - 1);;
val factorielle : int -> int = <fun>
# factorielle 5;;
- : int = 120
```

La fonction `factorielle` fait appel à elle-même. Cette construction doit utiliser le mot-clé **`rec`**.

2.6 Fonctionnelles

Une fonctionnelle est une fonction qui prend des fonctions en argument.

```
# let h f g x = if x mod 2 = 1 then f x else g x ;;
```

Ici le compilateur comprend que f et g sont des fonctions (non encore définies), par conséquent `h f g` est la composée $h \circ f \circ g$. Dans ce cas la fonction h est mal définie.

Sous la forme suivante, h est fonction des fonctions f et g et de l'entier x :

```
# let h (f, g, x) = if x mod 2 = 1 then f x else g x ;;
val h : (int -> 'a) * (int -> 'a) * int -> 'a = <fun>
# h((function x->3*x+1),(function x->x/2), 5);;
- : int = 16
```

ou

```
# let f = function x -> 3*x + 1;;
val f : int -> int = <fun>
# let g = function x -> x / 2;;
val g : int -> int = <fun>
# h(f,g,4);;
- : int = 2
```

Autre version :

```
# let q x = 4*x+2;;
val q : int -> int = <fun>
# let p x = 2*x+4;;
val p : int -> int = <fun>
# let h (f, g, x) = if x mod 2 = 1 then f x else g x ;;
```

```
val h : (int -> 'a) * (int -> 'a) * int -> 'a = <fun>
# h(p,q,4);;
- : int = 18
```

2.7 Symboles, séparateurs, identificateurs

Les identificateurs sont des suites de lettres, chiffres, apostrophes, soulignés, commençant par une lettre et séparés par des espaces, tabulations, retours à la ligne, caractères spéciaux (+,-,*). Certain identificateurs sont réservés, par exemple les mots clés de la syntaxe (`and, type...`).

Par convention, les constantes commencent par une majuscule et les variables par une minuscule (Pascal, C). En Caml toutes les variables ont une valeur constante. Il est d'usage de faire commencer les variables par une minuscule.

2.8 Types de base

La valeur *rien* le type prédéfini `unit` et est notée `()`. On lit *void* (c'est de l'anglais, lire *voïde*). Exemples d'emploi : `else ()` ou `print_newline ()`.

Les *booléens* ont le type `bool` qui contient deux constantes : `true` et `false`.

Les *entiers* ont le type `int` : `-1514`, 1788.

Les *flottants* ont le type `float` : `6.5596`, `15244832.E-8`.

Les *caractères* ont le type `char`. Ce sont `'a', 'b'...,'z'`, les symboles comme `'+'`, `'\\'` qui désigne `'\'`, `'\r'` le « retour chariot », `'\n'` le changement de ligne *etc... tous entourés par des apostrophes*. On peut appeler un caractère par son code de *American Standard Codes for Information Interchange*, par exemple le code de `é` est `'\233'`. La fonction `int_of_char` donne le code d'un caratère et `char_of_int` est la fonction réciproque.

Les *chaînes de caractères* ont le type `string`, ce sont les suites de caractères entourées de guillemets américains ", le caractère « guillemet » se note `\"`. Si `c` est une chaîne de caractères, le *i*^e^caractère est `s.[i]` et *i* varie à partir de 0.
Pour changer un caractère on utilise `<-`, par exemple :

```
# let s = "caillous";;
val s : string = "caillous"
# s.[7] <- 'x' ;;
- : unit = ()
# s;;
- : string = "cailloux"
```

L'initialisation d'une chaîne de caractères se fait avec la fonction `make_string` (`String.make` du module `String`) :

```
# let chaine_car = String.make 5 'c';;
val chaine_car : string = "ccccc"
```

Les *vecteurs* ont le type `vect`, ce sont des suites d'éléments du même type, la syntaxe de la définition est :

```
# let v = [| 1; 2; 3 |];;
 v : int vect = [| 1; 2; 3 |]
```

Initialisation (make_vect, Array.make) :

```
# let v = Array.make 4 1;;
val v : int array = [|1; 1; 1; 1|]
```

Les éléments d'un vecteur sont indexés à partir de 0.

Une coordonnée est modifiée de la façon suivante :

```
# v.(3) <- 2 ;;
- : unit = ()
# v ;;
- : int array = [|1; 1; 1; 2|]
```

Le type des vecteur a une notation suffixe : int vect est le type d'un vecteur d'entiers.

Les *références* ont le type ref. Le constructeur de type des références utilise la notation suffixe :

```
# let r = ref 3. ;;
val r : float ref = {contents=3}
# r :=  4. ;;
- : unit = ()
# !r ;;
- : float = 4
```

Ici r est une référence de flottant.

Pour modifier une référence, on utilise := et pour accéder à la valeur d'une référence, on met ! devant.

Les *paires* et plus généralement les *n-uplets*, exemple :

```
# let triple = (1, 'a', 3.);;
val triple : int * char * float = 1, 'a', 3
```

2.9 Expressions

Les expressions artithmétiques s'écrivent comme en mathématiques avec +, -, *, /, mod. Les opérations sur les flottant se font avec les opérateurs avec un point en suffixe. Donc :

```
# 2. + 3 ;;
```

donnera un message d'erreur. *Il faut faire les conversions explicitement*. Par exemple avec int_of_float.

Une expression *conditionnelle* s'écrit :

```
if condition then E else F
```

où `condition` est une expression de type booléen, `E` et `F` étant des expressions de même type.

Les expressions *booléennes* sont construites avec les opérateurs `||, &&, not` (soient respectivement : ou, et, non), des variables booléennes et des opérateurs de comparaison.

`||, &&, not` ont des priorités comme, respectivement, `*, +, -`.

Exemple :

```
(b && not c) || (not b && c)
```

est le « ou » exclusif.

Lors de l'appel d'une fonction, les arguments sont évalués avant tout, mais *l'ordre d'évaluation n'est pas spécifié*.

L'opérateur d'égalité est `=`, il est *polymorphe* car il s'applique à tout les types. C'est une égalité `structurelle` car elle teste la valeur, contrairement à l'égalité `==` qui teste l'égalité physique :

```
# "oui" = "oui" ;;
- : bool = true
# "oui" == "oui" ;;
- : bool = false
# "oui" != "oui" ;;
- : bool = true
```

`!=` est l'équivalent de `<>`.

```
# let x = ref 2 ;;
val x : int ref = {contents=2}
# let y = ref 2 ;;
val y : int ref = {contents=2}
# x = y ;;
- : bool = true
# x == y ;;
- : bool = false
# x == x ;;
- : bool = true
```

2.10 Blocs et portée des variables

Dans le corps d'un fonction (ou d'une procédure) on définit souvent des valeurs locales, introduites par la construction

```
let identifiant = expression in ...
```

Il n'y a aucune restriction sur les valeurs locales. Lorsqu'un identificateur `x` est cité, il fait référence à la dernière définition de `x` (après un `let` ou comme argument de fonction après `function`). Il est d'usage, parce que plus prudent (la lisibilité est meilleure) de minimiser les variables globales (une locale devient globale dans un sous programme).

Exemple de liaison des identificateurs (*portée statique* :

```
# let plus_un x = x - 1;;
val plus_un : int -> int = <fun>
# let plus_deux x = plus_un ( plus_un x );;
val plus_deux : int -> int = <fun>
# plus_deux 3 ;;
- : int = 1
# let plus_un x = x + 1;;
val plus_un : int -> int = <fun>
# plus_deux 3 ;;
- : int = 1
```

La première définition ne correspond pas au nom de la fonction, pourtant on définit une seconde fonction avec celle-ci. On modifie la première mais cela ne change pas la définition de la seconde. *La fonction `plus_deux` a été définie par la première version de la fonction `plus_un` (le compilateur a fonctionné une fois, lors de la validation de la définition).*

2.11 Débogage

La fonction `trace` permet de suivre pas à pas le déroulement d'un programme :

```
# trace ''programme_test'';;
```

Pour avoir des indications sur l'évolution de la mémoire, on peut intercaler l'impression de messages.

2.12 Instructions

Une suite, ou séquence est formée d'une succession d'expressions *express_i* séparées par des points-virgules. On peut délimiter une suite par `begin...end`. La valeur d'une suite est sont dernier élément.

Dans une suite, on admet les alternatives :

```
if e then e_1
```

et

```
if e then e_1 else e_2
```

L'alternative sans le *else* est en fait une alternative avec un *else* caché :

```
if e then e_1 else ()
```

ce qui explique les erreurs de type

```
#if true then 1;;
```

`1` n'a pas le type `unit`...

Pour lever les ambiguïtés il faut `begin...end` :

```
if e then if e' then e_1 else e_2
```

par exemple :

```
if e then begin if e' then e_1 else e_2 end
```

2.13 Filtrage

Le filtrage évite une cascade de *if*. Syntaxe :

```
match e with
| v_1 -> e_1
| v_2 -> e_2
  ...
| v_n -> e_n
| _ -> defaut
```

2.14 Boucles

Il y a deux constructions impératives pour l'itération, la boucle *for* et la boucle *while*. Dans les deux cas le corps de la boucle est délimité par `do ... done`.

```
for i = e_1 to e_2 do e done
```

L'indice de boucle, *i*, ne peut être modifié par une affectation dans le corps de la boucle. Le nombre de boucle est fixe et les seuls pas d'itération sont 1 et -1.

```
while e_1 do e done
```

Si *e_1* est fausse, la boucle n'est pas exécutée, si elle est toujours vraie la boucle est infinie, attention donc.

2.15 Exceptions

En appliquant la primitive *raise* à une valeur *exceptionnelle*, on déclenche une erreur.

#

La construction

```
try calcul with filtrage
```

permet de récupérer les erreurs qui se produiraient dans *calcul* avec *filtrage*. La valeur renvoyée en cas d'erreur doit être de même type que celle renvoyée sans erreur.

#

On peut définir des exceptions personnalisées.

2.16 Entrées – sorties

```
#read_line ();;
tape au clavier
- : string = "tape␣au␣clavier"
```

2.16.1 Impressions simples

`print_chaîne` où `chaîne` est l'un des mots : `int, char, float, string, newline`. On verra plus tard `printf`.

2.16.2 Lecture et écriture sur fichiers

On ouvre un *canal* par `open_in` et `open_out` suivi d'un nom de fichier pour, respectivement lire et écrire. La lecture se fait avec `input_char` et `input` ou `input_line`. L'écriture avec `output_char` et `output` ou `output_string`. Enfin les canaux sont fermés avec `close_in` et `close_out`.

Des exemples sont traités dans le paragraphe suivant.

2.17 Copie de fichiers

2.18 Définitions de types

2.19 Modules

2.20 Bibliothèques

2.21 Fonctions graphiques

3 Itération

3.1 Boucles

Une *boucle* est la répétition d'un même algorithme. Il y a deux sorte de boucles : les boucles effectuées un nombre fixe de fois et les boucles effectuées un nombre variable de fois.

Boucle « pour »

Boucle « pour »	: :=	`for` *ident* = *expression*	(for : pour)
		(`to` | `downto`) *expression*	(to : jusqu'à, down : en bas)
		`do` *expression* `done`	(do : faire, done : fait

Boucle « tant que »

Boucle « tant que »	: :=	`while` *expression*	(while : tant que)
		`do` *expression* `done`	(do : faire, done, fait)

Exemples

```
#let imp_chif () =
        for i = 0 to 9 do
                print_int i
        done;
        print_newline ();;
imp_chif : unit -> unit = <fun>
#imp_chif ();;
0123456789
- : unit = ()

#let imp_chif2 () =
        let i = ref 0 in
        while !i < 10 do
                print_int !i;
                i := !i + 1;
        done;
        print_newline ();;
imp_chif2 : unit -> unit = <fun>
#imp_chif2 ();;
0123456789
- : unit = ()
```

3.2 Exemples

Écrire les programmes de calcul de puissances, de factorielles, du calcul du plus petit diviseur d'un nombre.

3.3 Polynômes pleins

3.3.1 Manipulation de polynômes

Tableaux
Les tableaux sont aussi appelés *vecteurs*, ce sont des suites indexées d'un nombre fini d'éléments. Il y a deux façons de définir les tableaux : on le définit par ses éléments ou on définit les éléments ultérieurement. Soit respectivement :

```
#let v1 = [| 1; 2; 3 |];;
v1 : int vect = [|1; 2; 3|]

#let v2 = make_vect 3 2;;
v2 : int vect = [|2; 2; 2|]
```

La commande `make_vect` définit un vecteur à trois coordonnées initialisées à 2. La fonction `vect_lentgh` renvoie la longueur du vecteur, c'est à dire le nombre de coordonnées *moins un*, car les indices débutent à 0.

On accède aux coordonnées (ou *cases*) d'un tableau par

```
# v1.(2);;   (* troisi\`eme case *)
```

On modifie le contenu des cases par

```
# v2.(2) <- 3;;
```

On représentera un polynôme $\sum_{0 \leq k \leq n} a_k X^k$ par le tableau `[| a_0; ...; a_n |]`.

Impression des polynômes
Commençons par imprimer un monôme. On représente un monôme aX^k par `a k`.

```
#let imprime_monome coeff deg =
        if deg = 0 then print_int coeff else
        if coeff <> 0 then
                begin
                        print_string "+";
                        if coeff <> 1 then print_int coeff;
                        print_string "X";
                        if deg <> 1 then
                                begin
                                        print_string "^"; print_int deg
                                end
        end;;
imprime_monome : int -> int -> unit = <fun>
#imprime_monome 3 2;;
+3X^2- : unit = ()
```

Alors l'impression d'un polynôme plein est facile :

```
#imprime_polynome_plein v2;;
2+2X+3X^2- : unit = ()
```

On a un problème avec les signes ?

```
#let v3 = make_vect 3 1;;
v3 : int vect = [|1; 1; 1|]
#v3.(0) <- -1;;
- : unit = ()
#v3.(1) <- -3;;
- : unit = ()
#v3;;
- : int vect = [|-1; -3; 1|]
#imprime_polynome_plein v3;;
-1+-3X+X^2 - : unit = ()
```

Exercice Écrire un programme qui gère les signes négatifs.

Addition des polynômes L'addition des polynômes p et q revient à additionner deux vecteurs. Pour cela on définit un tableau s dont la longueur est égale à la plus grande des longueurs des tableaux qui représentent p et q.

Algorithme 3 Somme

---

initialisation : $s \leftarrow 0$ « 0 est le vecteur nul »
pour i de 0 à longueur de p **faire**
  $s.(i) \leftarrow p.(i)$
fin pour
pour i de 0 à longueur de q **faire**
  $s.(i) \leftarrow s.(i) + q.(i)$
fin pour

On utilisera la fonction `max` de *Caml*.

Produit de polynômes

Algorithme 4 Produit

---

initialisation : $t \leftarrow 0$ « 0 est le vecteur nul »
pour i de 0 à longueur(p) **faire**
  pour j de 0 à longueur(q) **faire**
    $t.(i+j) \leftarrow t.(i+j) + p.(i) * q.(j)$
  fin pour
fin pour

Exercice
Écrire les programmes d'évaluation d'un polynôme en a (entier ou réel).

3.4 Preuves de programmes

La preuve d'un programme c'est la démonstration qu'il fait bien ce qu'on lui demande de faire et en un temps fini. Évidemment il s'agit de démonstration mathématique.

La preuve consiste en deux étapes :

1. La preuve de correction partielle : si le programme s'arrête, il fournit le bon résultat ;
2. La preuve d'arrêt : le programme s'arrête !

3.4.1 La correction partielle

Une *action* c'est une opération (algorithme, fonction, procédure...). Un *invariant*, I est une propriété qui est vraie en entrée et en sortie. Une *précondition* est une propriété vraie en entrée et une *post-condition* est vraie en sortie.

On a :

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

Si C est vraie et I est vérifiée en entrée, alors C est vraie en sortie.
Tant que C est vraie, on peut effectuer l'action.
$\neg C$ désigne *non C*, alors dès que C est fausse :

$$I \rightarrow I \wedge \neg C$$

Il suffit qu'alors :

$$I \wedge \neg C \Rightarrow P$$

pour que la propriété P, la post-condition, soit vérifiée.

3.4.2 Preuve d'arrêt

Pour prouver l'arrêt, l'existence d'une fonction « taille » : t, à valeurs dans $\mathbb{N}$, telle que :

$$t = k \xrightarrow{action} t < k$$

montre que le programme s'arrête puisque qu'une suite décroissante d'entiers est finie.

3.4.3 Exemples

Algorithme 5 x puissance n

```
si n=0 alors
   1
sinon
   pour j de 1 à n faire
   puissance := puissance * x
fin si
```

L'invariant de boucle est x^j, car le passage dans la boucle donne x^{j+1}, c'est un invariant car $j := j + 1$ donne la même formule.

La fonction t est $t(j) = n - j$, donc l'algorithme se termine.

4 Récursivité

Un algorithme est « récursif » s'il est défini en fonction de lui-même.

Plus précisément :

On dit qu'un ensemble E est *gradué* s'il existe une fonction strictement croissante φ de E dans $\mathbb{N}$ telle que $\varphi^{-1}(\mathbb{N}) = E$. On note $E_j = \varphi^{-1}(j)$. Donc pour tout j :

$$E_j \subset E_{j+1}$$

et

$$\bigcup_{j \geq 0} E_j = E$$

On peut également définir des graduations à partir de relations d'ordres plus générales, puis des bigraduations... On ne retiendra que les relations d'ordre qui permettent d'effectuer un raisonnement par récurrence, elles sont définies généralement sur les ensembles dénombrables $\mathbb{N}^p$ car le principe du raisonnement par récurrence d'étend à ces ensembles.

Soit un algorithme $\mathcal{A}$ qui prend en entrée un élément x de E. On dit que $\mathcal{A}(x)$ est bien défini s'il se termine et fournit le résultat attendu.

On suppose qu'il existe une fonction F telle que :

- Pour tout x dans E_0 : $\mathcal{A}(x)$ est bien défini ;
- Pour x dans $E \setminus E_0$, il existe $y_1, \ldots, y_k$ dans E vérifiant pour tout j, $\varphi(y_j) < \varphi(x)$: $\mathcal{A}(x) = F(\mathcal{A}(y_1, \ldots, \mathcal{A}(y_k))$.

Le principe de raisonnement par récurrence appliqué à la propriété « $\mathcal{A}(x)$ est bien défini », montre qu'alors $\mathcal{A}(x)$ est défini pour tout x dans E. On dira que $\mathcal{A}$ est bien défini.

Remarque : F et les y_j peuvent dépendre de x.

Exemple 4.1 Considérons $\mathbb{N}^2$ et ordonnons ses éléments de la façons suivante : $(m, n) < (m', n')$ si $m + n < m' + n'$ et sinon $m < m'$. On obtient une graduation avec la fonction

$$\varphi(m, n) = \frac{(m + n - 1)(m + n)}{2} + m + 1$$

(on compte les éléments dans l'ordre, ainsi (m, n) est le $\varphi(m, n)$ [e] élément).

4.1 Exemple : exponentiation rapide

Pour calculer x^n nous utilisons l'algorithme suivant

Preuve d'arrêt

La fonction de « taille » est $t = n$, donc si $n \geq 2$ alors $1 \leq \frac{n}{2} < n$ car $n > 0$, donc t décroît strictement.

Preuve partielle

Soit l'invariant $I(p) = (yz^p = x^n)$, si n est pair $yz^p = y(z^2)^{\frac{p}{2}}$ et $z \leftarrow z^2$ et si p est impair $yz^p = yz(z^2)^{\frac{p-1}{2}}$ donc $z \leftarrow z^2$ et $y \leftarrow yz$.

Algorithme 6 Puissance rapide

si n=0 **alors**
  1
ou si n pair **alors**
  $y \leftarrow x^{\frac{n}{2}} \star x^{\frac{n}{2}}$
sinon
  $y \leftarrow x^{\frac{n-1}{2}} \star x^{\frac{n-1}{2}} * x$
fin si

Majoration du nombre de calculs
Il s'agit ici de ce que l'on appelle la complexité d'un algorithme. On calcule le nombre de calculs élémentaires (il faut alors définir précisément ce que cela signifie). Il existe aussi une notion de complexité en terme de place en mémoire physique. Nous estimons ici le coût en calculs.

On se place dans le pire des cas. Il faut au plus 2 multiplications (p impair) par appel récursif. Soit u_n, le nombre d'appels pour n. Alors

$$u_n = 1 + u_{\lfloor \frac{n}{2} \rfloor}$$

puisque suivant que n est pair ou impair u_n est égal à $1 + u_{\frac{n}{2}}$ ou $1 + u_{\frac{n-1}{2}}$.

En base 2 : $n = \sum_{0 \leq j \leq N} a_j 2^j \leq 2^{N+1}$ et si $\lfloor n \rfloor = 2^N$ alors $\lfloor \frac{n}{2} \rfloor = N - 1$. On suppose donc $n = 2^N$, il vient :

$$u_{2^N} = 1 + u_{2^{N-1}}$$

donc la suite u est arithmétique de raison 1 et ainsi $u_{2^N} = N + u_1$ donc $u_{2^N} = \mathrm{O}(\log_2 n)$.

4.2 Récursivité mutuelle

Définition : Deux algorithmes sont dit récursifs si chacun fait appel à l'autre.

```
let rec pair n = match n with
0 -> true
| _ -> impair (n - 1)
and impair n = match n with
0 -> false
| _ -> pair (n - 1);;
```

Exercice : Comparer avec le programme :

```
let rec pair n = match n with
0 -> true
| _ -> not(impair n)
and impair n = pair(n - 1);;
```

Une troisième fonction peut éliminer la récursivité mutuelle :

```
let rec pair_impair x = match x with
| 0 -> (true, false)
| _ -> let (p, i) = pair_impair (x - 1) in (i, p);;
```

```
let pair n =
  let (p, _) = pair_impair n in p;;
```

```
let impair n =
  let (_, i) = pair_impair n in i;;
```

4.3 Récursivité terminale

Prenons l'exemple de la factorielle avec le programme récursif :

```
let rec fact n =
        if n < 0 then failwith "erreur" else
        match n with
  | 0 -> 1
  | _ -> n * fact(n - 1);;
```

Le calcul se fait de la façon suivante :

```
fact(4)
4 * fact(3)
4 * 3 * fact(2)
4 * 3 * 2 * fact(1)
4 * 3 * 2 * 1
4 * 3 * 2
4 * 6
24
```

```
let rec fact2 n m =
if n < 0 then failwith "erreur" else
if n = 0 then m else
fact2(n - 1) (n * m);;
```

```
  #fact 3;;
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
- : int = 6
```

```
#fact2 3 1;;
fact2 <-- 3
fact2 --> <fun>
fact2* <-- 1
fact2 <-- 2
fact2 --> <fun>
fact2* <-- 3
fact2 <-- 1
fact2 --> <fun>
fact2* <-- 6
fact2 <-- 0
fact2 --> <fun>
fact2* <-- 6
fact2* --> 6
fact2* --> 6
fact2* --> 6
fact2* --> 6
- : int = 6
```

A priori il y a moins de calcul dans la seconde version puisque le résultat est trouvé à la fin de l'empilement. Pratiquement... ça dépend.

On dit que dans le second cas la récursivité est *terminale* (il n'y a rien à faire après l'appel récursif).

4.4 Preuves de terminaison

Prenons pour exemple la fonction de Ackermann : $ack(m, n)$ est définie sur $\mathbb{N}^2$ et vaut :

$n + 1$	si $m = 0$
$ack(m - 1, 1)$	si $n = 0$
$ack(m, n) = ack(m - 1, ack(m, n - 1))$	sinon

La complexité de l'algorithme étant plus qu'exponentielle, ne pas lancer le programme suivant pour n'importe quelles valeurs de (m, n).

```
let rec ack = fun
 (0, n) -> n + 1
|(m, 0) -> ack(m - 1, 1)
|(m, n) -> ack(m - 1, ack(m, n - 1));;
```

On munit $\mathbb{N}^2$ de l'ordre lexicographique, *i.e.* :

$(m, n) < (m', n')$	si et seulement si $m < m'$
	ou $m = m'$ et $n < n'$

C'est une relation d'ordre total (tous les éléments sont comparables) et *bien fondée*, c'est à dire qu'il n'existe pas de suite décroissante infinie.

Pour démontrer la terminaison du programme déduit de la définition de la fonction d'Ackermann, il suffit de démontrer que si $ack(m', n')$ se termine pour tout (m', n') strictement plus petit que (m, n) alors $ack(m, n)$ se termine.

Ainsi :

1. si $m = 0$ alors $ack(0, n)$ se termine (c'est évident).
2. si $m \geq 1$ alors $(m - 1, 1) < (m, 0)$, donc $ack(m - 1, 1)$ se termine.
3. si $m \geq 1$ et $n \geq 1$ alors $(m, n - 1) < (m, n)$ donc $ack(m, n - 1)$ se termine et vaut p, comme $(m - 1, p) < (m, n)$ alors $ack(m - 1, p)$ se termine.

Si l'ensemble $F = \{(m, n) \in \mathbb{N}^2 / ack(m, n) \text{ ne se termine pas}\}$ est non vide, alors il possède un plus petit élément (m_0, n_0). En effet $\{m \in \mathbb{N} / \exists n \in \mathbb{N}, \quad (m, n) \in F\}$ est une partie non vide de $\mathbb{N}$, elle a donc un plus petit élément m_0. De même $\{n \in \mathbb{N} / (m_0, n) \in F\}$ est une partie non vide de $\mathbb{N}$, qui a un plus petit élément n_0. Il est clair que (m_0, n_0) est le plus petit élément de F.

Or $ack(m_0, n_0)$ se calcule en fonction des $ack(m, n)$ où $(m, n) < (m_0, n_0)$, donc $ack(m_0, n_0)$ se termine et $(m_0, n_0) \notin F$. Donc F est vide, ce qui prouve la terminaison de l'algorithme d'Ackermann (qui est donc bien un algorithme).

Remarque : donner un sens précis à « se termine » est faisable mais un peu compliqué.

5 Diviser pour régner

La méthode « diviser pour régner » (*divide and conquer*, en anglais) consiste à ramener la résolution d'un problème à celle de plusieurs problèmes identiques mais de tailles strictement plus petites. Généralement le problème dépend d'un entier n et on le découpe en deux problèmes de tailles n' et n'', des entiers proches de $n/2$.

Un exemple typique est donné dans le paragraphe de récursivité avec la puissance rapide. Il y a bien d'autres exemples donnés dans la liste d'exercices, comme le produit de polynômes ou de matrices. On verra plus tard l'usage de cette méthode dans les tris.

5.1 Puissance rapide

Algorithme 7 La puissance rapide, version récursive : *puissance_rapide*

si $n = 0$ **alors**
  1
  si $n = 1$ **alors**
    x
  sinon
    si n est pair **alors**
      $\left(x^{\frac{n}{2}}\right)^2$
    sinon
      $x * \left(x^{\frac{n-1}{2}}\right)^2$
    fin si
  fin si
fin si

Voici une version *Caml* :

```
# let rec puissance_rapide (x, n) = match n with
| 0 -> 1
| 1 -> x
| _ -> let y = puissance_rapide (x, n / 2) in
      if n mod 2 = 0 then y * y else x * y * y ;;
```

```
#puissance_rapide (2, 29);;
- : int = 536870912
```

Cet algorithme est relié à la décomposition en base 2 de l'exposant n. Un 0 correspond à une élévation au carré, un 1 à une élévation au carré et à un multiplication par x. Au pire il n'y a que des 1 d'où $\lfloor \ln_2 n \rfloor$ multiplications par x et $\lfloor \ln_2 n \rfloor - 1$ élévatons au carré, la complexité est donc en $\ln n$.

5.2 Produits de polynômes

Tout polynôme P de degré inférieur ou égal à n s'écrit sous la forme $P_0 + X^{\lfloor \frac{n}{2} \rfloor} P_1$ où P_0 et P_1 sont de degrés inférieurs ou égaux à $n - \lfloor \frac{n}{2} \rfloor \leq \lfloor \frac{n}{2} \rfloor + 1$.

La complexité du calcul du produit de deux polynômes de degrés m et n est de l'ordre de mn.

Pour simplifier nous supposons que les polynômes ont des degrés inférieurs à n et que $n = 2m$.

Soient $a = a_0 + X^m a_1$ et $b = b_0 + X^m b_1$ deux polynômes sous la forme indiquée ci-dessus, leur produit :

$$ab = a_0 b_0 + (a_0 b_1 + a_1 b_0) X^m + a_1 b_1 X^{2m}$$

nécessite le calcul de quatre produits de polynômes de degrés inférieurs à m. Si C est la fonction de coût :

$$C(2m) = 4C(m) + \lambda(m+1)$$

où $\lambda(m+1)$ représente le coût de l'addition. Le gain est nul puisque C est un $\mathrm{O}(n^2)$.

Or une simple transformation du produit permet d'obtenir une meilleure performance :

$$a_0 b_1 + a_1 b_0 = (a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1$$

Maintenant il suffit de calculer le produit de trois polynômes de degrés inférieurs ou égaux à m et de calculer quatre sommes. Le coût vérifie une équation de la forme :

$$C(2m) = 3C(m) + 4m$$

La solution de l'équation homogène est : $u_k = 3^k u_0 = 3^k$, soit alors $C(2^k) = v_k 3^k$, il vient :

$$v_k - v_{k-1} = 2\left(\frac{2}{3}\right)^k$$

où $v_0 = 1$. Ainsi :

$$C(2^k) = 3^k + 6(3^k - 2^k)$$

Or $3^k = \exp k \ln 2 \left(\frac{\ln 3}{\ln 2}\right) = \left(2^k\right)^{\frac{\ln 3}{\ln 2}}$. On peut démontrer, par récurrence, que C est croissante ce qui donne

$$C(n) = \mathrm{O}(n^{\frac{\ln 3}{\ln 2}})$$

ce qui est un gain appréciable relativement à un coût quadratique.

6 Listes

6.1 Suites ordonnées

En mathématiques les éléments d'une suite u sont ordonnés naturellement par leur indice. Il y a des suites finies ou infinies, on appelera *longueur* de u l'indice maximal des éléments de u. On peut considérer des ensembles de suites dont la longueur est constante ou dont la longueur est variable ou encore infinie, autrement dit des ensembles de la forme :

$$(m \in \mathbb{N}) \quad \mathbb{R}^m \qquad \text{ou} \qquad \bigcup_{m \geq 0} \mathbb{R}^m \qquad \text{ou} \qquad \mathbb{R}^{\mathbb{N}}$$

En informatique les éléments d'une suite peuvent être rangés de plusieurs manières dans la mémoire de l'ordinateur.

1. les tableaux (vecteurs, matrices) de dimensions fixes
2. les suites et les listes de de longueurs variables

On peut accéder directement aux éléments d'un tableaux ou d'une suite avec son indice, cependant il existe un type de listes pour lesquelles c'est impossible, ce sont les *listes chaînées*.

Un objet est accessible à l'aide de son adresse dans la mémoire de l'ordinateur, un tableaux a une seule adresse, tous ses éléments sont alors accessible par cette adresse. La place en mémoire étant réservée, on ne peux plus modifier sa dimension car la case mémoire suivant le dernier élément peut être indisponible. Une liste chaînée est aussi accessible par une adresse mais c'est celle de son premier élément et du pointeur vers l'élément suivant qui peut occuper une case mémoire éloignée. Le pointeur est une partie de la case mémoire qui contient une adresse. On peut donc modifier la longueur d'une liste en insérant des éléments, mais pour accéder à un élément quelconque on doit parcourir la liste à partir du premier élément. Voir les schémas 1 et 2.

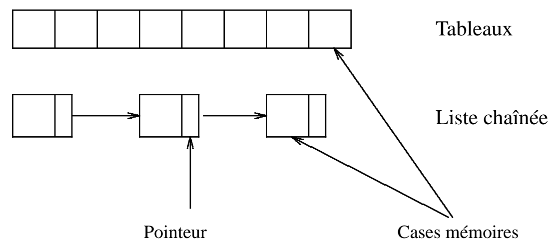


FIG. 1 – Structures des tableaux et des listes

6.2 Listes chaînées

Une liste en *Caml* est de la forme `[2;3;4]`.

```
let l = [9;8;7;6;5;4;3;2;1;0];;
```

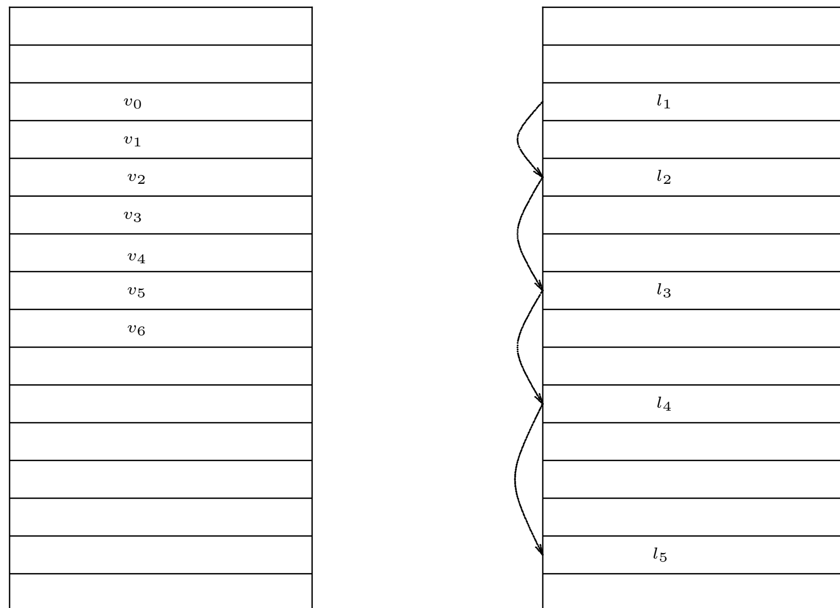


FIG. 2 – Représentation en mémoire d'un vecteur et d'une liste chaînée

ou encore par

4 : : 5 : : 6 : : [ ] ; ;

`[ ]` est la liste vide (*nil* en anglais, du latin *nihil*) et `: :` est l'opérateur infixe cons (constructeur de liste) qui ajoute un élément en tête de liste.

La liste `l` dont les éléments sont `l_j` est représentée par un peigne (figure 3).

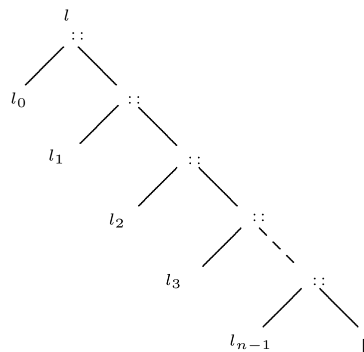


FIG. 3 – Liste chaînée, en peigne

On accède au premier élément par la fonction `tête`

```
let tete = function
```

```
  | t :: r -> t
  | _ -> failwith "liste vide";;
```

et au reste avec

```
let reste = function
  | _ :: r -> r
  | _ -> failwith "pas de reste";;
```

En fait, il y a des primitives qui font ces actions ce sont respectivement `hd` (*head*) et `tl` (*tail*).

L'opérateur de concaténation est `@`. On peut le définir par

```
  let rec concat_lists l1 l2 = match l1 with
| [] -> l2
| x :: reste -> x :: concat_lists reste l2;;
```

Exercice : somme des éléments d'une liste.

```
let rec somme = function
 | [] -> 0
 | x :: l -> x + somme l;;
```

On remarque que le produit est semblable

```
let rec produit = function
 | [] -> 1
 | x :: l -> x * produit l;;
```

Exercice : écrire des versions avec accumulateur.

```
let rec somme_accu accu =
function
[] -> accu
| x :: reste -> somme_accu (accu + x) reste;;
```

etc...

Exercice : retourner une liste.

```
let rec retourne = function
      [] -> []
      | x :: reste -> retourne reste @ [x];;
```

Avec un accumulateur :

```
let rec reto_accu accu = function
      [] -> accu
      | x :: reste -> reto_accu ([x] @ accu) reste;;
```

On remarquera que $[x]$ est concaténée à la fin du reste dans la première version et au début de l'accumulateur dans la seconde.

6.2.1 Recherche dans une liste

Recherche par le rang :

$j \leftarrow 0$ et r est le rang de l'élément cherché
tant que $j < r$ **faire**
  $j \leftarrow j + 1$
fin tant que

Version itérative :

```
let cherche_rang r l =
if r <= 0 or r > list_length l then failwith "erreur" else
let reste = ref l in
for j = 1 to r-1 do
reste := tl !reste
done;
hd !reste;;
```

Version récursive :

```
let rec cherch_rang r l =
 if r <= 0 or r > list_length l then failwith "erreur" else
 if r = 1 then hd l else
 cherche_rang (r - 1) (tl l);;
```

Exercice : comparer avec la recherche dans un vecteur.

Recherche par contenu : Par exemple et pour fixer les idées :

```
let test m = m mod 2 = 0;;

let rec cherche =
  function
        [] -> failwith "liste␣vide"
        | (x :: reste) -> if test x then x else
       cherche reste ;;
```

Si en plus on veut le rang, à partir du rang m inclus, on définit une fonction auxiliaire :

```
let rec cherche_aux m n =
  let r = ref 1 in
  function
        [] -> failwith "ya␣pas"
        | (x :: reste) -> if test x & !r >= m   then x, n else
       cherche_aux (m - 1) (n + 1) reste ;;
```

et pour chercher un élément et son rang à partir du rang m :

```
let cherche_cont m = cherche_aux m 1
```

Version itérative :

```
let rec cherche_cont test l =
  let liste = ref [] in
  for i = 1 to list_length l do
    liste := cherche_aux i l :: !liste
  done;
  !liste;;
```

Pour un vecteur, la recherche du contenu a seule un intérêt :

```
let cherche_cont_vect test v =
 let patrouve = ref true and j = ref 0 in
 while !patrouve & !j <= vect_length v - 1 do
   if test v.(!j) then patrouve := false else j := !j + 1
       done;
 !j;;
```

À partir du rang m :

```
  let cherche_cont_et_rang_vect test m v =
 let patrouve = ref true and j = ref m in
 while !patrouve & !j <= vect_length v - 1 do
   if test v.(!j) then patrouve := false else j := !j + 1
       done;
 v.(!j),!j;;
```

6.2.2 Insérer ou supprimer un élément dans une liste

Suppression :

```
let rec supprime test  = function
[] -> []
| a :: reste -> if test a then supprime test reste else
a :: supprime test reste;;
```

Insertion :

```
let rec insere j a l = match l with
   [] -> if j = 1 then [a] else []
 | x :: reste -> if j = 1 then a :: l else
   x :: insere (j - 1) a reste;;
```

6.3 Tris

Trier une liste c'est transformer cette liste par des opérations élémentaires pour obtenir une liste ordonnée. On trie aussi des vecteurs en ordonnant les coordonnées.

Dans tous les cas, si la liste est vide ou ne contient qu'un élément, elle est triée.

6.3.1 Tri insertion

On débute avec la liste 1 qui est la liste à trier et la liste 2 qui est vide et que l'on va remplir au fur et à mesure que l'on vide la première. On choisit un élément de la liste 1 pour démarrer la liste 2. Puis, par récurrence, on choisit un élément de la liste 1 que l'on insère à la bonne place dans la liste 2. Lorsque la liste 1 est vide, la liste 2 est la liste triée cherchée.

6.3.2 Tri sélection

On cherche le plus petit élément (ou le plus grand) de la liste et on le met en première position. Puis par récurrence, on cherche le k^e plus petit élément que l'on place en k^e position (le plus petit élément parmi les $n - k + 1$ restants, n étant le nombre d''éléments de la liste).

6.3.3 Tri rapide

On choisit un élément appelé le pivot et on divise la liste en deux sous-listes, la première contient les éléments plus petits que le pivot et la seconde contient les autres. On obtient une liste avec la première liste à laquelle on ajoute le pivot puis la seconde liste. Le pivot a donc atteint sa place définitive. On recommence la même opération avec les deux listes. Ce tri se programme donc facilement de manière récursive.

On a besoin d'une fonction qui découpe une liste en deux listes selon une propriété P. On met les éléments d'une liste l qui vérifient P dans la liste l_1 et les autres dans l_2.

x vérifie P si $\mathrm{test}\, x$ est vrai.

```
let rec partition test l1 l2 = function
      [] -> l1 , l2
    | x :: reste -> if test x then
        partition test l1 (x :: l2) reste else
        partition test (x :: l1) l2 reste;;

#partition test [] [] [0;1;2;3;4;5;6;7;8;9];;
- : int list * int list = [9; 7; 5; 3; 1], [8; 6; 4; 2; 0]
```

(Ici, dans l'exemple, la fonction *test* vérifie qu'un entier est pair, attention les éléments sont inversés car le premier élément est mis en fin de liste).

Une fonction de comparaison, par exemple :

```
let compare a = function
b -> if a < b then true else false;;
```

Exemple :

```
#partition (compare 4) [] [] l1;;
- : int list * int list = [9; 8; 7; 6; 5], [4; 3; 2; 1]
```

Le programme de tri rapide est donc :

```
let rec tri_rap compare = function
[] -> []
| [a] -> [a]
| (a :: l) -> let l1, l2 = partition (compare a) [] [] l in
tri_rap compare l1 @ (a :: tri_rap compare l2);;

#tri_rap compare [3;5;1;2;8;6];;
- : int list = [1; 2; 3; 5; 6; 8]
```

Rappel : quand une erreur est réparée dans un programme, il faut revalider tous les autres programmes qui en dépendent.

Une version paramétrable de `compare`

```
let compare2 ordre a = function
    b -> if ordre a b then true else false;;
```

avec un exemple :

```
#compare2 (prefix >) 4 2;;
- : bool = true
```

6.3.4 Tri bulle

Le principe du tri bulle est de comparer deux éléments consécutifs et à les échanger s'ils ne sont pas dans le bon ordre, en commençant par un bout de la liste et en itérant jusquà la fin. Après un premier tour un élément est à sa place définitive. On recommence jusqu'à ce que tous les éléments soient à leur place.

Trions une liste (d'entiers) suivant l'ordre croissant. La fonction `aux` renvoie un booléen qui indique si la liste à été parcourue et la liste éventuellement après inversion.

```
let rec aux l = match l with
  |[] -> false, l
  |[a] -> false, [a]
  |a :: r -> let test, r' = aux r in
       if a <= hd r' then test, a :: r' else
         true, hd r' :: a :: tl r'
;;
```

Après un passage, le premier élément est à sa place (à cause de la récursion).

```
let rec tribul l = match aux l with
  |true, a :: r  -> a :: tribul r
  |_, _ -> l
;;

#tri_bulle [9; 5; 7; 3; 4; 6; 1; 2; 0; 8];;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Soit n la longueur de la liste. Dans le tri bulle il y a $n-1$ appels à la fonction *deplace* qui s'appelle $m-1$ fois pour m variant de n à 2. À chaque appel, il y a deux

Cons, d'où une complexité en $O(n^2)$. La place en mémoire requiert une pile de listes qui ne dépasse pas $n - 1$.

Version itérative sur les vecteurs :

```
let tri_bulle2 v =
  let  r = ref 0 in
  for m = vect_length v - 1 downto 1 do
  for j = 0 to m - 1 do
    if v.(j) > v.(j + 1) then
      begin
        r := v.(j);
        v.(j) <- v.(j + 1);
        v.(j + 1) <- !r
            end;
    done;
    done;
  v;;
```

Cette fois ci, on compte $m + 1$ échanges au maximum pour m variant de 2 à $n - 1$, donc la complexité est encore un $O(n^2)$. Par contre la complexité spatiale est constante puisque l'on a qu'un vecteur en mémoire.

6.3.5 Tri fusion

Le principe est couper la liste en deux, de trier les deux listes puis de les fusionner. Donc, récursivement, on découpe une liste en deux listes, puis ces listes en deux et ainsi de suite jusqu'à ce que les listes obtenues n'aient qu'un élément au plus, elles sont alors triées, il reste à fusionner ces listes triées.

Fonction de partition, on l'appelle sur une liste l et la liste vide, elle renvoie deux listes de longueurs égales à 1 près :

```
let rec partition = function
  |[] -> [], []
  |[a] -> [a], []
  |a :: b :: r -> let l1, l2 = partition r in a :: l1, b :: l2
;;

#partition [4;5;6;8;7;2;1;9];;
- : int list * int list = [7; 2; 1; 9], [8; 6; 5; 4]
```

Fonction de fusion de listes triées :

```
let rec fusion l l' = match l, l' with
  | [], l' -> l'
  | l, [] -> l
  | a :: l1, b :: l2 ->
      if a <= b then a :: fusion (l1, (b :: l2))
      else  b :: fusion ((a :: l1), l2);;
```

Le programme de tri est :

```
let tri_fusion l = let l1 , l2 = partition l in
  fusion (tri_fusion l1) (tri_fusion l2)
;;
```

Remarque : (Cousineau page 65) La structure de liste est coûteuse car la recherche d'un élément est, en moyenne, proportionnelle à la longueur de la liste. Si les éléments d'un ensemble ne peuvents être ordonnés ou ne sont pas trop nombreux, on pourra représenter leur ensemble par une liste, car le temps de recherche sera linéaire quelle que soit la représentation.

Tri fusion pour les vecteurs :

```
let tri_fusion_vect u =
  let n = vect_length u  in
  match n with
  | 1 -> u
  | 2 ->  if u.(0) > u.(1) then [|u.(1); u.(0)|]
      else u
  | _ ->  fusion_vect
          (tri_fusion_vect (sub_vect u 0 (n / 2)))
          (tri_fusion_vect (sub_vect u (n / 2) (n - n / 2)) );;
```

avec une fonction de fusion de deux vecteurs ordonnés, avec répétitions.

```
let fusion_vect u v =
  let i = ref 0 and j = ref 0 and k = ref 0 and
  m = vect_length u  and n = vect_length v  in
  let w = make_vect (m + n) 0 in
  while !i < m & !j < n do
    if u.(!i) < v.(!j) then
      begin
        w.(!k) <- u.(!i);
        i := !i + 1
      end
    else
      begin
        w.(!k) <- v.(!j);
        j := !j + 1
      end;
    k := !k + 1
  done;
  while !j < n do
    w.(!k) <- v.(!j);
    j := !j + 1;
    k := !k + 1;
  done;
  while !i < m do
```

```
    w.(!k) <- u.(!i);
    i := !i + 1;
    k := !k + 1;
 done;
w;;
```

7 Piles

Une *pile* est une suite d'éléments que l'on peut modifier en ajoutant un élément ou en retirant un élément en première position.

Ainsi la structure de liste de prête naturellement à décrire les opérations sur les listes.

7.1 Évaluation des formules postfixes

Réductions élémentaires :
valeur x unaire $f \to fx$
valeur x valeur y binaire$f \to fxy$

F est une valeur bien formée s'il existe une suite de réductions élémentaires qui transforment F en $[x]$.

Soit une liste $l = [l_1; \ldots; l_n]$, p est une pile vide en entrée.

pour $i = 1$ à n **faire**
  si l_1 est un nombre **alors**
    On l'empile
  fin si
  si l_1 est un opérateur unaire **alors**
    Dépiler $p = x$ empiler $l_1(x)$ sur p
  fin si
  si l_1 est un opérateur binaire **alors**
    Dépiler $p = x$, dépiler $p = y$, empiler $l_1(x, y)$ sur p
  fin si
fin pour
Dépiler $p = x$
Afficher x

La valeur d'une formule postfixe bien formée ne dépend pas de l'ordre des réductions. On vérifie que deux réductions distinctes portent sur des éléments distincts.

Analyse syntaxique : analyse grammaticale d'une langue.

Analyse lexicale : séparation en mots (uyn mot est une unité lexicale ou lexème).

l'analyseur syntaxique regroupe les mots en phrases selon les règles de la grammaire du langage (informatique), c'est la syntaxe concrète.

Pour les exemples voir la partie « programmes ».

8 Logique

L'art de la persuasion est très ancien, il a conduit les grecs de l'antiquité à tenter de définir des règles de raisonnement.
Plus récemment, lorsque l'on a voulu établir les mathématiques sur des bases solides, on a été amené à développer la logique. L'avènement de l'informatique a profité de ces travaux et inversement les mathématiciens ont trouvé matière à réflexion dans l'informatique.

8.1 Rudiments de logique booléenne

Une phrase mathématique est une suite de mots reliés par des connecteurs (conjonctions de coordination par exemple) et à laquelle on peut donner une et une seule valeur parmi deux : *vrai* ou *faux*, par exemple, mais on peux aussi utiliser n'importe quel ensemble ayant deux éléments. Dorénavant, une telle phrase sera appelée *proposition*.

Exemple 8.1

1. $1 + 1 = 2$
2. « *la logique c'est facile* »

On comprendra qu'il faut toujours préciser les conditions d'application...

Définition 8.1 *Un booléen (ou constante booléenne) est un élément d'un ensemble à deux éléments, par exemple* $\{\text{vrai}, \text{faux}\}$ *ou* $\{0, 1\}$.

Définition 8.2 *Un opérateur booléen peut être unaire ou binaire, il est défini par sa table de vérité (*i.e. *son graphe).*

8.1.1 Tables de vérité

p et q désignent les valeurs numériques (0 ou 1) des propositions P et Q :

1. non : on note $\text{non}\, P$ ou $\neg P$ ou $\overline{P}$. Le tableaux des valeurs de $\neg P$ (ou table de vérité) est :

P	$\neg P$
0	1
1	0

La valeur de $\neg P$ est : $1 - p$

2. et : on note $P \wedge Q$ ou PQ

P	Q	$P \wedge Q$
0	0	0
0	1	0
1	0	0
1	1	1

La valeur de $P \wedge Q$ est : pq

3. ou : $P \vee Q$ ou $P + Q$

P	Q	$P \vee Q$
0	0	0
0	1	1
1	0	1
1	1	1

La valeur de $P \vee Q$ est : $p + q - pq$

4. oubien : $P \oplus Q$

P	Q	$P \oplus Q$
0	0	0
0	1	1
1	0	1
1	1	0

La valeur de $P \oplus Q$ est : $p + q - 2pq$

$P \oplus Q$ est la *disjonction de* P et Q, on dit aussi ou exclusif, en théorie des ensembles il correspond à la différence symétrique.

5. $\Rightarrow$: par définition
$$(P \Rightarrow Q) \quad = \quad ((\text{non}\, P)\, \text{ou}\, Q)$$
6. $\Leftarrow$:
$$(P \Leftarrow Q) \quad = \quad (P\, \text{ou}\, (\text{non}\, Q))$$
7. $\Leftrightarrow$:
$$(P \Leftrightarrow Q) \quad = \quad (P \Leftarrow Q)\, \text{et}\, (P \Rightarrow Q)$$
8. nand : (« non et »)
$$(P\, \text{nand}\, Q) \quad = \quad \text{non}\, (P\, \text{et}\, Q)$$
9. nor : (« non ou »)
$$(P\, \text{nor}\, Q) \quad = \quad \text{non}\, (P\, \text{ou}\, Q)$$

Remarque 8.1 *Les valeurs correspondent aux fonctions indicatrices en théorie des ensembles. Par exemple si P est une proposition qui dépend d'un paramètre x et A est l'ensemble des éléments x pour lesquels $P(x)$ est vraie, alors la valeurs booléenne $p(x)$ de $P(x)$ est égale à la fonction indicatrice de A calculée en x.*

8.1.2 Tautologies

$\text{non}(\text{non}\, P) \equiv P$

$P \oplus Q \equiv \text{non}(P \Leftrightarrow Q)$, valeur : $p\overline{q} + \overline{p}q$ (on note $\overline{p} = 1 - p$).

Lois de Morgan :

$\text{non}\, (P\, \text{et}\, Q) \equiv (\text{non}\, P)\, \text{ou}\, (\text{non}\, Q)$

$\text{non}\, (P\, \text{ou}\, Q) \equiv (\text{non}\, P)\, \text{et}\, (\text{non}\, Q)$

De plus :

et et ou sont des lois associatives et elles sont mutuellement distributives l'une par rapport à l'autre.

Exercice 8.1 Vérifier
$$P \oplus Q = \overline{\overline{\overline{P}Q}\,\overline{P\overline{Q}}}$$

8.1.3 Fonctions booléennes

Définition 8.3 *Une formule booléenne est une expression contenant les constantes, des variables booléennes, des opérateurs booléens : non, et, ou, oubien,* $\Leftrightarrow$, $\Leftarrow$, $\Rightarrow$, *nand, nor.*
En particulier une fonction booléenne s'écrit à l'aide d'une formule booléenne.

Proposition 8.1 *Toute fonction booléenne peut être représentée par une formule (non unique) ne comportant que des et et des non.*

La démonstration consiste à exprimer tous les connecteurs logiques en fonction de ces deux là.

Définition 8.4 *Un littéral est une variable ou la négation d'une variable.*
La forme normale conjonctive *de la formule* f *est une formule qui représente* f *comme conjonctions de disjonctions de littéraux, chaque variable n'apparaissant qu'uns seule fois dans chaque disjonction.*
Respectivement forme normale disjonctive *en échangeant les mots conjonction et disjonction.*

Exemple 8.2 Forme conjonctive de disjonctions :

$$P_1\overline{P_2} + P_2\overline{P_3}$$

Forme disjonctive de conjonctions :

$$(P_1 + \overline{P_3})(P_1 + P_2)$$

Remarque 8.2 *Les formes normales sont uniques à permutation près.*

Lemme 8.1 *Toute fonction sur un corps commutatif fini est égale à une fonction polynomiale.*

Preuve : Soit $\mathbb{K}$ un corps commutatif fini. On raisonne par récurrence. Soit f une fonction de zéro variable, elle est constante, donc le lemme est vérifié. On suppose le lemme vérifié pour toute fonction de $n-1$ variables au plus. Soit f une fonction à n variables ($n \geq 1$), pour tout k dans $\mathbb{K}$ on considère l'application :

$$(x_1, \ldots, x_{n-1}) \mapsto f(x_1, \ldots, x_{n-1}, k)$$

C'est une fonction de $n-1$ variables à laquelle on peut appliquer l'hypoyhèse de récurrence, il existe donc un polynôme P_k à $n-1$ indéterminées tel que pour tous $x_1, \ldots, x_{n-1}$ dans $\mathbb{K}$

$$f(x_1, \ldots, x_{n-1}, k) = P_k(x_1, \ldots, x_{n-1})$$

La formule de Lagrange, par exemple, permet d'écrire

$$f(x_1, \ldots, x_n) = \sum_{k \in \mathbb{K}} c_k(x_1, \ldots, x_n) P_k(x_1, \ldots, x_n)$$

qui démontre le lemme car

$$c_k(x_1, \dots, x_n) = \left(\prod_{r \in \mathbb{K}, r \neq k} (k - r) \right) \prod_{r \in \mathbb{K}, r \neq k} (x_k - r)$$

□

Remarque 8.3 *On peut identifier une fonction booléenne de n variables avec une fonction (polynomiale) de n variables sur $\mathbb{Z}/2\mathbb{Z}$. Il y a donc 2^{2^n} fonctions booléennes de n variables.*

8.2 Circuits logiques

Un circuit logique est un dispositif physique muni d'entrées et de sorties. Les entrées et les sorties sont 0 ou 1. Nous utiliserons les circuits de la figure 4 et ceux qui s'en déduisent.

Le sens de parcourt d'un circuit élémentaire est indiqué par la forme de flèche des composants, appelés aussi *portes*. Les portes de la figure 4 ont une ou deux entrées et une sortie. Le petit disque en sortie indique la négation.

Toute fonction booléenne peut être représentée par un circuit logique et réciproquement.

8.2.1 Exemple de construction de circuit

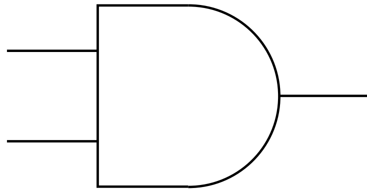
On se pose le problème de construire un circuit logique qui effectue la somme de trois bits : a, b et c. Voici les possibilités :

a	b	c	$a+b+c$	binaire
0	0	0	0	00
1	0	0	1	01
0	1	0	1	01
0	0	1	1	01
1	1	0	2	10
1	0	1	2	10
0	1	1	2	10
1	1	1	3	11

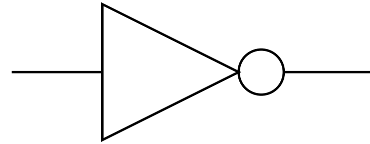
Il est clair qu'une fonction booléenne de a, b, c ne prenant que deux valeurs et $a+b+c$ en prenant quatre, il est nécessaire d'utiliser deux fonctions booléennes pour résoudre la problème.
La fonction f renverra le premier bit et g le second. f et g sont des fonctions polynômiales en a, b et c, elles sont donc de la forme :

$$x_1 + x_2 a + x_3 b + x_4 c + x_5 ab + x_6 bc + x_7 ca + x_8 abc$$

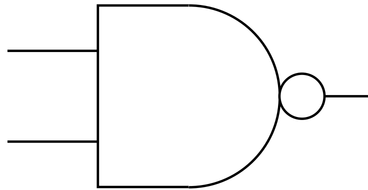
(puisque, pour tout t dans $\{0, 1\}$: $t^2 = t$).



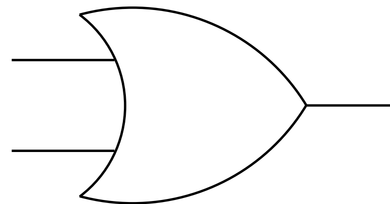
et



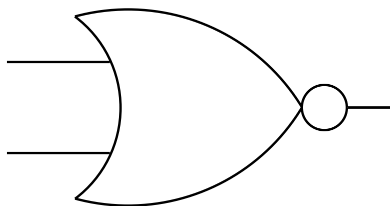
non



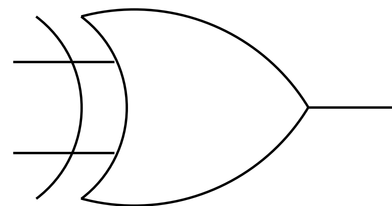
nand



ou



nor



oubien

FIG. 4 – Circuits élémentaires

Les coefficients des polynômes f et g sont déterminés grâce au tableau précédent, nous effectuons les calcul avec le logiciel libre Maxima (clone de MacSyma) :

```
(C3) ARRAY(x,8);
(D3)                x
(C4) f(a, b, c) :=x[1]+x[2]*a+x[3]*b+x[4]*c+x[5]*a*b+x[6]*b*c+
                  x[7]*c*a+x[8]*a*b*c;
(D4) f(a, b, c) := x_1+ x_2 a+ x_3 b+ x_4 c+ x_5 a b+ x_6 b c+
                  x_7 c a+ x_8 a b c
(C5) f(0,0,0)=0$
(C6) f(1,0,0)=1$
(C7) f(0,1,0)=1$
(C8) f(0,0,1)=1$
(C9) f(1,1,0)=0$
(C10) f(1,0,1)=0$
(C11) f(0,1,1)=0$
(C12) f(1,1,1)=1$
(C13) LINSOLVE([D5,D6,D7,D8,D9,D10,D11,D12],[x[1],x[2],x[3],x[4],
                x[5],x[6],x[7],x[8]]);
(D13) [x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 1, x_5 = - 2, x_6 = - 2,
                x_7 = - 2, x_8 = 4]
```

d'où

$$f(a,b,c) = a + b + c - 2ab - 2bc - 2ca + 4abc$$

Puis :

```
(C14) g(0,0,0)=0$
(C15) g(0,1,0)=0$
(C16) g(1,0,0)=0$
(C17) g(0,0,1)=0$
(C18) g(1,1,0)=1$
(C19) g(1,0,1)=1$
(C20) g(0,1,1)=1$
(C21) g(1,1,1)=1$
(C22)LINSOLVE([D14,D15,D16,D17,D18,D19,D20,D21],[x[1],x[2],x[3],
                                          x[4],x[5],x[6],x[7],x[8]]);
(D22)   [x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = 1, x_6 = 1,
                                                  x_7 = 1, x_8 = - 2]
```

d'où

$$g(a,b,c) = ab + bc + ca - 2abc$$

Ainsi f et g sont des fonctions booléennes, donc à valeurs dans $\{0,1\}$, par conséquent elles sont égales à leurs valeurs *modulo 2* :

$$f(a,b,c) = a + b + c \quad \text{modulo}\, 2$$
$$g(a,b,c) = ab + bc + ca \quad \text{modulo}\, 2$$

Or, modulo 2 : $A \oplus B \oplus C = a + b + c$ (en fait $f(a,b,c)$ est exactement la valeur de $A \oplus B \oplus C$ dans $\{0,1\}$).

D'autre part : $AC + BC = C(A + B)$ et $C(A + B) = C(A \oplus B) \quad \text{modulo}\, 2$
donc $AC + BC + CA = C(A \oplus B) \oplus AB$

En écrivant $A \oplus B \oplus C = (A \oplus B) \oplus C$ on voit que l'on peut utiliser cinq portes : $A \oplus B$, $(A \oplus B) \oplus C$, $C(A \oplus B)$, AB et $C(A \oplus B) \oplus AB$, d'où le circuit :

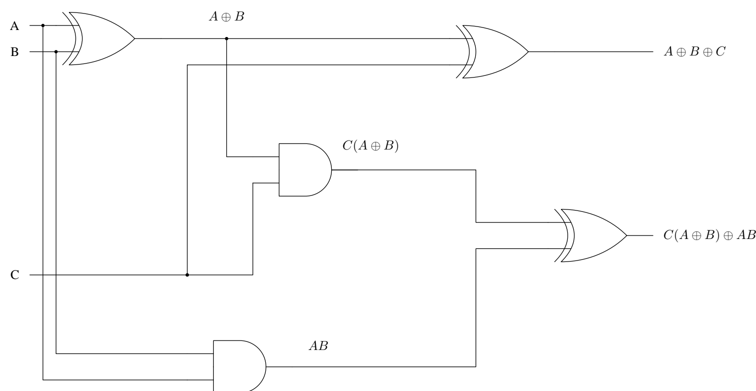


FIG. 5 – Additionneur 1 bits

Remarque 8.4 *Le circuit n'est pas unique, il dépend des connecteurs utilisés.*

9 Arbres

De manière générale, les définitions varient suivant les ouvrages, c'est pourquoi les définitions utilisées sont fixées au début de chaque épreuve de concours. Nous nous intéresserons principalements aux arbres binaires finis (*i.e. dont l'ensemble des nœuds est fini*). Dans la suite, après avoir défini les arbres généraux, tous les arbres seront binaires et finis sauf mention contraire.

9.1 Généralités

Définition 9.1 *Un arbre est un ensemble A d'éléments d'un ensemble ordonné $(N, \leq)$ appelés* nœuds *et de couples de nœuds (a, b), appelés* arêtes *(ou* liens*), tels que $a \leq b$, a est le* père *de b qui est le* fils *de a.*

- *(a, b) est une arête si et seulement si l'implication suivnte est vraie : $a \leq c \leq b$ alors $c = a$ ou $c = b$;*
- *A a un plus petit élément unique appelé* racine *;*
- *Tout nœud n'a qu'un père, sauf la racine qui n'a pas de père.*

Un nœud qui a un fils est dit nœud interne. *Un élément maximal est appelé* feuille. *Un nœud sans fils est maximal (ou nœud externe). L'ensemble vide est maximal.*

Cette définition implique qu'un arbre n'est pas vide.

Définition 9.2 *L'ensemble des éléments plus petits qu'un nœud a est l'ensemble de ses* ascendants *et l'ensemble des éléments plus grands que a est l'ensemble de ses* descendants.

Définition 9.3 *Une réunion d'arbres est une* forêt.

Définition 9.4 *Soit a un nœud d'un arbre A. L'ensemble des descendants de a est muni d'une structure de forêt dont les éléments sont appelés* branches *issues de a.*

Définition 9.5 *Le degré d'un nœud est le nombre de fils.*
Un arbre de degré d est un arbre dont le maximum des degrés des nœuds est égal à d.

9.2 Arbres binaires

Définition 9.6 *Un arbre* binaire *est un arbre de degré 2 tel que les fils de ses nœuds soient ordonnés : chaque nœud a un fils* droit *et un fils* gauche.

La figure 6 représente un arbre non binaire car le nœud N_5 est de degré 3.
Les arbres binaires ont une définition récursive :

Définition 9.7 *Un arbre binaire est soit un ensemble vide, soit de la forme (r, A_1, A_2) où r est un élément d'un ensemble ordonné N appelé* racine *et A_1, A_2 sont des arbres binaires dont les nœuds sont plus grands que r.*

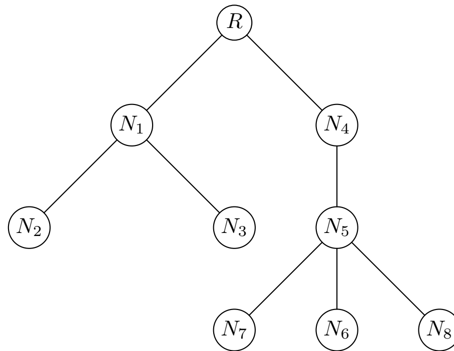


FIG. 6 – Représentation d'un arbre (non binaire)

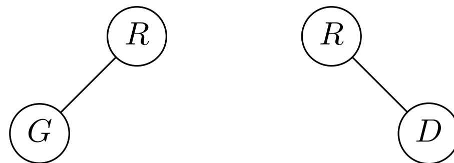


FIG. 7 – Deux arbres binaires à ne pas confondre

Remarque 9.1 *On prendra garde que les deux fils d'un nœud d'un arbre binaire ne sont pas interchangeables, voir la figure 7.*

Définition 9.8 *Un arbre* étiqueté *est une application définie sur un arbre A et à valeurs dans un ensemble E appelé ensemble des* étiquettes*.*

Un arbre homogène $\mathcal{A}$*, est une application d'un arbre A dans un ensemble N.*

Un arbre hétérogène *est un arbre étiqueté tel que l'ensemble des étiquettes soit une réunion $N \cup F$ où N est l'ensembles des valeurs des nœuds internes et F l'ensemble des valeurs des feuilles.*

En Caml les arbres binaires hétérogènes se définissent par exemple de la façon suivante :

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

où les feuilles sont de type `'f` et les nœuds de type `'n`.

Remarque 9.2 *Il y a d'autres façons de définir les arbres avec les listes, les vecteurs ou les enregistrements. On choisit la structure de données adaptée à la situation (éléments et traitement de ces éléments).*

Exercice 9.1 Un arbre à n nœuds a $n - 1$ arêtes.
(Raisonner par récurrence, en retirant la racine, ou bien en retirant une feuille).

Définition 9.9 *Un chemin c dans un arbre A est un $(p+1)$-uplet $(a_0, \ldots, a_p)$ où, pour tout j, a_{j+1} est le père de a_j ou bien a_j est le père de a_{j+1}.*
Le chemin est monotone si pour tout j, a_{j+1} est le père de a_j ou bien si pour tout j, a_j est le père de a_{j+1}.
La longueur *(ou* profondeur*) de c est p.*

Définition 9.10 *La hauteur d'un arbre est la plus grande longueur des chemins monotones ou le nombre maximal de nœuds (internes) sur un branche. On note $\mathrm{haut}(A)$.*

Exercice 9.2 Encadrement du nombre n de nœuds d'un arbre en fonction de la hauteur (formule valable pour tout arbre ; $\ell = \mathrm{haut}(A)$ et d désigne le degré de l'arbre, soit $d = 2$ pour un arbre binaire) :

$$\ell \leq n \leq d^\ell - 1$$

Définition 9.11 *On dit qu'un arbre binaire est* équilibré *si tous les chemins (croissants) de la racine à une feuille sont de hauteur $\mathrm{haut}(A)$ ou $\mathrm{haut}(A) - 1$.*

Définition 9.12 *On dit qu'un arbre binaire est* complet *si tous ses nœuds, qui ne sont pas des feuilles, ont deux fils.*

Par exemple, un arbre binaire complet ayant n nœuds internes a $n + 1$ feuilles (preuve par récurrence : c'est vrai si $n = 0$, sinon la racine n'est pas une feuille et on applique l'hypothèse de récurrence aux deux branches de la racine), on en déduit :

Proposition 9.1 *Il y a bijection entre l'ensemble des arbres binaires ayant n nœuds et l'ensemble des arbres binaires complets ayant $2n + 1$ nœuds.*

On considère un arbre a comme homogène et étiqueté par un ensemble N, on ajoute à chaque feuille deux fils dont les étiquettes n'appartiennent pas à N, on obtient un arbre hétérogène complet. L'application réciproque consiste à «effeuiller»[1].

Exercice 9.3 Nombre d'arbres binaires de taille n : $\dfrac{1}{n+1}\binom{2n}{n}$.
La racine est liée à deux arbres binaires de tailles k et $n-1-k$, d'où, en appelant b_m le nombre d'arbres binaires de taille m :

$$b_0 = 1, \qquad b_n = \sum_{0 \leq k \leq n-1} b_k b_{n-1-k}$$

Posons $b(x) = \sum_{k \geq 0} b_k x^k$, il vient : $b(x)^2 = \sum_{m \geq 0} (\sum_{0 \leq k \leq m} b_k b_{m-k}) x^m$ d'où $b(x)^2 = \sum_{m \geq 0} b_{m+1} x^m$, soit $x b(x)^2 = -1 + b(x)$. On en déduit que $b(x) = \dfrac{1 - \sqrt{1-4x}}{2x}$ car $b(0)$ est défini et vaut 1. b est donc une fonction C^∞ au voisinage de 0 et son développement en série entière en 0 est :

$$b(x) = \sum_{k \geq 0} \frac{1}{n+1} \binom{2n}{n} x^n$$

[1] dans hétérogène, il y a érogène.

Donc
$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

Remarque 9.3 *On peut représenter un arbre binaire en binaire de la façon suivante :*
- 1 *est la racine ;*
- *Si* $\overline{a_m \cdots a_0}$ *est le nombre binaire qui représente le nœud* $\mathcal{N}$ *alors le nombre binaire* $\overline{a_m \cdots a_0 0}$ *représente le fils gauche et* $\overline{a_m \cdots a_0 1}$ *le fils droit.*

*(*si 0 est la racine, son fils gauche est 00*).*

9.2.1 Opérations élémentaires sur les arbres binaires hétérogènes

On reprend l'implémentation 9.2 pour calculer

1. la hauteur

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

2. le nombre de nœuds

```
let rec nb_noeuds = function
  |Feuille _ -> 0
  |Noeud (gauche, _, droit) ->
    1 + nb_noeuds gauche + nb_noeuds droit;;
```

3. le nombre de feuilles

```
let rec nb_feuilles = function
  |Feuille _ -> 1
  |Noeud (gauche, _, droit) ->
    nb_feuilles gauche + nb_feuilles droit;;
```

9.2.2 Parcours d'un arbre binaire hétérogène complet

Il y a essentiellement trois façons de parcourir un arbre en traitant ses nœuds :

1. le parcours préfixe ;
   - traitement de la racine ;
   - parcours de la branche gauche ;
   - parcours de la branche droite.
2. le parcours infixe :
   - parcours de la branche gauche ;
   - traitement de la racine ;
   - parcours de la branche droite.
3. le parcours postfixe :
   - parcours de la branche gauche ;

– parcours de la branche droite ;
– traitement de la racine.

Par exemple, on définit les actions `traitement1` et `traitement2` puis :

```
let rec parcours = function
    |Feuille f -> traitement1 f
    |Noeud (gauche, _, _) -> parcours gauche
    |Noeud (_, _, droite) -> parcours droite
    |Noeud (_, n, _) -> traitement2 n
;;
```

On peut évidemment inverser droite et gauche. Il existe aussi le parcours en largeur (voir plus loin et les travaux dirigés).

9.3 Arbres binaires homogènes

En Caml un arbre homogène complet peut se définir comme

```
type 'f Arbre =
  |Feuille of 'f
  |Noeud of ('f Arbre) * 'f * ('f Arbre);;
```

(tous les nœuds internes ont deux fils, feuilles et nœuds ont le même type d'étiquette).

On peut considérer un arbre homogène comme un arbre hétérogène complet dont les feuilles sont vides :

```
type 'f Arbre =
  |Vide
  |Noeud of ('f Arbre) * 'f * ('f Arbre);;
```

Par la suite, sauf mention du contraire, on utilisera les arbres homogènes étiquetés par les entiers :

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

9.3.1 Accès aux nœuds d'un arbre binaire homogène

Dans un arbre binaire quelconque : Soit A un arbre binaire de taille n, on calcule la longueur moyenne $L(n)$ d'un chemin (monotone) de la racine a_0 à un nœud a. A_1 et A_2, les branches gauche et droite issues de a_0, sont des arbres ayant respectivement j et $n-1-j$ nœuds avec une probabilité de $\frac{1}{n}$ (on suppose que les n possibilités sont équiprobables). La longueur moyenne est la somme des produits de la longueur d'un chemin par la probabilité que ce chemin soit choisi (c'est l'espérance de la variable aléatoire dont les valeurs sont les longueurs des chemins), par exemple si A_1 est vide $L(j) = 0$, sinon $j = \#(A_1)$ et :

$$L(j) = \sum_{a \in A_1} \frac{1}{j} \ell(a)$$

(où $\ell(a)$ est la longueur du chemin de la racine de A_1 à a).

Soit Ω l'espace de probabilité dont les évènements élémentaires sont les nœuds de l'arbre (chaque chemin de la racine à un nœud étant identifié avec son extrémité). Notons X la variable aléatoire dont les valeurs sont les longueurs des chemins et ϵ la variable aléatoire égale à l'extrémité. Ainsi :

$$\begin{aligned}(X=\ell) &= \cup_{0\leq j\leq n}(X=l,\#A_1=j)\\ &= \cup_{0\leq j\leq n}(X=l,\#A_1=j,\epsilon\in A_1)\cup(X=l,\#A_1=j,\epsilon\in A_2)\end{aligned}$$

Prenons les probabilités

$$P(X=\ell)=$$
$$\sum_{0\leq j\leq n} P(X=l,\#A_1=j,\epsilon\in A_1)+P(X=l,\#A_1=j,\epsilon\in A_2)$$

utilisant les probabilités conditionnelles :

$$\begin{aligned}&=\sum_{0\leq j\leq n} P(X=l,\epsilon\in A_1/\#A_1=j)\frac{1}{n}+P(X=l,\epsilon\in A_2/\#A_1=j)\frac{1}{n}\\ &=\sum_{0\leq j\leq n} P(X=l/\epsilon\in A_1,\#A_1=j)\frac{j}{n}\frac{1}{n}+P(X=l/\epsilon\in A_2,\#A_1=j)\frac{n-1-j}{n}\frac{1}{n}\end{aligned}$$

On remarque que $P(X=l,\epsilon\in A_1/\#A_1=j)$ est la probabilité qu'un chemin ait une longueur ℓ sachant que son extrémité est dans A_1, c'est aussi la probabilité que, dans A_1, un chemin ait une longueur égale à $\ell-1$. Donc

$$\sum_{\ell\geq 0} P(X=l,\epsilon\in A_1/\#A_1=j)(\ell-1)=L(j)$$

On en déduit que l'espérance de X qui est la valeur moyenne cherchée vérifie :

$$\begin{aligned}L(n) &= \sum_{\ell\geq 0}\sum_{0\leq j\leq n-1} P(X=l/\epsilon\in A_1,\#A_1=j)((\ell-1)+1)\frac{j}{n}\frac{1}{n}+\\ &\quad P(X=l/\epsilon\in A_2,\#A_1=j)((\ell-1)+1)\frac{n-1-j}{n}\frac{1}{n}\\ &=\sum_{0\leq j\leq n-1}(L(j)+1)\frac{j}{n^2}+(L(n-1-j)+1)\frac{n-1-j}{n^2}\\ &=n\frac{n-1}{n^2}+\sum_{0\leq j\leq n-1} L(j)\frac{j}{n^2}+L(n-1-j)\frac{n-1-j}{n^2}\end{aligned}$$

or

$$\sum_{0\leq j\leq n-1} L(n-1-j)\frac{n-1-j}{n^2}=\sum_{0\leq j\leq n-1} L(j)\frac{j}{n^2}$$

(parcourir l'intervalle dans les deux sens). Finalement :

$$L(n) = \frac{n-1}{n} + 2 \sum_{0 \leq j \leq n-1} L(j) \frac{j}{n^2}$$

Remarquons qu'en posant $f(n) = nL(n)$ et en calculant $(n+1)f(n+1) - nf(n)$ on trouve une équation assez simple dont $n \mapsto n+1$ est solution. Ceci conduit à poser : $g(n) = \frac{n}{n+1} L(n)$, il vient :

$$g(n+1) - g(n) = \frac{-2}{n+1} + \frac{4}{n+2}$$

Cette équation se résout aisément, on trouve ainsi une expression équivalente à $2 \ln n$; posant $h(n) = \sum_{1 \leq k \leq n} \frac{1}{k}$ il vient :

$$\begin{aligned}
g(n) &= g(1) + \sum_{1 \leq k \leq n} \frac{-2}{k} + \frac{4}{k+1} \\
\sum_{2 \leq k \leq n} \frac{-2}{k} &= -2(h(n) - 1) \\
\sum_{2 \leq k \leq n} \frac{4}{k+1} &= 4 \sum_{k+1=3}^{k+1=n+1} \frac{1}{k+1} \\
&= 4 \left(h(n+1) - \left(1 + \frac{1}{2} \right) \right)
\end{aligned}$$

d'où, en tenant compte que $g(1) = 0$:

$$\begin{aligned}
g(n) &= g(1) + 2h(n) + \frac{4}{n+1} - 4 \\
L(n) &= \frac{2(n+1)}{n} h(n) - 4
\end{aligned}$$

Or $h(n) = \ln n + \gamma + o(1)$ (γ est la constante d'Euler).

Dans un arbre binaire complet équilibré ($2^{m+1} - 1$ nœuds) : Il y a 2^k nœuds de profondeur k donc la longueur moyenne des chemins est :

$$\frac{1}{2^{m+1} - 1} \sum_{1 \leq k \leq m} k 2^k$$

Après un calcul bien connu :

$$\frac{m2^m - (m+1)2^m + 1}{2^{m+1} - 1} \underset{m \to +\infty}{\sim} m$$

Or $m = \log_2 n$ donc la complexité est un $O(\ln n)$.

9.3.2 Opérations sur les arbres binaires homogènes

Vue l'identification entre arbre binaire homogène et arbre binaire hétérogène dont les feuilles sont de type vide, les calculs de nombres de feuilles ou de nœuds se déduisent du cas hétérogène (exercice).

9.4 Arbres binaires de recherche

Définition 9.13 *Soit* $(E, \leq)$ *un ensemble ordonné. On appelle* arbre binaire de recherche *dans* E *tout arbre binaire* A *étiqueté par* E *tel que (*ϵ *est la fonction d'étiquetage) :*

1. *pour tout nœud* r *de* A *et tout nœud* g *de la branche gauche de* r*, on a :* $\epsilon(g) \leq \epsilon(r)$ *;*
2. *pour tout nœud* r *de* A *et tout nœud* d *de la branche droite de* r*, on a :* $\epsilon(r) < \epsilon(d)$*.*

Remarque 9.4 *Dans la plupart des exemples* E *est l'ensemble des entiers positifs.*

Exemple 9.1 La figure 8 montre un arbre binaire de recherche et un arbre binaire qui n'est pas un arbre binaire de recherche parce que la branche droite du nœud d'étiquette 2 contient un nœud dont l'étiquette *n'est pas* strictement supérieure à 2.

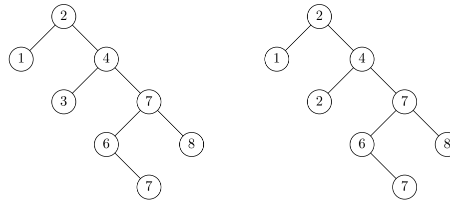


FIG. 8 – L'arbre binaire de recherche est à gauche

9.4.1 Test d'un arbre binaire de recherche

On prend la définition suivante des arbres binaires

```
type Arbre =
  |Vide
  |Noeud of Arbre) * int * Arbre;;
```

On écrit une fonction qui renvoie un triplet formé d'un booléen (*oui* si l'arbre est un arbre binaire de recherche et *non* sinon), la plus petite étiquette et la plus grande. On convient que la borne supérieure des étiquettes est $-\infty$ (soit `-1000000`) et la borne inférieure $+\infty$ (soit `1000000`).

Le programme est écrit d'après une définition récursive des arbres binaires de recherche :

– un arbre vide est un arbre binaire de recherche ;
– les fils gauche et fils droit de la racine d'un arbre binaire de recherche sont des arbres binaires de recherche ;
– l'étiquette de la racine d'un arbre binaire de recherche non vide est plus grande que les étiquettes de son fils gauche et strictement plus petite que les étiquettes de son fils droit.

Ainsi

```
let rec arbre_bin_rech = function
  |Vide -> (true, 1000000, -1000000)
  |Noeud (gauche, n, droit) ->
    match (arbre_bin_rech gauche, arbre_bin_rech droit) with
     ((testg, ming, maxg), (testd, mind, maxd)) ->
                                          (testg && testd && (maxg <= n) &&
            (n < mind), min (min ming mind) n, max (max maxg maxd) n)
;;
```

Commentaire : `arbre_bin_rech gauche` renvoie le triplet

```
(booléen, plus petite étiquette, plus grande étiquette)
```

appelés (avec des noms « parlants ») :

```
(testg, ming, maxg)
```

où `testg` vaut `vrai`, si la branche gauche est un arbre binaire de recherche. De même pour `droit` : `(testd, mind, maxd)`.
Le booléen `testg && testd && (maxg <= n) && (n < mind)` vaut vrai si et seulement si l'arbre est un arbre binaire de recherche.

Exemple 9.2 Soit a un arbre binaire de recherche :

```
let a =
Noeud (
Noeud (Vide, 1, Vide), 2,
Noeud (
Noeud (Vide, 3, Vide), 4,
Noeud(
Noeud (Vide, 6,
Noeud(Vide, 7, Vide)),7,
Noeud (Vide, 8, Vide) ) ) );;
```

et b un arbre binaire qui n'est pas un arbre binaire de recherche

```
let b =
Noeud (
Noeud (Vide, 1, Vide), 2,
Noeud (
Noeud (Vide, 2, Vide), 4,
Noeud(
```

```
   Noeud (Vide, 6,
   Noeud(Vide, 7, Vide)),7,
   Noeud (Vide, 8, Vide) ) ) );;

#arbre_bin_rech b;;
 - : bool * int * int = false, 1, 8
#arbre_bin_rech a;;
 - : bool * int * int = true, 1, 8
```

9.4.2 Recherche dans un arbre binaire de recherche

Dans le programme suivant, on recherche l'élément x dans l'arbre a. Si $a = \emptyset$, x n'est pas dans l'arbre. Ensuite si x est égal à l'étiquette n du nœud, x figure dans l'arbre. Enfin si les filtrages précédents ont échoués, c'est que $x < n$ et il faut le chercher dans la branche gauche, ou bien $x > n$ et il faut le chercher dans la branche droite.

```
 let rec y_est x a = match a with
   |Vide -> false
   |Noeud ( _, n, _) when x = n -> true
   |Noeud (gauche, n, _) when x < n -> y_est x gauche
   |Noeud (_, _, droit) -> y_est x droit
;;
```

Remarque 9.5 *La recherche dans un arbre binaire de recherche nécessite au plus n comparaisons (une par nœud), ce nombre est atteint dans le cas d'un peigne. Le nombre moyen de comparaison est égal à la hauteur moyenne d'un arbre binaire (ou longueur moyenne des chemins), ce nombre est donc un $O(\ln n)$.*

9.4.3 Insertion dans un arbre binaire de recherche

L'insertion d'un élément x dans un arbre binaire de recherche A, a pour algorithme (récursif) :

- si A est vide, le résultat de l'insertion est l'arbre ayant un seul nœud, étiqueté par x ;
- sinon soient $\epsilon(r)$ l'étiquette de la racine r, A_g, A_d les branches gauche et droites de A et
  - si $x \leq \epsilon(r)$, on insère x dans A_g ;
  - si $x > \epsilon(r)$, on insère x dans A_d.

En Caml :

```
let rec insere_dans_arbre x a = match a with
   |Vide -> Noeud (Vide, x, Vide)
   |Noeud (gauche, r, droit) when x <= r ->
                      Noeud (insere_dans_arbre x gauche, r, droit)
   |Noeud (gauche, r, droit) ->
```

```
                    Noeud (gauche, r, insere_dans_arbre x droit)
;;
```

La complexité est la même que pour la recherche.

Exercice 9.4 Écrire un programme de transformation de liste en arbre binaire de recherche et un programme de calcul des hauteurs. Comparer les arbres construits avec les listes $[5; 1; 8; 4]$ et $[4; 8; 1; 5]$.

9.4.4 Suppression dans un arbre binaire de recherche

Il s'agit de supprimer la première occurrence d'un élément x dans un arbre binaire de recherche. L'algorithme est semblable aux précédents. On suppose que l'arbre n'est pas vide. Si l'élément à supprimer est la racine r, on supprime la racine, si $x < \epsilon(r)$, on recherche x dans le fils gauche, ce qui revient à supprimer une racine, si $x > \epsilon(r)$ on fait de même dans la branche droite.

Supprimer la racine d'un ABR est facile si l'une des deux branches est vide. On suppose qu'elles ne le sont pas. On cherche alors *l'élément le plus à droite* de la branche gauche, son étiquette est la plus grande possible, ce nœud n'a pas de fils droit par définition, il est relié éventuellement à son père et à son fils gauche. Étudions les quatre cas de figure (les trois premiers sur la figure 9). On note n le nœud le plus à droite :

1. n a un père k et un fils m tels que $\epsilon(k) < \epsilon(m) \leq \epsilon(n)$, on supprime n, k devient le père de m ;
2. n est la racine et a un fils m, on supprime n, alors m devient la nouvelle racine ;
3. n a un père m et n'a pas de fils, on supprime n, m a un fils droit vide ;
4. n n'a ni père ni fils, on supprime n pour obtenir l'arbre vide.

L'algorithme de suppression du terme le plus à droite est :
- si l'arbre est vide , on ne supprime rien ;
- si la branche droite est vide, on a le bon nœud (on renvoie le nouvel arbre qui n'est autre que la branche gauche et le nœud) ;
- sinon, on supprime le terme le plus à droite dans la branche droite.

```
let rec supp_droit = function
  |Vide -> raise Not_found
  |Noeud (gauche, n, Vide) -> (gauche, n)
  |Noeud (gauche, n, droit) ->
     let (nv_droit, plus_a_droite) = supp_droit droit in
Noeud (gauche, n, nv_droit), plus_a_droite
;;
```

Algorithme de suppression de la racine d'un ABR :
- si l'arbre est vide on ne fait rien ;
- si la branche gauche est vide on renvoie la branche droite ;
- si la branche droite est vide on renvoie la branche gauche ;

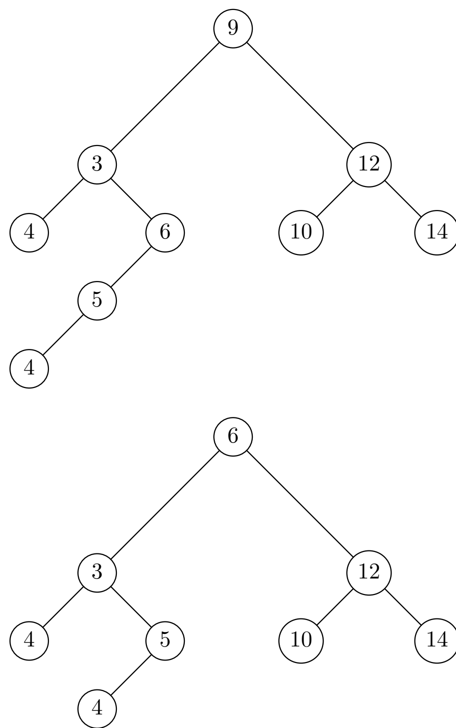


FIG. 9 – Suppression d'un nœud

– sinon on renvoie l'arbre dont la branche droite est la même et la branche gauche est l'arbre dans lequel on a retiré le nœud le plus à droite de l'ancienne branche gauche, la racine ayant pour étiquette l'étiquette du nœud le plus à droite de la branche gauche.

Voir un exemple figure 10 En Caml :

```
let supp_racine = function
  |Vide -> raise Not_found
  |Noeud (gauche, n, Vide) -> gauche
  |Noeud (Vide, n, droit) -> droit
  |Noeud (gauche, r, droit) ->
     let (nv_gauche, plus_a_droite) = supp_droit gauche in
       Noeud (nv_gauche, plus_a_droite, droit)
;;
```

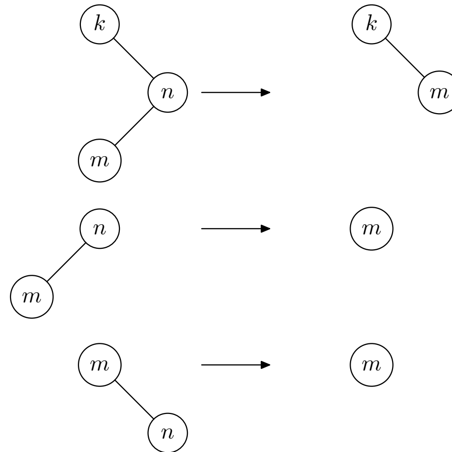


FIG. 10 – Suppression de la racine

Algorithme de suppression d'un élément x dans un ABR a :

– si l'arbre est vide il n'y a rien à supprimer ;
– si x est égal à l'étiquette de la racine de l'arbre, on supprime la racine ;
– si x est inférieur à l'étiquette de la racine de l'arbre, on supprime x dans la branche gauche ;
– si x est strictement supérieur à l'étiquette de la racine de l'arbre, on supprime x dans la branche droite.

En Caml :

```
let rec supp_elem x a = match a with
|Vide -> raise Not_found
|Noeud (gauche, r, droit) when x = r -> supp_racine a
|Noeud (gauche, n, droit) when x <= n ->
                              Noeud (supp_elem x gauche, n, droit)
```

```
  |Noeud (gauche, n, droit) ->
                              Noeud (gauche, n, supp_elem x droit)
;;
```

9.5 Files de priorité et tas

9.5.1 Files de priorité

Dans une pile le dernier arrivé est le premier à sortir (LIFO *last in, first out*). Dans une file d'attente, le premier arrivé est le premier à sortir (FIFO *first in, first out*).
Dans une file de priorité (appelée aussi *maximier*), l'arrivant est classé suivant sa priorité, le premier à sortir est celui qui a la priorité la plus grande. Cette structure est facilement décrite par un arbre homogène.

Définition 9.14 *Soit* $(X, \leq)$ *un ensemble ordonné, on appelle* file de priorité *dans* X*, tout arbre* A *étiqueté par* X *tel que tout nœud ait une étiquette supérieure à celles de ses fils.*

Dans la suite toutes les files de priorité seront représentées par des arbres binaires. Les opérations que l'ont doit savoir faire sont l'insertion et la suppression de la racine.

Suppression de la racine.

La suppression va s'opérer de la manière suivante :

- échange de la racine avec une feuille ;
- suppression de cette feuille ;
- placer l'étiquette de la racine à la bonne place en faisant des échanges d'étiquettes entre père et fils le long d'une branche (*percolation*).

Algorithme de percolation : remarquons que la suppression d'une feuille ne change pas la structure de file de priorité, que l'échange ait eu lieu ou non.

- si la racine de l'arbre a une étiquette strictement plus grande que les étiquettes de ses fils, on échange l'étiquette de la racine avec la plus grande étiquette des fils puis on effectue la percolation sur la branche modifiée ;
- sinon on ne fait rien.

Programme de percolation :

```
let rec percole = function
  |Noeud( Noeud( gg, ng, gd), n, Noeud( dg, nd, dd))
      when ng > n && ng >= nd  ->
    Noeud( percole (Noeud( gg, n, gd)), ng, Noeud( dg, nd, dd))
  |Noeud(  Noeud( gg, ng, gd), n, Noeud( dg, nd, dd))
      when nd > n && nd >= ng ->
    Noeud(  Noeud( gg, ng, gd), nd, percole (Noeud( dg, n, dd)))
  |a -> a
;;
```

La preuve du programme se fait par induction : si l'arbre a moins d'un nœud, c'est évident, sinon la suppression d'une feuille ne changeant pas la structure de file de priorité de l'arbre, les fils de la racine sont des files de priorité ayant un élément de

moins. Si l'étiquette de la racine est supérieure aux étiquettes des racines des fils, on a une file de priorité. Sinon, on échange l'étiquette de la racine avec l'étiquette du fils qui est la plus grande. Ainsi l'étiquette de la racine a la priorité la plus élevée, l'un des fils est inchangé et est donc une file de priorité. On applique l'algorithme de percolation à l'autre arbre, d'après l'hypothèse de récurrence on obtient une file de priorité. Par définition, l'arbre obtenu est une file de priorité.
La complexité de la percolation est un $O(m)$ où m est la taille de l'arbre (au pire, on a un peigne et on fait $n-1$ échanges).
Algorithme de suppression de la racine d'une file de priorité : on doit écrire l'algorithme de recherche et suppression d'un nœud terminal en mettant son étiquette à la racine.

Suppression du nœud terminal :
- si l'arbre est vide on ne fait rien ;
- si la racine a deux fils vides, on renvoie l'arbre vide ;
- si le fils gauche n'est pas vide, on supprime un nœud terminal du fils gauche ;
- si le fils droit n'est pas vide, on supprime un nœud terminal du fils droit.

Comme il faut garder l'étiquette du nœud supprimé, on utilise une référence.

```
let etiquet = ref 0 in
let rec supp_feuille = function
  |Vide -> invalid_arg "arbre␣vide"
  |Noeud( Vide, n, Vide) -> etiquet := n; Vide
  |Noeud( gauche, n, Vide) -> Noeud( supp_feuille gauche, n, droit)
  |Noeud( gauche, n, droit) -> Noeud( gauche, n, supp_feuille droit)
;;
```

Programme de suppression de la racine dans une file de priorité non vide : on supprime une feuille quelconque en copiant son étiquette dans une référence. Cette étiquette est placée à la racine de l'arbre, il suffit alors d'appliquer l'algorithme de percolation.

```
let supp_raci_file a =
  let etiquet = ref 0 in
  let rec supp_feuille = function
    |Vide -> invalid_arg "arbre␣vide"
    |Noeud( Vide, n, Vide) -> etiquet := n; Vide
    |Noeud( gauche, n, Vide) -> Noeud( supp_feuille gauche, n, Vide)
    |Noeud( gauche, n, droit) -> Noeud( gauche, n, supp_feuille droit)
  in let t = supp_feuille a in match t with
    |Vide -> invalid_arg "arbre␣vide"
    |(Noeud(g, n, d)) -> percole (Noeud(g, !etiquet, d))
;;
```

Insertion dans une file de priorité.
Soit une étiquette x à insérer dans un arbre de priorité a.
- si a est vide, on renvoie l'arbre dont la racine a pour étiquette x et a deux fils vides ;
- si x est supérieur à la racine de a, on échange x avec l'étiquette de la racine et on insère l'étiquette de la racine dans l'un des fils ;

– si x est inférieur à l'étiquette de la racine, on l'insère dans l'un des fils.

d'où

```
let rec insere_file x = function
  |Vide -> Noeud( Vide, x, Vide)
  |Noeud( gauche, n, droit) when x > n ->
     Noeud( insere_file n gauche, x, droit)
  |Noeud( gauche, n, droit) -> Noeud( gauche, n, insere_file x droit)
;;
```

Dans ce programme on choisi de faire l'insertion dans la branche gauche ou la branche droite suivant la taille de l'élément à insérer. Cela peut conduire à des arbres déséquilibrés. Pour obtenir, en moyenne, des arbre équilibrés, on peut insérer *au hasard* (à faire en exercice : utiliser la fonction `random__int 2` qui renvoie un nombre pseudo aléatoire égal à 0 ou 1).

La complexité de l'insertion est encore de l'ordre de $O(m)$ où m est la taille de la file.

On remarque que le nombre maximal d'échanges est minimal lorsque l'arbre est équilibré, *i.e.* lorsque la hauteur de l'arbre est minimale relativement à sa taille.

9.5.2 Tas

On a vu que dans le cas des files de priorité, l'insertion ou la suppression pouvait déséquilibrer les arbres et augmenter la complexité.

Définition 9.15

- *Le niveau d d'un arbre binaire homogène est l'ensemble des éléments situés à une distance d de la racine.*
- *On dit qu'un arbre binaire homogène de hauteur h est* parfait *(ou* complet*, mais on a déjà utilisé ce mot) si pour tout d dans $[0; h-1]$, le niveau d de l'arbre possède 2^d éléments et si les nœuds du niveau h sont à gauche.*

(on pourra utiliser un arbre binaire hétérogène dont les feuilles sont vides).

Définition 9.16 *On appelle* tas *(*heap *en anglais) un arbre binaire homogène parfait qui est une file de priorité.*

(c'est très voisin des files vues précédemment).

De manière générale, on représente les arbres par des tableaux en numérotant les nœuds de gauche à droite. Pour les arbres binaires complets, les nœuds du niveau d sont numérotés de 2^d à $2^{d+1}-1$, le nœud numéro j a pour père le nœud numéro $\lfloor \frac{j}{2} \rfloor$ et inversement le nœud numéro k a pour fils gauche le nœud numéro $2k$ et pour fils droit le nœud numéro $2k+1$.

Dans l'exemple de la figure 11, le tas est représenté par le tableau

$$[|20; 17; 16; 15; 13; 14; 12; 11; 7; 9|]$$

On dit que l'arbre est parcouru en *largeur*.

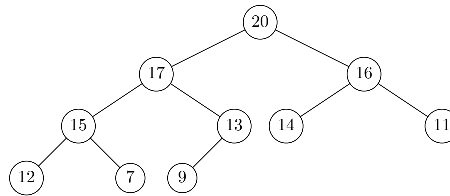


FIG. 11 – Un tas

Pour définir un type Tas il faut prendre en compte que l'on doit pouvoir préciser la taille maximale de l'arbre et modifier le nombre d'éléments du tas. Posons :

```
type Tas = {mutable taille_tas : int ; contenu_tas : int vect};;
# let t = {taille_tas = 10; contenu_tas = make_vect 15 0};;
```

`taille_tas` est le nombre d'éléments du tas, soit le nombre d'étiquettes non vides de l'arbre. `contenu_tas` est la définition du tableau et contient donc la taille maximale de l'arbre (les coefficients sont initialisés à 0, la distinction entre le nœud d'étiquette 0 et le nœud vide est faite avec `taille_tas`).

9.5.3 Transformation de tas en arbres

Ou, ce qui revient au même, on transforme un vecteur v en arbre a. L'algorithme est le suivant : on parcourt l'arbre en largeur et on procède par récurrence. Construction du i^{e}nœud :

- si i est supérieur à la taille de l'arbre, le nœud est vide ;
- sinon on place le coefficient v_i comme étiquette du nœud numéro i puis on traite le fils gauche (nœud $2i + 1$) et le fils droit (nœud $2i + 2$).

```
let vect_vers_arbre v = traite_noeud 0
                        where
                        rec traite_noeud i =
if i >= (vect_length v) then Vide
else
  Noeud( traite_noeud (2 * i + 1), v.(i), traite_noeud (2 * i + 2))
;;
```

9.5.4 Suppression de la racine dans un tas

Le principe est simple, on échange l'étiquette de la racine avec l'étiquette du dernier élément, on supprime le dernier élément, puis on percole l'étiquette de la racine jusqu'à ce qu'elle trouve sa place.

On a besoin d'un programme d'échange :

```
let echange i j v =
let u = ref v.(i) in
  v.(i) <- v.(j); v.(j) <- !u;;
```

La percolation :

```
let percole tas =
  let n = tas.taille_tas and v = tas.contenu_tas in
    traite_noeud 0 v where
      rec traite_noeud i v =
                      let j = 2 * i + 1 in
                      let  p = ref j in
                        if j < n then
                          begin
                            if j + 1 < n && v.(j) < v.(j + 1) then p := j + 1;
                            echange i !p v;
                            traite_noeud j v
                          end;;
```

La suppression de la racine :

```
let supp_rac_tas t =
  let m = t.taille_tas - 1 in
    t.contenu_tas.(0) <- t.contenu_tas.(m);
    t.taille_tas <- t.taille_tas - 1;
    percole t;;
```

Autre version :

```
let supp_racine t =
  let v = t.contenu_tas and m = t.taille_tas in
    echange 0 (m - 1) v;
     t.taille_tas <- m - 1;
     traite_noeud 0 where
      rec traite_noeud i =
        let j = 2 * i + 1 in let  p = ref j in
          if j + 1 < m && v.(j) < v.(j + 1) then p := j + 1;
          if j < m && v.(i) < v.(!p) then echange i !p v;;
```

9.5.5 Insertion dans un tas

Insertion dans un tas : on met l'élément à insérer à la première place libre puis on le remonte pour le mettre à la bonne place.

```
  let insere x t =
  if t.taille_tas >= vect_length t.contenu_tas then
    failwith "le␣tas␣est␣plein";
  t.contenu_tas.(t.taille_tas) <- x;
  t.taille_tas <- t.taille_tas + 1;
  refait_tas t where
    refait_tas tas =
  let d = ref (tas.taille_tas - 1) in
  let p = ref ((!d - 1) / 2)  and
```

```
v = ref tas.contenu_tas in
while (!d >= 1) && (!v.(!d) > !v.(!p)) do
  echange !d !p !v;
  d := !p;
  p := (!d - 1) / 2
done;
!v;;
```

Preuve : l'arbre étant complet, il le reste. L'algorithme se termine car d décroît strictement et il donne le résultat attendu. On considère l'invariant de boucle suivant :

On note p le père, d et d' les fils gauche et droit.

Les branches de racines d et d' sont des files de priorité et l'étiquette de p est supérieure à celle de d' : $\epsilon(p) \geq \epsilon(d')$.

C'est vérifié en entrée de boucle. Dans la boucle : si $\epsilon(p) epsilon(d)$, on ne fait rien, l'arbre de racine p est une file de priorité donc t est un tas et l'invariant est bien conservé puisque rien n'a changé.

Si $k = \epsilon(p) < \epsilon(d) = j$, comme $\epsilon(d') \leq \epsilon(p) < \epsilon(d)$, en échangeant les étiquettes de p et d on aura : $\epsilon(p) = j$ et $\epsilon(d) = k$, donc $\epsilon(d') < \epsilon(d) < \epsilon(p)$ par conséquent l'arbre de racine p est une file de priorité.

En sortie de boucle l'étiquette de p est supérieure à l'étiquette de ses fils.

9.6 Programme de représentation des arbres

La structure utilisée est celle de tas, donc de vecteur. Nous écrivons un programme qui affiche un arbre lorsqu'il est représenté par un vecteur. Dans le cas où la structure est celle d'arbre vue plus haut, il faut réécrire l'arbre en tas, pour cela il faut parcourir l'arbre en largeur.

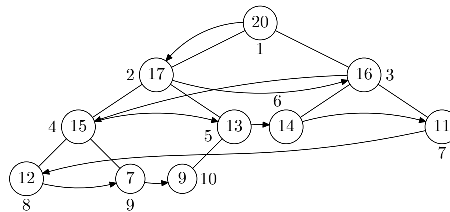


FIG. 12 – Parcours en largeur

9.6.1 Transformations diverses

Transformation d'arbre en tas (complet) :

```
let arbre_vers_tas a t =
  traite_noeud 0 a
  where rec traite_noeud i = function
    |Vide -> ()
    |Noeud(gauche, n, droit) ->
```

```
          begin
            if i >= t.taille_tas then t.taille_tas <- i + 1;
            t.contenu_tas.(i) <- n;
            traite_noeud (2 * i + 1) gauche;
            traite_noeud (2 * i + 2) droit;
          end
;;
```

ou en vecteur (qui représente l'arbre complété) :

```
let arbre_vers_vect a =
let
  rec hauteur_arbre = function
    |Vide -> 0
    |Noeud (gauche, r, droit) ->
       1 + max (hauteur_arbre gauche) (hauteur_arbre droit)
in let                      (* and, ne marche pas *)
  rec expos m n = match n with
    |0 -> 1
    |1 -> m
    |_ -> let y = expos m (n / 2) in
         if n mod 2 = 0 then y * y else m * y * y
in
let v = make_vect (expos 2 (hauteur_arbre a) - 1) 0 in
  traite_noeud 0 a
    where rec traite_noeud i = function
      |Vide -> v                  (* renvoyer un : int vect, ici *)
      |Noeud(gauche, n, droit) ->
         begin
           v.(i) <- n;
           traite_noeud (2 * i + 1) gauche;
           traite_noeud (2 * i + 2) droit;
         end;
;;
```

9.6.2 Programme Caml d'affichage

Notations et relations :
- espace entre deux nœuds à la ligne n : u_n (`espace(n)`);
- marge entre le bord de l'écran et le premier nœud : v_n (`marge(n)`);
- largeur d'un nœud : alpha $= \alpha$.

$u_{n-1} = 2u_n + \alpha$ donc, si on choisit $u_h = \alpha$, $u_n = \alpha 2^{h+1-n} - \alpha$. D'autre part : $v_{n-1} = v_n + \frac{\alpha}{2} + \frac{u_n}{2}$ et par convention $v_h = 0$ d'où $v_0 = \alpha 2^h - \alpha$ ou $v_0 = \frac{u_0}{2} - \frac{\alpha}{2}$.

On remarque que $v_{n-1} - v_n$ est le nombre de lignes entre deux niveaux.

```
<- v_{n-1} ->   845 <---- u_{n-1} ----> 493      (ligne n-1)
                ---                     ---
```

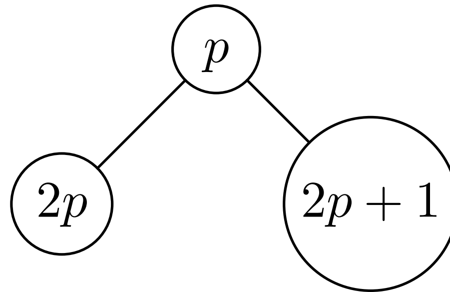
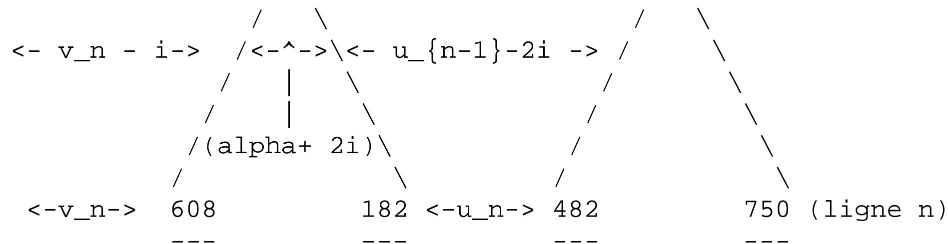


FIG. 13 – mesures dans l'arbre



Le programme écrit en Caml tient compte de la taille des étiquettes. Il prend en argument un vecteur et renvoie le dessin de l'arbre en caractères. Il faut que l'arbre soit complet. `expos m n` est m^n.

```
let dess_arbre vecteur =
let
  hauteur t =
  int_of_float(floor(log(float_of_int(vect_length t))/. log(2.)))
and
  v = map_vect string_of_int vecteur
in
let
  h = hauteur v
in
let espace = ref (2 * (expos 2 h) - 1) (* n = 0 *)
in
let marge = ref ((!espace - 1) / 2) (* n = 0 *)
and
  vs = map_vect string_length v
in
let alpha = ref vs.(0) and gauche = ref 0
                       and droite = ref 0
                       and marge2 =ref 0
                       and espace2 = ref 0
in
```

```
    for i = 1 to vect_length vs - 1 do
      if vs.(i) > !alpha then alpha := vs.(i)
    done;
    for i = 0 to vect_length v - 1 do   (* normalise l'affichage des noeuds *)
      droite := !alpha - vs.(i);
      gauche := 0; (* !alpha - !droite; *)
      v.(i) <- (make_string (!gauche) ' ')^v.(i)^(make_string (!droite) ' ') ;
    done;                                   (* fin normalise *)
    marge := !alpha * !marge;        (* mise à l'echelle *)
    espace := !alpha * !espace;
    print_string ((make_string (!marge) ' ')^v.(0)); (* affichage de la racine *)
    print_newline();    (* souligne *)
    print_string ((make_string (!marge) ' ')^(make_string !alpha '-'));
    print_newline();
    marge2 := !marge; (* marge de la ligne n-1 *)
    espace := (!espace - !alpha) / 2 ;
    marge := !marge - (!espace + !alpha) / 2;
    espace2 := !espace;
    for i = 0 to !marge + 1 do  (* affichage des liens sous la racine *)
      print_string ((make_string (!marge2 - i - 1) ' ')^("/")^(
      make_string (!alpha + 2 * i) ' ')^("\\"));
␣␣␣␣(*␣-1␣dans␣!marge2␣-␣i␣-␣1␣pour␣centrer␣*)
␣␣␣␣print_newline();
␣␣done;␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(*␣fin␣affichage␣des␣liens␣*)
␣␣for␣n␣=␣1␣to␣h␣do␣␣(*␣affichage␣des␣noeud␣a␣la␣ligne␣n␣*)
␣␣␣␣for␣k␣=␣(expos␣2␣n)␣to␣(expos␣2␣(n␣+␣1)␣-␣1)␣do
␣␣␣␣␣␣if␣k␣=␣(expos␣2␣n)
␣␣␣␣␣␣then
␣␣␣␣␣␣␣␣print_string␣((make_string␣(!marge)␣'␣')^v.(k␣-␣1))
␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣print_string␣((make_string␣(!espace)␣'␣')^v.(k␣-␣1));
␣␣␣␣done;␣␣␣␣␣␣␣␣␣␣␣␣␣(*␣fin␣affichage␣des␣noeud␣*)
␣␣␣␣print_newline();
␣␣␣␣for␣k␣=␣(expos␣2␣n)␣to␣(expos␣2␣(n␣+␣1)␣-␣1)␣do␣␣(*␣souligne␣les␣noeuds␣*)
␣␣␣␣␣␣if␣k␣=␣(expos␣2␣n)
␣␣␣␣␣␣then
␣␣␣␣␣␣␣␣print_string␣((make_string␣(!marge)␣'␣')^(make_string␣!alpha␣'-'))
␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣print_string␣((make_string␣(!espace)␣'␣')^(make_string␣!alpha␣'-'));
␣␣␣␣done;␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(*␣fin␣souligne␣*)
␣␣␣␣print_newline();
␣␣␣␣if␣n␣<␣h␣then␣␣(*␣pour␣la␣derniere␣ligne,␣pas␣de␣fils␣*)
␣␣␣␣␣␣begin
␣␣␣␣␣␣␣␣espace2␣:=␣!espace;␣(*␣espace␣de␣la␣ligne␣n␣*)
␣␣␣␣␣␣␣␣marge2␣:=␣!marge;␣␣␣(*␣marge␣de␣la␣ligne␣n␣*)
␣␣␣␣␣␣␣␣espace␣:=␣(!espace␣-␣!alpha)␣/␣2␣;␣(*␣espace␣de␣la␣ligne␣suivante␣*)
```

```
␣␣␣␣␣␣␣␣marge␣:=␣!marge␣−␣(!espace␣+␣!alpha)␣/␣2;
        (* marge de la ligne suivante *)
         for i = 1 to !marge2 − !marge do
         (* dessins des liens entre les noeuds de la ligne n et leurs fils *)
           print_string ((make_string (!marge2 − i) ‘ ‘)^("/")^(
␣␣␣␣␣␣␣␣␣␣make_string␣(!alpha␣+␣2␣*␣i␣−␣2)␣‘␣‘)^("\\"));
␣␣␣␣␣␣␣␣␣␣for␣k␣=␣(expos␣2␣n␣+␣1)␣to␣(expos␣2␣(n␣+␣1)␣−␣1)␣do
␣␣␣␣␣␣␣␣␣␣␣␣print_string␣((make_string␣(!espace2␣−␣2␣*␣i)␣‘␣‘)^("/")^(
␣␣␣␣␣␣␣␣␣␣␣␣make_string␣(␣!alpha␣+␣2␣*␣i␣−␣2)␣‘␣‘)^("\\"));
␣␣␣␣␣␣␣␣␣␣done;
␣␣␣␣␣␣␣␣␣␣print_newline();
␣␣␣␣␣␣␣␣done;␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(*␣fin␣dessins␣*)
␣␣␣␣␣␣end;
␣␣done;;
```

Il faut donc un programme qui complète un arbre représenté en vecteur :

```
let complete_arbre t =
let n = vect_length t in
let h =
  int_of_float(floor(log(float_of_int(n))/. log(2.)))
in
let m = expos 2 (h + 1) − 1
in
let v = make_vect m 0
in
  for i = 0 to m − 1 do
    if i < n then v.(i) <− t.(i)
  done;
  v;;
```

Le vecteur est initialisé avec des 0, on distingue les nœuds à étiquettes nulles des nœuds vides par la taille de l'arbre d'origine.

Pour dessiner un arbre quelconque on doit donc le parcourir en largeur pour le représenter en vecteur. Nous allons d'abord transformer un arbre en liste par un parcours en profondeur à gauche, comme pour la suppression d'une feuille d'un tas (figure 14). Le programme suivant renvoie l'étiquette du terme « le plus à gauche » et l'arbre sans la feuille correspondante :

```
let cherche_feuille_g a =
  let liste = ref 0 in
  let rec supp_feuille = function
    |Vide −> Vide
    |Noeud( Vide, n, Vide) −> liste := n ; Vide
    |Noeud( Vide, n, droit) −> Noeud(Vide, n, supp_feuille droit)
    |Noeud( gauche, n, droit) −> Noeud( supp_feuille gauche, n, droit)
  in (!liste, supp_feuille a);;
```

puis ce programme renvoie la liste de toutes les étiquettes :

```
let rec arbr2list_g = function
|Vide -> []
|a -> let (n, b) = cherche_feuille_g a
    in
      n :: arbr2list_g b;;
```

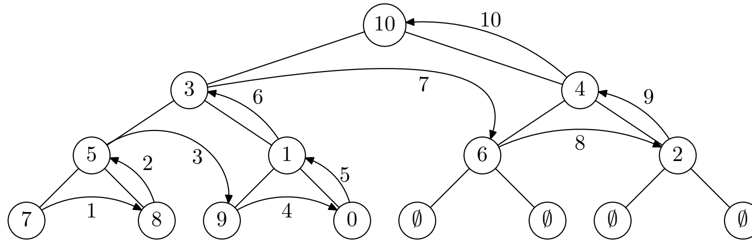


FIG. 14 – Parcours en profondeur à gauche

Nous allons maintenant transformer la liste en vecteur en tenant compte du fait que le parcours de l'arbre a été fait suivant un ordre précis. On remarque en effet que h étant la profondeur de l'arbre a que l'on veut représenter, le premier nœud retiré est celui d'indice 2^h. De manière générale on part du nœud d'indice $2^h + n$ où $0 \leq n \leq 2^h - 1$ et l'on remonte en diagonale, par récurrence, du nœud d'indice $k = 2^i + j$ vers le nœud d'indice $\frac{k-1}{2} = 2^{i-1} + \frac{j-1}{2}$. Si $\frac{j-1}{2}$ est pair le nœud n'est pas un fils droit (figure 15) et on incrémente n, sinon on décrémente i et j de 1 :

pour $n = 0$ à $2^h - 1$ **faire**
  $k \leftarrow 2^h + n$
  tant que k est impair et positif **faire**
    $v_{k-1} \leftarrow$ tête de liste
    liste $\leftarrow$ queue de liste
    $k \leftarrow (k-1)/2$
  fin tant que« maintenant k est pair »
  si k est inférieur à la taille de l'arbre et strictement positif **alors**
    $v_{k-1} \leftarrow$ tête de liste « attention la queue peut être vide »
    si la queue n'est pas vide : liste $\leftarrow$ queue de liste
  fin si
  $k \leftarrow (k-1)/2$
fin pour

Le programme Caml est

```
let liste2vecteur l =
let liste = ref l
and
  m = list_length l
and
  k = ref 0
in
```

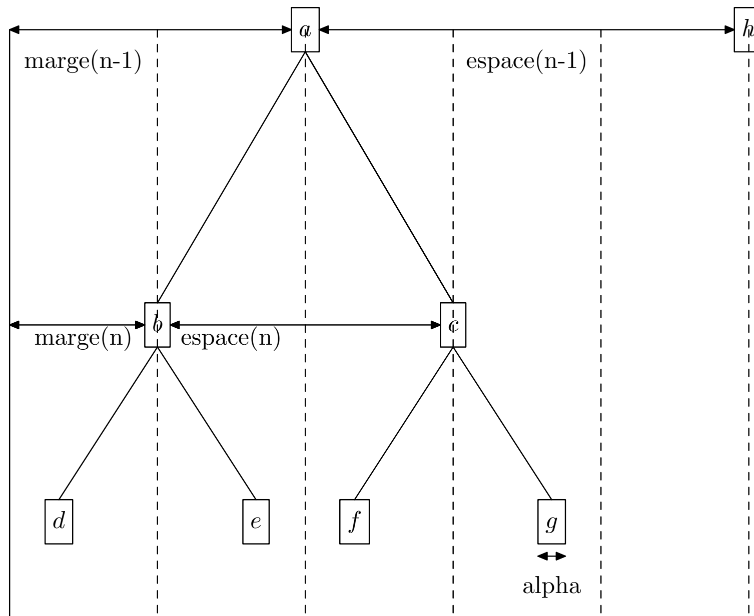
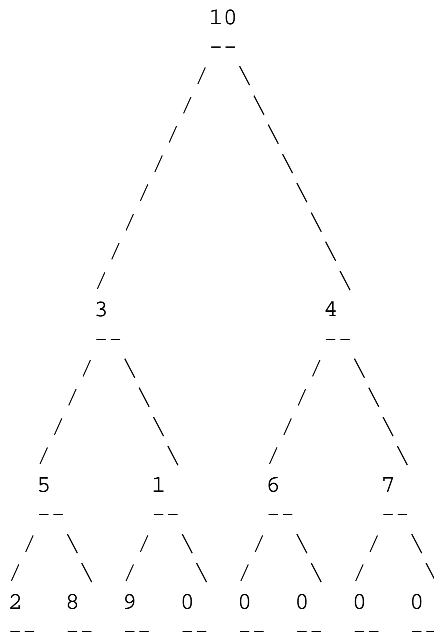


FIG. 15 – Un nœud pair est un fils gauche

```
let
  h =  int_of_float(floor(log(float_of_int m)/. log(2.)))
in
let b = (expos 2 h)
and
  v = make_vect m 0
in
  for n = 0 to b - 1 do
    k := b + n;
    while !k mod 2 = 1 && !k > 0 do
      if !k<= m then
        begin
          v.(!k - 1) <- hd !liste;
          liste := tl !liste;
        end;
        k := (!k - 1) / 2;
    done;
    if !k<= m && !k > 0 then
      begin
        v.(!k - 1) <- hd !liste;
        if  list_length (!liste) > 1 then liste := tl !liste;
      end;
    k := (!k - 1) / 2;
```

```
  done;
v;;

#let arbuste =
  Noeud(
    Noeud(
      Noeud(
        Noeud(Vide,2,Vide),5,
        Noeud(Vide,8,Vide)),3,
      Noeud(
        Noeud(Vide,9,Vide),1,
        Noeud(Vide,0,Vide))),10,
    Noeud(
      Noeud(Vide,6,Vide),4,
      Noeud(Vide,7,Vide))
  );;
#arbr2list_g arbuste;;
- : int list = [2; 8; 5; 9; 0; 1; 3; 6; 7; 4; 10]
#liste2vecteur (arbr2list_g arbuste);;
- : int vect = [|10; 3; 4; 5; 1; 6; 7; 2; 8; 9; 0|]
#let vv = liste2vecteur (arbr2list_g arbuste);;
vv : int vect = [|10; 3; 4; 5; 1; 6; 7; 2; 8; 9; 0|]
#let vvv = complete_arbre vv;;
vvv : int vect = [|10; 3; 4; 5; 1; 6; 7; 2; 8; 9; 0; 0; 0; 0; 0|]
#dess_arbre vvv;;
```



Les quatre derniers 0 représentent $\emptyset$.
Exemple qui utilise un autre programme :

```
#dess_arbre(arbre_vers_vect ( Noeud(
    Noeud(
      Noeud(
        Noeud(Vide,2,Vide),5,
        Noeud(Vide,8,Vide)),3,
      Noeud(
        Noeud(Vide,9,Vide),1,
        Noeud(Vide,0,Vide))),10,
    Noeud(
      Noeud(Vide,6,Vide),4,
      Noeud(Vide,7,Vide))
  )));;
```

10 Étude de la complexité de quelques tris

Les tris que l'on étudie ici sont des tris *internes*, c'est-à-dire que les données sont suffisamment petites pour être enregistrées en mémoire et être triées sur place. On doit donc prendre en compte la place mémoire utilisée. Ces tris utilisent principalement deux opérations : la *comparaison* entre deux éléments et l'*échange* ou le *transfert* d'éléments (par affectation).

Pour la plupart de ces tris il s'agit de tris vu en première année.

10.1 Méthodes par sélection

10.1.1 Le tri sélection

Principe : On recherche le plus petit élément de la liste à trier, on le met en première position puis on recommence cette opération sur la fin de liste.

On suppose que l'on trie un vecteur, donc que l'on peut accéder directement à n'importe quelle coordonnée (on emploiera le mot « liste » dans son sens usuel).

Soit u_n le nombre de comparaisons pour trier un vecteur à n coordonnées. On doit effectuer $n-1$ comparaisons, ensuite on doit trier un vecteur à $n-1$ coordonnées. On a donc la relation de récurrence :

$$u_1 = 0, \qquad u_n = u_{n-1} + n - 1$$

Or

$$u_n = u_1 + \sum_{2 \le k \le n} u_k - u_{k-1} = \sum_{2 \le k \le n} k - 1$$

d'où $u_n = \dfrac{n(n-1)}{2}$. La complexité est en $O(n^2)$.

Si on utilise un second vecteur pour la liste triée, on fait n transferts mais on utilise n places supplémentaires en mémoire.

10.1.2 Le tri à bulles

En commençant par la fin, on échange deux éléments consécutifs qui ne sont pas dans le bon ordre. À la fin du premier passage, le plus petit élément est à sa place définitive, en tête. On a donc fait $n-1$ comparaisons et, dans le pire des cas $n-1$ échanges. La complexité, comme pour le tri sélection est en $O(n^2)$.

Le tri bulle se fait sans appel à un vecteur auxilliaire.

10.2 Méthodes par insertion

10.2.1 Le tri insertion

On insére le i^{e}élément parmi les $i-1$ éléments précédemment triés.

On fait $i-1$ comparaisons, i variant de 1 à n et un échange au plus à chaque fois donc n échanges en tout (en comptant le déplacement du premier élément). La complexité est encore en $O(n^2)$. On utilise un vecteur auxilliaire.

10.2.2 L'insertion dichotomique

Il s'agit d'une variante du tri insertion : on recherche le rang de l'élément à insérer par recherche dichotomique, *i.e.* on compare l'élément à insérer avec l'élément qui est au milieu de la liste déjà triée.

Si m éléments sont triés, pour insérer le $m + 1^{\text{e}}$, on effectuera au plus k comparaisons où :

$$2^{k-1} < m \leq 2^k$$

Donc

$$\frac{\ln m}{\ln 2} \leq k < \log_2 m + 1$$

soit $k = \lfloor \log_2 m \rfloor$. Le nombre de comparaisons est majoré par

$$n - 1 + \frac{1}{\ln 2} \sum_{0 \leq m \leq n-1} \ln m$$

Par comparaison avec une intégrale, on trouve que la complexité est en $O(n \ln n)$.

10.3 Le tri rapide

On choisit un élément dans la liste, le pivot, on le met à sa place définitive, puis on trie de la même façon les éléments plus petits et les éléments plus grands. Pour placer le pivot, on balaye à gauche jusqu'à trouver un élément plus grand que le pivot, on balaye à droite jusqu'à trouver un élément plus petit que le pivot, puis on échange ces deux éléments (pivot compris).

Échanges : au pire, il y a $\lceil \frac{n}{2} \rceil$ échanges dans le cas où le pivot est au milieu, est le plus petit élément, les éléments plus grands sont à gauche et les plus petits à droite. On peut alors établir une formule de récurrence. Prendre un exemple de quinze nombres disposés de telle sorte que le nombre d'échanges soit maximal :

$$11, 10, 9, 12, 15, 14, 13, 8, 3, 2, 1, 4, 7, 5, 6$$

(sauf erreur).

Comparaisons : au pire, le pivot est en première position et est le plus petit élément (aucun échange mais un nombre maximal de comparaisons, la liste est triée !). La complexité est en $O(n^2)$.

Remarque 10.1 *En moyenne, on peut montrer que la complexité est en $O(n \ln n)$. Enfin, on améliore la complexité du tri rapide en choisissant le pivot ailleurs qu'en première position.*

10.4 Le tri fusion

Principe : on partage la liste en deux sous listes que l'on trie puis on les fusionne. Pour trier les deux sous listes, on applique la même procédure.

Ayant à disposition deux fonctions `partition` et `fusion`, le programme peut s'écrire (ici `l` est de type list) sous la forme :

```
let tri_fusion l = match l with
  |[] -> []
  |[a] -> [a]
  |_ ->
   let (l1, l2) = (partition l) in
      fusion( tri_fusion l1, tri_fusion l2);;
```

Soit u_n le nombre de comparaisons, dans le pire des cas, pour fusionner deux listes triées de $\frac{n}{2}$ éléments en une liste de n. Alors :

$$u_n = n - 1 + u_{\lfloor \frac{n}{2} \rfloor} + u_{\lceil \frac{n}{2} \rceil}$$

Pour simplifier, on commence par supposer que $n = 2^m$. La formule devient

$$u_n = n - 1 + 2u_{\frac{n}{2}}$$

On résout d'abord l'équation

$$v_n = 2v_{\frac{n}{2}}$$

Qui donne $v_{2^m} = 2^m v_1$. On pose alors $u_{2^m} = a_m 2^m v_1$. La suite a vérifie

$$a_m = a_{m-1} + \frac{1}{v_1}\left(1 - \frac{1}{2^m}\right)$$

D'où, après sommation :

$$a_m = \frac{m}{v_1} - \frac{1}{v_1}\left(1 - \frac{1}{2^m}\right)$$

il vient :

$$u_{2^m} = \frac{m2^m}{v_1} - \frac{2^m}{v_1} + \frac{1}{v_1}$$

On vérifie que $u_2 = 1$, donc $v_1 = 1$, finalement :

$$u_{2^m} = m2^m - 2^m + 1$$

On démontre ensuite le cas général par récurrence :

$$u_n = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$$

10.5 Tri en tas

Le principe est d'insérer les éléments à trier dans un tas puis de retirer la racine jusqu'à ce que le tas soit vide.

10.6 Arbres de décision

Ordonner un tableau de taille n c'est trouver une permutation parmi $n!$. Pour chaque comparaison et transfert de deux éléments on effectue une permutation. Chaque décision est associée à un nœud d'un arbre binaire (on échange ou on n'échange pas ces éléments). Cet arbre binaire a $n!$ feuilles, autant que de permutations, sa profondeur est de l'ordre de $n \log_2 n$ d'après la formule de Stirling. On en déduit que la complexité optimale d'une procédure de tri est du même ordre.

11 Langages et automates

11.1 Introduction

La théorie des langages étudie ce que l'on appelle les *langages formels* par opposition aux langages naturels comme le français, le tamoul le louchebem. Pour définir un langage il faut un ensemble de caractères (ou symboles) $\mathcal{A}$, appelé *alphabet*, l'ensemble des mots sur $\mathcal{A}$ qui est l'ensemble $\mathcal{A}^*$ des suites finies d'éléments de $\mathcal{A}$ et qui contient le mot vide noté ε. Un langage (formel) d'alphabet $\mathcal{A}$ est une partie $\mathcal{L}$ de $\mathcal{A}^*$. La syntaxe de $\mathcal{L}$ est l'ensemble des règles de construction des mots de $\mathcal{L}$. La sémantique de $\mathcal{L}$ est la signification des mots.

La définition donnée est mathématique, les mots utilisés pour les définitions sont imagés car la théorie a été construite pour répondre à des problèmes pratiques.

On se pose deux questions :

1. comment construire les mots du langage ?
2. comment vérifier qu'un mot donné appartient au langage ?

On peut y répondre de manière automatique : on peut construire un algorithme, on dira, ici, une *machine* appelée *automate* dont les plus simples sont les automates finis que nous étudions.

Les applications sont variées et font appels à des problèmes de *décision* (réponse boléenne) ou qualitatifs (réponse numérique).

- reconnaître le lexique d'un langage de programmation : les mots-clés, vérifier la syntaxe etc ...
- reconnaissance d'un motif dans une image (empreinte digitale, iris de l'œil, pour les identifications), d'un mot pour les traitements de textes ou les bases de données (étude de l'ADN etc...).

Exemple 11.1 Soit $\mathcal{A}$ l'alphabet des caractères accessibles par un clavier d'ordinateur. On peut considérer les langages suivants :

- représentation décimale d'entiers (+,-, et les dix chiffres).
- représentation des nombres flottants (+,-,E, et les dix chiffres).
- les mots qui ne comportent que des lettres et des chiffres ne commençant que par une lettre.
- les mots qui contiennent les lettres c,o,n,f,i,g dans cet ordre.

Exemple 11.2 Le langage d'alphabet $\mathcal{A} = \{0, 1\}$ dont les mots sont la représentation binaire d'un nombre divisible par 3 (ce langage est noté $\{0, 1\}^\star$).

Notation 11.1 *Le langage consitué de tous les mots de l'alphabet $\mathcal{A}$ est noté $\mathcal{A}^\star$.*

11.2 Automates finis

11.2.1 Définition

Un automate fini est le modèle mathématique d'une machine qui peut se trouver dans un nombre fini d'*états* (de configurations internes), qui reçoit des signaux externes

qui provoquent un changement d'état, appelé *transition*. On pourra envisager qu'un automate émette un message lors d'une transition.

Un automate est un graphe orienté et étiqueté, plus précisément :

Définition 11.1 *Un automate M est la donnée d'un quintuplet (A, Q, δ, q_0, F) constitué :*

1. *d'un alphabet fini A ;*
2. *d'un ensemble non vide d'états Q ;*
3. *d'une fonction de transition $\delta : Q \times A \to Q$;*
4. *d'un état initial noté q_0 ;*
5. *d'un sous ensemble F de l'ensemble des états, appelé ensembles des états finaux ou états* acceptants.

Un automate M étant fixé, on note souvent $p \xrightarrow{a} q$ à la place de $\delta(p, a) = q$.

On dit que l'automate est *complet* si pour tout état p et tout a dans A, il existe $\delta(p, a)$.

Notons $|u|$ la *longueur* d'un mot, c'est-à-dire le nombre de symboles qu'il contient. On définit la concaténation de deux mots u et v comme étant le mot uv obtenu en disposant la suite des symboles de v après la suite des symboles de u. On définit une fonction $\overline{\delta}$ par récurrence sur la longueur des mots par :

1. $\overline{\delta}(q, \varepsilon) = q$;
2. pour w dans $\mathcal{A}^*$ et a dans $\mathcal{A}$: $\overline{\delta}(q, wa) = \delta(\overline{\delta}(q, w), a)$.

On écrit : $p \xrightarrow{w} q$ pour $\overline{\delta}(p, w) = q$ et $p \xrightarrow{w} F$ s'il existe q dans F tel que $p \xrightarrow{w} q$.
On définit alors $L(M)$ le langage *reconnu* par M par :

$$L(M) = \{w \in \mathcal{A}^* / q_0 \xrightarrow{w} F\}$$

Si w appartient à $L(M)$, on dit que M *accepte* w.

11.2.2 Interprétation

On considère le traitement d'un mot par un automate comme le déchiffrement des informations sur une bande magnétique par une tête de lecture. La figure (16) représente la tête de lecture qui se déplace de gauche à droite, à partir de son état initial et qui change d'état suivant les signaux reçus.

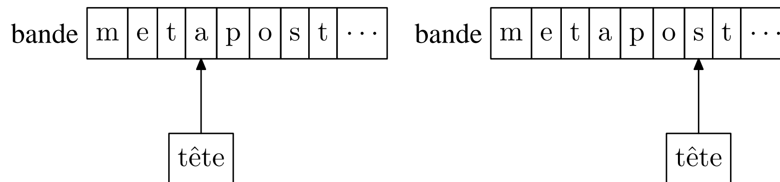


FIG. 16 – Représentation d'un automate

11.2.3 Diagramme d'un automate

On considère l'automate M d'alphabet $A = \{a, b\}$ qui admet cinq états $Q = \{q_0, \ldots, q_4\}$ et dont la fonction de transition est donnée par le tableau suivant :

	a	b
q_0	q_1	q_4
q_1	q_4	q_2
q_2	q_3	q_4
q_3	q_4	q_2
q_4	q_4	q_4

Pour suivre plus aisément le fonctionnement de l'automate, on le représente par un diagramme. Les états sont symbolisés par des disques, les transitions par des flèches étiquetées par des lettres de l'alphabet, voir la figure (17).

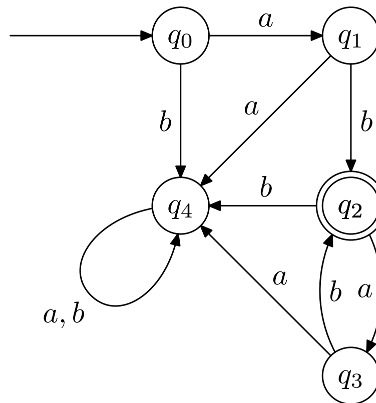


FIG. 17 – Diagramme d'un automate

L'état initial est marqué par une flèche qui arrive de nulle part et les états finaux sont doublements cerclés. Parfois on n'écrira pas les noms des états. L'état noté q_4 n'est pas final pourtant les flèches y aboutissent et aucune n'en part, c'est un état *rebut*, on ne le fait pas figurer sur le diagramme en général, on obtient alors la figure (18). Par convention, toutes les transitions qui ne figurent pas, aboutissent à un état rebut.

À chaque mot est associé un chemin sur le diagramme, un mot accepté est un mot qui aboutit à un état final et inversement, les chemins de l'état initial à un état final définissent des mots reconnus.

L'automate décrit par la figure (18) reconnait les mots : ab, $abab$, *etc...* donc tous les mots de la forme $(ab)^n$.

11.2.4 Implémentation d'un automate en Caml

Les états sont étiquetés par des entiers, les mots sont, ici, des suites de caractères.

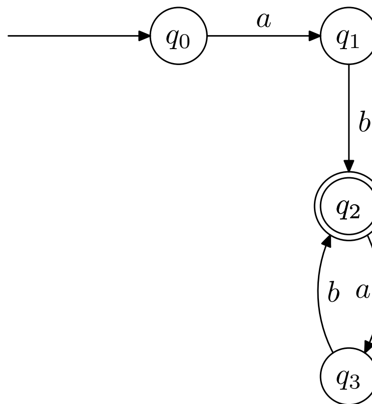


FIG. 18 – Diagramme d'un automate

```
type etat == int;;
type caractere == char;;
```

Dans l'exemple suivant, la fonction de transition est représentée par une liste de type `(etat * caractere) * etat` qui contient les éléments de la forme $((p, a), \delta(p, a))$ (graphe de δ). On définit conventionnellement l'état rebut égal à -1.

```
let delta liste_valeurs = function
  |(-1, _) -> -1
  |couple -> try assoc couple liste_valeurs
    with Not_found -> -1;;
```

Extension de la fonction δ aux mots.

```
let extension_mot delta =
let rec aux = function
    |(p,[]) -> p
    |(p, h :: r) -> aux (delta (p, h), r)
    in aux;;
```

11.3 Automates non déterministes

11.3.1 Généralités

Définition 11.2 *Un automate fini* non déterministe M *(en abrégé AFND) est la donnée :*

1. *d'un alphabet $\mathcal{A}$;*
2. *d'un ensemble non vide d'états E ;*
3. *d'une fonction de transition δ :* $E \times \mathcal{A} \to \mathcal{P}(E)^\star$ *;*
   ($\mathcal{P}(E)^\star$ désigne l'ensemble des parties non vides de E).
4. *d'un état initial p_0 (ou un ensemble d'états initiaux) ;*

5. *d'un ensemble d'états finaux F.*

$\delta(p, a)$ est l'ensemble des états auquels on peut aboutir en partant de p et en lisant a.

On écrit $p \xrightarrow{a} q$ si $q \in \delta(p, a)$. Comme pour les automates déterministes, la fonction de transition se prolonge en une fonction $\overline{\delta}$ de $E \times \mathcal{A}^\star$ dans $\mathcal{P}(E)^\star$. Ainsi, le mot w est accepté par l'automate s'il existe un état final dans $\overline{\delta}(p_0, w)$ ou, en plus imagé, si, lisant w, il existe un chemin dans le graphe qui aboutit à un état final.

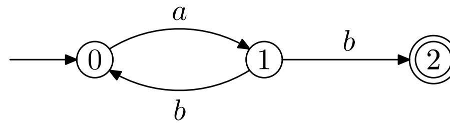


FIG. 19 – Automate non déterministe

La figure (19) représente le diagramme d'un automate non déterministe.

Théorème 11.1 *Tout langage reconnu par un automate non déterministe si et seulement s'il est reconnu par un automate déterministe.*

Preuve : Étant donné un automate non déterministe M, nous allons construire un automate déterministe $M^\star$ qui accepte les mêmes mots.

1. l'alphabet est $\mathcal{A}$;
2. l'ensemble d'états est $\mathcal{E}^\star$;
3. la fonction de transition est

$$\Delta : \qquad \mathcal{P}(E)^\star \times \mathcal{A} \to \mathcal{P}(E)^\star$$

   où

$$\Delta(Q, a) = \bigcup_{p \in Q} \delta(p, a)$$

   $\Delta(Q, a)$ représente l'ensemble des états auxquels on aboutit en partant d'un état dans Q et en lisant a.
   Le prolongement de Δ aux mots est noté $\overline{\Delta}$.
4. l'état initial est $\{p_0\}$;
5. d'un ensemble d'états finaux :

$$\mathcal{F} = \{A \in \mathcal{P}(E)^\star / A \cap F \neq \emptyset\}$$

   Autrement dit, le mot w est accepté si dans $\overline{\Delta}(\{p_0\}, w)$ il y a un état final (*i.e.* parmi tous les chemins étiquetés par w, partant de p_0 et arrivant dans Q, il y en a un qui aboutit sur un état final).

– Soit m un mot reconnu par M, on a : $p_0 \xrightarrow{m} q$ où $q \in \overline{\delta}(p_0, m)$ et $\overline{\delta}(p_0, m) \cap F \neq \emptyset$. Donc $\overline{\delta}(p_0, m)$ est un état final de $M^\star$:

$$\overline{\Delta}(\{p_0\}, m) = \overline{\delta}(p_0, m)$$

Ainsi m est accepté par $M^\star$.

– Inversement, soit m un mot accepté par $M^\star$, il existe un état final Q de $M^\star$ tel que $Q = \overline{\Delta}(\{p_0\}, m)$ or $\overline{\Delta}(\{p_0\}, m) = \overline{\delta}(p_0, m)$, donc comme l'intersection de Q avec F est non vide, l'intersection de $\overline{\delta}(p_0, m)$ avec F est aussi non vide (c'est la même). Ceci signifie que M accepte m.

Donc M et $M^\star$ acceptent les mêmes mots, par conséquent ils reconnaissent le même langage.

□

Remarque 11.1 *La construction de $M^\star$ fait passer d'un ensemble d'état à n éléments à un ensemble d'états à $2^n - 1$ éléments. La croissance est donc exopnentielle. Mais en pratique, on n'a pas besoin de toutes les parties de E.*

11.3.2 Exemple de déterminisation

Nous allons construire un automate déterministe $M^\star$ (voir la figure (20)) qui reconnait le même langage que l'automate non déterministe M représenté sur la figure (19).

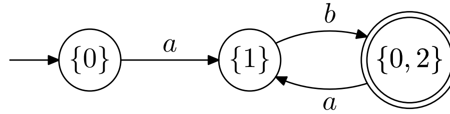


FIG. 20 – Automate déterministe $M^\star$

On commence par écrire la matrice de transition en notant d'abord les états singletons. Par convention le rebut est noté -1.

	a	b
0	1	-1
1	-1	$\{0, 2\}$
2	-1	-1

On constate que les états qui interviennent sont :

$$\{0\}, \qquad \{1\}, \qquad \{0, 2\}$$

L'état final est la partie qui contient l'état final de l'automate non déterministe.

Nous pouvons écrire la matrice de transition de l'automate déterministe obtenu (l'image d'une partie est l'ensemble des images de ses éléments) :

	a	b
$\{0\}$	$\{1\}$	-1
$\{1\}$	-1	$\{0, 2\}$
$\{0, 2\}$	$\{1\}$	-1

11.3.3 Simulation d'automates non déterministes

La bibliothèque caml contient les fonctions `mem`, `exists` et `union` qui agissent sur les listes et permettent de simuler les ensembles.

Supposons que les transitions soient données comme partie de $(E \times \mathcal{A}) \times \mathcal{P}(E)^\star$, l'application de transition δ est définie par :

```
let delta (p, a) =
  aux [] (p,a) graphe_delta
    where rec aux l d graphe = match graphe with
      |[] -> l
      |(e, q) :: reste when e = d -> aux (union [q] l) d reste
      |_ :: reste -> aux l d reste;;
```

L'ensemble des états de l'automate déterminisé étant connu, la transition Δ est donnée par :

```
let rec Delta (e, a) = match e with
  |[] -> []
  |t :: r -> union (delta(t, a)) (Delta(r, a));;
```

11.3.4 Transitions instantanées

Une transition instantanée est une transition étiquetée par le mot vide ε, une telle transition permet de passer d'un état à un autre sans lecture d'une lettre.

Définition 11.3 *Un automate non déterministe à transitions instantanées (en abrégé ANDε) est un automate non déterministe muni d'une application τ :*

$$\begin{array}{cccc} \tau : & E & \rightarrow & \mathcal{P}(E) \\ \tau : & q & \mapsto & \tau(q) \end{array}$$

Intuitivement si M est un automate ANDε dans un état $q_1 \in A \subset E$, alors M peut se retrouver dans n'importe quel état q_2 issu de q_1 par une, ou plusieurs, transitions instantanées, l'ensemble de ces états est appelé *clôture instantanée* de A, on la détermine par récurrence, posons :

$$\tau^\star(A) = A \cup \left(\bigcup_{q \in A} \tau(q) \right)$$

puis on définit une suite (A_n) de parties de E par : $A_0 = A$ et pour $k \geq 1$, $A_k = \tau^\star(A_{k-1})$. (A_n) est une suite croissante majorée par E, ensemble fini, donc elle est stationnaire, cette limite est la clôture instantanée de A, on la note $c\ell(A)$.
On note $\mathcal{C}(E)$ l'ensemble des clôtures instantanées des éléments de $\mathcal{P}(\mathcal{E})^\star$.
Les autres définitions vues pour les AND ont des versions semblables pour les ANDε et on peut démontrer que tout langage reconnu par un ANDε est reconnu par un automate déterministe :

L'ensemble des états de l'automate déterministe est $\mathcal{C}(E)$ et la fonction de transition Δ est donnée par :

$$\Delta(Q, a) = c\ell\left(\bigcup_{q \in Q} \delta(q, a)\right)$$

On dit que M (ANDε) accepte la lettre a si $c\ell(\delta(p_0, a))$ contient un état final (ou $\Delta(c\ell(\{p_0\}), a)$).

Un exemple d'AFNDε est donné par le diagramme (21) et le déterminisé de cet automate est donné par le diagramme (22).

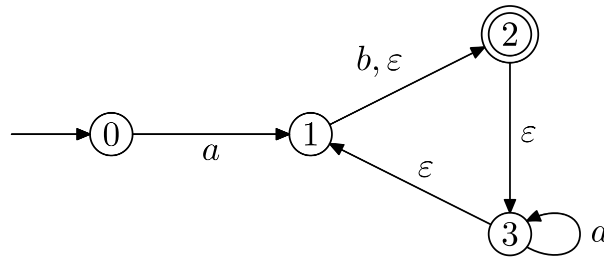


FIG. 21 – ANFDε.

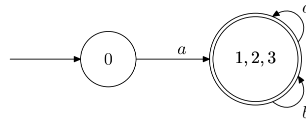


FIG. 22 – Automate déterminisé.

Remarque 11.2 *Soit $A \in \mathcal{P}(E)^\star$, $c\ell(A)$ est la réunion des orbites des éléments de A par l'action de ε.*

11.4 Automates et langages

11.4.1 Opérations sur les langages

Soient L_1 et L_2 deux langages sur un alphabet $\mathcal{A}$. On définit :

- la réunion $L_1|L_2$ par $L_1 \cup L_2$;
- le complémentaire $\neg L_1$ par : $m \in \neg L_1 \quad \Leftrightarrow m \notin L_1$;
- la concaténation $L_1.L_2$ (ou sans le point L_1L_2) par $\{m_1m_2/(m_1, m_2) \in L_1 \times L_2\}$;
- la clôture par : $L^\star = \bigcup_{i=0}^{\infty} L^i$ où $L^0 = \{\varepsilon\}$ et pour $n \geq 1$ $L^n = LL^{n-1}$. La clôture positive est $L^+ = LL^\star$.

11.4.2 Expressions rationnelles

Définition 11.4 *Une* expression rationnelle *(ou* expression régulière*) est une formule constituée de lettres de l'alphabet $\mathcal{A}$ et des symboles qui n'appartiennent pas à l'alphabet $\emptyset, \varepsilon, +, ., \star, |, (,), \ldots\}$ (… signifiant* etc*) et définie par récurrence :*

1. *$\emptyset$ est une expression rationnelle qui désigne le langage vide ;*
2. *ε est une expression rationnelle qui désigne le langage $\{\varepsilon\}$;*
3. *a, où $a \in \mathcal{A}$ est une expression rationnelle qui désigne le langage $\{a\}$;*
4. *r et s désignant les langages L_r et L_s, alors*
   (a) *$r + s$ désigne le langage $L_1|L_2$;*
   (b) *rs désigne le langage $L_1.L_2$;*
   (c) *$r^\star$ désigne le langage $L_1^\star$.*

Les régles d'écriture et de priorités sont les même qu'en algèbre, cependant l'étoile a la priorité la plus élevée.

On trouve diverses conventions d'écriture, par exemple : $r|s$ a la place de $r + s$.

Exemple 11.3 $0(1 + 0)^\star$ représente le langage dont tous les mots commencent par 0 et comportent ensuite un nombre quelconque de 0 ou de 1.

11.4.3 Langages rationnels

Ce paragraphe traite le théorème de Kleene)

Définition 11.5 *Un* langage rationnel *est un langage défini par une expression rationnelle.*

Théorème 11.2 *Tout langage rationnel est reconnu par un automate fini.*

Preuve : On démontre le théorème par récurrence sur le nombre n d'opérateurs. On note L un langage rationnel quelconque.

Si $n = 0$, alors L est $\emptyset$ ou $\{\varepsilon\}$ ou $\{a\}$ ($a \in \mathcal{A}$. L est reconnu respectivement par les automates des figures (23), (24) et (25).

FIG. 23 – Reconnait le langage vide.

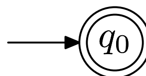


FIG. 24 – Reconnait $\{\varepsilon\}$.

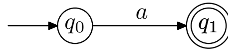


FIG. 25 – Reconnait $\{a\}$.

On suppose maintenant que $n > 0$, il suffit de prouver que si les expressions rationnelle r et s définissent des langages L_r et L_s reconnus par des automates finis M_r et M_s, alors les expressions régulières $r + s$, rs et $r^\star$ définissent des langages reconnus par des automates finis.

Remarquons tout d'abord, que l'on peut supposer

– que les automates reconnaissants sont des automates finis non déterministes à transitions instantanées.
– que les automates n'ont qu'un état initial : il suffit de définir un nouvel automate en ajoutant un état q_0 et des transitions instantanées vers les états initiaux de l'automate d'origine.
– qu'il n'y a qu'un état final : en ajoutant un état q_f et des transitions instantanées des états finaux vers q_f.

Un tel automate est appelé automate de Thompson.

On remarque que le nouvel automate reconnait le même langage.

Nous allons utiliser la représentation simplifiée de la figure (26) d'un automate fini n'ayant qu'un état initial et un état final. On ne représente que l'état initial et l'état final.

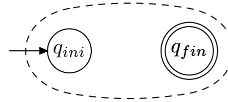


FIG. 26 – Automate de Thompson.

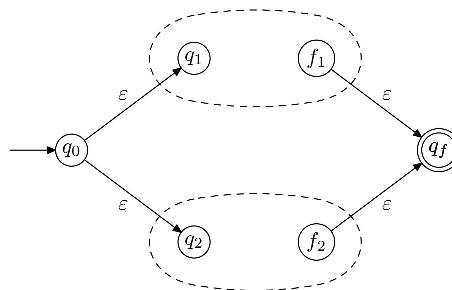


FIG. 27 – Automate reconnaissant $L_1|L_2$.

Ainsi, la figure (27) représente l'automate reconnaissant $L_1|L_2$, la figure (28) représente l'automate reconnaissant $L_1.L_2$ et la figure (29) représente l'automate reconnaissant $L_1^\star$.

□

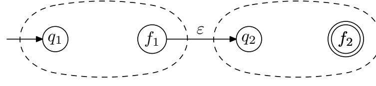


FIG. 28 – Automate reconnaissant $L_1.L_2$.

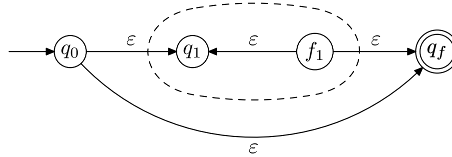


FIG. 29 – Automate reconnaissant $L_1^\star$.

Il y a une réciproque :

Théorème 11.3 *Un langage reconnu par un automate fini est rationnel.*

Preuve : Soit L un langage reconnu par l'automate M. Il suffit de supposer M déterministe.

Hypothèse : $M = (\mathcal{A}, \{q_1, \ldots, q_n\}, q_1, F, \delta)$ reconnait L. On note $L_{i,j}^k$ le langage constitué des mots m tels que :

$$\overline{\delta}(q_i, m) = q_j$$

et tels que si $m = um'$ alors $\overline{\delta}(q_i, u) = q_\ell$ et $\ell \leq k$.
Autrement dit, c'est l'ensemble des mots qui font passer M de l'état q_i à l'état q_j en passant par des états qui sont dans $\{q_1, \ldots, q_k\}$.

Par exemple $L_{i,j}^n$ est simplement l'ensemble des mots qui font passer M de q_i à q_j. D'autre part la formule se simplifie si $i = k$ ou $j = k$.

On a :

$$L_{i,j}^k = L_{i,k}^{k-1} \left(L_{k,k}^{k-1}\right)^\star L_{k,j}^{k-1} \cup L_{i,j}^{k-1}$$

Cette formule s'interprète de la façon suivante : M passe de l'état q_i à l'état q_k sans être passé par q_k auparavant, ensuite il peut y avoir des cycles ; M repassant par l'état q_k après avoir été dans des états q_ℓ où $\ell < k$. Enfin M quitte l'état q_k pour ne plus y revenir et passe par des états q_ℓ où $\ell < k$ avant d'arriver dans l'état q_j. Sinon M passe de l'état q_i à l'état q_j en transitant par des états q_ℓ où $\ell < k$.

Notons $r_{i,j}^k$ l'expression rationnelle qui désigne $L_{i,j}^k$, la formule des langages se traduit en une formule d'expressions rationnelles :

$$r_{i,j}^k = r_{i,k}^{k-1} \left(r_{k,k}^{k-1}\right)^\star r_{k,j}^{k-1} + r_{i,j}^{k-1}$$

Initialisation de la suite $L_{i,j}^k$:

$$L_{i,j}^0 = \begin{cases} \{a \in \mathcal{A} / \delta(q_i, a) = q_j\} & \text{si} \quad i \neq j \\ \{a \in \mathcal{A} / \delta(q_i, a) = q_i\} \cup \{\varepsilon\} & \text{si} \quad i = j \end{cases}$$

Ainsi

$$L = \bigcup_{q_j \in F} L^n_{1,j}$$

Si $F = \{q_{j_1}, \ldots, q_{j_h}\}$ alors L est défini par l'expression rationnelle :

$$r^n_{1,j_1} + \ldots + r^n_{1,j_h}$$

$\square$

Propriétés de fermeture :

On dit qu'un ensemble est *fermé* par une opération ou une application s'il contient les images de ses éléments.

Ainsi, nous avons vu que :

Théorème 11.4 *Les langages rationnels sont fermés par réunion, concaténation et clôture.*

De plus :

Théorème 11.5 *Si L est rationnel, alors $\neg L$ est rationnel.*

Enfin, on appelle homomorphisme de langage, toute application f définie sur un langage L_1 et à valeurs dans un langage L_2 telle que pour tous mots m et m' de L_1 :

$$f(mm') = f(m)f(m')$$

On peut démontrer que l'image et l'image réciproque d'un langage rationnel est un langage rationnel.

11.4.4 Lemme de l'étoile

Le résultat suivant (*pumping lemma*), qui donne une propriété des mots des langages rationnels fournit un critère pour montrer qu'un langage n'est pas rationnel.

Théorème 11.6 (lemme de l'étoile) *Soit L un langage rationnel.*
Alors il existe un entier positif n tel que pour tout mot m de L :
si $|m| > n$ alors il existe des mots u, x et v de L tels que $m = uxv$ et

1. $|ux| \leq n$
2. $|x| \geq 1$
3. $\forall i \geq 0 : \quad ux^iv \in L$.

Preuve : Soit M un automate déterministe à n états reconnaissant L.

Soit $m = a_0 \ldots a_\ell$ un mot de L de longueur $\ell + 1 > n$. On définit une suite finie de mots (m_i) par :

$$m_0 = \varepsilon, \quad \text{et si} \quad i \geq 1 : \quad m_i = a_0 \ldots a_i$$

On note q_i l'état tel que $q_i \stackrel{a_i}{\rightarrow} q_{i+1}$ $(0 \leq i \leq \ell)$, q_0 étant l'état initial et $q_{\ell+1}$ un état final .

Par hypothèse : $\ell \geq n$, donc l'application $i \mapsto q_i$ de $\{0, \ldots, n\}$ (qui a $n+1$ éléments) dans l'ensemble des états E de M n'est pas injective. Par conséquent, il existe deux indices i et j distincts et inférieurs à n tels que $q_i = q_j$ et $i < j$ (si $j = n$ on a visité $n+1$ états, donc au moins un deux fois). Posons $u = m_{i-1}$ et $x = a_i \ldots a_{j-1}$ de sorte que $q_0 \stackrel{u}{\rightarrow} q_i$ et $q_i \stackrel{x}{\rightarrow} q_j$. Enfin, posant $v = a_j \ldots a_\ell$, on a : $q_j \stackrel{v}{\rightarrow} q_{\ell+1}$.
Comme $j \leq n$ et $i \neq j$, on a : $|ux| \leq n$ et $|x| \geq 1$

Puisque $q_i = q_j$, on constate que pour tout entier k, le mot ux^kv fait passer l'automate de l'état initial q_0 à l'état final $q_{\ell+1}$. Donc $ux^kv \in L$.
□

Remarquons que n est inférieur au nombre d'états de tout automate reconnaissant L.

11.4.5 Limitation des automates finis

Considérons l'alphabet $\mathcal{A} = \{a, b\}$ et le langage $L = \{a^nb^n / n \in \mathbb{N}\}$. Supposons qu'il existe un automate fini reconnaissant L. Soient q_0 l'état initial et q_f l'état final de M.

Il existe donc deux entiers distincts k et n tels que $q_0 \stackrel{a^n}{\rightarrow} q_n$, $q_0 \stackrel{a^k}{\rightarrow} q_k$ et $q_k = q_n$. Or a^nb^n est accepté par M donc $q_0 \stackrel{a^nb^n}{\rightarrow} q_f$ mais alors on a aussi $q_0 \stackrel{a^kb^n}{\rightarrow} q_f$ bien que $a^kb^n \notin L$.
Donc L n'est pas rationnel.

Utilisons le lemme de l'étoile pour démontrer ce résultat : si L est rationnel, il est reconnu par un automate M à n états. Soit $m = a^nb^n$, d'après le lemme de l'étoile, il existe u, x et v tels que : $m = uxv$ et $|ux| \leq n$, donc u et x ne comportent que la lettre a, il existe donc des entiers i et j $(j > 0)$ tels que $u = a^i$ et $x = a^j$ d'où $v = a^{n-i-j}b^n$. D'après le lemme $uv \in L$ or $uv = a^{n-j}b^n$ ceci implique que $j = 0$, ce qui est contradictoire.

11.4.6 Calcul d'un automate à partir d'une expression rationnelle

Dans ce qui suit on écrira souvent $r|s$ à la place de $r + s$.

On considère l'expression rationnelle $(ba|a)^\star bb(a|ab)^\star$. L'automate reconnaissant le langage décrit par a est donné par le diagramme (30) et celui du diagramme (31) reconnaît (ba).

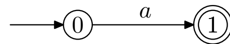


FIG. 30 – Reconnaît a

On en déduit que l'automate (32) reconnaît $(a|ba)$.

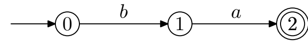


FIG. 31 – Reconnaît (ba)

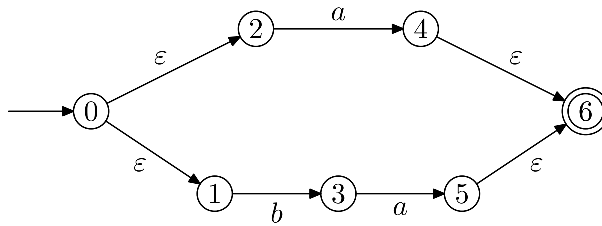


FIG. 32 – Reconnaît $(a|ba)$

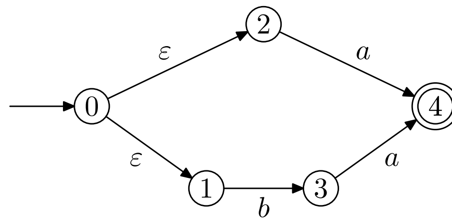


FIG. 33 – Reconnaît $(a|ba)$ (forme simplifiée)

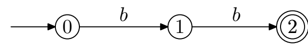


FIG. 34 – Reconnaît (bb)

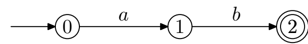


FIG. 35 – Reconnaît (ab)

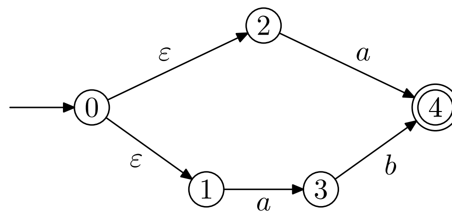


FIG. 36 – Reconnaît $(a|ab)$

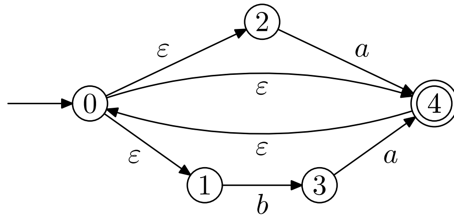


FIG. 37 – Reconnaît $(ba|a)^\star$

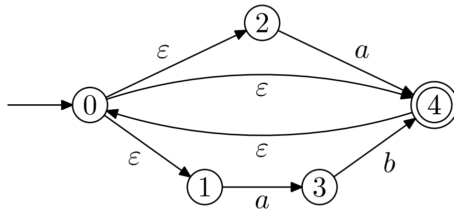


FIG. 38 – Reconnaît $(a|ab)^\star$

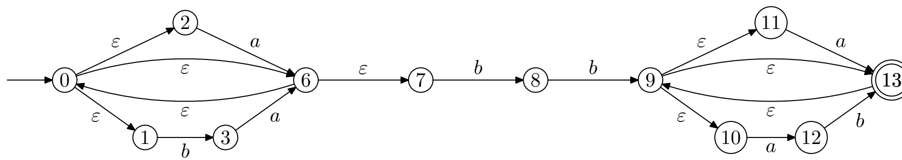


FIG. 39 – Reconnaît $(ba|a)^\star bb(a|ab)^\star$

On remarque que l'automate (32) peut être simplifié en l'automate (33) en identifiant les états 4 et 5.

L'automate (39) à transitions spontanées reconnaît donc $(ba|a)^{\star}bb(a|ab)^{\star}$, en le déterminisant et après simplification, on trouve que l'automate déterministe (40) reconnaît $(ba|a)^{\star}bb(a|ab)^{\star}$.

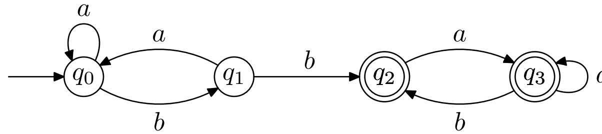


FIG. 40 – Automate déterministe reconnaissant $(ba|a)^{\star}bb(a|ab)^{\star}$

11.4.7 Exemple de calcul d'une expression rationnelle

On considère l'automate de la figure (41) et il s'agit de déterminer l'expression rationnelle du langage qu'il reconnaît.

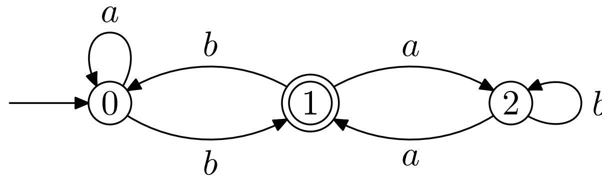


FIG. 41 – Automate déterministe.

L'expression rationnelle est calculée par récurrence en suivant la démonstration du théorème de Kleene (11.3).

On dresse le tableau initial $(L^0_{i,j})_{\substack{1\leq i\leq n\\ 1\leq j\leq n}}$:

(*les états sont étiquetés de 0 à 2 mais numérotés de 1 à 3.*

	$j=1$	$j=2$	$j=3$
$i=1$	$\{\varepsilon\}\|\{a\}$	$\{b\}$	$\emptyset$
$i=2$	$\{b\}$	$\{\varepsilon\}$	$\{a\}$
$i=3$	$\emptyset$	$\{a\}$	$(\{\varepsilon\}\|\{b\})$

Puis grâce aux relations : $L^k_{i,j} = L^{k-1}_{i,k}\left(L^{k-1}_{k,k}\right)^{\star} L^{k-1}_{k,j} \cup L^{k-1}_{i,j}$, on établit les tableaux de $L^1_{i,j}$, $L^2_{i,j}$ en utilisant les formules : $r^k_{i,j} = r^{k-1}_{i,k}\left(r^{k-1}_{k,k}\right)^{\star} r^{k-1}_{k,j} + r^{k-1}_{i,j}$.

Exemples : $L^0_{1,1} = \{a,\varepsilon\} = \{a\}\cup\{\varepsilon\}$, on note $r^0_{1,1} = (\varepsilon|a)$, puis $r^0_{1,2} = r^0_{2,1} = b$, $r^0_{1,3} = r^0_{3,1} = \emptyset$ *etc...*

Ainsi, après calculs, le tableau de $(r^1_{i,j})_{\substack{1\leq i\leq n\\1\leq j\leq n}}$ est :

	$j=1$	$j=2$	$j=3$
$i=1$	$a^\star$	$ba^\star$	$\emptyset$
$i=2$	$a^\star b$	$(\varepsilon\|ba^\star b)$	a
$i=3$	$\emptyset$	a	$(\varepsilon\|b)$

Ce tableau permet le calcul de $(r^2_{i,j})_{\substack{1\leq i\leq n\\1\leq j\leq n}}$:

	$j=1$	$j=2$	$j=3$
$i=1$	$(a^\star\|a^\star b(ba^\star b)^\star ba^\star$	$(ba^\star b)^\star ba^\star$	$a(ba^\star b)^\star ba^\star$
$i=2$	$a^\star b(ba^\star b)^\star$	$(ba^\star b)^\star$	$a(ba^\star b)^\star$
$i=3$	$a^\star b(ba^\star b)^\star a$	$(ba^\star b)^\star a$	$(\varepsilon\|b\|a(ba^\star b)^\star a)$

Enfin ce dernier tableau fournit l'expression rationnelle du langage reconnu ($k=3$ le nombre d'états, $i=1$ est le numéro de l'état initial et $j=2$ le numéro de l'état final) :

$$r^3_{0,1} = (a^\star b(ba^\star b)^\star | a^\star b(ba^\star b)^\star a(b|a(ba^\star b)^\star a)^\star a(ba^\star b)^\star)$$

ou

$$r^3_{0,1} = (a^\star b(ba^\star b)^\star + a^\star b(ba^\star b)^\star a(b + a(ba^\star b)^\star a)^\star a(ba^\star b)^\star)$$

11.5 Compléments

11.5.1 Constructions d'automates

On considère deux langages L_1 et L_2 sur un même alphabet $\mathcal{A}$ reconnus respectivement par des automates finis déterministes $M_1 = (E_1, p_{1,0}, F_1, \mathcal{A}, \delta_1)$ et $M_2(E_2, p_{2,0}, F_2, \mathcal{A}, \delta_2)$.

Il existe des constructions d'automates déterministes reconnaissant les langages $L_1 \cap L_2$ et $L_1 \cup L_2$.

- Le langage $L_1 \cap L_2$ est reconnu par l'automate :

$$M = (E_1 \times E_2, (p_{1,0}, p_{2,0}), F_1 \times F_2, \mathcal{A}, \delta)$$

  où δ est définie par $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$.
- Le langage $L_1 | L_2$ est reconnu par l'automate :

$$M = (E_1 \times E_2, (p_{1,0}, p_{2,0}), (E_1 \times F_2) \cup (F_1 \times E_2), \mathcal{A}, \delta)$$

  où δ est définie par $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$.

11.5.2 Minimisation

Étant donné un automate M, il est possible quelques fois de trouver un automate qui reconnaisse le même langage et qui possède moins d'états.

Définition 11.6 *Soit L un langage d'alphabet $\mathcal{A}$. On dit que L sépare deux mots u et v si ($u \in L$ et $v \notin L$) ou ($u \notin L$ et $v \in L$).*

Définition 11.7 *On dit que les mots u et v sont égaux modulo L si pour tout mot m de $\mathcal{A}^\star$, L ne sépare pas um et vm. On note :*

$$u \equiv v \mod L$$

C'est une relation d'équivalence sur $\mathcal{A}^\star$. On note $\mathcal{A}^\star/L$ l'ensemble des classes d'équivalence (appelé ensemble quotient).

Cette définition permet de caractériser les langages rationnels.

Théorème 11.7 *Un langage est reconnu par un automate fini si et seulement si les classes de $\mathcal{A}^\star$ modulo L sont en nombre fini.*

Preuve

Définition 11.8 *On dit qu'un état q d'un automate M, de langage $\mathcal{A}$ d'état initial p_0 et de transition δ, est* accessible *s'il existe un mot m de $\mathcal{A}^\star$ tel que $q = \overline{\delta}(p_0, m)$. q est inaccessible dans le cas contraire.*

Soit $M = (E, p_0, F, \delta, \mathcal{A})$ un automate fini reconnaissant L, sans état inaccessible. Soit $\mathcal{R}$ la relation sur $\mathcal{A}^\star$ définie par :

$$\overline{\delta}(p_0, x) = \overline{\delta}(p_0, y)$$

Si $(x, y) \in \mathcal{R}$ alors $u \equiv v \mod L$ (L ne sépare pas x et y) car pour tout mot m : $\overline{\delta}(p_0, xm) = \overline{\delta}(p_0, ym)$, donc $xm \in L$ si et seulement si $ym \in L$.
Ainsi la relation $\mathcal{R}$ est plus fine que la congruence modulo L.
Donc l'ensemble quotient $\mathcal{A}^\star/L$ a moins d'éléments que l'ensemble quotient $\mathcal{A}^\star/\mathcal{R}$ des classes d'équivalence suivant $\mathcal{R}$.

Or $\mathcal{A}^\star/\mathcal{R}$ est en bijection avec E car l'application $x \mapsto \overline{\delta}(p_0, x)$ est surjective (pas d'élément inaccessible) et l'application qui s'en déduit de $\mathcal{A}^\star/\mathcal{R}$ dans E est injective, donc le nombre de classes modulo L est fini.
Inversement, soit L un langage définissant un nombre fini de classes $\{c_0, \ldots, c_{n-1}\}$ modulo L. On définit un automate M d'alphabet $\mathcal{A}$ par son ensemble d'états $E = \{c_0, \ldots, c_{n-1}\}$ et son état initial c_0 (classe du mot vide). Sa fonction de transition δ étant telle que pour tout mot x et tout caractère a :

$$\delta(\overline{x}, a) = \overline{xa}$$

où $\overline{m}$ désigne la classe modulo L du mot m. δ est bien définie, *i.e.* $\delta(\overline{x}, a)$ ne dépend que de la classe de x et pas de son représentant x (en effet, si $\overline{x} = \overline{y}$, x et y ne sont pas séparés par L donc pour tout mot m' on a $\overline{xm'} = \overline{ym'}$, en prenant $m' = am$ on voit que xa et ya ne sont pas séparés et donc $\overline{xa} = \overline{ya}$). L'ensemble F des états finaux de M est défini par :

$$F = \{\overline{x}/x \in L\}$$

Vérifions que M reconnaît L par récurrence sur la longueur des mots. On a :

$$\delta(c_0, a) = \delta(\overline{\varepsilon}, a) = \overline{\varepsilon a} = \overline{a}$$

Donc pour tout mot m : $\overline{\delta}(c_0, m) = \overline{m}$. Donc un mot m est accepté si et seulement si $\overline{m} \in F$ donc si et seulement si $m \in L$. Ce qui achève la démonstration.

Remarque 11.3 *Un automate M reconnaissant L a un nombre d'état supérieur ou égal au nombre de classes modulo L.*

Cette remarque motive la définition suivante :

Définition 11.9 *On dit que M est* minimal *si le nombre d'états de M est égal au nombre de classes modulo L.*

Donc si M est minimal on peut identifier les états de M avec les classes modulo L :

$$c(p) = \{m \in \mathcal{A}^\star / \overline{\delta}(p_0, m) = p\} = \overline{m}$$

Définition 11.10 *On dit que deux états p et q d'un automate M sont équivalents si pour tout mot m : $\overline{\delta}(p, m) \in F \Leftrightarrow \overline{\delta}(q, m) \in F$.*

Minimisation d'un automate :

Attention aux différentes définitions de $x \mapsto \overline{x}$.

Soit M un automate ayant E pour ensemble d'états : $M = (A, E, \delta, p_0, F)$. Nous allons construire un automate minimal reconnaissant le même langage. La première étape consiste à retirer les états inaccessibles, on obtient un automate d'états E' et de transition δ', qui reconnaît encore le même langage. Ensuite on pose $E'' = E'/\mathcal{R}$ où $\mathcal{R}$ est la relation d'équivalence des états (définition 11.10). L'alphabet est toujours A, l'état initial est $\overline{p_0}$, classe de l'état initial de M. L'ensemble des états finaux F'' est défini comme ensemble des $\overline{q}$ tels que $\overline{q} \cap F \neq \emptyset$. Enfin, la transition δ'' est définie par :

$$\forall \overline{q} \in E'', \quad \forall a \in A \ : \qquad \delta''(\overline{q}, a) = \overline{\delta(q, a)}$$

Vérifions la validité de cette définition ; il suffit de montrer que si p et q sont équivalents alors $\delta(p, a)$ et $\delta(q, a)$ le sont aussi.

Soit p équivalent à q, alors pour tout mot m :

$$\overline{\delta}(q, m) \in F \Leftrightarrow \overline{\delta}(p, m) \in F$$

δ'' sera bien définie si $\overline{\delta(p, a)} = \overline{\delta(q, a)}$ soit si

$$\overline{\delta}(\delta(p, a), m) \in F \Leftrightarrow \overline{\delta}(\delta(q, a), m) \in F$$

or $\overline{\delta}(\delta(p, a), m) = \overline{\delta(p, am)}$ et $\overline{\delta}(\delta(q, a), m) = \overline{\delta(q, am)}$ comme par hypothèse, pour tout mot m' :

$$\overline{\delta}(p, m') \in F \Leftrightarrow \overline{\delta}(q, m') \in F$$

en posant $m' = am$ on voit que les états $\delta(p, a)$ et $\delta(q, a)$ sont équivalents.

On définit l'automate M'' par :

$$M'' = (A, E'', \delta'', \overline{p_0}, F'')$$

Remarquons qu'un état q équivalent à un état final est un état final (prendre $m = \varepsilon$).

Alors un mot m est accepté par M si et seulement s'il est accepté par M'' car $\overline{\delta''}(\overline{p_0}, m) = \overline{\overline{\delta}(p_0, m)}$ et par définition de F'' :

$$\overline{\overline{\delta}(p_0, m)} \in F'' \Leftrightarrow \overline{\overline{\delta}(p_0, m)} \cap F \neq \emptyset$$

or $\overline{\delta}(p_0, m)$ étant équivalent à un état final est un état final d'après la remarque ci-dessus, donc :

$$\overline{\delta}(p_0, m) \in F \Leftrightarrow \overline{\delta''}(\overline{p_0}, m) \in F''$$

Vérifions enfin que M'' est minimal. Deux états u et v sont équivalents modulo L si et seulement si :

$$\forall m \in A^\star : \qquad \overline{\delta}(p_0, um) \in F \Leftrightarrow \overline{\delta}(p_0, vm) \in F$$

or $\overline{\delta}(p_0, um) = \overline{\delta}(\overline{\delta}(p_0, u), m)$ et $\overline{\delta}(p_0, vm) = \overline{\delta}(\overline{\delta}(p_0, v), m)$ donc

$$\forall m \in A^\star : \overline{\delta}(\overline{\delta}(p_0, u), m) \in F \Leftrightarrow \overline{\delta}(\overline{\delta}(p_0, v), m) \in F$$

autrement dit les états $\overline{\delta}(p_0, um)$ et $\overline{\delta}(p_0, vm)$ sont équivalents, d'où :

$$\overline{\delta''}(\overline{p_0}, u) = \overline{\delta''}(\overline{p_0}, v)$$

À une classe modulo L on a associé un état de M''.
Inversement à un état $\overline{q} = \overline{\delta''}(\overline{p_0}, m) = \overline{\overline{\delta}(p_0, m)}$ de M'' (on rappelle que $\overline{q}$ est accessible), on associe l'ensemble des mots u tels que $\overline{\overline{\delta}(p_0, u)} = \overline{q}$, soient tels que :

$$\overline{\delta}(p_0, m) \in F \Leftrightarrow \overline{\delta}(p_0, u) \in F$$

autrement dit : la classe de m modulo L.
M'' a donc le nombre minimal d'états.

Algorithme de minimisation :
On considère le tableau des couples d'états de M :

$$T = \{(p_i, q_j) \in E^2 / i > j\}$$

ou plutôt le demi-tableau, pour n'avoir chaque couple qu'une fois.
On procède ensuite de la manière suivante :

1. on *coche* (p, q) si p ou q est un état final ;
2. pour chaque lettre a on coche le couple (p, q) si le couple $(\delta(p, a), \delta(q, a))$ est coché ;
3. on recommence jusqu'à ce que le tableau ne se modifie plus.

Alors p et q sont équivalents si et seulement si (p, q) n'est pas coché.

Preuve : on démontre par récurrence que si p et q ne sont pas équivalents, il existe m de longueur minimale tel que $\overline{\delta}(p, m) \in F$ et $\overline{\delta}(q, m) \notin F$, ou inversement, et (p, q) est coché.

Si $m = \varepsilon$, $p \in F$ et $q \in F$, alors (p, q) est coché, par définition de l'algorithme.
On suppose $m = au$ et on pose :

$$\begin{array}{rcl} p' & = & \delta(p, a) \\ q' & = & \delta(q, a) \end{array}$$

ainsi :

$$\begin{array}{rcl} \delta(p', u) & \in & F \\ \delta(q', u) & \notin & F \end{array}$$

donc (p', q') est coché d'après l'hypothèse de récurrence, ce qui conduit à cocher (p, q) par définition de l'algorithme.

Amélioration : second algorithme :

On procède par raffinement de partitions de l'ensemble des états. La première partition est : $F, \quad E \setminus F$.

Puis on fait une récurrence : soit $X_1, \ldots, X_k$ une partition de E et a une lettre. Pour tout indice i on partitionne X_i en $\{p \in X_i / \delta(p, a) \in X_j\} \cup \{p \in X_i / \delta(p, a) \notin X_j\}$ s'il existe un indice j tel que ces ensembles soient non vides. Lorsque l'on a épuisé les lettres, la partition obtenue est la partition en états équivalents.

Preuve : par récurrence. Hypothèse : si $p \in X_i$ et $q \in X_j$ où $i \neq j$, alors p et q ne sont pas équivalents.

Soient alors p et q dans X_i. S'il existe une lettre a telle que $\delta(p, a) \in X_j$ et $\delta(q, a) \notin X_j$ alors $\delta(p, a)$ et $\delta(q, a)$ ne sont pas équivalent d'après l'hypothèse de récurrence. Mais alors p et q ne sont pas équivalents car l'équivalence de p et q implique celle de $\delta(p, a)$ et $\delta(q, a)$.

Soit $\mathcal{P} = \{X_1, \ldots\}$, la dernière partition : s'il existe une lettre a telle que $\delta(p, a)$ et $\delta(q, a)$ ne sont pas dans le même ensemble X_i, alors p et q ne sont pas équivalents.

Dans le cas contraire, pour toute lettre a, il existe i tel que :

$$\delta(p, a) \in X_i \Leftrightarrow \delta(q, a) \in X_i$$

or $X_i \subset F$ ou $X_i \subset E \setminus F$. Donc :

$$\delta(p, a) \in F \Leftrightarrow \delta(q, a) \in F$$

on en déduit que pour tout mot m :

$$\overline{\delta}(p, m) \in F \Leftrightarrow \overline{\delta}(q, m) \in F$$

i.e. p et q sont équivalents.

12 Analyses lexicale, syntaxique, sémantique

12.1 Termes

Une signature est la donnée d'un ensemble Σ dont les éléments sont appelés *éléments fonctionnels* et d'une application $\sigma : \Sigma \to \mathbb{N}$. Pour tout f dans Σ, $\sigma(f)$ est l'*arité* du symbole f. Un symbole d'arité n (entier positif) est dit n-*aire* ; si $n = 1$ on dit *unaire*, si $n = 2$, *binaire etc...* L'ensemble des symboles d'arité n est noté Σ_n. Soit V un ensemble disjoint de Σ, ensemble des *variables*, un terme sur V est est un arbre t étiqueté par $\Sigma \cup V$ tel que l'étiquette d'une feuille est un symbole constant ou une variable et l'étiquette d'un nœud de degré n est un symbole d'arité n. L'ensemble $T(\Sigma, V)$ des termes est défini par induction par :

- si $x \in V$ alors $x \in T(\Sigma, V)$;
- si $f \in \Sigma_n \wedge t_1, \dots, t_n \in T(\Sigma, V)$ alors $f(t_1, \dots, t_n) \in T(\Sigma, V)$.

Un terme est dit *clos* si ses feuilles sont des constantes.

Exemple 12.1 (Expressions logiques) Soit Σ défini par $\Sigma_0 = \{0, 1\}$, $\Sigma_1 = \{\neg\}$, $\Sigma_2 = \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$ et $\Sigma_n = \infty$ pour $n \geq 3$. Dans ce cas un terme est une expression logique. L'expression logique $\neg(\vee(x, y))$ est représentée par le terme t (figure 12.1)

Autres exemples : expressions rationnelles, expressions arithmétiques.

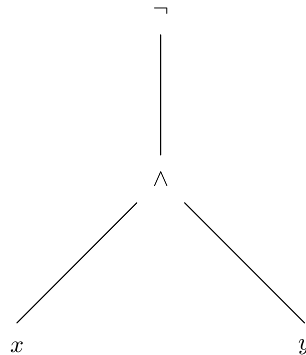


FIG. 42 – t

12.2 Sémantique

Définition 12.1 *Soit un ensemble A, une Σ-algèbre $\mathcal{A}$ sur A est une application qui à tout symbole $f \in \Sigma_n$ d'arité n associe une application, appelée* interprétation *de f, $f^{\mathcal{A}}$ de A^n dans A :*

$$f^{\mathcal{A}} : (a_1, \dots, a_n) \longrightarrow f^{\mathcal{A}}(a_1, \dots, a_n)$$

Définition 12.2 *Une* liaison *(ou* environnement*) ℓ est une application de V dans A (*i.e. *un élément de A^V).*

L'interprétation d'un terme est définie par induction :
– si $t = x \in V$ alors $t^{\mathcal{A}}(\ell) = \ell(x)$;
– si $t = f(t_1, \dots, t_n)$ alors $t^{\mathcal{A}}(\ell) = f^{\mathcal{A}}(t_1^{\mathcal{A}}(\ell), \dots, t_n^{\mathcal{A}}(\ell))$.

Exemple 12.2 Reprenons l'exemple précédent et interprétons les éléments de Σ. On représente les fonctions constantes par les constantes elles-mêmes, comme de coutume.
– Σ_0 : $0^{\mathcal{A}} = 0$, $1^{\mathcal{A}} = 1$;
– Σ_1 : $\neg^{\mathcal{A}}(0) = 1$, $\neg^{\mathcal{A}}(1) = 0$;
– Σ_2 :
  – $\wedge^{\mathcal{A}}(x, y) = 1$ si et seulement si $x = y = 1$;
  – $\vee^{\mathcal{A}}(x, y) = 1$ si et seulement si $x = 1$ ou $y = 1$;
  – $\Rightarrow^{\mathcal{A}} (x, y) = 1$ si et seulement si $x = 0$ ou $y = 1$;
  – $\Leftrightarrow^{\mathcal{A}} (x, y) = 1$ si et seulement si $x = y$.

Ainsi une tautologie est une expression logique dont l'interprétation est 1.

12.3 Syntaxes

La syntaxe *concrète* est l'écriture de phrases compréhensibles par l'humain, ces phrases sont analysées et traduites en phrases obéissant à la syntaxe *abstraite* pour être traitées par la machine. La première phase de l'analyse syntaxique est l'analyse *lexicale* qui découpe la phrase en mots.

Autrement dit : la chaîne d'entrée (syntaxe concrète) passe par l'analyseur syntaxique qui fournit en sortie la syntaxe abstraite.

Exemple 12.3 $\neg(\vee(x, y))$ est la forme abstraite préfixée de $\neg(x \vee y)$. Préfixée car le parcours du terme est préfixe.

Exemple 12.4 Les expressions arithmétiques sont définies par une signature Σ dont les symboles constants sont les mots qui représentent un réel en base 10 (plus précisément un nombre décimal), les symboles unaires sont $\Sigma_1 = \{\ln, \exp, \sin\cos, \arctan, \mathrm{moins}\}$ et les symboles binaires sont $\Sigma_2 = \{+, -, \times, /, \}$. Soit $V = \{x, y\}$ l'ensemble des variables.
L'expression infixe parenthésée $(x + 1)\ln y$ est représentée par un arbre binaire (figure 12.3) dont les parcours préfixes, infixes et postfixes donnent respectivement les expressions :
– préfixes : $\times + x 1 \ln y$;
– infixes : $x + 1 \times \ln y$ (un nœud de degré 1 n'est rencontré que deux fois) ;
– postfixes : $x 1 + y \ln \times$.

Exercice 12.1 Écrire des programmes de transformation d'expressions concrètes en expressions abstraites postfixées.

12.4 Exemple : dérivation formelle

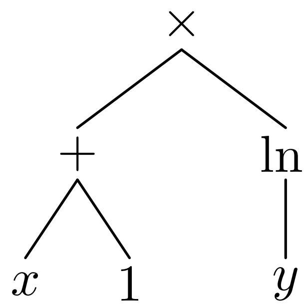


FIG. 43 – t