

# Cours d'informatique de MPSI (2011-2012)

Stéphane Flon



# Table des matières

<b>Avant de commencer</b>	5
Introduction à l'option informatique	7
1. Objectifs de formation	7
2. Choisir concrètement l'option informatique	7
3. Bref aperçu du programme de SUP	8
4. FAQ	8
<b>partie A. Programmation en Caml</b>	9
Chapitre I. Introduction	11
1. Le langage Caml	11
2. Quelques conseils	13
Chapitre II. Programmation en Caml	15
1. Brève présentation de Caml light	15
2. Notion d'effet de bord	15
3. Identificateurs	16
4. Types élémentaires	18
5. Deux exemples de types non élémentaires	20
6. Programmation fonctionnelle	20
7. Introduction à la récursivité	25
8. Programmation impérative	25
9. Types prédéfinis en Caml	28
10. Bref retour sur le filtrage	32
11. Types définis par l'utilisateur	34
<b>partie B. Méthodes de programmation</b>	37
Chapitre III. Programmation impérative	39
1. Correction et terminaison	39
2. Notion d'invariant de boucle	39
Chapitre IV. Récursivité	43
1. Introduction	43
2. Compléments sur les relations d'ordre	43
3. Terminaison et correction des fonctions récursives	46
4. Récursivité croisée	50
5. Gestion d'une fonction récursive	50
6. Ensembles inductifs, induction structurelle	57
<b>partie C. Analyse des algorithmes</b>	59
Chapitre V. Complexité	61
1. Généralités	61
2. Un premier exemple	62
3. Diviser pour régner	62
Chapitre VI. Tris	67
1. Introduction	67
2. Utilité du tri : recherche dans un tableau trié	67
3. Insertion dans une liste triée	68

---

4. Fusion de deux listes triées	68
5. Exemples de tris	69
<b>partie D. Structures de données et algorithmes</b>	<b>81</b>
Chapitre VII. Structures de données	83
1. Notion de structure de données	83
2. Listes	84
3. Piles	87
4. Files	90
5. Arbres	92
6. Dictionnaires	93
<b>partie E. Exercices</b>	<b>95</b>
Feuille de TD 1. Premiers pas en Caml	97
Feuille de TD 2. Premiers pas en récursivité et itération	99
Feuille de TD 3. Itération	101
Feuille de TD 4. Terminaison et correction des fonctions récursives	103
Feuille de TD 5. Récursivité terminale	105
Feuille de TD 6. Induction structurelle	107
Feuille de TD 7. Complexité	109
Feuille de TD 8. Diviser pour régner	111
Feuille de TD 9. Tris	113
Feuille de TD 10. Listes	115
Feuille de TD 11. Piles et files	117
Feuille de TD 12. Arbres	119
<b>partie F. Solution des exercices</b>	<b>121</b>
Feuille de TD 1. Solution de premiers pas en Caml	123
Feuille de TD 2. Solution de premiers pas en récursivité et itération	125
Feuille de TD 3. Solution d'itération	129
Feuille de TD 4. Solution de terminaison et correction des fonctions récursives	133
Feuille de TD 5. Solution de récursivité terminale	135
Feuille de TD 6. Induction structurelle	137
Feuille de TD 7. Solution de complexité	139
Feuille de TD 8. Solution de diviser pour régner	141
Feuille de TD 9. Solution de tris	143
Feuille de TD 10. Solution de listes	145
Feuille de TD 11. Solution de piles et files	149

**Avant de commencer**



# Introduction à l'option informatique

## 1. OBJECTIFS DE FORMATION

L'informatique, telle qu'enseignée en option en MPSI, est envisagée comme une science, et non comme un amas de techniques ou de compétences : comme le dit le programme officiel, « l'informatique est une science opérant sur des représentations rigoureuses de concepts bien définis ».

Cela signifie notamment que l'enseignement de cette option ne consistera pas seulement à vous faire programmer : une grande part du travail s'effectuera loin de tout ordinateur. Bien entendu, et heureusement, l'enseignement ne sera pas purement théorique, et vous pourrez souvent confronter votre apprentissage à un principe de réalité, en l'implémentant en salle info.

Pour cela, nous utiliserons le langage Caml (light), tout à fait approprié à l'esprit de l'option info (beaucoup plus que Maple par exemple ...).

Nous aurons souvent recours aux mathématiques, afin de poser les bases d'une programmation rigoureuse. Elles nous permettront notamment :

- (1) de *prouver* un programme (comme on prouve un théorème mathématique).
- (2) d'évaluer l'efficacité d'un programme, en particulier le temps qu'il met à s'exécuter (étude de la *complexité* d'un programme).
- (3) de confronter des parti-pris de programmation, de choisir, parmi eux, le plus adapté à la situation rencontrée (récuratif *vs* impératif, étude des *structures de données*).

Il s'agira donc, grâce à cette démarche rigoureuse, de réfléchir –en décortiquant ce que l'on veut faire, ce dont on dispose, et comment on va s'y prendre–, avant de se lancer dans la programmation proprement dite.

Dans le cadre de cette option, ce sont les mathématiques qui sont au service de l'informatique, et non l'inverse.

L'idée est, entre autres, d'éviter la technique de programmation suivante, si souvent rencontrée : « écrire un programme au hasard qui, avec un peu de chance, va donner le bon résultat, puis on se rend compte qu'en fait non, puis on passe des heures à tenter de le corriger ».

En cela, cette option est très utile à qui veut bien programmer plus tard, même si elle va sans doute trop loin pour un ingénieur dont la programmation serait une activité secondaire. Cependant, cette discipline, par la rigueur et la logique qu'elle requiert, reste toujours bénéfique, au même titre que les mathématiques : en tant qu'ingénieur, vous ne vérifierez pas la continuité d'une fonction avant d'appliquer le théorème des valeurs intermédiaires, mais votre formation vous permettra de prendre conscience d'un cas retors, voire de le résoudre (cf. le phénomène de Gibbs).

## 2. CHOISIR CONCRÈTEMENT L'OPTION INFORMATIQUE

### 2.1. CONSÉQUENCES SUR L'ORIENTATION

En choisissant l'option informatique, vous fermez la voie de la PSI : vous ne pourrez aller qu'en MP (ou MP\*). Bien entendu, cela ne vous limitera pas plus tard à une carrière dans l'informatique, toutes les écoles restent accessibles avec cette option, et, au sein de ces écoles, toutes les voies sont possibles.

L'option informatique permet de passer tous les concours MP, indifféremment de l'option SI, à l'exception notable de Polytechnique, où les listes d'admis sont séparées selon l'option.

### 2.2. ORGANISATION DU TRAVAIL

Vous aurez informatique le vendredi matin de 8h à 10h : une heure de cours, puis une heure de TD. L'heure de TD se fera parfois devant l'ordinateur mais pas systématiquement, loin de là.

Certaines de ces séances seront consacrées aux devoirs surveillés (deux par trimestre).

Vous aurez également quelques colles Caml au long de l'année (horaires à définir).

## 3. BREF APERÇU DU PROGRAMME DE SUP

- *Initiation à Caml*  
On présente le langage Caml, ses qualités, sa philosophie, sa syntaxe.
- *Méthodes de programmation*  
Programmation impérative (boucles for, while, conditionnelle if, etc.). Comment prouver un programme impératif.  
Programmation récursive : une nouvelle philosophie de programmation (la fonction « s'appelle elle-même » dans sa définition). Compléments mathématiques (sur les relations d'ordre), ensembles inductifs. Comment prouver un programmer récursif.
- *Analyse des algorithmes*  
Complexités spatiale et temporelle (performance théorique des algorithmes conçus).  
Méthode diviser pour régner.  
Algorithmes classiques de tri (comment ranger des valeurs selon un ordre donné).
- *Structures de données et algorithmes*  
Notion de structure de données. Listes, piles. Comment choisir la structure de données adaptée à un problème, quelles conséquences cela a-t-il sur la programmation ?
- *Logique*  
Calcul propositionnel. Distinction entre syntaxe et sémantique. Circuits logiques.
- *Et en Spé ?*  
Bien sûr, vous reprendrez en partie ce qui a été fait en Sup, mais vous étudierez également de nouveaux sujets, parfois très abstraits, comme les automates finis.

## 4. FAQ

## 4.1. QUELLES SONT LES BONNES RAISONS DE SUIVRE L'OPTION INFORMATIQUE ?

On peut suivre l'option informatique

- par goût de l'informatique, par exemple parce qu'on
  - veut devenir informaticien, ou intégrer une école d'informatique (désolé pour l'évidence ...)
  - l'a déjà pratiqué (C/C++, php, etc.)
  - a apprécié la partie programmation de Maple.
  - n'a pas apprécié la partie programmation de Maple, parce qu'on l'a trouvée trop peu rigoureuse.
  - voudrait commencer la programmation sur de bonnes bases.
- par goût des (et facilités en) mathématiques, surtout algébriques : relations d'ordre, structures algébriques, récurrence, dénombrement.
- par goût de l'abstraction, de la rigueur.

## 4.2. QUELLES SONT LES BONNES RAISONS DE NE PAS SUIVRE L'OPTION INFORMATIQUE ?

On a bien raison de ne pas suivre l'option informatique si :

- on surkiffe la SI.
- on veut aller en PSI (ou s'en garder la possibilité).
- veut intégrer une école spécialisée dans un domaine précis, relevant de la SI (cela dit, SI comme informatique sont certainement utiles, au moins comme bagage culturel, à tout ingénieur qui se respecte).
- on a toutes les difficultés du monde à programmer une boucle for en Maple, ou si on est exaspéré car Maple nous en veut de ne pas avoir mis « ; ».

Pour les écoles généralistes, les deux options se valent.

## 4.3. QUELLES SONT LES MAUVAISES RAISONS DE SUIVRE, OU DE NE PAS SUIVRE, L'OPTION INFORMATIQUE

Il ne faut pas se déterminer à cause :

- des horaires.
- de la « stratégie » : choisissez votre option selon vos goûts, et non parce qu'untel est une méga-torche et prend telle option ...
- de l'élitisme : l'option info est ouverte à tous ceux qui sont motivés, même s'ils ne sont pas géniaux en maths (mais ils s'engagent en connaissance de cause).



Première partie

**Programmation en Caml**



## CHAPITRE I

# Introduction

### 1. LE LANGAGE CAML

Bien que l'enseignement de l'option informatique ne se résume pas à la programmation, il nous faut disposer d'un langage dans lequel travailler, afin de mettre en pratique la théorie et de se confronter à un certain principe de réalité. Cette année, nous utiliserons le langage Caml, et plus précisément Caml light, que vous pourrez trouver facilement (et gratuitement) sur le net.

Sans rentrer dans une analyse poussée de ce langage, on peut lui reconnaître de nombreuses qualités pédagogiques, que nous passons brièvement en revue, et sur lesquelles nous reviendrons plus en détail au chapitre suivant :

#### 1.1. RIGUEUR

En Caml, tous les objets (y compris les fonctions) ont un *type*, c'est-à-dire que chacun appartient à une catégorie bien précise. On dit que ce langage est *fortement typé*.

Par exemple, comme en mathématiques, toute fonction a une source (un ensemble de départ) et un but (un ensemble d'arrivée) déterminés.

Après chaque définition, Caml répond sous la forme

```
nom : type = valeur
```

Après chaque expression, Caml répond sous la forme

```
- : type = valeur
```

(la réponse étant parfois précédée d'effets de bord, voir plus loin).

Dans l'exemple

```
# 2 + 2;;
```

```
- : int = 4
```

Caml nous informe que le résultat 4 de l'expression  $2 + 2$  est de type entier (int).

Ces contraintes de type confèrent à Caml une certaine rigidité : Caml n'aime pas mélanger les choux et les carottes<sup>1</sup>. Par exemple, Caml n'accepte pas que nous additionnions un entier (type int) avec un nombre à virgule flottante (type float) :

```
# 2 + 2.4;;
```

Toplevel input :

```
>2 + 2.4;;
```

```
>
```

```
This expression has type float ,  
but is used with type int .
```

Cette rigidité doit toutefois être mise au crédit de Caml, car elle permet d'éviter ou détecter de nombreuses erreurs de programmation.

#### 1.2. SIMPLICITÉ

Les bases de ce langage – dont nous nous contenterons – sont assez réduites, et la plupart des programmes que nous écrirons ne prendront que quelques lignes. Pourtant, nous pourrions répondre à de nombreux problèmes relativement compliqués.

1. sauf si on choisit de les mélanger en définissant un type plus général

Nous avons vu que chaque objet avait un type : faut-il pour autant déclarer le type de chaque objet en Caml? Non, car le typage est automatique : Caml devine, autant que possible, de quel type d'objet vous parlez.

---

```
# let f x = x + 2;;

f : int -> int = <fun>
```

---

J'ai ici défini la fonction  $f$ , dont Caml a automatiquement déterminé le type, à savoir une fonction qui à un entier associe un entier : il a deviné que l'argument  $x$  de  $f$  était entier, puisque je lui ai ajouté 2, et sait que le résultat obtenu est de type entier.

### 1.3. FONCTIONNEL

Nous venons de voir un exemple de fonction en Caml. En fait, un programme Caml n'est pour ainsi dire qu'une succession de fonctions, que nous nous contentons d'appliquer en fin de programme. Dans un programme Caml, il s'agit bien plus de décrire les fonctions que nous souhaitons employer (programmation fonctionnelle) que de donner des instructions de gestion des données ou de la mémoire de l'ordinateur (programmation impérative). Il impose donc une vision assez mathématique de la programmation.

Il faut bien comprendre que les fonctions en Caml sont des objets comme les autres, ce qui fait qu'on peut les prendre en argument, comme en mathématiques : ainsi,

---

```
# let translate_de_2_depart_arrivee f x = f (x + 2) + 2;;

translate_de_2_depart_arrivee : (int -> int) -> int -> int = <fun>
```

---

est la fonction qui, à une fonction  $f$  de type  $\text{int} \rightarrow \text{int}$ , associe la fonction de même type, qui à  $x$  (entier) associe  $f(x + 2) + 2$ . C'est donc, avec les notations usuelles, un élément de  $(\mathbb{N}^{\mathbb{N}})^{(\mathbb{N}^{\mathbb{N}})}$ .

### 1.4. POLYVALENT

Caml n'est pas un langage purement fonctionnel, dans le sens où il a également des traits impératifs, comme les boucles `for` et `while` par exemple, que vous connaissez sans doute déjà :

---

```
# for i = 1 to 10 do print_int i done;;

12345678910- : unit = ()

# let i = ref 2;;
i : int ref = ref 2

# while !i < 1000 do i := !i * !i done;;
- : unit = ()

# i;;
- : int ref = ref 65536
```

---

Il admet aussi des définitions par filtrage très puissantes et élégantes :

---

```
# let est_voyelle_minuscule = function
  | 'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
  | _ -> false;;
est_voyelle_minuscule : char -> bool = <fun>

# est_voyelle_minuscule 'e';;
- : bool = true

# est_voyelle_minuscule 's';;
- : bool = false

# est_voyelle_minuscule 'Y';;
- : bool = false
```

---

## 1.5. SÛR

Comme on l'a déjà observé, le typage automatique en Caml permet d'éviter de nombreuses erreurs. De plus, Caml n'hésite pas à produire des avertissements en cas de programmation tendancieuse. Par exemple, il prévient lorsqu'un cas de filtrage est oublié :

---

```
let f (x, y) = match (x, y) with
  | (0, y) -> y
  | (x, 0) -> x;;
```

Toplevel input :

```
.....match (x, y) with
  | (0, y) -> y
  | (x, 0) -> x..
```

Warning: this matching is **not** exhaustive.

```
f : int * int -> int = <fun>
```

---

Sans rentrer dans les détails, cette fonction associe à un couple d'entiers un entier. Cependant, sa définition ne donne la valeur que lorsque l'une des coordonnées est nulle.

## 2. QUELQUES CONSEILS

Donnons ici quelques recommandations de bon sens, afin de faciliter la programmation, et la compréhension de vos programmes par quelqu'un d'autre (moi, par exemple).

## 2.1. RÉFLÉCHIR AVANT DE PROGRAMMER

Il est recommandé, avant de se lancer dans l'écriture d'un programme, de bien réfléchir à la structure que l'on veut lui donner, éventuellement par écrit, au brouillon. Plutôt que d'écrire un premier jet incorrect, puis de lui appliquer des rustines jusqu'à ce qu'il fonctionne (ou semble fonctionner), au risque de le rendre incompréhensible, essayez de savoir où vous allez.

Pour bien structurer votre programme, répondant à un problème précis, vous pouvez essayer de décomposer ce dernier en sous-problèmes, se décomposant éventuellement à leur tour en (sous-)sous-problèmes, jusqu'à tomber sur des briques élémentaires faciles à programmer et à assembler : il s'agit donc de programmer par raffinements successifs, ou encore par *analyse descendante*.

Un autre avantage d'avoir écrit des sous-programmes élémentaires consiste en leur possible réutilisation et partage.

## 2.2. PRATIQUER

Un langage de programmation s'apprend en le pratiquant : il ne faut pas vous contenter de comprendre les programmes trouvés ici ou là, il vous faudra tester votre bonne assimilation en programmant, en les testant, en devinant ce qu'un programme va produire, etc.

Comme en mathématiques, il faudra travailler activement en TD, et non se contenter de lire un corrigé.

## 2.3. RENDRE VOTRE PROGRAMME LISIBLE

N'oubliez pas que le programme sera sans doute lu par quelqu'un d'autre, et que sa compréhension n'est pas forcément immédiate : il vous faut donc faire des efforts de lisibilité. D'ailleurs, même si votre programme est à usage personnel, il vaut mieux le clarifier, car vous pouvez avoir du mal à le comprendre une fois que vous l'avez laissé de côté pendant quelques temps.

N'hésitez pas à insérer des commentaires dans vos programmes (utiliser `(* ... *)` en Caml), expliquant votre démarche, pourquoi votre algorithme fonctionne, les notations, etc.

On simplifie grandement la compréhension d'un programme en donnant des noms parlants aux objets introduits : de même qu'en mathématiques, il est possible mais malvenu de noter un certain nombre complexe  $x$  et  $z$  sa partie réelle (ou un entier naturel  $\varepsilon$ ), il vaut mieux éviter en informatique de noter « addition » ou même « k » la loi de multiplication des entiers.

Pensez également à aérer et hiérarchiser le texte, notamment en utilisant les indentations.

Vous pouvez aussi rendre votre programmation plus abstraite, ce qui rend parfois le texte plus lisible. Attention cependant, dans certains cas, l'abstraction ne simplifie pas vraiment la compréhension.

## 2.4. LAISSER L'ORDINATEUR TRAVAILLER

La récursivité et l'aspect fonctionnel de Caml permettent d'écrire des programmes qui ne semblent être que des reformulations du problème posé. Ces programmes sont le plus souvent clairs, efficaces, élégants, et simples à écrire. Il est parfois possible, au prix d'un effort mathématique, d'enlaidir et compliquer lesdits programmes, mais on n'a pas grand chose à y gagner. Les intoxiqués de la programmation impérative doivent apprendre à laisser l'ordinateur faire le job !

## Programmation en Caml

Nous ne présentons ici qu'une petite partie des fonctionnalités de Caml light. Vous pouvez consulter l'aide en ligne pour approfondir certains aspects de ce langage.

### 1. BRÈVE PRÉSENTATION DE CAML LIGHT

L'implémentation Caml light que nous utiliserons se présente comme une boucle interactive :

- Caml invite l'utilisateur à écrire une phrase par le « prompt » # ;
- l'utilisateur tape une phrase, c'est-à-dire, le plus souvent, une *expression* ou une *déclaration*, qu'il termine par « ; », et valide en appuyant sur entrée ;
- Caml analyse alors la syntaxe de cette phrase ;
- lorsque cette syntaxe est correcte, il détermine le type de l'expression ;
- si cette expression est bien typée, il l'évalue, puis renvoie une réponse sous la forme - : **type** = valeur.

**Remarque :** à chacune des étapes évoquées, Caml peut renvoyer un message d'erreur.

**Remarque :** cette manière de programmer, au sein de cette boucle interactive, est conviviale. Elle permet d'utiliser Caml comme une calculatrice, ou un peu comme Maple (en moins souple). Elle a cependant des limites évidentes, dès que le programme écrit est élaboré. On pourra donc, le cas échéant, écrire un fichier en .ml, que l'on ouvrira dans l'implémentation Caml light.

### 2. NOTION D'EFFET DE BORD

#### 2.1. DÉFINITION

En informatique, on appelle *effet (de bord)* d'une fonction une modification de l'environnement : il peut s'agir par exemple de la modification d'une variable ou de l'argument de la fonction appliquée, de l'écriture sur un fichier, d'un affichage (« print »). Une fonction qui n'agit que par effet de bord (qui ne renvoie aucune valeur) est appelée *procédure*.

Caml dispose de plusieurs fonctions prédéfinies, qui n'agissent que par effet de bord, comme la fonction `print_int` :

---

```
# print_int 3;;
```

```
3- : unit = ()
```

---

qui a pour effet d'afficher l'entier pris en argument.

#### 2.2. NÉCESSITÉ D'UN TYPE UNIT EN CAML

En Caml, toutes les fonctions ont un argument, et renvoient un résultat : il doit en être ainsi des procédures, qui ne renvoient pourtant pas de résultat. Pour lever cette difficulté, on a défini le type `unit`, dont l'unique élément est `()`, et que l'on peut comprendre comme « rien ». En Caml, une procédure n'est rien d'autre qu'une fonction dont le résultat est `()`. C'est le cas par exemple de `print_int`, utilisée ci-dessus.

**Remarque :** il faudra bien distinguer par exemple les deux fonctions suivantes :

---

```
# let successeur n = n + 1;;
successeur : int -> int = <fun>
```

```
# let affiche_successeur n = print_int (n + 1);;
affiche_successeur : int -> unit = <fun>
```

---

En effet, `affiche_successeur` est une procédure, elle renvoie `()`. En particulier, on ne peut pas la composer avec elle-même, contrairement à `successeur` :

---

```
# successeur 3;;
- : int = 4

# successeur (successeur 3);;
- : int = 5
```

---

```
# affiche_successeur 3;;
4- : unit = ()

# affiche_successeur (affiche_successeur 3);;
Toplevel input:
>affiche_successeur (affiche_successeur 3);;
>
This expression has type unit ,
but is used with type int.
```

---

**Remarque :** de même qu'une fonction peut renvoyer (), une fonction peut prendre ce terme en argument : c'est par exemple le cas de la fonction suivante

---

```
# let Hello_world () = print_string "Hello_world";;
Hello_world : unit -> unit = <fun>

# Hello_world ();;
Hello world- : unit = ()

# Hello_world (Hello_world ());;
(* on peut composer cette procédure par elle-même,
   puisque son argument est de type unit *)
Hello worldHello world- : unit = ()
```

---

### 3. IDENTIFICATEURS

Il est souvent utile de donner un nom au résultat d'un calcul effectué par Caml. Comme en mathématiques, on peut avoir besoin de « variables » *globales* et *locales*.

#### 3.1. IDENTIFICATEURS GLOBAUX

On parle aussi, en se conformant à l'usage, de variables globales, même si nous verrons que cette expression peut induire en erreur. Les identificateurs globaux sont conservés en mémoire, et donc utilisables dans l'intégralité du programme. Ils sont à réserver aux objets dont nous avons besoin tout au long de notre travail.

Pour introduire (ou définir, ou déclarer) un identificateur global, la syntaxe est

<pre>let identificateur = valeur</pre>
--

Une telle définition est une *liaison* d'un identificateur (autrement dit d'un nom, d'une abréviation) à une valeur (qui devient ainsi identifiée), que l'on peut utiliser dans la suite de notre programme. C'est l'équivalent du « soit » en mathématiques, tel qu'employé dans l'expression « Soit  $M$  la borne supérieure de  $f$  sur  $[0, 1]$ . »

---

```
# let y = 7;; (* 7 a pour nom y *)
y : int = 7

# let successeur n = n + 1;;
successeur : int -> int = <fun>

# successeur y;; (* Caml sait qui est désigné par le symbole y *)
- : int = 8
```

---

```
# let y = 7;;
y : int = 7
```



```
# y + 2;;
- : int = 9
```

**Remarque :** voici un point relativement compliqué. Il ne faut pas confondre cette *définition* de l'*affectation*. **Une fois défini, un nom conserve la valeur donnée, à moins de le redéfinir**, c'est pourquoi le terme de « variable » est plutôt à éviter dans ce cadre, et il vaut mieux parler d'identificateur.

```
# let y = 7;; (* 7 a pour nom y *)
y : int = 7
```

```
# let x = y + 2;; (* 7 + 2 a pour nom x *)
x : int = 9
```

```
# let y = 2;; (* y n'identifie plus 7 mais 2 *)
y : int = 2
```

```
# x;; (* x est toujours le nom de 9 *)
- : int = 9
```

Pour une raison obscure, nous avons redéfini y. Nous pouvons même le redéfinir à partir de sa définition actuelle :

```
# let y = 7;; (* 7 a pour nom y *)
y : int = 7
```

```
# let y = y + 2;; (* 7+2 a pour nom y *)
y : int = 9
```

Cependant, cette programmation horrible est à proscrire, tant elle contrevient à l'esprit Caml. D'une manière générale, il ne semble pas opportun de redéfinir un identificateur <sup>1</sup>.

### 3.2. IDENTIFICATEURS LOCAUX

Ce sont les identificateurs auxiliaires, permettant de réaliser et de donner un sens à un calcul secondaire : ce sont des définitions « jetables » (et « jetées »). Ainsi en est-il des indices muets que nous utilisons dans les sommes : quand on écrit  $\sum_{k=0}^n k^3$  en mathématiques,  $n$  est un entier naturel qu'il faut avoir introduit dans le corps du texte, tandis que  $k$  est une variable muette ne servant qu'à proprement définir cette somme. Il faut savoir de quel (type de)  $n$  on parle, mais, inversement, il faut que  $k$  n'ait pas de valeur affectée (la phrase « Soit  $k = 3$  et  $n = 5$ . Calculons  $\sum_{k=0}^n k^3$ . » est rejetée en mathématiques).

La syntaxe est la même que pour les identificateurs globaux, mais on lie la déclaration de l'identificateur local à l'expression dans laquelle il est utilisé par **in** :

```
let identificateur_local = valeur in expression
```

```
# let y = 7 in y + 2;;
- : int = 9
```

```
# y;;
Toplevel input:
>y;;
>^
The value identifier y is unbound.
```

On peut aussi observer que cette phrase est une expression et non une déclaration.

On peut par ailleurs utiliser un identificateur local pour définir un identificateur global :

```
# let x = (* x est global *)
      let y = 7 in (* y est local, et sert à définir x *)
      y + 2;;
x : int = 9
```

```
# x;;
```

1. mais on peut le faire par mégarde si le programme est long

```
- : int = 9

# y;;
Toplevel input:
>y;;
>^
The value identifier y is unbound.
```

En reprenant l'analogie avec l'exemple de la somme en mathématiques, que se passe-t-il si nous avons la mauvaise idée d'utiliser un identificateur global pour identificateur local ?

```
# let y = 3;;
y : int = 3

# let x =
    let y = 7 in
    y + 2;;
x : int = 9

# x;;
- : int = 9

# y;;
- : int = 3
```

L'identificateur local l'a emporté au sein de l'expression à laquelle il est associé, ce qui est normal (à quoi servirait-il sinon ?). Bien sûr, il vaut mieux éviter cette réutilisation pour le moins maladroite.

Pour mieux insister sur le caractère secondaire d'un identificateur local, on peut inverser les ordres de la définition et de l'expression, en utilisant **where** au lieu de **in**, c'est-à-dire en écrivant

```
expression where identificateur_local = valeur
```

On peut ainsi écrire

```
# let x = y + 2 where y = 7;;
x : int = 9

# x;;
- : int = 9
```

**Remarque :** dans la mesure du possible, on privilégiera les identificateurs locaux. Cela limitera les erreurs.

Afin d'éviter de trop nombreuses définitions emboîtées, on pourra utiliser **and** pour introduire plusieurs identificateurs (qu'ils soient locaux ou globaux, et que ce soit avec **in** ou **where**) :

```
# z * z - x * x - y * y where z = 13 and x = 5 and y = 12;;
- : int = 0
```

## 4. TYPES ÉLÉMENTAIRES

On recense ici les types élémentaires en Caml, *i.e.* ceux qui ne sont pas construits à partir d'autres types : par exemple, le type `int` (celui des entiers) est élémentaire, pas le type `int -> int` (celui des fonctions qui à un entier associent un entier). Nous avons déjà parlé du type `unit`. Il ne s'agit pas ici de décrire de manière exhaustive ces types et les fonctions associées, mais de présenter les plus utiles d'entre eux.

### 4.1. LE TYPE BOOLÉEN

Le type booléen ne prend que deux valeurs : `true` et `false`. Un *prédicat* est une fonction à valeurs booléennes (c'est surtout à cela que servent les booléens). On dispose du connecteur logique « et », s'écrivant `&` ou `&&`, du connecteur logique « ou », s'écrivant `or` ou `||`, et de l'opérateur unaire de négation **not**.

**Remarque :** ces opérateurs binaires « et » et « ou » sont dits *infixes* car ils se placent entre les arguments.

**Remarque :** faites attention, **and** ne désigne pas le connecteur logique « et ».

**Remarque :** il faut savoir que Caml effectue une évaluation dite *paresseuse*, ce qui signifie que dès qu'il dispose d'informations suffisantes pour déterminer la valeur du booléen considéré, il arrête le parcours. Concrètement, Caml répond `true` à `0 = 0 || (1 / 0 = 0)`;;, alors que `(1 / 0 = 0)` produirait une erreur.

De même, on peut définir la fonction factorielle ainsi (peu importe si vous ne comprenez pas encore cette définition)

---

```
# let rec fact = function
  | 0 -> 1
  | n -> n * fact (n - 1);;
fact : int -> int = <fun>
```

---

ce qui permet d'effectuer quelques calculs :

```
# fact 10;;
- : int = 3628800
```

---

mais pas tous :

```
# fact 5345346;;
Uncaught exception: Out_of_memory
```

---

Pourtant, Caml évalue sans problème, grâce à sa « paresse », le booléen suivant :

```
# 1 = 0 && fact 5345346 = 123;;
- : bool = false
```

---

alors qu'il n'évaluerait pas `fact 5345346 = 123 && 1=0`.

Les connecteurs logiques ne sont donc pas vraiment commutatifs : on pensera à mettre en premier lieu ce qui se calcule vite.

#### 4.2. LE TYPE INT

C'est, comme nous l'avons dit, le type des entiers. Il faut cependant prendre garde, car Caml gère seulement les entiers compris entre  $-2^{30}$  et  $2^{30}-1$ , et se ramène à un tel entier par congruence modulo  $2^{31}$  (si vous préférez, il travaille dans  $\mathbb{Z}/2^{31}\mathbb{Z}$ ). On obtient donc des résultats étonnants si on ne fait pas attention :

```
# 10000 * 10000 * 10000;;
- : int = -727379968
```

---

On dispose des opérateurs binaires (à deux arguments) infixes `+` `-` `*` `/` `mod`, dont les définitions sont claires. Précisons simplement que `a / b` et `a mod b` désignent respectivement le quotient et le reste de la division euclidienne de `a` par `b` lorsque `a` et `b` sont des entiers naturels (`b` non nul), mais que `(-7) mod 5` par exemple vaut `-2`.

**Remarque :** on peut, et c'est un procédé général, donner une version *préfixe* de ces fonctions en utilisant la primitive **prefix** :

```
# let plus = prefix +;;
plus : int -> int -> int = <fun>
```

---

```
# plus 3 5;;
- : int = 8
```

---

On a également l'opérateur unaire (à un argument) `-`, de passage à l'opposé.

Caml intègre les opérateurs binaires préfixes `min` et `max` (`min a b` et `max a b` donnent respectivement le minimum et le maximum de `a` et `b`).

On dispose en outre d'opérateurs de comparaison `=` `<` `>` `<=` `>=` `<>` entre entiers, qui sont des prédicats (ils renvoient des booléens).

**Remarque :** nous verrons que ces opérateurs sont en fait *polymorphes*, et permettent donc d'autres comparaisons que celles entre entiers.

#### 4.3. LE TYPE FLOAT

Le type float est celui des nombres à virgule flottante (en bref : des flottants), qui sont des approximations des nombres réels.

**Remarque :** la virgule dont il est question s'écrit avec un point (`3.5`, `4.` sont acceptés, mais pas `.6`) :

---

```
# 2.6;;
- : float = 2.6
```

---

Les opérateurs sur ces nombres sont pour l’essentiel ceux définis pour les entiers `int`, suivis d’un point, *i.e.* `+`, `-`, `*`, `/`. On dispose aussi de l’exponentiation `**` (par exemple, `2. ** 10.;` renvoie `- : float = 1024.0`).

**Remarque :** pour de grands nombres, Caml passe en notation scientifique mantisse-exposant. Il répond par exemple à `3. ** 45.;` par `- : float = 2.95431270655e+21`.

On peut aussi utiliser les opérateurs de comparaison `=`, `<`, `<=`, `>`, `>=`, `<>`.

**Remarque :** en réalité, il n’est pas nécessaire de mettre de point après ces symboles lorsqu’on traite de `float`. Nous développerons cette remarque en 6.5.

Enfin, les fonctions usuelles `sqrt` `exp` `log` `sin` `cos` `tan` `acos` `asin` `atan`, dont il est inutile de donner la définition, sont des primitives Caml.

**Remarque :** retenez bien que `**` s’emploie pour les nombres à virgule flottante, et non les entiers.

#### 4.4. LE TYPE CHAR

C’est le type des caractères. On les écrit entre accents graves (pour ne pas les confondre avec un identificateur).

#### 4.5. CONVERSION DE TYPES ÉLÉMENTAIRES

Caml dispose de nombreuses fonctions de conversion de type, que l’on considère comme inélégantes. Leurs noms sont explicites : `int_of_float` , `float_of_int` , `string_of_float` , `float_of_string` , `string_of_int` , `int_of_string` , etc. Précisons que `type1_of_type2` convertit un objet de type `type2` en un objet de type `type1`.

#### 4.6. AFFICHAGE DE TYPES ÉLÉMENTAIRES

On utilise les procédures suivantes, aux noms transparents : `print_int` , `print_float` , `print_char` , `print_string` . À ce propos, on peut produire un retour à la ligne (comme effet de bord) grâce à `print_newline ()` , `print_char '\n'` ou `print_string "\n"` au choix.

### 5. DEUX EXEMPLES DE TYPES NON ÉLÉMENTAIRES

#### 5.1. LES $n$ -UPLETS

Un couple de première composante  $a$  et de seconde composante  $b$  s’écrit  $(a, b)$ . Son type est `type_de_a * type_de_b`.

**Remarque :** les objets  $a$  et  $b$  n’ont pas nécessairement le même type.

**Remarque :** on accède à la première (resp. seconde) composante par les fonctions `fst` et `snd`, mais il est facile de redéfinir ces fonctions.

**Remarque :** on peut dans certaines circonstances se passer des parenthèses.

De manière analogue, on définit plus généralement un  $n$ -uplet (où  $n$  est un entier supérieur ou égal à deux) avec la syntaxe évidente  $(a_1, \dots, a_n)$ .

---

```
# snd (3, (5, 7));;
- : int * int = 5, 7
```

---

#### 5.2. LES FONCTIONS

Nous allons voir à la section suivante comment définir des fonctions. Mentionnons simplement ici que le type `type1 -> type2` est celui des fonctions dont l’argument est de type `type1`, dont les valeurs sont de type `type2`.

### 6. PROGRAMMATION FONCTIONNELLE

Un programme Caml est souvent une suite de fonctions, que l’on applique à la fin. Il est donc très important de savoir définir et utiliser les fonctions en Caml. La plupart des fonctions que nous définirons seront récursives, mais on s’intéresse seulement dans cette section aux fonctions non récursives (on écrira simplement `let rec f` au lieu de `let f` pour une fonction récursive).

## 6.1. LES FONCTIONS : DES OBJETS COMME LES AUTRES

Les fonctions sont des objets Caml au même titre que les nombres à virgule flottante ou les listes d'entiers par exemple. Nous en déduisons les remarques suivantes :

**Remarque :** nous pouvons définir des fonctions prenant comme argument une fonction, ou ayant pour valeurs des fonctions. Une telle fonction est appelée *fonctionnelle*.

---

```
# let successeur x = x + 1;; (* définition d'une fonction simple *)
successeur : int -> int = <fun>
```

```
# let ajoute_deux_a_l_argument f = function x -> f (x + 2);;
(* définition d'une fonctionnelle *)
ajoute_deux_a_l_argument : (int -> 'a) -> int -> 'a = <fun>
```

```
# let ajoute_trois = ajoute_deux_a_l_argument successeur;;
(* application de la fonctionnelle *)
ajoute_trois : int -> int = <fun>
```

```
# ajoute_trois 5;;
- : int = 8
```

---

**Remarque :** il n'est pas nécessaire qu'une fonction ait un identificateur (vous n'imposez pas que l'entier 3 ait un identificateur pour travailler avec) :

---

```
# (function x -> x + 1)(3);; (* application d'une fonction sans la déclarer *)
- : int = 4
```

---

## 6.2. FONCTIONS D'UNE VARIABLE

Supposons que nous voulions définir une fonction  $f$  à un argument  $x$ . Caml accepte les syntaxes suivantes :

<pre>let f = function x -&gt; valeur_de_f_en_x  let f = fun x -&gt; valeur_de_f_en_x  let f x = valeur_de_f_en_x</pre>
--

où `valeur_de_f_en_x` est la formule donnant  $f(x)$ .

Pour définir par exemple la fonction `successeur`, de type `int -> int`, nous pouvons donc écrire

---

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

```
# let successeur = fun x -> x + 1;;
successeur : int -> int = <fun>
```

```
# let successeur x = x + 1;;
successeur : int -> int = <fun>
```

---

Pour appliquer la fonction  $f$  en  $x$ , on tape simplement `f(x)`, ou même `f x` dans certains cas (lorsqu'il n'y a pas d'ambiguïté) : par exemple, avec la fonction précédente, `successeur (5)`;;, `successeur 8`;; et `successeur (-3)`;; renvoient respectivement, 6, 9 et -2, mais Caml répond à `successeur -2`;; par le message d'erreur

---

```
Toplevel input:
>successeur -2;;
>^^^^^^^^^^
This expression has type int -> int,
but is used with type int.
```

---

Caml a constaté un problème en essayant de retrancher 2 à `successeur`, qui n'est pas un entier.

## 6.3. FONCTIONS DE PLUSIEURS VARIABLES

Nous avons donné une syntaxe pour une fonction d'un argument. Notons que cet argument peut très bien être un couple ou un triplet par exemple. Une loi de composition interne prend un unique argument, qui est un couple. Pour l'addition, on obtient ainsi :

---

```
# let addition (x, y) = x + y;; (* (x, y) est un couple d'entiers *)
addition : int * int -> int = <fun>
```

```
# addition (3, 5);;
- : int = 8
```

---

Cependant, les informaticiens préfèrent souvent voir l'addition comme une application qui à un entier  $x$  associe l'application qui à un entier  $y$  associe  $x+y$ . On obtient ainsi une fonction à deux arguments  $x$  et  $y$ . Caml autorise des syntaxes analogues à celles vues dans le cas d'une fonction d'une variable, à l'exception de la première (celle avec **function**) :

---

```
# let addition = fun x y -> x + y;;
addition : int -> int -> int = <fun>
```

```
# addition 3 6;;
- : int = 9
```

---

Nous aurions tout aussi bien pu écrire **let** addition  $x\ y = x + y$ ;;  
 En revanche, **let** addition = **function**  $x\ y -> x + y$ ;; renvoie le message d'erreur

---

```
Toplevel input:
>let addition = function x y -> x + y;;
>
The constructor x is unbound.
```

---

Tout cela est bien joli, mais quel est l'intérêt de considérer l'addition ainsi? Le principal avantage est que nous pouvons appliquer addition en un entier  $x$  fixé, obtenant ainsi l'application de translation (ou d'ajout) de  $x$ , *i.e.* la fonction  $y \mapsto x + y$  :

---

```
# let addition x y = x + y;;
addition : int -> int -> int = <fun>
```

```
# let ajout_de_quatre = addition 4;;
ajout_de_quatre : int -> int = <fun>
```

```
# ajout_de_quatre 9;;
- : int = 13
```

---

**Remarque** : on tente<sup>2</sup> le plus souvent de se débarrasser du parenthésage. Pour ce faire, on peut retenir que

$f\ x + g\ y$ équivaut à $f(x)+g(y)$ $f\ x\ y$ équivaut à $(f\ x)\ y$
--

On dit que l'application associée à gauche.

(addition 2) 9 donne 11, mais addition (2 9) produit un message d'erreur. En effet, dans ce second cas, on demande à Caml d'appliquer 2 à 9, ce qui n'a pas grand sens :

---

```
# let add x y = x + y;;
add : int -> int -> int = <fun>
```

```
# add (2 9);;
Toplevel input:
>add (2 9);;
>
This expression is not a function, it cannot be applied.
```

---

2. on ne vous impose pas toutefois de chasser toutes les parenthèses inutiles

6.4. CURRYFICATION

Caml permet de définir l'addition sous la forme `let addition x y = x + y;;`. Cette forme de l'addition est dite *curryfiée*. Formalisons un peu le procédé ayant permis de passer de l'addition usuelle à sa version curryfiée (ce procédé s'appelant *curryfication*).

Considérons trois ensembles  $A, B, C$ , et soit  $\mathcal{F}(A \times B, C)$  l'ensemble des applications de  $A \times B$  dans  $C$  (c'est aussi  $C^{A \times B}$ ). Cet ensemble est en bijection canonique avec  $\mathcal{F}(A, \mathcal{F}(B, C))$ , (*i.e.* avec  $(C^B)^A$ ), par :

$$\begin{aligned} \nabla : \mathcal{F}(A \times B, C) &\rightarrow \mathcal{F}(A, \mathcal{F}(B, C)) \\ f &\mapsto \left( \begin{array}{l} \nabla(f) : A \rightarrow \mathcal{F}(B, C) \\ a \mapsto \left( \begin{array}{l} \nabla(f)(a) : B \rightarrow C \\ b \mapsto f(a, b) \end{array} \right) \end{array} \right) \end{aligned}$$

6.5. LE POLYMORPHISME

En 4.2, nous remarquons que les opérateurs de comparaison sont polymorphes. Qu'entend-on vraiment par cela? Vous vous rappelez que tout objet Caml a un type, et que toute fonction est un objet : toute fonction a donc un type. Cependant, le type de certaines fonctions (ou d'autres objets) n'est pas toujours figé. Si on veut écrire la version préfixe de `<` par exemple, Caml répond :

```
# let inferieur_strict a b = a < b;;
inferieur_strict : 'a -> 'a -> bool = <fun>
```

Il signale ainsi que l'important est que `a` et `b` aient le même type, qu'il note `'a` (et qui est un *paramètre de type*), mais sans préciser ce type.

C'est pourquoi Caml répond sans broncher aux requêtes suivantes : `4 < 2;;`, `5.3 < 2.1;;`, “ `petit` ” `<` “ `grand` ”;; (et il répond `false` à chaque fois).

**Remarque :** le polymorphisme en Caml fonctionne par « tout ou rien » : si un objet n'a pas un type précis, il a potentiellement tous les types.

6.6. FILTRAGE DE MOTIFS

C'est l'un des traits les plus agréables de Caml. Le filtrage permet souvent des définitions très claires et simples de fonctions. Supposons que nous voulions définir une fonction `f`. Les syntaxes suivantes sont possibles

```
let f n = match n with
| n_0 -> valeur_de_f_en_n_0
| n_1 -> valeur_de_f_en_n_1
| ..
| _ -> valeur_de_f_dans_les_autres_cas
```

```
let f = fun (* ou fonction pour une fonction à un unique argument *)
| n_0 -> valeur_de_f_en_n_0
| n_1 -> valeur_de_f_en_n_1
| ..
| _ -> valeur_de_f_dans_les_autres_cas
```

**Remarque :** les valeurs filtrées (comme `n0`) sont bien des valeurs, ou des motifs, mais ils ne peuvent pas être des identificateurs déjà définis (tout identificateur dans un filtre est un nouvel identificateur).

**Remarque :** la dernière ligne règle le sort de tous les cas non encore traités. Elle n'est pas toujours utile, mais permet de s'assurer que le filtrage est exhaustif (tous les cas possibles sont étudiés).

**Remarque :** le filtrage s'effectue dans l'ordre de lecture, et peut très bien être redondant (c'est le filtrage antérieur qui l'emporte alors).

**Remarque :** Caml prévient si le filtrage n'est pas exhaustif, et il prévient également si un cas de filtrage est inutile.

**Remarque :** si plusieurs éléments de la source ont la même valeur, on peut les regrouper dans le filtrage avec `|`. Voir l'exemple de `est_voyelle_minuscule` page 12.

```
# let f = fun
| 0 y -> 1
| _ -> 0;;
f : int -> 'a -> int = <fun>
```

```
# f 5 2;;  
- : int = 0
```

---

On peut filtrer en changeant le type

---

```
# let f x y = match (x,y) with  
| (0, _) -> 1  
| (_, 0) -> 1  
| _ -> 0;;  
f : int -> int -> int = <fun>
```

---

```
# f 5 2;;  
- : int = 0
```

---

On peut filtrer selon une seule variable

---

```
# let ou_logique a b = match a with  
| true -> true  
| false -> b;;  
ou_logique : bool -> bool -> bool = <fun>
```

---

```
# ou_logique false true;;  
- : bool = true
```

---



## 7. INTRODUCTION À LA RÉCURSIVITÉ

Il ne s'agit pas ici d'étudier la récursivité du point de vue théorique, mais d'en comprendre le principe sur des exemples simples.

Une fonction est dite *récursive* si elle s'appelle dans sa propre définition. Cela n'empêche pas la fonction d'être parfaitement définie. En fait, ce procédé est souvent utilisé en mathématiques, lorsqu'on définit une fonction (par exemple une suite récurrente), une notation (par exemple  $a^n$  où  $a$  est un élément d'un anneau,  $n$  un entier naturel) ou un symbole (par exemple celui de somme) par récurrence.

En Caml, pour déclarer une fonction récursive  $f$ , on utilise la syntaxe :

```
let rec f = définition_réursive_de_f
```

On peut ainsi définir la fonction factorielle de la manière suivante (à la fois récursive et par filtrage) :

```
# let rec fact = function
  | 0 -> 1
  | n -> n * fact (n - 1);;
fact : int -> int = <fun>
```

```
# fact 5;;
- : int = 120
```

**Remarque :** si l'on omet le mot **rec**, Caml prévient que l'identificateur de la fonction n'est pas lié :

```
# let fact = function
  | 0 -> 1
  | n -> n * fact (n - 1);;
Toplevel input:
>      | n -> n * fact (n - 1);;
>      ^^^^
The value identifier fact is unbound.
```

**Remarque :** si on met le mot **rec** pour déclarer une fonction non récursive, Caml ne dit rien de particulier :

```
let rec carre = function n -> n * n;;
carre : int -> int = <fun>

carre 5;;
- : int = 25
```

Cependant, il serait bien sûr malvenu d'employer cette construction **let rec** pour définir une fonction non récursive ...

## 8. PROGRAMMATION IMPÉRATIVE

La plupart des programmes que nous écrirons en Caml seront récursifs, et nous exploiterons surtout les aspects fonctionnels de ce langage.

Cependant, la programmation impérative (ou itérative) est possible, et parfois souhaitable (dans beaucoup de problèmes sur les vecteurs par exemple).

## 8.1. LA CONDITIONNELLE

La conditionnelle est la classique construction « Si condition alors expression1 sinon expression2 ». Elle s'écrit

```
if condition then expression1 else expression2
```

**Remarque :** condition est de type booléen, expression1 et expression2 sont nécessairement de même type. N'oubliez pas de vérifier la cohérence du typage.

En fait, la partie else est facultative, et

```
if condition then expression1
```

se lit comme

---

```
if condition then expression1 else ()
```

---

Comme les expressions de la conditionnelle ont nécessairement le même type, cette syntaxe sans « else » n'est possible que si expression1 est de type unit.

---

```
# let vaut_un x = if x = 1 then true;;
Toplevel input:
>let vaut_un x = if x = 1 then true;;
>
This expression has type unit,
but is used with type bool.
```

---

Pour pallier cette incohérence de typage, on peut écrire (mais pas indifféremment) :

---

```
# let vaut_un_effet x = if x = 1 then print_string "true";;
vaut_un_effet : int -> unit = <fun>

# let vaut_un_predicat x = if x = 1 then true else false;;
vaut_un_predicat : int -> bool = <fun>

# vaut_un_effet 1;;
true- : unit = ()

# vaut_un_predicat 1;;
- : bool = true
```

---

**Remarque :** la conditionnelle n'est pas tant employée que cela, puisque le filtrage la remplace avantageusement le plus souvent. En particulier, il n'y a pas en Caml de « elif ».

## 8.2. SÉQUENCEMENT

Pour regrouper plusieurs expressions, on peut effectuer un *séquencement* : la syntaxe Caml est

<pre>begin     expression1;     expression2;     ...     expressionN; end</pre>
---

Le résultat de cette séquence est celui de la dernière expression. Un tel séquencement n'a de sens que si les expressions précédentes agissent par effet de bord (Caml produit un avertissement dans le cas contraire).

**Remarque :** ce séquencement est intéressant dans le cas d'une conditionnelle, car la conditionnelle risque de s'arrêter trop tôt sinon.

Comparer par exemple les phrases suivantes :

---

```
# if 0 = 1 then begin
    print_int 2;
    print_int 3;
end;;
- : unit = ()

# if 0 = 1 then
    print_int 2;
    print_int 3;
;;
3- : unit = ()
```

---

**Remarque :** on peut remplacer **begin** instructions **end** par (instructions).

## 8.3. BOUCLES

8.3.1. *Boucle inconditionnelle.* C'est la classique boucle `for`, dont la syntaxe est la suivante

```
for indice = valeur_initiale to valeur_finale do expression_pour_l_indice_i done
```

**Remarque :** on peut aussi décrémenter l'indice en mettant **downto** au lieu de **to**. L'indice, de type `int`, joue le rôle d'un identificateur local, et il n'apparaît pas forcément dans le corps de la boucle.

**Remarque :** la boucle `for` joue un peu le rôle d'un séquençement dont on aurait ordonné (et décrit) les instructions à l'aide d'un indice. En particulier, elles ne tolèrent pas les déclarations globales, et produisent un avertissement dans le cas où les expressions ne sont pas de type `unit`.

**Remarque :** la boucle `for` se terminant par **done**, il n'est pas utile d'utiliser le séquençement pour regrouper les éventuelles sous-instructions.

```
# for i = 1 to 5 do print_int i; print_newline () done;;
1
2
3
4
5
- : unit = ()
```

8.3.2. *Boucle conditionnelle.* C'est la non moins classique boucle `while`, dont voici la syntaxe :

```
while condition do expression done
```

**Remarque :** `condition` est un booléen, testé en entrée de boucle (contrairement à la boucle de type « repeat ... until ... », non implémentée en Caml).

Il est difficile de donner un exemple d'utilisation de boucle `while` sans utiliser la notion de référence, à laquelle nous consacrons la sous-section suivante.

```
# let i = ref 1 in
while !i < 10 do print_int !i; i := !i + 1 done;;
123456789- : unit = ()
```

## 8.4. RÉFÉRENCES

La notion de référence, similaire à celle de pointeur en C, sert à définir un contenant, que l'on donne avec son contenu initial. Il faut bien comprendre que le contenu de ce contenant peut être changé (mais que l'on doit conserver le même type).

La syntaxe est

```
let contenant = ref contenu
```

Pour accéder à `contenu` à partir de `contenant`, on écrit `!contenant`.

**Remarque :** `contenant` est de type `type_du_contenu ref`.

Pour modifier (ou réaffecter) le contenu de `contenant`, la syntaxe est

```
contenant := nouveau_contenu
```

**Remarque :** `nouveau_contenu` doit être du même type que `contenu`. Cette expression est de type `unit` (on agit par effet de bord).

```
# let y = ref 5;;
y : int ref = ref 5
```

```
# y := 7;;
- : unit = ()
```

```
# y := 5.6;;
Toplevel input:
>y := 5.6;;
>      ^^^
This expression has type float ,
```



---

```
# let rec je_sais = fonction
    | 0 -> "Je_sais";
    | n -> je_sais (n-1) ^ "_que_je_sais";;
je_sais : int -> string = <fun>

# je_sais 3;;
- : string =
  "Je_sais_que_je_sais_que_je_sais_que_je_sais"
```

---

## 9.2. LES VECTEURS OU TABLEAUX

Les vecteurs (ou tableaux) sont des emplacements consécutifs en mémoire, où l'on peut stocker des éléments de même type. Si un tableau stocke des objets de type `type_1`, son type est `type_1 vect`.

Un tableau s'écrit

`[| element_0; element_1; ... ; element_{N-1} |]`

**Remarque :** un tableau a une taille fixée, à laquelle on peut accéder grâce à la fonction `vect_length` (de type `'a vect -> int`).

On accède à l'élément d'indice  $i \in \llbracket 0, N-1 \rrbracket$  d'un tableau `T` de taille `N` par l'instruction `T.(i)`.

**Remarque :** pour les chaînes de caractères, on utilise les crochets, et on utilise les parenthèses pour les tableaux.

**Remarque :** faites attention aux indices (le premier indice est 0 et non 1).

**Remarque :** les tableaux étant de taille fixe, il est fréquent qu'on traite un problème les concernant par la programmation impérative.

Pour construire un tableau initial de taille `N` dont tous les termes valent `a`, on utilise la fonction `make_vect` en écrivant `make_vect N a` :

---

```
# make_vect 3 2.5;;
- : float vect = [|2.5; 2.5; 2.5|]
```

---

On peut aussi réaffecter l'élément d'indice `i` d'un tableau `T` en lui donnant la valeur `b` (de même type que la valeur actuelle) selon la syntaxe :

`T.(i) <- b`

La liaison entre vecteurs est la même que pour les chaînes de caractères :

---

```
# let T = [|3; 10; 3|];;
T : int vect = [|3; 10; 3|]

# let T' = T;;
T' : int vect = [|3; 10; 3|]

# T'.(1) <- 3;;
- : unit = ()

# T;;
- : int vect = [|3; 3; 3|]
```

---

## 9.3. LES LISTES

La définition d'une liste est récursive : la liste vide est une liste de type `'a list`, et si `L` est une liste dont les termes sont de type `type_1`, et `a` un objet de type `type_1`, alors on obtient une nouvelle liste, de même type, en plaçant `a` devant `L`, grâce au constructeur infixé « `::` ». On dit que `a` (resp. `L`) est la *tête* (resp. la *queue*, ou la *fin*) de la liste obtenue `a :: L`.

---

```
# let l = [];;
l : 'a list = []

# let l' = 3 :: l;;
l' : int list = [3]
```

```
# let l'' = 5 :: 8 :: l';;
l'' : int list = [5; 8; 3]
```

**Remarque :** à l'inverse des tableaux, il est facile d'ajouter un terme en tête d'une liste. En contrepartie, l'accès à un terme d'une liste ne s'effectue pas en temps constant (contrairement au cas des tableaux). C'est le terme en tête de liste que l'on obtient le plus rapidement, puis le suivant, etc.

**Remarque :** par nature, les listes privilégient une programmation récursive.

On accède à la taille (ou longueur) d'une liste par la fonction `list_length`, que l'on peut reprogrammer ainsi :

```
# let rec taille_liste = function
  | [] -> 0
  | a :: q -> 1 + taille_liste q;;
taille_liste : 'a list -> int = <fun>
```

```
# taille_liste [2; 5; 8; 9];;
- : int = 4
```

**Remarque :** il est inutile d'identifier la tête de la liste dans le second cas de filtrage, de sorte que `a :: q` peut être remplacé par `_ :: q`.

On accède à la tête (resp. la queue) d'une liste grâce à la fonction `hd` (resp. `tl`).

On dispose aussi d'un opérateur de concaténation entre listes (de même type), noté `@`.

Enfin, les fonctions `mem` et `index` de types respectifs (`'a -> 'a list -> bool`) et (`'a -> 'a list -> int`) teste l'appartenance (resp. donne le premier indice) d'un objet dans une liste.

#### 9.4. EXCEPTIONS

Il existe en Caml un type tout à fait particulier, celui des exceptions `exn`. Dans une première approche, on peut voir une exception comme une valeur que l'on renvoie lorsque l'expression n'est pas évaluable.

```
# 1 / 0;;
Uncaught exception: Division_by_zero
```

```
# index 3 [1; 1; 1];;
Uncaught exception: Not_found
```

Pour *déclencher* (ou *lever*) une exception, on utilise la primitive `raise`, qui, toutes affaires cessantes (cette primitive est prioritaire), renvoie l'exception détectée :

```
# 1 / 0 = 0 || 0 = 0;;
Uncaught exception: Division_by_zero
```

Il est possible de définir soi-même une exception, en l'introduisant selon l'une des syntaxes suivantes

```
exception ma_nouvelle_exception
```

```
exception ma_nouvelle_exception of type_de_cette_exception
```

Dans le premier cas, un déclenchement de l'exception produira `Uncaught exception: ma_nouvelle_exception`, alors que Caml répondra `Uncaught exception: ma_nouvelle_exception valeur_déclenchant_l_exception` dans l'autre cas :

```
# exception egalite;;
Exception egalite defined.
```

```
# let taux f x y = if x = y then raise egalite else (f y -. f x) /. (y -. x);;
taux : (float -> float) -> float -> float -> float = <fun>
```

```
# taux cos 0. 0.01;;
- : float = -0.00499995833347
```

```
# taux sin 3. 3.;;
Uncaught exception: egalite
```

```

# exception egalite of float;;
exception egalite defined.

# let taux' f x y = if x = y then raise (egalite' x) else (f y -. f x) /. (y -. x);;
taux' : (float -> float) -> float -> float -> float = <fun>

# taux' cos 0. 0.01;;
- : float = -0.00499995833347

# taux' sin 3. 3.;;
Uncaught exception: egalite' 3.0

```

---

Les deux exceptions les plus courantes sont `Failure` et `Invalid_argument`. On a donc implémenté des fonctions `failwith` et `invalid_arg`, de type `string -> 'a`, qui lèvent directement ces exceptions :

---

```

# let racines_de_ax_plus_b = fun
  | 0. 0. -> failwith "tout_reel_est_solution"
  | 0. _ -> raise (Failure "aucune_solution")
  | a b -> -. b /. a;;
racines_de_ax_plus_b : float -> float -> float = <fun>

# racines_de_ax_plus_b 0. 0.;;
Uncaught exception: Failure "tout_reel_est_solution"

# racines_de_ax_plus_b 0. 1.;;
Uncaught exception: Failure "aucune_solution"

# racines_de_ax_plus_b 2. 1.;;
- : float = -0.5

```

---

uncaught **exception**: une\_certaine\_exception signifie que l'exception une\_certaine\_exception n'a pas été *rat-trapée* : comment rattraper une exception, et qu'entend-on au juste par là ? On peut dire à Caml quelle attitude adopter dans le cas où une exception serait levée, selon la syntaxe

```

try expression with attitude_à_adopter_en_cas_d_exception

```

où `attitude_à_adopter_en_cas_d_exception` est un filtrage dont les filtres sont des exceptions, que Caml utilise si `expression` a déclenché une exception :

---

```

# let f x y = try x / y with Division_by_zero -> 42;;
f : int -> int -> int = <fun>

# f 3 2;;
- : int = 1

# f 4 0;;
- : int = 42

```

---

Ce filtrage sur les exceptions revêt un grand intérêt dans le cas d'exceptions paramétrées. Par exemple, on peut prolonger la fonction `taux` d'accroissement de la fonction sinus de la manière suivante :

---

```

# exception egalite of float;;
Exception egalite defined.

# let taux_de_sinus x y = try if x = y then
  raise (egalite x)
  else (sin y -. sin x) /. (y -. x)
  with egalite x -> cos x;;
taux_de_sinus : float -> float -> float = <fun>

# taux_de_sinus 0. 0.01;;

```

```
- : float = 0.999983333417
```

```
# taux_de_sinus 1. 1.;;
- : float = 0.540302305868
```

### 10. BREF RETOUR SUR LE FILTRAGE

Il ne vous a pas échappé, dans l'exemple précédent, que la fonction `taux_de_sinus` pouvait être écrite, de manière bien plus simple, de la manière suivante :

```
# let taux_de_sinus x y = if x = y then
                           cos x
                           else (sin y -. sin x) /. (y -. x);;
taux_de_sinus : float -> float -> float = <fun>
```

Le seul avantage que l'on pourrait concéder à la première programmation est la meilleure mise en valeur du fait que l'on soit tombé sur un cas « interdit ».

Il est naturel de vouloir remplacer dans la seconde écriture la conditionnelle par un filtrage :

```
# let taux_de_sinus = fun
  | x x -> cos x
  | x y -> (sin y -. sin x) /. (y -. x);;
Toplevel input:
> | x x -> cos x
> |
The variable x is bound several times in this pattern.
```

Pourquoi Caml n'a-t-il pas accepté ce filtrage ? Chaque identificateur ne peut apparaître qu'une seule fois dans un filtre. Les seuls tests d'égalité possibles dans un filtrage simple sont ceux avec une constante.

Pour effectuer malgré tout un test comme dans la fonction ci-dessus, on peut utiliser une *garde*, avec le mot-clé `when` :

```
# let taux_de_sinus = fun
  | x y when x = y -> cos x
  | x y -> (sin y -. sin x) /. (y -. x);;
taux_de_sinus : float -> float -> float = <fun>
```

Cherchons maintenant à écrire une fonction  $f$  de type `'a list -> 'a list`, qui, à une liste d'au moins deux termes, enlève les deux premiers termes s'ils sont distincts, ajoute le premier terme s'ils sont égaux, et laisse toute autre liste invariante.

Parmi les nombreuses programmations possibles, on propose :

```
# let f = function
  | a :: a' :: q when a = a' -> a :: a :: a' :: q
  | a :: a' :: q -> q
  | l -> l;;
f : 'a list -> 'a list = <fun>

# f [1; 1; 3];;
- : int list = [1; 1; 1; 3]

# f [1; 2; 3];;
- : int list = [3]
```



Le premier filtre est lisible, mais l'écriture du résultat obtenu ne rend pas justice à la simplicité de l'opération effectuée, à savoir ajouter le premier terme en tête de liste. Pour mieux mettre en valeur cette simplicité, et rendre le programme plus clair, on peut utiliser le mot-clé **as**, qui permet d'identifier le cas de filtrage. Ainsi peut-on réécrire la fonction  $f$  de la manière suivante :

---

```
# let f = function
  | a :: a' :: q as l when a = a' -> a :: l
  | a :: a' :: q -> q
  | l -> l;;
f : 'a list -> 'a list = <fun>
```

---

L'opération effectuée dans le premier cas de filtrage est maintenant beaucoup plus lisible.

## 11. TYPES DÉFINIS PAR L'UTILISATEUR

En Caml, on peut définir ses propres types.

## 11.1. TYPE CONSTANT, TYPE PARAMÉTRÉ

On peut définir un type en lui donnant un nom, en écrivant

```
type nom_du_type = valeur_constante_du_type
```

```
# type Isole = unique;;
Type Isole defined.

# let f = function unique -> 1;;
f : Isole -> int = <fun>

# let a = unique;;
a : Isole = unique

# f a;;
- : int = 1
```

**Remarque :** le type unit est un exemple de type constant (déjà défini).

On peut créer également un type paramétré, en écrivant

```
type nom_du_type of type_du_type
```

```
# type Param = p of int;;
Type Param defined.

# let g = function p m -> p (m * m);;
g : Param -> Param = <fun>

# let a = p 3;;
a : Param = p 3

# g a;;
- : Param = p 9
```

Notons que l'on peut définir des types polymorphes, en utilisant les habituelles variables de type comme 'a et 'b, en les déclarant comme ceci :

```
type 'a nom_du_type
```

```
# type 'a Polym = q of 'a;;
Type Polym defined.

# let x = q 3;;
x : int Polym = q 3

# let h (q a) = [q a];;
h : 'a Polym -> 'a Polym list = <fun>

# h (q 3);;
- : int Polym list = [q 3]
```

**Remarque :** on peut également effectuer une abréviation de type selon l'instruction :

```
type nom_de_l_abreviation = type_deja_defini
```

```
# type couple = int * float;;
Type couple defined.

# let phi = function (a : couple) -> fst a;;
phi : couple -> int = <fun>

# phi (4, 5.5);;
- : int = 4
```

**Remarque :** cet exemple vous montre également comment forcer le type d'un argument.

## 11.2. TYPE SOMME ET TYPE PRODUIT

Un type somme correspond à une union disjointe de types déjà connus. La syntaxe est

```
type nom_du_type = | type_1 | type_2 | ... | type_n
```

**Remarque :** la première barre verticale est facultative.

```
# type nombre = | Entier of int | Reel of float;;
Type nombre defined.

# let somme = fun
  | (Entier a) (Entier b) -> Entier (a + b)
  | (Reel a) (Entier b) -> Reel (a +. float_of_int b)
  | (Entier a) (Reel b) -> Reel (float_of_int a +. b)
  | (Reel a) (Reel b) -> Reel (a +. b);;
somme : nombre -> nombre -> nombre = <fun>
```

```
# somme (Entier 3) (Reel 5.7);;
- : nombre = Reel 8.7
```

Un type produit (ou enregistrement) correspond à un produit cartésien de types déjà connus. La syntaxe est

```
type nom_du_type = {Etiquette_1 : type_1; ... ; Etiquette_n : type_n}
```

La déclaration d'un objet ayant ce type s'écrit

```
let identificateur = {Etiquette_1 = valeur_de_type_1; ... ;
  Etiquette_n = valeur_de_type_n}
```

On peut préciser qu'un type enregistrement est *mutable* (on peut en modifier le contenu, à la manière d'un tableau), en insérant le mot clef **mutable** avant l'étiquette concernée :

Pour modifier le contenu d'une étiquette mutable, on écrira

```
nom_du_type.etiquette <- nouvelle_valeur
```

```
# type bulletin = {Maths : float; Physique : float; Informatique : float;
mutable Appreciation : string; LeReste : int };;
Type bulletin defined.

# let eleve = {Maths = 18.8; Physique = 19.2; Informatique = 16.7;
Appreciation = "nul"; LeReste = 14 };;
eleve : bulletin = {Maths = 18.8; Physique = 19.2; Informatique = 16.7;
Appreciation = "nul"; LeReste = 14}

# eleve.LeReste;;
- : int = 14

# eleve.Appreciation <- "convenable";;
- : unit = ()
```

```
# eleve;;
- : bulletin =
  {Maths = 18.8; Physique = 19.2; Informatique = 16.7;
  Appreciation = "convenable"; LeReste = 14}
```

---

## 11.3. TYPE RÉCURSIF

On peut définir un type de manière récursive, exactement comme on le ferait pour un ensemble inductif :

```
# type entier = | Zero | succ of entier;;
Type entier defined.

# let rec addition a b = match a with
  | Zero -> b
  | succ c -> succ (addition c b);;
addition : entier -> entier -> entier = <fun>

# addition (succ (succ Zero)) (succ (succ Zero));;
- : entier = succ (succ (succ (succ Zero)))

# type liste = | Liste_Vide | Cons of int * liste;;
Type liste defined.

# let queue = function
  | Liste_Vide -> failwith "Pas_de_queue_pour_la_liste_vide"
  | Cons (a, li) -> li;;
queue : liste -> liste = <fun>

# queue (Cons(3, Cons(5, Cons(7, Liste_Vide))));;
- : liste = Cons (5, Cons (7, Liste_Vide))

# type arbre_binaire = | Feuille of int | Noeud of int * arbre_binaire * arbre_binaire;;
Type arbre_binaire defined.

# let bonsai = Noeud (1, Noeud (2, Feuille 3, Feuille 4), Feuille 5);;
bonsai : arbre_binaire =
  Noeud (1, Noeud (2, Feuille 3, Feuille 4), Feuille 5)
```

---

**Remarque :** pour de tels types, la programmation récursive s'impose naturellement, et la structure des programmes découle simplement de la structure inductive (*i.e.* des cas de base et des constructeurs).

**Remarque :** on peut également définir des types mutuellement récursifs, de la même manière que l'on définit des fonctions récursives croisées.

Deuxième partie

Méthodes de programmation



## Programmation impérative

### 1. CORRECTION ET TERMINAISON

Étant donné un algorithme, il est important de vérifier sa *correction* (ou sa validité), *i.e.* qu'il renvoie bien le résultat souhaité, et sa *terminaison*, *i.e.* qu'il ne boucle pas indéfiniment (pour certains algorithmes récursifs ou pour des boucles conditionnelles).

**Remarque :** on peut également se poser la question de son efficacité (*i.e.* de sa rapidité et de l'espace qu'il prend), ce que nous ferons lors du cours sur la complexité.

En réalité, on ne peut pas prouver, en toute généralité, la terminaison d'un algorithme donné. Penser à la suite de Syracuse. Pire, on peut prouver qu'on ne peut pas le prouver.

Nous ne rentrerons pas dans les détails de la preuve d'un programme impératif : c'est un sujet délicat, qui peut vite devenir très compliqué. Donnons le principe général de preuve d'une boucle.

### 2. NOTION D'INVARIANT DE BOUCLE

Une boucle, qu'elle soit conditionnelle ou inconditionnelle (on dit aussi indexée), peut se schématiser de la manière suivante :

On part d'une précondition (ou condition d'entrée) dans le module, qui décrit l'état des variables avant d'entrer dans la boucle.

On effectue un certain nombre de passages dans la boucle, non nécessairement connu à l'avance.

En sortie de boucle, la condition de sortie (ou postcondition), qui doit correspondre au résultat souhaité.

On appelle *invariant de boucle* un prédicat sur les arguments de la boucle, renvoyant vrai avant passage dans la boucle, restant vrai après passage dans la boucle : en sortie de boucle, l'invariant de boucle est vrai, ce qui doit nous permettre de prouver qu'on a obtenu le résultat souhaité.

**Remarque :** il s'agit donc de montrer un résultat par récurrence finie.

Prenons l'exemple de la fonction factorielle, de manière impérative avec boucle indexée :

```
# let fact n =
  let temp = ref 1 in
    for i = 1 to n do temp := !temp * i done;
    !temp;;
fact : int -> int = <fun>
```

```
# fact 5;;
- : int = 120
```

Pourquoi cette fonction renvoie-t-elle bien la bonne valeur ? La précondition est : « temp a pour valeur 1 ». Montrons que pour tout  $k \in \llbracket 0, n \rrbracket$ , la valeur de temp après la  $k$ -ème itération est  $k!$ .

C'est vrai pour  $k = 0$  (avant l'entrée dans la boucle), et, supposant la propriété vraie à un rang  $k \in \llbracket 0, n-1 \rrbracket$ , alors après un nouveau passage dans la boucle (pour  $i = k + 1$ ), la valeur de temp est  $(k + 1)!$ .

En sortie de boucle, *i.e.* après la  $n$ -ième itération, on obtient  $n!$  comme valeur pour temp.

En fait, il n'est pas nécessaire de tant détailler la correction de cette boucle (une preuve n'est pas systématique). On se contentera de mettre en commentaire l'invariant de boucle :

```
# let fact n =
  let temp = ref 1 in
    for i = 1 to n do temp := !temp * i done;
    (* invariant : !temp = i! *)
    !temp;;
fact : int -> int = <fun>
```

**Remarque :** bien sûr, cette fonction produit un résultat étrange si on prend un argument négatif (on aurait pu lever une exception).

**Remarque :** que se passe-t-il si on omet la dernière ligne ? Et si on la remplace par `temp` ?

**Remarque :** en réalité, il est souvent plus pratique de déterminer le corps de la boucle, d'en trouver un invariant, et d'en déduire la précondition.

**Remarque :** l'invariant permet de faire le pont entre la précondition et la postcondition.

Pour donner un exemple avec une boucle conditionnelle, reprenons la fonction factorielle :

---

```
# let factw n =
    let i = ref n in
      let temp = ref 1 in
        while !i > 1 do temp := !temp * !i; i := !i - 1 done;
        (* invariant : (!i)! * !temp = n! *)
        !temp;;
fact : int -> int = <fun>

# factw 5;;
- : int = 120
```

---

La fonction termine (même si  $n < 1$ ) car `!i` est décrémenté à chaque passage dans la boucle, et ne peut donc rester indéfiniment strictement supérieur à 1. En sortie, `!i` vaut 1 : grâce à l'invariant de boucle, on constate que `!temp` vaut bien la valeur souhaitée.

Fonction puissance de manière naïve :

---

```
# let puissance a b = let temp = ref 1 in
    for i = 1 to b do temp := !temp * a done;
    (* invariant : !temp = a ^ i *)
    !temp;;
puissance : int -> int -> int = <fun>

# puissance 3 4;;
- : int = 81
```

---

**Remarque :** on aurait également pu introduire l'itératrice  $f : x \mapsto ax$ , que l'on effectue à chaque passage dans la boucle.

Fonction somme

---

```
# let somme t = let temp = ref 0 in
    for i = 0 to vect_length t - 1 do temp := !temp + t.(i) done;
    (* invariant : !temp = somme des termes de t d'indices 0 à i *)
    !temp;;
somme : int vect -> int = <fun>

# somme [|3; 5; 11|];;
- : int = 19
```

---



---

```
# let fact n =
  let temp = ref 1 in
    for i = 1 to n do temp := !temp * i done;
  !temp;;
fact : int -> int = <fun>

# let factw n =
  let i = ref n in
    let temp = ref 1 in
      while !i > 1 do temp := !temp * !i; i := !i - 1 done;
    !temp;;
fact : int -> int = <fun>

# factw 5;;
- : int = 120

# let puissance a b = let temp = ref 1 in
  for i = 1 to b do temp := !temp * a done;
  !temp;;
puissance : int -> int -> int = <fun>

# puissance 3 4;;
- : int = 81

# let somme t = let temp = ref 0 in
  for i = 0 to vect_length t - 1 do temp := !temp + t.(i) done;
  !temp;;
somme : int vect -> int = <fun>

# somme [|3; 5; 11|];;
- : int = 19
```

---



## Récurtivité

### 1. INTRODUCTION

Informellement, une fonction est dite *réursive* lorsqu'elle s'appelle elle-même dans sa définition. Le problème de terminaison, rencontré pour les boucles conditionnelles, se rencontre évidemment dans ce cadre. Nous ne pourrions d'ailleurs pas déterminer si une fonction réursive quelconque termine (voir l'exemple de la suite de Syracuse plus loin). En réalité, il n'est même pas envisageable de pouvoir répondre à cette question un jour

Pour préciser à Caml que l'on définit une fonction réursive, on utilise la déclaration :

---

```
let rec fonction_reursive =
```

---

Pour prouver la terminaison et la correction d'une fonction réursive, nous allons introduire un cadre mathématique adapté, qui s'appuie sur la notion de relation d'ordre.

Pour comprendre le fonctionnement ou déboguer les fonctions réursives, on peut utiliser la fonction trace de type string  $\rightarrow$  unit.

### 2. COMPLÉMENTS SUR LES RELATIONS D'ORDRE

Dans la suite, nous considérerons un ensemble  $E$  muni d'un ordre total ou partiel  $\preceq$ , dont nous noterons  $\prec$  l'ordre strict associé.

La réursivité est liée à la notion de preuve par récurrence. On a vu en mathématiques des exemples de preuves par récurrence simple, cette méthode permettant de montrer la terminaison et la correction des fonctions suivantes :

---

```
# let rec fact = function
  | 0 -> 1
  | n -> n * fact (n - 1);;
fact : int -> int = <fun>
```

```
# trace "fact";;
The function fact is now traced.
- : unit = ()
```

```
# fact 5;;
fact <- 5
fact <- 4
fact <- 3
fact <- 2
fact <- 1
fact <- 0
fact -> 1
fact -> 1
fact -> 2
fact -> 6
fact -> 24
fact -> 120
- : int = 120
```

---

```
# let rec f = function
  | 0 -> 5.
  | n -> (f (n - 1) +. 5. /. f (n - 1)) /. 2.;;
f : int -> float = <fun>
```

```
# f 4;;
- : float = 2.23606889564

# sqrt 5.;;
- : float = 2.2360679775
```

La preuve de ces fonctions utilise de manière naturelle les notions de prédécesseur d'un entier naturel non nul et de successeur d'un entier naturel.

On a également vu la notion de preuve par récurrence forte (ou avec prédécesseurs), qui permet de prouver des fonctions récursives avec plusieurs appels :

```
# let rec fibo = function
  | 0 | 1 -> 1
  | n -> fibo (n - 1) + fibo (n - 2);;
fibo : int -> int = <fun>

# fibo 5;;
- : int = 8
```

En réalité, la démonstration par récurrence simple est difficile à généraliser, puisque fondée sur ces notions de prédécesseur et de successeur. Par exemple, pour l'ordre lexicographique sur  $\mathbb{N}^2$ , pourtant total, certains éléments non nuls n'admettent pas de prédécesseur.

De plus, il peut être utile de travailler sur un ensemble partiellement ordonné, tel que  $\mathbb{N} \setminus \{0, 1\}$  pour la divisibilité (dans la recherche de la décomposition d'un entier en produit de facteurs premiers par exemple).

C'est donc la récurrence forte que nous allons généraliser.

#### Définition (Élément minimal)

Un élément  $m$  d'une partie  $\Omega$  de  $E$  est dit *minimal (dans  $\Omega$ )* s'il n'en existe pas de strictement plus petit, *i.e.*

$$\forall x \in \Omega, (x \preceq m) \Rightarrow (x = m)$$

2.a

**Remarque :** ne pas confondre élément minimal et élément minimum (sauf si l'ordre est total).

**Remarque :** on peut bien sûr définir la notion d'élément maximal, mais elle ne nous sera pas très utile ici.

#### Exemple (Éléments minimaux)

- (1) Le seul élément minimal de  $\mathbb{N}$  usuel est 0.
- (2) Dans  $\mathbb{Z}$  usuel, il n'y a pas d'élément minimal (il n'y a pas nécessairement existence d'un élément minimal).
- (3) Les éléments minimaux de  $\mathbb{N} \setminus \{0, 1\}$  pour la divisibilité sont les nombres premiers. On observe en particulier qu'il n'y a pas nécessairement unicité d'un élément minimal.

i

#### Définition (Ensemble bien fondé)

On dit que l'ensemble ordonné  $(E, \preceq)$  est *bien fondé* (et que l'ordre  $\preceq$  est bien fondé) si toute partie non vide de  $E$  admet un élément minimal.

2.b

**Remarque :** bien sûr, on ne demande pas l'existence d'un élément minimal pour  $E$  seulement.

Exemple (Ensemble bien fondé)

- (1)  $\mathbb{N}$  usuel est bien fondé. Mieux, toute partie non vide admet un plus petit élément (on le dit *bien ordonné*). En fait  $(E, \preceq)$  est bien ordonné si et seulement si  $(E, \preceq)$  est bien fondé et totalement ordonné.
- (2)  $\mathbb{Z}$  n'est pas bien fondé.
- (3)  $\mathbb{N}$  pour la divisibilité est bien fondé.
- (4) Toute partie d'un ensemble bien fondé  $(E, \preceq)$  est bien fondée pour l'ordre induit.
- (5) L'ordre lexicographique est bien fondé (et bien ordonné).

ii

Théorème (Caractérisation des ensembles bien fondés)

$(E, \preceq)$  est bien fondé si et seulement si il n'existe pas de suite infinie strictement décroissante d'éléments de  $E$ .

2.a

Démonstration

S'il existe une suite infinie strictement décroissante  $(x_n)$  d'éléments de  $E$ , alors  $\{x_n, n \in \mathbb{N}\}$  est une partie de  $E$  n'admettant pas d'élément minimal. S'il n'existe pas de telle suite, alors soit  $\Omega$  partie non vide de  $E$  : soit  $x_0 \in \Omega$ . Supposons pour  $n \in \mathbb{N}$  fixé, avoir construit  $x_0, \dots, x_n$ . Si  $x_n$  n'est pas minimal dans  $\Omega$ , on prend  $x_{n+1} \in \Omega$  tel que  $x_{n+1} \prec x_n$ . La suite ainsi construite est nécessairement finie, s'arrêtant à un indice  $N$ , et  $x_N$  est minimal dans  $\Omega$ .

□

**Remarque :** si  $E$  est fini, alors il est bien fondé. Plus généralement, si, pour tout  $e \in E$ ,  $E$  n'a qu'un nombre fini d'éléments inférieurs à  $e$ , alors  $E$  est bien fondé.

Théorème d'induction

On suppose  $(E, \preceq)$  bien fondé. Soit  $\mathcal{M}$  l'ensemble des éléments minimaux de  $E$ . Soit  $p$  un prédicat sur  $E$  tel que :

- (1)  $\forall x \in \mathcal{M}, p(x)$ .
- (2)  $\forall x \in E, (\forall y \prec x, p(y)) \Rightarrow p(x)$ .

$p$  est alors vérifié sur tout  $E$ .

2.b

Démonstration

Soit  $\Omega$  l'ensemble des éléments de  $E$  où  $p$  est faux : supposons  $\Omega$  non vide.  $\Omega$  admet un élément minimal  $x$ .  $x$  n'est pas minimal dans  $E$  (car  $\Omega \cap \mathcal{M} = \emptyset$ ). De plus, pour tout élément  $y$  de  $E$  tel que  $y \prec x$ , on a  $p(y)$  : d'après (2),  $p(x)$  est vérifié, c'est absurde.

□

**Remarque :** en toute logique, la seconde condition entraîne la première, qui est donc superflue. Cependant, on préfère cette version plus lisible.

**Remarque :** on pouvait aussi raisonner en termes de parties de  $E$  :  $E$  est la seule partie de  $E$  contenant  $\mathcal{M}$  et qui comprend tout élément dont elle comprend les éléments strictement inférieurs dans  $E$ .

**Remarque :** on parle d'hypothèse d'induction.

## 3. TERMINAISON ET CORRECTION DES FONCTIONS RÉCURSIVES

On considère une fonction récursive  $f$  de  $A$  dans  $B$ . On suppose disposer d'une fonction  $\Phi$  de  $A$  dans un ensemble bien fondé  $E$ , que l'on appelle parfois *graduation*.

## Définition (Cas de base)

On appelle *cas de base* de  $f$  tout élément  $a$  de  $A$  tel que  $\Phi(a)$  soit minimal dans  $\Phi(A)$ .

3.a

Notons  $\mathcal{B}$  l'ensemble des cas de base, et  $\mathcal{M} = \Phi(\mathcal{B})$ .

**Remarque :** on n'impose pas  $\Phi(a)$  minimal dans  $E$  mais dans  $\Phi(A)$ .

## Théorème (Terminaison d'une fonction récursive)

On suppose que  $f$  termine pour les cas de base, et que pour tout  $x \in A$ , l'exécution de  $f(x)$  ne produit <sup>a</sup> qu'un nombre fini d'appels à  $y_1, \dots, y_k \in A$  vérifiant  $\Phi(y_j) \prec \Phi(x)$  pour tout  $j \in \llbracket 1, k \rrbracket$ .

La fonction  $f$  termine alors pour toute valeur  $x \in A$  de son argument.

3.a

a. outre un nombre fini d'opérations élémentaires

## Démonstration

Notons  $p$  le prédicat sur  $\Phi(A)$  :  $p(e)$  (est vrai) si et seulement si  $f$  termine pour tout élément de  $\Phi^{-1}(\{e\})$ . D'après le théorème d'induction,  $p$  est vérifié sur  $\Phi(A)$  : le théorème est prouvé.

□

De la même manière, on montre :

## Théorème (Correction d'une fonction récursive)

Soit  $p_f$  le prédicat défini, pour tout  $x \in A$ , par : «  $f(x)$  retourne (en temps fini) le résultat souhaité ».

On suppose :

$$(1) \forall x \in \mathcal{B}, p_f(x)$$

$$(2) \text{ pour tout } x \in A, \text{ l'exécution de } f(x) \text{ ne produit qu'un nombre fini d'appels, à } y_1, \dots, y_k \in A \text{ vérifiant } \Phi(y_j) \prec \Phi(x) \text{ pour tout } j \in \llbracket 1, k \rrbracket, \text{ et :}$$

$$(\forall j \in \llbracket 1, k \rrbracket, p_f(y_j)) \Rightarrow p_f(x)$$

La fonction  $f$  est alors correcte (et termine) pour toute valeur  $x \in A$  de son argument.

3.b

**Remarque :** en pratique, il peut être avantageux de chercher d'abord un ordre bien fondé adapté à notre problème, puis de programmer, que l'inverse.

D'ailleurs, la difficulté réside parfois dans le choix de  $E$  et de la graduation  $\Phi$ .

```

# let rec fact = function
    | 0 -> 1
    | n -> n * fact (n - 1);;
fact : int -> int = <fun>

# trace "fact";;
The function fact is now traced.
- : unit = ()

# fact 5;;
fact <- 5
fact <- 4
fact <- 3
fact <- 2
fact <- 1
fact <- 0
fact -> 1
fact -> 1
fact -> 2
fact -> 6
fact -> 24
fact -> 120
- : int = 120

# let rec f = function
    | 0 -> 5.
    | n -> (f (n - 1) +. 5. /. f (n - 1)) /. 2.;;
f : int -> float = <fun>

# f 4;;
- : float = 2.23606889564

# sqrt 5.;;
- : float = 2.2360679775

# let rec fibo = function
    | 0 | 1 -> 1
    | n -> fibo (n - 1) + fibo (n - 2);;
fibo : int -> int = <fun>

# fibo 5;;
- : int = 8

```

---

---

```
# let rec syracuse = function
  | 1 -> print_string "true"; print_newline ()
  | n when n mod 2 = 0 -> syracuse (n / 2)
  | n -> syracuse (3 * n + 1);;
syracuse : int -> unit = <fun>
```

```
# for i = 100 to 110 do syracuse i done;;
true
true
true
true
true
true
true
true
true
true
true
- : unit = ()
```

```
# trace "syracuse";;
The function syracuse is now traced.
- : unit = ()
```

```
# syracuse 10;;
syracuse <- 10
syracuse <- 5
syracuse <- 16
syracuse <- 8
syracuse <- 4
syracuse <- 2
syracuse <- 1
true
syracuse -> ()
syracuse -> ()
syracuse -> ()
syracuse -> ()
syracuse -> ()
syracuse -> ()
syracuse -> ()
- : unit = ()
```

```
# syracuse 36;;
syracuse <- 36
syracuse <- 18
syracuse <- 9
syracuse <- 28
syracuse <- 14
syracuse <- 7
syracuse <- 22
syracuse <- 11
syracuse <- 34
syracuse <- 17
syracuse <- 52
syracuse <- 26
syracuse <- 13
syracuse <- 40
syracuse <- 20
```





## 4. RÉCURSIVITÉ CROISÉE

Il arrive que l'on définisse deux ou plusieurs fonctions de manière récursive, qui s'appellent mutuellement : on parle de récursivité *croisée*. La syntaxe Caml est

```
let rec fonction 1 = définition de fonction1
    and
fonction2 = définition de fonction2
```

Par exemple, on peut définir les fonctions pair et impair de type `int -> bool` de la manière suivante :

---

```
# let rec pair = function | 0 -> true | n -> impair (n - 1)
    and
impair = function | 0 -> false | n -> pair (n - 1);;
pair : int -> bool = <fun>
impair : int -> bool = <fun>

# trace "pair";;
The function pair is now traced.
- : unit = ()

# trace "impair";;
The function impair is now traced.
- : unit = ()

# pair 5;;
pair <- 5
impair <- 4
pair <- 3
impair <- 2
pair <- 1
impair <- 0
impair -> false
pair -> false
impair -> false
pair -> false
impair -> false
pair -> false
- : bool = false
```

---

## 5. GESTION D'UNE FONCTION RÉCURSIVE

## 5.1. NOTION DE PILE D'EXÉCUTION

Pour gérer les appels récursifs, Caml effectue un empilement des valeurs à calculer (on parle de pile d'exécution), qu'il finit par dépiler pour arriver au calcul final. Par exemple, pour la fonction factorielle, nous obtenons le comportement suivant :

---

```
# let rec fact = function
    | 0 -> 1
    | n -> n * fact (n - 1);;
fact : int -> int = <fun>

# trace "fact";;
The function fact is now traced.
- : unit = ()

# fact 12;;
fact <- 12
fact <- 11
fact <- 10
```

```

fact <- 9
fact <- 8
fact <- 7
fact <- 6
fact <- 5
fact <- 4
fact <- 3
fact <- 2
fact <- 1
fact <- 0
fact -> 1
fact -> 1
fact -> 2
fact -> 6
fact -> 24
fact -> 120
fact -> 720
fact -> 5040
fact -> 40320
fact -> 362880
fact -> 3628800
fact -> 39916800
fact -> 479001600
- : int = 479001600

```

---

En cas de plusieurs appels récursifs à chaque passage, cela se complique, comme le montre l'exemple de la suite de Fibonacci :

```

# let rec fibo = function
  | 0 | 1 -> 1
  | n -> fibo (n - 1) + fibo (n - 2);;
fibo : int -> int = <fun>

```

```

# trace "fibo";;
The function fibo is now traced.
- : unit = ()

```

```

# fibo 4;;
fibo <- 4
fibo <- 2
fibo <- 0
fibo -> 1
fibo <- 1
fibo -> 1
fibo -> 2
fibo <- 3
fibo <- 1
fibo -> 1
fibo <- 2
fibo <- 0
fibo -> 1
fibo <- 1
fibo -> 1
fibo -> 2
fibo -> 3
fibo -> 5
- : int = 5

```

---

Pour mieux comprendre le calcul de `fibo 4`, on peut dessiner l'*arbre des appels* de `fibo 4`.

La fonction trace est donc très utile pour comprendre le fonctionnement d'une fonction récursive. Cependant, il faut savoir qu'elle ne gère pas (ou mal en tout cas) les fonctions polymorphes, et donc beaucoup de fonctions sur les listes chaînées :

---

```
# let rec pasderepet = function
  | [] -> []
  | [a] -> [a]
  | a :: q -> if a = hd q then pasderepet q else a :: pasderepet q;;
pasderepet : 'a list -> 'a list = <fun>

# trace "pasderepet";;
The function pasderepet is now traced.
- : unit = ()

# pasderepet [1; 3; 3; 3; 5; 5; 10; 10; 10];;
pasderepet <- [<poly>; <poly>; <poly>; <poly>; <poly>; <poly>; <poly>;
             <poly>; <poly>]
pasderepet <- [<poly>; <poly>; <poly>; <poly>; <poly>; <poly>; <poly>;
             <poly>]
pasderepet <- [<poly>; <poly>; <poly>; <poly>; <poly>; <poly>; <poly>]
pasderepet <- [<poly>; <poly>; <poly>; <poly>; <poly>; <poly>]
pasderepet <- [<poly>; <poly>; <poly>; <poly>; <poly>]
pasderepet <- [<poly>; <poly>; <poly>]
pasderepet <- [<poly>; <poly>]
pasderepet <- [<poly>]
pasderepet -> [<poly>]
pasderepet -> [<poly>]
pasderepet -> [<poly>]
pasderepet -> [<poly>; <poly>]
pasderepet -> [<poly>; <poly>]
pasderepet -> [<poly>; <poly>; <poly>]
pasderepet -> [<poly>; <poly>; <poly>]
pasderepet -> [<poly>; <poly>; <poly>]
pasderepet -> [<poly>; <poly>; <poly>]
pasderepet -> [<poly>; <poly>; <poly>; <poly>]
- : int list = [1; 3; 5; 10]
```

---

Pour remédier à ce problème, on peut forcer le typage de la fonction pasderepet, en précisant le type de son argument

---

```
# let rec pasderepet (l : int list) = match l with
  | [] -> []
  | [a] -> [a]
  | a :: q -> if a = hd q then pasderepet q else a :: pasderepet q;;
pasderepet : int list -> int list = <fun>

# trace "pasderepet";;
The function pasderepet is now traced.
- : unit = ()

# pasderepet [1; 3; 3; 3; 5; 5; 10; 10; 10];;
pasderepet <- [1; 3; 3; 3; 5; 5; 10; 10; 10]
pasderepet <- [3; 3; 3; 5; 5; 10; 10; 10]
pasderepet <- [3; 3; 5; 5; 10; 10; 10]
pasderepet <- [3; 5; 5; 10; 10; 10]
pasderepet <- [5; 5; 10; 10; 10]
pasderepet <- [5; 10; 10; 10]
pasderepet <- [10; 10; 10]
pasderepet <- [10; 10]
pasderepet <- [10]
```

```

pasderepet → [10]
pasderepet → [10]
pasderepet → [10]
pasderepet → [5; 10]
pasderepet → [5; 10]
pasderepet → [3; 5; 10]
pasderepet → [3; 5; 10]
pasderepet → [3; 5; 10]
pasderepet → [1; 3; 5; 10]
- : int list = [1; 3; 5; 10]

```

**Remarque :** il est intéressant de retenir cette manière de forcer le typage. On peut d'ailleurs observer que l'on peut aussi forcer le typage à l'arrivée :

```

# let eval x f = (f x : int);;
eval : 'a -> ('a -> int) -> int = <fun>

```

## 5.2. RÉCURSIVITÉ TERMINALE

Parmi les fonctions récursives, il existe une sous-classe, particulièrement facile à gérer : celle des fonctions à récursivité dite *terminale*. Une fonction est dite récursive terminale si, dans le corps de sa définition, *l'unique appel* récursif se fait en toute *fin de parcours*.

**Remarque :** en particulier, on n'effectue aucune opération sur l'appel récursif (par exemple, la fonction `fact` n'est pas récursive terminale).

Dans le cas d'une fonction récursive terminale, les calculs ou effets de bord précédant cet unique appel récursif deviennent jetables, et il n'y a plus pour Caml qu'à calculer la valeur pour cet appel. Il n'y a donc pas d'empilement, et la complexité spatiale d'une telle fonction est réduite (comme dans le cas d'une programmation impérative). On ne risque pas dans ce cas de dépasser la taille limite de la pile d'exécution (« stack overflow »).

```

# let rec term = function
  | 0 -> print_int 0
  | n -> print_int n; term (n - 1);;
term : int -> unit = <fun>

```

```

# term 10;;
109876543210- : unit = ()

```

```

# trace "term";;
The function term is now traced.
- : unit = ()

```

```

# term 5;;
term ← 5
5term ← 4
4term ← 3
3term ← 2
2term ← 1
1term ← 0
0term → ()
term → ()
term → ()
term → ()
term → ()
term → ()
- : unit = ()

```

En revanche, la fonction suivante n'est pas récursive terminale :

```

# let rec nonterm = function
  | 0 -> print_int 0
  | n -> nonterm (n - 1); print_int n;;

```

```

nonterm : int -> unit = <fun>

# trace "nonterm";;
The function nonterm is now traced.
- : unit = ()

# nonterm 5;;
nonterm <- 5
nonterm <- 4
nonterm <- 3
nonterm <- 2
nonterm <- 1
nonterm <- 0
0nonterm -> ()
1nonterm -> ()
2nonterm -> ()
3nonterm -> ()
4nonterm -> ()
5nonterm -> ()
- : unit = ()

```

---

Bien sûr, les fonctions fact et fibo définies ci-dessus ne sont pas récursives terminales.

### 5.3. DE LA RÉCURSIVITÉ NON TERMINALE À LA RÉCURSIVITÉ TERMINALE

Il y a toute légitimité à vouloir privilégier les fonctions récursives terminales, qui constituent souvent le juste milieu entre la clarté des fonctions récursives, et la rapidité supposée de la programmation impérative. Une idée générale pour tenter de programmer de manière récursive terminale est d'utiliser un *accumulateur*, qui joue le rôle de « mémoire » de notre programme.

Par exemple, pour la fonction factorielle, nous pouvons proposer :

---

```

# let rec fact_term = fun
    | (1, x) -> x
    | (n, x) -> fact_term (n - 1, x * n);;
fact_term : int * int -> int = <fun>

# trace "fact_term";;
The function fact_term is now traced.
- : unit = ()

# fact_term (5, 1);;
fact_term <- 5, 1
fact_term <- 4, 5
fact_term <- 3, 20
fact_term <- 2, 60
fact_term <- 1, 120
fact_term -> 120
fact_term -> 120
fact_term -> 120
fact_term -> 120
fact_term -> 120
- : int = 120

```

---

**Remarque :** il ne faut pas être impressionné par les nombreux renvois, qui sont en réalité tous identiques.

**Remarque :** pour retrouver une fonction factorielle de type `int -> int`, il suffit de définir fact n comme fact (n, 1), par exemple localement :

---

```

# let fact =
    let rec fact_term = fun
        | (1, x) -> x
        | (n, x) -> fact_term (n - 1, x * n)

```

---

```
in function n -> fact_term (n, 1);;
```

---

**Remarque :** on peut se demander pourquoi je n'ai pas employé une fonction curryfiée.

---

```
# let rec curry_fact_term = fun
  | 1 x -> x
  | n x -> curry_fact_term (n - 1) (x * n);;
curry_fact_term : int -> int -> int = <fun>
```

```
# trace "curry_fact_term";;
The function curry_fact_term is now traced.
- : unit = ()
```

```
# curry_fact_term 5 1;;
curry_fact_term <- 5
curry_fact_term -> <fun>
curry_fact_term* <- 1
curry_fact_term <- 4
curry_fact_term -> <fun>
curry_fact_term* <- 5
curry_fact_term <- 3
curry_fact_term -> <fun>
curry_fact_term* <- 20
curry_fact_term <- 2
curry_fact_term -> <fun>
curry_fact_term* <- 60
curry_fact_term <- 1
curry_fact_term -> <fun>
curry_fact_term* <- 120
curry_fact_term* -> 120
curry_fact_term* -> 120
curry_fact_term* -> 120
curry_fact_term* -> 120
curry_fact_term* -> 120
- : int = 120
```

---

Voici un exemple de programmation récursive terminale de la suite de Fibonacci :

---

```
# let rec fibo_term = fun
  | (0, x, y) -> x
  | (n, x, y) -> fibo_term ((n - 1), y, x + y);;
fibo_term : int * int * int -> int = <fun>
```

```
# trace "fibo_term";;
The function fibo_term is now traced.
- : unit = ()
```

```
# fibo_term (5, 1, 1);;
fibo_term <- 5, 1, 1
fibo_term <- 4, 1, 2
fibo_term <- 3, 2, 3
fibo_term <- 2, 3, 5
fibo_term <- 1, 5, 8
fibo_term <- 0, 8, 13
fibo_term -> 8
fibo_term -> 8
fibo_term -> 8
fibo_term -> 8
fibo_term -> 8
fibo_term -> 8
- : int = 8
```

---

**Remarque :** on donnera plus tard une méthode encore bien plus efficace grâce à la stratégie « diviser pour régner ».



6. ENSEMBLES INDUCTIFS, INDUCTION STRUCTURELLE

Définition (Ensemble inductif)

Soit  $E$  un ensemble,  $\mathcal{B}$  une partie de  $E$ ,  $\mathcal{C}$  un ensemble d'applications  $\Phi : E^{a(\Phi)} \rightarrow E$ , où  $a(\Phi)$  est l'arité de  $\Phi$ .

On appelle *ensemble défini inductivement* par  $\mathcal{B}$  et  $\mathcal{C}$  la plus petite partie  $\mathcal{T}$  de  $E$  (pour l'inclusion) contenant  $\mathcal{B}$ , et stable par tous les éléments de  $\mathcal{C}$ , dans le sens où pour tout  $\Phi \in \mathcal{C}$ , tout  $(t_1, \dots, t_{a(\Phi)}) \in \mathcal{T}^{a(\Phi)}$ ,  $\Phi(t_1, \dots, t_{a(\Phi)}) \in \mathcal{T}$ .

De plus, les éléments de  $\mathcal{B}$  sont les *éléments de base* de  $\mathcal{T}$ , et les éléments de  $\mathcal{C}$  sont appelés *constructeurs* de  $\mathcal{T}$ .

6.a

On fixe pour la suite les notations de cette définition

Démonstration

Justification de l'existence de  $\mathcal{T}$  (l'unicité étant claire).

□

Exemple (Ensembles définis inductivement)

Ensemble des entiers naturels,  $2\mathbb{N}$ .

Listes

i

Définition (Éléments de complexité au plus  $n$ )

On pose  $\mathcal{T}_0 = \mathcal{B}$ , et, pour tout  $n \in \mathbb{N}$ ,  $\mathcal{T}_{n+1} = \mathcal{T}_n \cup \{y \in E, \exists(\Phi, (t_k)) \in \mathcal{C} \times \mathcal{T}_n^{a(\Phi)}, y = \Phi(t_k)\}$ .

Si  $y \in \mathcal{T}_n$ , on dit que  $y$  est de complexité au plus  $n$ .

6.b

**Remarque :** cette complexité ne dépend pas seulement de l'ensemble  $\mathcal{T}$ , mais surtout de la façon dont il est construit, *i.e.* dépend des constructeurs et des cas de base.

Théorème Description d'un ensemble inductif par niveaux de complexité

On a  $\mathcal{T} = \bigcup_{n \in \mathbb{N}} \mathcal{T}_n$ .

6.a

Démonstration

facile.

□

Définition (Complexité)

Pour tout  $y \in \mathcal{T}$ , on appelle *complexité* du terme  $y$  de l'ensemble inductif  $\mathcal{T}$  comme  $\min\{n \in \mathbb{N}, y \in \mathcal{T}_n\}$ .

6.c

## Théorème Induction structurelle

Soit  $p$  un prédicat sur  $E$ . On suppose

(1)  $\forall b \in \mathcal{B}, p(b)$ .

(2) Pour tout  $(\Phi, (t_1, \dots, t_{a(\Phi)})) \in \mathcal{C} \times \mathcal{T}^{a(\Phi)}$  :

$(\forall k \in \llbracket 1, a(\Phi) \rrbracket, p(t_k)) \Rightarrow p(\Phi(t_1, \dots, t_{a(\Phi)}))$ .

Le prédicat est alors vrai sur  $\mathcal{T}$ .

6.b

## Démonstration

Il suffit de constater que l'ensemble  $\Omega$  des éléments de  $E$  pour lesquels le prédicat est vrai contient  $\mathcal{B}$  et est stable par application des éléments de  $\mathcal{C}$ .

□

**Remarque :** on peut appliquer ce résultat à la définition et égalité de fonction(s).

**Remarque :** lorsqu'on programme une fonction définie sur un ensemble inductif, il est très fréquent que la structure inductive dicte la programmation.

Troisième partie

Analyse des algorithmes



## CHAPITRE V

# Complexité

### 1. GÉNÉRALITÉS

#### 1.1. COMMENT MESURER LA PERFORMANCE D'UN PROGRAMME ?

Bien entendu, la première vertu d'un programme est sa correction, la suivante étant sa terminaison. Une fois ces propriétés satisfaites, il reste à savoir dans quelle mesure le programme proposé est efficace : c'est l'objet du domaine de l'informatique appelé *complexité*.

On distingue surtout deux types de complexité : la *complexité temporelle*, qui évalue la rapidité de l'algorithme, et la *complexité spatiale*, qui évalue l'occupation mémoire de l'algorithme.

Les progrès de l'informatique ont fait perdre de l'importance à la complexité spatiale : nous nous concentrerons surtout sur la complexité temporelle.

Bien entendu, il est hors de question d'évaluer concrètement cette complexité temporelle, en chronométrant un programme : cela n'aurait qu'une valeur empirique, non prédictive, dépendrait fortement du hardware sur lequel le programme tourne, dépendrait des données initiales (par exemple, il est facile de tester si  $2^{43112609}$  est premier, ça l'est moins pour  $2^{43112609} - 1$ ). Il n'est pas non plus utile de donner précisément la complexité de l'algorithme, mais plutôt son ordre de grandeur.

Il nous faut donc trouver un cadre d'étude théorique pour évaluer la rapidité d'un programme.

#### 1.2. NOTION DE TAILLES DE DONNÉES, CLASSES DE COMPLEXITÉ

La plupart des algorithmes ont un argument entier (test de primalité, factorisation), plusieurs (algorithme d'Euclide, exponentiation) ou leur exécution dépend d'un entier naturel (taille d'une liste, d'un vecteur) : nous noterons  $n$  un entier, représentant la taille de données, dont les algorithmes dépendront. Pour un entier, ce peut être le nombre de bits.

Outre la notation de Landau  $O$ , nous utiliserons également la notation  $\Theta$ , signifiant que deux suites ont même ordre :  $u_n = \Theta(v_n)$  signifie  $u_n = O(v_n)$  et  $v_n = O(u_n)$ .

Selon la taille de données  $n$ , l'algorithme va effectuer un certain nombre de tâches, dont certaines auront un poids bien plus grand dans le temps d'exécution. Nous ne compterons que le nombre  $c_n$  de ces opérations coûteuses.

On dit qu'un algorithme est

- *logarithmique* si  $c_n$  est de l'ordre de  $\log_2(n)$ .
- *linéaire* si  $c_n$  est de l'ordre de  $n$ .
- *quasi-linéaire* si  $c_n$  est de l'ordre de  $n \log n$ .
- *quadratique* si  $c_n$  est de l'ordre de  $n^2$ .
- *polynomial* si  $c_n$  est de l'ordre de  $n^k$ , pour un entier non nul  $k$ .
- *exponentiel* si  $c_n$  est de l'ordre de  $a^n$ , où  $a > 1$ .

Ces classes de complexité sont données en ordre croissant : les algorithmes exponentiels sont très peu utiles, les logarithmiques finissent en temps raisonnable pour n'importe quelle taille de l'entrée.

**Remarque :** bien sûr, il faut tempérer ce jugement, puisqu'il peut y avoir des coûts occultes (si par exemple  $c_n \sim 10^{100} \log_2(n)$ , l'algorithme n'est pas si pratique que cela ...).

#### 1.3. UN RAFFINEMENT NÉCESSAIRE

En réalité, la vitesse d'exécution d'un algorithme peut être très sensible à la valeur des données, pas seulement à la taille  $n$  de ces données (penser encore une fois à un test de primalité). Ceci nous conduit à introduire des complexités selon les cas : complexité dans le meilleur, le pire des cas, complexité en moyenne.

Dans ce dernier cas, il est important de préciser le contexte probabiliste, afin de pouvoir pondérer les différentes entrées possibles.

## 2. UN PREMIER EXEMPLE

Nous prenons l'exemple de la recherche d'un élément dans un tableau :

---

```
# let cherche elt t = let i = ref 0 and trouve = ref false in
  while (!i < vect_length t & !trouve = false) do
    if elt = t.(!i) then trouve := true else i := !i + 1
  done;
  !trouve;;
cherche : 'a -> 'a vect -> bool = <fun>

# cherche 3 [| 5; 6; 9 |];;
- : bool = false
```

---

**Exercice :** programmer par exception la recherche dans un tableau.

$n$  est ici la taille du tableau, et nous allons compter le nombre de comparaisons  $elt = t.(!i)$  : dans le meilleur des cas, il n'y en a qu'une, dans le pire, il y en a  $n$ .

Supposons que  $elt$  apparaisse avec une probabilité  $q$ , que les éléments de  $t$  soient distincts, et que chaque position pour  $elt$  soit équiprobable.

Pour trouver l'élément en position  $i$ , on effectue  $i$  comparaisons. La complexité est donc  $(1 - q)n + q\frac{n+1}{2}$ . Elle est donc linéaire, comme dans le pire des cas.

## 3. DIVISER POUR RÉGNER

## 3.1. LE PRINCIPE

La méthode ou le paradigme « diviser pour régner » consiste, étant donné un problème de taille de données  $n$ , à :

- (1) Diviser ce problème en  $q$  sous-problèmes de taille  $n/2$  environ.
- (2) Traiter ces  $q$  sous-problèmes.
- (3) Fusionner ces  $q$  sous-problèmes

Le gain en complexité peut être énorme dans certains cas. Bien sûr, on applique ceci récursivement (on coupe chaque sous-problème en  $q$  sous-problèmes, etc.).

**Remarque :** cette méthode se rencontre souvent en informatique, et il faut apprendre à la mettre en œuvre et à la proposer spontanément.

**Remarque :** le plus souvent,  $q = 2$ .

Un premier exemple d'utilisation de ce paradigme pourrait être la recherche dichotomique (dans un tableau trié, d'un zéro d'une fonction), dont nous reparlerons lors du cours sur les tris. Nous allons voir d'autres exemples : l'exponentiation, la multiplication des polynômes, des grands entiers, des matrices. Nous rencontrerons également des algorithmes de tri efficaces fondés sur cette méthode.

## 3.2. GAIN EN TERMES DE COMPLEXITÉ

Pour évaluer le gain en termes de complexité, nous allons traiter le cas où  $n = 2^m$ , où le coût de la division et de la fusion est de  $f(n)$ , nous notons  $C(n) = T(m)$  le coût temporel de l'algorithme.

Nous avons donc la relation de récurrence :

$$T(m+1) = qT(m) + f(2^m)$$

que l'on peut réécrire

$$\frac{T(m+1)}{q^{m+1}} - \frac{T(m)}{q^m} = \frac{f(2^m)}{q^{m+1}}$$

On obtient donc, en sommant :

$$\frac{T(m)}{q^m} = T(0) + \sum_{k=0}^{m-1} \frac{f(2^k)}{q^{k+1}}.$$

**Remarque :** observons que  $q^m = n^{\log_2(q)}$  : si on connaît déjà un algorithme en  $\Theta(n^{\log_2(q)})$ , le gain sera nul. Tout l'art consistera à minimiser  $q$ .

**Premier cas :**  $\frac{f(2^m)}{q^m} = O(\frac{1}{m^2})$ , par exemple  $f(n)$  est polynomial en  $m$ , ou en  $\alpha^m$ , où  $\alpha < q$ . On montre alors que  $T(m) \sim \lambda q^m = \lambda n^{\log_2(q)}$ , de sorte que l'algorithme a une complexité au pire linéaire lorsque  $q = 2$ .

**Deuxième cas :**  $f(n)$  a un coût en  $n^{\log_2(q)}$ . On montre alors que  $C(n) = \Theta(n^{\log_2(q)} \log_2(n))$ , de sorte que l'algorithme a une complexité quasi-linéaire lorsque  $q = 2$ .

**Troisième cas :**  $f(n)$  a un coût en  $\alpha^m$ , où  $\alpha > q$ . On montre alors que  $T(m)$  a une complexité dominée par  $q^m \left(\frac{\alpha}{q}\right)^{m+1}$ , i.e. par  $\alpha^m = n^{\log_2(\alpha)}$ .

**Remarque :** le fait de prendre une puissance de deux ne nuit pas à la généralité du calcul de complexité, car un encadrement d'un autre entier entre deux puissances de deux permet de conclure.

### 3.3. L'EXPONENTIATION RAPIDE

L'idée la plus évidente pour calculer les puissances d'un entier est la suivante :

---

```
# let puissance_imperative a b = let p = ref 1 in
  for i = 1 to b do p := !p * a done; !p;;
(* invariant après la i-ème itération : !p = a ^ i *)
puissance_imperative : int -> int -> int = <fun>
```

```
# let rec puissance a = function
  | 0 -> 1
  | b -> a * puissance a (b - 1);;
puissance : int -> int -> int = <fun>
```

---

Ces fonctions donnent bien le résultat voulu, mais leur complexité (comptée en nombre de multiplications) est linéaire en  $b$ . Le paradigme diviser pour régner permet de faire bien mieux :

---

```
# let rec puissance a b = match b with
  | 0 -> 1
  | - -> let y = puissance a (b / 2) in
    if b mod 2 = 0 then y * y else y * y * a;;
puissance : int -> int -> int = <fun>
```

```
# puissance 2 10;;
- : int = 1024
```

---

De manière impérative, cela pourrait donner

---

```
# let puiss a b = let res = ref a and expo = ref b and reste = ref 1 in
  while !expo > 1 do
    if !expo mod 2 = 1 then reste := !reste * a;
    res := !res * !res;
    expo := !expo / 2
  done;
  (* Invariant : a^b = !res ^ !expo * !reste *)
  !res * !reste;;
puiss : int -> int -> int = <fun>
```

```
# puiss 3 5;;
- : int = 243
```

---

Dans les deux cas, on trouve une complexité logarithmique (en nombre de multiplications) : le gain est donc conséquent. Ce gain s'explique par le fait que le coût de la fusion vaut 1 ou 2, mais surtout que dans le paradigme diviser pour régner, on a ici  $q = 1$  !

### 3.4. L'ALGORITHME DE KNUTH DE MULTIPLICATION DES POLYNÔMES

On s'intéresse à la multiplication de polynômes à coefficients entiers, représentés par des tableaux d'entiers (l'indice correspond au degré du monôme associé).

On cherche à multiplier deux polynômes  $A = \sum_{k=0}^p a_k X^k$  et  $B = \sum_{k=0}^q b_k X^k$ . On sait que :

$$AB = \sum_{i=0}^p \sum_{j=0}^q a_i b_j X^{i+j},$$

ce qui permet de proposer une première fonction réalisant ce produit :

---

```

# let produit_poly_simple a b = let p = vect_length a and q = vect_length b in
  let res = make_vect (p + q - 1) 0 in
    for i = 0 to p - 1 do
      for j = 0 to q - 1 do
        res.(i + j) <- res.(i + j) + a.(i) * b.(j)
      done
    done;
  res;;
produit_poly_simple : int vect -> int vect -> int vect = <fun>

# produit_poly_simple [|1; 1; 1|] [|1; 1; 1|];;
- : int vect = [|1; 2; 3; 2; 1|]

```

---

Cependant, la complexité est clairement de l'ordre de  $pq$ , donc au pire quadratique dans le cas de polynômes de même degré.

Knuth a proposé un algorithme de meilleure complexité, en utilisant l'astuce de Karatsuba : on suppose les deux polynômes de degré au plus  $2n - 1$ . On écrit  $P = P_0 + X^n P_1$  et  $Q = Q_0 + X^n Q_1$  (on a effectué les divisions euclidiennes de  $P$  et de  $Q$  par  $X^n$ ). On a donc

$$PQ = P_0 Q_0 + X^n (P_0 Q_1 + P_1 Q_0) + X^{2n} P_1 Q_1.$$

Ceci donne quatre multiplications, ce qui n'apporte rien en termes de complexité. L'astuce de Karatsuba consiste cependant à remarquer que les trois produits  $P_0 Q_0$ ,  $P_1 Q_1$ , et  $(P_0 + P_1)(Q_0 + Q_1)$  suffisent à calculer ces quatre produits (au prix d'additions supplémentaires, de coût négligeable par rapport à une multiplication). La complexité (comptée en nombre de multiplications) est de l'ordre de  $n^{\log_2(3)} \simeq n^{1.58}$ .

**Remarque :** cet algorithme est utile pour effectuer la multiplication de grands entiers en base  $b$  (représentés par des vecteurs d'entiers entre 0 et  $b - 1$ ).

### 3.5. L'ALGORITHME DE STRASSEN DE MULTIPLICATION DES MATRICES

Pour multiplier deux matrices carrées de taille  $n$ , la méthode standard, revenant à la définition d'un tel produit, est de complexité en  $n^3$  (on compte encore une fois seulement les multiplications).

On peut mieux faire, en effectuant un produit matriciel par blocs. On cherche à multiplier deux matrices carrées de taille paire  $n$

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \quad \text{et} \quad B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

Les formules de produit par bloc donnent

$$AB = \begin{pmatrix} A_1 B_1 + A_2 B_3 & A_1 B_2 + A_2 B_4 \\ A_3 B_1 + A_4 B_3 & A_3 B_2 + A_4 B_4 \end{pmatrix},$$

ce qui semble nécessiter huit multiplications, ce qui serait sans grand intérêt. L'idée de Strassen, un peu similaire à l'astuce de Karatsuba, est de réduire le nombre de multiplications à sept. On note :

$$P_1 = A_1(B_2 - B_4), P_2 = (A_1 + A_2)B_4, P_3 = (A_3 + A_4)B_1, P_4 = A_4(B_3 - B_1), \\ P_5 = (A_1 + A_4)(B_1 + B_4), P_6 = (A_2 - A_4)(B_3 + B_4), P_7 = (A_1 - A_3)(B_1 + B_2)$$

On a

$$AB = \begin{pmatrix} -P_2 + P_4 + P_5 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 - P_3 + P_5 - P_7 \end{pmatrix}.$$

On obtient donc une complexité en  $n^{\log_2(7)}$ .



---

```

# let puissance_imperative a b = let p = ref 1 in
    for i = 1 to b do p := !p * a done; !p;;
(* invariant après la i-ème itération : !p = a ^ i *)
puissance_imperative : int -> int -> int = <fun>

# let rec puissance a = function
    | 0 -> 1
    | b -> a * puissance a (b - 1);;
puissance : int -> int -> int = <fun>

# let rec puissance a b = match b with
    | 0 -> 1
    | - -> let y = puissance a (b / 2) in
        if b mod 2 = 0 then y * y else y * y * a;;
puissance : int -> int -> int = <fun>

# puissance 2 10;;
- : int = 1024

# let puiss a b = let res = ref a and expo = ref b and reste = ref 1 in
    while !expo > 1 do
        if !expo mod 2 = 1 then reste := !reste * a;
        res := !res * !res;
        expo := !expo / 2
    done;
    (* Invariant : a^b = !res ^ !expo * !reste *)
    !res * !reste;;
puiss : int -> int -> int = <fun>

# puiss 3 5;;
- : int = 243

# let produit_poly_simple a b = let p = vect_length a and q = vect_length b in
    let res = make_vect (p + q - 1) 0 in
        for i = 0 to p - 1 do
            for j = 0 to q - 1 do
                res.(i + j) <- res.(i + j) + a.(i) * b.(j)
            done
        done;
    res;;
produit_poly_simple : int vect -> int vect -> int vect = <fun>

# produit_poly_simple [[1; 1; 1]] [[1; 1; 1]];;
- : int vect = [[1; 2; 3; 2; 1]]

```

---



# Tris

## 1. INTRODUCTION

Le tri est à la fois l'un des plus anciens et courants problèmes d'informatique. De nombreux algorithmes ont été proposés afin d'effectuer le tri d'un tableau ou d'une liste d'éléments d'un ensemble totalement ordonné.

La complexité d'un algorithme de tri s'effectue en prenant pour taille de données la taille de la liste ou du vecteur à trier, et en comptant le nombre d'échanges ou de comparaisons effectuées (on pourra privilégier, selon le but cherché, l'un ou l'autre). On s'intéressera également à l'espace employé pour l'application de tel ou tel algorithme (un tri sur place étant un tri n'employant qu'un espace constant), et, éventuellement, à la stabilité d'un tel algorithme, *i.e.* s'il lui arrive d'échanger des valeurs égales en des indices distincts.

La structure la plus naturellement adaptée aux tris est celle de vecteur, puisque l'accès aux éléments d'une telle structure s'effectue en temps constant, contrairement au cas des listes.

Nous allons motiver le tri par la recherche dichotomique, se poser le problème de la fusion de deux tableaux ou de deux listes triées, puis proposer de nombreux algorithmes de tri.

## 2. UTILITÉ DU TRI : RECHERCHE DANS UN TABLEAU TRIÉ

Nous avons déjà vu un algorithme linéaire de recherche dans un tableau. Cependant, si le tableau est trié, une recherche dichotomique (cas particulier de la stratégie diviser pour régner) permet d'effectuer cette recherche en temps logarithmique :

---

```
# let rec cherche_dicho t (elt : int) = let n = vect_length t in match n with
  | 0 -> false
  | 1 -> if t.(0) = elt then true else false
  | _ -> let n' = n / 2 in
          if t.(n') <= elt then cherche_dicho (sub_vect t n' (n - n')) elt
          else cherche_dicho (sub_vect t 0 n') elt;;
cherche_dicho : int vect -> int -> bool = <fun>
```

```
# trace "cherche_dicho";;
The function cherche_dicho is now traced.
- : unit = ()
```

```
# cherche_dicho [|1; 3; 7; 9; 12; 14|] 12;;
cherche_dicho <- [|1; 3; 7; 9; 12; 14|]
cherche_dicho -> <fun>
cherche_dicho* <- 12
cherche_dicho <- [|9; 12; 14|]
cherche_dicho -> <fun>
cherche_dicho* <- 12
cherche_dicho <- [|12; 14|]
cherche_dicho -> <fun>
cherche_dicho* <- 12
cherche_dicho <- [|12|]
cherche_dicho -> <fun>
cherche_dicho* <- 12
cherche_dicho* -> true
cherche_dicho* -> true
cherche_dicho* -> true
cherche_dicho* -> true
- : bool = true
```

---

**Remarque :** dans le cas d'une liste chaînée, la recherche dichotomique n'est pas adaptée, puisque l'accès à un élément ne s'effectue pas en temps constant, et que la longueur de la liste n'est pas connue.

### 3. INSERTION DANS UNE LISTE TRIÉE

Comment insérer un élément dans un tableau ou une liste déjà trié ?

---

```
# let insertion t elt = let n = vect_length t in let t' = make_vect (n + 1) elt in
  let i = ref 0 in (* on place les éléments inférieurs à elt *)
    while !i < n & t.(!i) < elt do
      t'.(!i) <- t.(!i); i := !i + 1
    done;
  let j = ref (n - 1) in (* on place les éléments supérieurs à elt *)
    while !j >= 0 & t.(!j) > elt do
      t'.(!j + 1) <- t.(!j); j := !j - 1
    done;
  t';;
insertion : 'a vect -> 'a -> 'a vect = <fun>
```

```
# insertion [|3; 4; 9|] 7;;
- : int vect = [|3; 4; 7; 9|]
```

On peut aussi effectuer une insertion pour les listes

---

```
# let rec insertion_liste l elt = match l with
| [] -> [elt]
| a :: q -> if a < elt then a :: insertion_liste q elt else elt :: l;;
insertion_liste : 'a list -> 'a -> 'a list = <fun>
```

```
# insertion_liste [3; 4; 9] 7;;
- : int list = [3; 4; 7; 9]
```

**Remarque :** on effectue  $n/2$  comparaisons en moyenne.

### 4. FUSION DE DEUX LISTES TRIÉES

Le problème de la fusion de deux tableaux triés (ou listes triées), qui généralise en quelque sorte celui de l'insertion, se pose naturellement. Comment programmer ?

Pour les tableaux, on peut proposer

---

```
# let fusion t t' = let n = vect_length t and n' = vect_length t' in
  let res = make_vect (n + n') 0 (* résultat *) in
  let j = ref 0 and j' = ref 0 in
  for i = 0 to n + n' - 1 do
    (* après le passage pour l'indice i, les termes du résultat,
    d'indices 0 à i, sont bien placés dans res *)
    if !j < n (* le tableau t n'est pas épuisé *)
    then
      if !j' < n' (* le tableau t' n'est pas épuisé *)
      then
        if t.(!j) <= t'.(!j') then
          (res.(i) <- t.(!j);
           j := !j + 1);
        else
          (res.(i) <- t'.(!j');
           j' := !j' + 1);
        else (* le tableau t' est épuisé *)
          (res.(i) <- t.(!j); j := !j + 1);
      else (* le tableau t est épuisé *)
        (res.(i) <- t'.(!j'); j' := !j' + 1);
    done;
```

```

    res ;;
fusion : int vect -> int vect -> int vect = <fun>

# fusion [|1; 2; 5; 15; 20|] [|0; 2; 6; 9; 11|];;
- : int vect = [|0; 1; 2; 2; 5; 6; 9; 11; 15; 20|]

```

La complexité est bien sûr en  $n + n'$ , donc linéaire.

**Remarque :** on pouvait bien sûr effectuer une succession d'insertions.

Pour les listes, cela pourrait donner :

```

# let rec fusion_listes l l' = match (l, l') with
  | (-, []) -> l
  | ([], -) -> l'
  | (a :: q, a' :: q') -> if a <= a' then a :: fusion_listes q l'
                          else a' :: fusion_listes l q';;
fusion_listes : 'a list -> 'a list -> 'a list = <fun>

# fusion_listes [|1; 2; 5; 15; 20|] [|0; 2; 6; 9; 11|];;
- : int list = [|0; 1; 2; 2; 5; 6; 9; 11; 15; 20|]

```

## 5. EXEMPLES DE TRIS

### 5.1. LE TRI PAR INSERTION

Le tri par insertion, celui du joueur de cartes, consiste à insérer un par un chaque terme, tout en conservant l'ordre.

On peut le programmer ainsi :

```

# let tri_insertion t = let t' = ref [||] in
  for i = 0 to vect_length t - 1 do t' := insertion !t' t.(i) done;
  !t';;
tri_insertion : 'a vect -> 'a vect = <fun>

# tri_insertion [|1; 3; -2; -1; 5; -3; 6|];;
- : int vect = [| -3; -2; -1; 1; 3; 5; 6|]

# let rec tri_insertion_liste = function
  | [] -> []
  | a :: q -> insertion_liste (tri_insertion_liste q) a;;
tri_insertion_liste : 'a list -> 'a list = <fun>

# tri_insertion_liste [|1; 3; -2; -1; 5; -3; 6|];;
- : int list = [ -3; -2; -1; 1; 3; 5; 6]

```

**Remarque :** cette programmation « à la lettre » du tri par insertion n'est pas optimale. On peut faire bien mieux en programmant par effets de bord.

**Remarque :** le coût d'une insertion étant linéaire, et puisque nous en faisons  $n$ , le coût du tri par insertion est quadratique, ce qui sera le cas de la plupart des tris simples.

**Remarque :** le tri par insertion est très efficace si le tableau ou le vecteur est presque trié.

### 5.2. LE TRI PAR SÉLECTION

Dans le tri par sélection, on procède de la manière suivante : on cherche d'abord le plus petit élément, que l'on place dans le résultat, puis le suivant, que l'on range en deuxième position, etc.

```

# let echange t i j = let c = t.(i) in t.(i) <- t.(j); t.(j) <- c;;
echange : 'a vect -> int -> int -> unit = <fun>

# let tri_selection t = let n = vect_length t in
  (* programmation par effets de bord *)
  for i = 0 to n - 2 do
    (* après le passage en boucle pour l'indice i, t (0 .. i)

```

```

    est constitué des (i + 1) plus petits termes de t ordonnés *)
    let temp = ref i in
      for j = i + 1 to n - 1 do
        if t.(j) < t.(!temp) then temp := j;
        done;
      échange t i !temp;
    done;;
tri_selection : 'a vect -> unit = <fun>

# let tab = [| -3; 1; -1; 16; 7; 12 |];;
tab : int vect = [| -3; 1; -1; 16; 7; 12 |]

# tri_selection tab;;
- : unit = ()

# tab;;
- : int vect = [| -3; -1; 1; 7; 12; 16 |]

```

---

Cet algorithme est encore quadratique pour les comparaisons, mais linéaire pour les échanges.  
Voici une version pour les listes :

```

# let rec isole_min = function
  | [] -> failwith "pas_de_minimum_pour_la_liste_vide"
  | [a] -> (a, [])
  | a :: q -> let l' = isole_min q in
    if a <= fst l' then (a, q) else (fst l', a :: snd l');;
isole_min : 'a list -> 'a * 'a list = <fun>

# isole_min [3; 6; -2; 7; 9];;
- : int * int list = -2, [3; 6; 7; 9]

# let rec tri_selection_liste = function
  | [] -> []
  | l -> let l' = isole_min l in
    (fst l') :: (tri_selection_liste (snd l'));;
tri_selection_liste : 'a list -> 'a list = <fun>

# tri_selection_liste [6; 3; -2; 9; 1; 10; 3];;
- : int list = [-2; 1; 3; 3; 6; 9; 10]

```

### 5.3. LE TRI BULLE

Le tri bulle est une variante du tri par sélection : on parcourt le tableau en échangeant deux éléments consécutifs mal placés.

```

# let tri_bulle t = let n = vect_length t in
  for i = 0 to (n - 1) do
    for j = 0 to (n - 2) do
      if t.(j) > t.(j + 1) then échange t j (j + 1)
      done;
    done;
  t;;
tri_bulle : 'a vect -> 'a vect = <fun>

(* Non optimal, car le tri bulle peut s'arrêter
dès qu'il ne modifie plus rien en une passe *)

# tri_bulle [| 3; 5; 1; 6; 3 |];;
- : int vect = [| 1; 3; 3; 5; 6 |]

```

---

**Remarque :** le tri bulle a peu d'intérêt, puisqu'il est quadratique en comparaisons et en échanges.

---

```
(* Recherche dichotomique pour les vecteurs *)

# let rec cherche_dicho t (elt : int) = let n = vect_length t in match n with
  | 0 -> false
  | 1 -> if t.(0) = elt then true else false
  | - -> let n' = n / 2 in
          if t.(n') <= elt then cherche_dicho (sub_vect t n' (n - n')) elt
          else cherche_dicho (sub_vect t 0 n') elt;;

cherche_dicho : int vect -> int -> bool = <fun>

# trace "cherche_dicho";;
The function cherche_dicho is now traced.
- : unit = ()

# cherche_dicho [|1; 3; 7; 9; 12; 14|] 12;;
cherche_dicho <- [|1; 3; 7; 9; 12; 14|]
cherche_dicho -> <fun>
cherche_dicho* <- 12
cherche_dicho <- [|9; 12; 14|]
cherche_dicho -> <fun>
cherche_dicho* <- 12
cherche_dicho <- [|12; 14|]
cherche_dicho -> <fun>
cherche_dicho* <- 12
cherche_dicho <- [|12|]
cherche_dicho -> <fun>
cherche_dicho* <- 12
cherche_dicho* -> true
cherche_dicho* -> true
cherche_dicho* -> true
cherche_dicho* -> true
- : bool = true
```

---

---

```

(* Insertion d'un élément dans un vecteur trié *)

# let insertion t elt = let n = vect_length t in let t' = make_vect (n + 1) elt in
  let i = ref 0 in (* on place les éléments inférieurs à elt *)
    while !i < n & t.(!i) < elt do
      t'.(!i) <- t.(!i); i := !i + 1
    done;
  let j = ref (n - 1) in (* on place les éléments supérieurs à elt *)
    while !j >= 0 & t.(!j) > elt do
      t'.(!j + 1) <- t.(!j); j := !j - 1
    done;
  t';;
insertion : 'a vect -> 'a -> 'a vect = <fun>

# insertion [|3; 4; 9|] 7;;
- : int vect = [|3; 4; 7; 9|]

(* Fusion de vecteurs triés *)

# let fusion t t' = let n = vect_length t and n' = vect_length t' in
  let res = make_vect (n + n') 0 (* résultat *) in
  let j = ref 0 and j' = ref 0 in
  for i = 0 to n + n' - 1 do
    (* après le passage pour l'indice i, les termes du résultat,
    d'indices 0 à i, sont bien placés dans res *)
    if !j < n (* le tableau t n'est pas épuisé *)
    then
      if !j' < n' (* le tableau t' n'est pas épuisé *)
      then
        if t.(!j) <= t'.(!j') then
          (res.(i) <- t.(!j);
           j := !j + 1;)
        else
          (res.(i) <- t'.(!j');
           j' := !j' + 1;)
        else (* le tableau t' est épuisé *)
          (res.(i) <- t.(!j); j := !j + 1;)
      else (* le tableau t est épuisé *)
        (res.(i) <- t'.(!j'); j' := !j' + 1;)
    done;
  res;;
fusion : int vect -> int vect -> int vect = <fun>

# fusion [|1; 2; 5; 15; 20|] [|0; 2; 6; 9; 11|];;
- : int vect = [|0; 1; 2; 2; 5; 6; 9; 11; 15; 20|]

```

---



---

```

(* Tri par insertion "à la lettre" pour les vecteurs *)

# let tri_insertion t = let t' = ref [||] in
  for i = 0 to vect_length t - 1 do t' := insertion !t' t.(i) done;
  !t';;
tri_insertion : 'a vect -> 'a vect = <fun>

# tri_insertion [|1; 3; -2; -1; 5; -3; 6|];;
- : int vect = [| -3; -2; -1; 1; 3; 5; 6|]

(* Tri par sélection pour les vecteurs *)

# let echange t i j = let c = t.(i) in t.(i) <- t.(j); t.(j) <- c;;
echange : 'a vect -> int -> int -> unit = <fun>

# let tri_selection t = let n = vect_length t in
  (* programmation par effets de bord *)
  for i = 0 to n - 2 do
    (* après le passage en boucle pour l'indice i, t (0 .. i)
     est constitué des (i + 1) plus petits termes de t ordonnés *)
    let temp = ref i in
      for j = i + 1 to n - 1 do
        if t.(j) < t.(!temp) then temp := j;
        done;
      echange t i !temp;
    done;;
tri_selection : 'a vect -> unit = <fun>

# let tab = [| -3; 1; -1; 16; 7; 12|];;
tab : int vect = [| -3; 1; -1; 16; 7; 12|]

# tri_selection tab;;
- : unit = ()

# tab;;
- : int vect = [| -3; -1; 1; 7; 12; 16|]

(* Tri bulle pour les vecteurs *)

# let tri_bulle t = let n = vect_length t in
  for i = 0 to (n - 1) do
    for j = 0 to (n - 2) do
      if t.(j) > t.(j + 1) then echange t j (j + 1)
      done;
    done;
  t;;
tri_bulle : 'a vect -> 'a vect = <fun>

# tri_bulle [|3; 5; 1; 6; 3|];;
- : int vect = [|1; 3; 3; 5; 6|]

```

---

Nous présentons maintenant deux algorithmes de tri évolués, fondés sur la stratégie diviser pour régner (dans lesquels la division et la fusion n'ont pas le même poids).

## 5.4. LE TRI FUSION

L'idée consiste à scinder la liste en deux listes de même taille (ou à peu près), à trier chacune de ces deux sous-listes, et à les fusionner (ceci de manière récursive bien entendu).

---

```
(* Tri fusion (on rappelle que la fusion a déjà été programmée) *)

# let rec tri_fusion t = let n = vect_length t in match n with
  | 0 | 1 -> t
  | _ -> let n' = n / 2 in
          fusion (tri_fusion (sub_vect t 0 n')) (tri_fusion (sub_vect t n' (n - n')));
tri_fusion : int vect -> int vect = <fun>

# trace "tri_fusion";;
The function tri_fusion is now traced.
- : unit = ()

# tri_fusion [|1; 3; 5; 1; 7; 4; 9; -1|];;
tri_fusion <- [|1; 3; 5; 1; 7; 4; 9; -1|]
tri_fusion <- [|7; 4; 9; -1|]
tri_fusion <- [|9; -1|]
tri_fusion <- [| -1|]
tri_fusion -> [| -1|]
tri_fusion <- [|9|]
tri_fusion -> [|9|]
tri_fusion -> [| -1; 9|]
tri_fusion <- [|7; 4|]
tri_fusion <- [|4|]
tri_fusion -> [|4|]
tri_fusion <- [|7|]
tri_fusion -> [|7|]
tri_fusion -> [|4; 7|]
tri_fusion -> [| -1; 4; 7; 9|]
tri_fusion <- [|1; 3; 5; 1|]
tri_fusion <- [|5; 1|]
tri_fusion <- [|1|]
tri_fusion -> [|1|]
tri_fusion <- [|5|]
tri_fusion -> [|5|]
tri_fusion -> [|1; 5|]
tri_fusion <- [|1; 3|]
tri_fusion <- [|3|]
tri_fusion -> [|3|]
tri_fusion <- [|1|]
tri_fusion -> [|1|]
tri_fusion -> [|1; 3|]
tri_fusion -> [|1; 1; 3; 5|]
tri_fusion -> [| -1; 1; 1; 3; 4; 5; 7; 9|]
- : int vect = [| -1; 1; 1; 3; 4; 5; 7; 9|]
```

---

Le tri fusion est de complexité (temporelle) quasi-linéaire, puisque fondé sur la stratégie diviser pour régner standard, que la division a un coût nul, et que la fusion a un coût linéaire en termes de comparaisons.

**Remarque :** malheureusement, le tri fusion prend beaucoup de place en mémoire, à cause de ses nombreux appels récursifs.

## 5.5. LE TRI RAPIDE

L'idée du tri rapide (ou de Hoare, ou par segmentation) consiste à choisir un élément  $x$  de la liste (souvent en tête), appelé pivot, à scinder ce qu'il reste de la liste en deux sous-listes, selon leur relation à  $x$ .

---

```
(* Tri rapide *)

(* La fonction pivot compare les éléments de t à son terme initial,
les répartit dans un tableau, et précise en outre
l'indice où se trouve désormais le pivot *)

# let pivot t ind = let a = t.(ind) and n = vect_length t in
  let j = ref 1 and k = ref (n - 1) and res = make_vect n 0 in
  res.(0) <- a;
  (* on insère les éléments de t dans res selon leur relation à a *)
  for i = 0 to n - 1 do
    if i <> ind then if t.(i) < a then (res.(!j) <- t.(i); j := !j + 1;)
    else (res.(!k) <- t.(i); k:= !k - 1;)
  done;
  echange res 0 !k; (* on replace a à sa place *)
  res,!k;;
pivot : int vect -> int -> int vect * int = <fun>

# pivot [[3; 5; 6; 1; 2; 4; 9]] 3;;
- : int vect * int = [[1; 9; 4; 2; 6; 5; 3]], 0

(* Fonction fusion_triple pour améliorer la lisibilité *)

# let fusion_triple t t' t'' = fusion (fusion t t') t'';;
fusion_triple : int vect -> int vect -> int vect -> int vect = <fun>

(* Fonction tri_rapide suivant l'algorithme à la lettre *)

# let rec tri_rapide t = let n = vect_length t in
  if n <= 1 then t
  else
  let tab,i = pivot t 0 in match i with
  | 0 -> fusion [[tab.(0)]] (tri_rapide (sub_vect tab 1 (n - 1)))
  | k when k = n - 1 -> fusion (* cas de filtrage n - 1 non valide *)
    (tri_rapide (sub_vect tab 0 (n - 1)))
    [[tab.(n - 1)]]
  | _ -> fusion_triple
    (tri_rapide (sub_vect tab 0 i))
    [[tab.(i)]]
    (tri_rapide (sub_vect tab (i + 1) (n - i - 1)));;
tri_rapide : int vect -> int vect = <fun>

# trace "tri_rapide";;
The function tri_rapide is now traced.
- : unit = ()

# tri_rapide [[4; 2; 1; 8; 4; 7; 10; 11; -3; 8; 12]];;
tri_rapide <- [[4; 2; 1; 8; 4; 7; 10; 11; -3; 8; 12]]
tri_rapide <- [[12; 8; 11; 10; 7; 4; 8]]
tri_rapide <- [[8; 8; 11; 10; 7; 4]]
tri_rapide <- [[10; 11; 8]]
tri_rapide <- [[11]]
tri_rapide -> [[11]]
tri_rapide <- [[8]]
```

```

tri_rapide -> [|8|]
tri_rapide -> [|8; 10; 11|]
tri_rapide <- [|4; 7|]
tri_rapide <- [|7|]
tri_rapide -> [|7|]
tri_rapide -> [|4; 7|]
tri_rapide -> [|4; 7; 8; 8; 10; 11|]
tri_rapide -> [|4; 7; 8; 8; 10; 11; 12|]
tri_rapide <- [| -3; 2; 1|]
tri_rapide <- [|1; 2|]
tri_rapide <- [|2|]
tri_rapide -> [|2|]
tri_rapide -> [|1; 2|]
tri_rapide -> [| -3; 1; 2|]
tri_rapide -> [| -3; 1; 2; 4; 4; 7; 8; 8; 10; 11; 12|]
- : int vect = [| -3; 1; 2; 4; 4; 7; 8; 8; 10; 11; 12|]

```

Dans le pire des cas, l'un des tableaux scindés est vide, ce qui conduit à une complexité quadratique en termes de comparaisons.

Dans le cas où chaque tableau scindé est de taille moitié, on se retrouve avec une stratégie diviser pour régner classique, et on trouve une complexité en  $n \log_2(n)$ . En effet, la division a un coût linéaire, et la fusion un coût nul en termes de comparaisons.

**Remarque :** en réalité, cette complexité quasi-linéaire est optimale pour un algorithme de tri générique fondé sur les comparaisons.

Notons  $c_n$  le nombre moyen de comparaisons effectuées dans le tri rapide pour la taille de données  $n$  : on a

$$\begin{aligned}
nc_n &= n(n-1) + \sum_{k=0}^{n-1} (c_k + c_{n-1-k}) \\
&= n(n-1) + 2 \sum_{k=0}^{n-1} c_k \\
&= n(n-1) + 2c_{n-1} + (n-1)c_{n-1} - (n-1)(n-2) \\
&= (n+1)c_{n-1} + 2(n-1)
\end{aligned}$$

d'où

$$\frac{c_n}{n+1} - \frac{c_{n-1}}{n} \sim \frac{2}{n},$$

puis  $c_n$  est quasi-linéaire.

```

# let rec partition l (pivot : int) = match l with
| [] -> [], []
| a :: q -> let l, l' = partition q pivot in
            if a < pivot then a :: l, l' else l, a :: l';;
partition : int list -> int -> int list * int list = <fun>

# let rec tri_rapide_listes = function
| [] -> []
| [a] -> [a] (* facultatif, mais simplifie la trace *)
| a :: q -> let l, l' = partition q a in
            tri_rapide_listes l @ [a] @ tri_rapide_listes l';;
tri_rapide_listes : int list -> int list = <fun>

# trace "tri_rapide_listes";;
The function tri_rapide_listes is now traced.
- : unit = ()

# tri_rapide_listes [-3; 5; -2; 9; 3; -1; -2; 3; 12];;
tri_rapide_listes <- [-3; 5; -2; 9; 3; -1; -2; 3; 12]
tri_rapide_listes <- [5; -2; 9; 3; -1; -2; 3; 12]
tri_rapide_listes <- [9; 12]
tri_rapide_listes <- [12]

```

```
tri_rapide_listes → [12]
tri_rapide_listes ← []
tri_rapide_listes → []
tri_rapide_listes → [9; 12]
tri_rapide_listes ← [-2; 3; -1; -2; 3]
tri_rapide_listes ← [3; -1; -2; 3]
tri_rapide_listes ← [3]
tri_rapide_listes → [3]
tri_rapide_listes ← [-1; -2]
tri_rapide_listes ← []
tri_rapide_listes → []
tri_rapide_listes ← [-2]
tri_rapide_listes → [-2]
tri_rapide_listes → [-2; -1]
tri_rapide_listes → [-2; -1; 3; 3]
tri_rapide_listes ← []
tri_rapide_listes → []
tri_rapide_listes → [-2; -2; -1; 3; 3]
tri_rapide_listes → [-2; -2; -1; 3; 3; 5; 9; 12]
tri_rapide_listes ← []
tri_rapide_listes → []
tri_rapide_listes → [-3; -2; -2; -1; 3; 3; 5; 9; 12]
- : int list = [-3; -2; -2; -1; 3; 3; 5; 9; 12]
```

---

---

```
(* Tri fusion (on rappelle que la fusion a déjà été programmée) *)

# let rec tri_fusion t = let n = vect_length t in match n with
  | 0 | 1 -> t
  | -  -> let n' = n / 2 in
          fusion (tri_fusion (sub_vect t 0 n')) (tri_fusion (sub_vect t n' (n - n')));
tri_fusion : int vect -> int vect = <fun>

# trace "tri_fusion";;
The function tri_fusion is now traced.
- : unit = ()

# tri_fusion [|1; 3; 5; 1; 7; 4; 9; -1|];;
tri_fusion <- [|1; 3; 5; 1; 7; 4; 9; -1|]
tri_fusion <- [|7; 4; 9; -1|]
tri_fusion <- [|9; -1|]
tri_fusion <- [| -1|]
tri_fusion -> [| -1|]
tri_fusion <- [|9|]
tri_fusion -> [|9|]
tri_fusion -> [| -1; 9|]
tri_fusion <- [|7; 4|]
tri_fusion <- [|4|]
tri_fusion -> [|4|]
tri_fusion <- [|7|]
tri_fusion -> [|7|]
tri_fusion -> [|4; 7|]
tri_fusion -> [| -1; 4; 7; 9|]
tri_fusion <- [|1; 3; 5; 1|]
tri_fusion <- [|5; 1|]
tri_fusion <- [|1|]
tri_fusion -> [|1|]
tri_fusion <- [|5|]
tri_fusion -> [|5|]
tri_fusion -> [|1; 5|]
tri_fusion <- [|1; 3|]
tri_fusion <- [|3|]
tri_fusion -> [|3|]
tri_fusion <- [|1|]
tri_fusion -> [|1|]
tri_fusion -> [|1; 3|]
tri_fusion -> [|1; 1; 3; 5|]
tri_fusion -> [| -1; 1; 1; 3; 4; 5; 7; 9|]
- : int vect = [| -1; 1; 1; 3; 4; 5; 7; 9|]
```

---

---

```

(* Tri rapide *)

(* La fonction pivot compare les éléments de t à son terme initial,
les répartit dans un tableau, et précise en outre
l'indice où se trouve désormais le pivot *)

# let pivot t ind = let a = t.(ind) and n = vect_length t in
  let j = ref 1 and k = ref (n - 1) and res = make_vect n 0 in
  res.(0) <- a;
  (* on insère les éléments de t dans res selon leur relation à a *)
  for i = 0 to n - 1 do
    if i <> ind then if t.(i) < a then (res.(!j) <- t.(i); j := !j + 1;)
    else (res.(!k) <- t.(i); k:= !k - 1;)
  done;
  echange res 0 !k; (* on replace a à sa place *)
  res,!k;;
pivot : int vect -> int -> int vect * int = <fun>

# pivot [|3; 5; 6; 1; 2; 4; 9|] 3;;
- : int vect * int = [|1; 9; 4; 2; 6; 5; 3|], 0

(* Fonction fusion_triple pour améliorer la lisibilité *)

# let fusion_triple t t' t'' = fusion (fusion t t') t'';;
fusion_triple : int vect -> int vect -> int vect -> int vect = <fun>

```

---

---

```
(* Fonction tri_rapide suivant l'algorithme à la lettre *)

# let rec tri_rapide t = let n = vect_length t in
  if n <= 1 then t
  else
    let tab, i = pivot t 0 in match i with
    | 0 -> fusion [|tab.(0)|] (tri_rapide (sub_vect tab 1 (n - 1)))
    | k when k = n - 1 -> fusion (* cas de filtrage n - 1 non valide *)
      (tri_rapide (sub_vect tab 0 (n - 1)))
      [|tab.(n - 1)|]
    | _ -> fusion_triple
      (tri_rapide (sub_vect tab 0 i))
      [|tab.(i)|]
      (tri_rapide (sub_vect tab (i + 1) (n - i - 1)));
tri_rapide : int vect -> int vect = <fun>

# trace "tri_rapide";;
The function tri_rapide is now traced.
- : unit = ()

# tri_rapide [|4; 2; 1; 8; 4; 7; 10; 11; -3; 8; 12|];;
tri_rapide <- [|4; 2; 1; 8; 4; 7; 10; 11; -3; 8; 12|]
tri_rapide <- [|12; 8; 11; 10; 7; 4; 8|]
tri_rapide <- [|8; 8; 11; 10; 7; 4|]
tri_rapide <- [|10; 11; 8|]
tri_rapide <- [|11|]
tri_rapide -> [|11|]
tri_rapide <- [|8|]
tri_rapide -> [|8|]
tri_rapide -> [|8; 10; 11|]
tri_rapide <- [|4; 7|]
tri_rapide <- [|7|]
tri_rapide -> [|7|]
tri_rapide -> [|4; 7|]
tri_rapide -> [|4; 7; 8; 8; 10; 11|]
tri_rapide -> [|4; 7; 8; 8; 10; 11; 12|]
tri_rapide <- [| -3; 2; 1|]
tri_rapide <- [|1; 2|]
tri_rapide <- [|2|]
tri_rapide -> [|2|]
tri_rapide -> [|1; 2|]
tri_rapide -> [| -3; 1; 2|]
tri_rapide -> [| -3; 1; 2; 4; 4; 7; 8; 8; 10; 11; 12|]
- : int vect = [| -3; 1; 2; 4; 4; 7; 8; 8; 10; 11; 12|]
```

---



Quatrième partie

**Structures de données et algorithmes**



## Structures de données

## 1. NOTION DE STRUCTURE DE DONNÉES

## 1.1. ALGORITHMS + DATA STRUCTURES = PROGRAMS

Comment convertir un algorithme en un programme dans un langage donné? Bien évidemment, il nous faut connaître les rudiments de ce langage. Un aspect à ne pas négliger est celui de la *structure de données* employée : choisir entre une liste chaînée et un tableau peut changer beaucoup d'aspects du programme, comme les complexités spatiale et temporelle du programme, sa limitation (pour initialiser un tableau, on doit en connaître la taille, par exemple). Cela dépend aussi du style de notre programme, récursif, fonctionnel, impératif, etc.

En fait, plutôt que de chercher la structure de données adaptée à notre algorithme, il serait plus judicieux de réfléchir en amont à la structure de données adaptée à notre problème, puis d'écrire les algorithmes correspondants. On doit donc d'emblée se poser la question de la modélisation du sujet étudié : comment représenter un entier, un réel, un produit cartésien, une application, un chemin, etc.

Au fil de l'essor de la science informatique, des structures de données classiques et fondamentales ont émergé : cette année, outre les listes et les tableaux, nous étudierons les piles, files, dictionnaires, files de priorité, et les arbres binaires.

Le fait d'avoir un nombre restreint de structures de données fondamentales permet de les employer de manière collective, sans avoir besoin de réinventer la roue à chaque fois.

On peut pousser plus loin cette volonté de généralité, de s'extraire le plus possible d'un langage, d'une machine, et d'une implémentation précise, en définissant la notion de *structure de données abstraite* (en anglais : ADT, pour abstract data structure), comme la donnée d'un type muni d'opérations. Cette généralité permet d'écrire des programmes pour ainsi dire identiques dans tous les langages (hormis quelques variations syntaxiques), du moment que les structures de données ont été implémentées selon les règles imposées.

Grâce aux bibliothèques de l'environnement de programmation, on peut même utiliser des structures de données avant d'avoir programmé leur réalisation concrète. Par exemple, il existe un module « stack » en Caml, permettant de travailler sur les piles.

Une fois une telle structure abstraite définie, il faudra réfléchir à la manière de les représenter en machine, au moyen d'une *structure de données concrète* (ou d'une *implémentation*). Les exemples concrets de réalisation de ces structures de données en seront appelées des *instances*.

## 1.2. STRUCTURES DE DONNÉES PERSISTANTES ET IMPÉRATIVES

Parmi les problématiques possibles, figure celle de la persistance : une structure de données dite *persistante* (ou *immuable*) si ses instances n'admettent pas plusieurs états. C'est par exemple le cas des listes en Caml : aucun effet de bord ne permettra de modifier une liste. En revanche, on peut « modifier » une liste, ou plutôt l'identifiant d'une liste, en la redéclarant :

---

```
# let a = [1; 2; 3];;
a : int list = [1; 2; 3]

# 0 :: a;;
- : int list = [0; 1; 2; 3]

# a;; (* a n'a pas été modifiée *)
- : int list = [1; 2; 3]

# let a = 0 :: a;;
a : int list = [0; 1; 2; 3]
```

---

Si une structure de données n'est pas persistante, elle est dite *impérative* (ou *modifiable*, ou *mutable*). C'est le cas des vecteurs en Caml :

---

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

# v.(0) <- 0;;
- : unit = ()

# v;; (* v a été modifié *)
- : int vect = [|0; 1; 2; 3|]
```

---

Dans la programmation fonctionnelle (où les programmes sont des suites d'appels de fonctions), on privilégie les structures persistantes. Comme on peut s'y attendre, on privilégie les structures impératives dans la programmation impérative.

Parmi les avantages des structures de données persistantes, on peut citer la sûreté, dans le sens où les effets de bord sont interdits, et donc où aucune opération cachée n'a modifié notre instance.

Les structures de données impératives permettent parfois un gain de temps et d'espace considérables. Par exemple, la concaténation de deux listes chaînées ne prend pas de place en style mutable, et est beaucoup plus coûteuse en style persistant.

### 1.3. DÉFINITION D'UNE STRUCTURE DE DONNÉES : SIGNATURE ET SÉMANTIQUE

Les fonctions de manipulation des structures de données sont de trois classes : les *constructeurs*, les *prédicats*, et les *fonctions de sélection*.

La précision, pour une structure de données précise, de ces fonctions en définit la *signature*. Cette signature serait sans grand intérêt si on ne connaissait pas le rôle précis des fonctions de manipulation : c'est l'objet de la *sémantique*.

Les structures de données, le plus souvent récursives/inductives, dictent les programmes qui les manipulent, comme nous l'avons déjà observé sur les listes.

De même, pour prouver des programmes utilisant des structures de données inductives, on utilisera souvent une démonstration par induction structurelle.

## 2. LISTES

### 2.1. DÉFINITION

Soit  $E$  un ensemble. On définit une suite d'ensembles  $(E_n)$  par  $E_0 = \{\emptyset\}$  et  $E_{n+1} = E \times E_n$ . On appelle *liste d'éléments de  $E$*  un élément de  $\mathcal{L}(E) = \cup E_n$ .

Par exemple,  $(3, (2, (5, \emptyset)))$  est une liste.

**Remarque :** on pourrait bien sûr confondre cette dernière liste avec  $(3, 2, 5)$ , mais il faut bien comprendre que l'accès à 3 est plus rapide que l'accès à 2 ou 5, comme nous n'avons cessé de le répéter pour les listes chaînées.

**Remarque :** bien que dans cet exemple, il ne soit pas simple de définir  $\mathcal{L}(E)$  par induction, il faut comprendre en quoi cette structure est néanmoins inductive. Pour prouver qu'un résultat est vrai pour toute liste, on prouvera qu'il l'est pour la liste vide, et qu'il reste vrai par adjonction d'un élément en tête de liste.

On peut définir les fonctions de sélection Tête et Queue, dont la sémantique est claire.

En informatique, la définition d'une liste pourrait être

$$Liste = nil + Element \times Liste$$

exprimant qu'une liste est vide (*nil*) ou s'écrit  $(a, q)$ , où  $a$  est un élément,  $q$  une liste.

En Caml, ces listes sont déjà implémentées, par les listes chaînées (le type `'a list`). Comme dit plus haut, il s'agit d'une structure de données persistante.

**Remarque :** en mémoire, la liste  $[1; 3; 8]$  par exemple, qui peut s'écrire  $1 :: 3 :: 8 :: []$ , est représentée par un premier couple  $(1, a)$  qui donne la valeur en tête de la liste, et un pointeur  $a$  pour déterminer où se trouve la valeur suivante, etc.

**Remarque :** le « cons » `::` est un constructeur<sup>1</sup> de la structure de données liste, contrairement à la concaténation `@`, c'est pourquoi on évitera si possible cette dernière, au profit du premier (qui est accepté en filtrage). En particulier, l'adjonction en queue de liste est bien plus coûteuse que l'adjonction en tête de liste.

**Remarque :** dans les prédicats, il faudrait rajouter le test d'égalité, ce qui peut poser plus de problèmes qu'on ne le pense, lorsqu'un même objet peut être représenté de manières différentes.

---

1. Plus exactement, pour tout  $x \in E$ ,  $l \mapsto x :: l$  est un constructeur.

Si l'on voulait redéfinir le type liste en Caml, on pourrait écrire :

---

```
(* Définition d'un type liste *)

# type 'a Liste = Nil | Cons of 'a * ('a Liste);;
Type Liste defined.

(* Sémantique des fonctions de manipulation d'une liste *)

(* Prédicat testant si une liste est vide *)

# let est_vide = function
  | Nil -> true
  | _ -> false;;
est_vide : 'a Liste -> bool = <fun>

# est_vide (Cons (1, Nil));;
- : bool = false

(* Constructeur d'ajout en tête de liste *)

# let ajout a l = Cons (a, l);;
ajout : 'a -> 'a Liste -> 'a Liste = <fun>

# ajout 3.8 Nil;;
- : float Liste = Cons (3.8, Nil)

(* Fonctions de sélection *)

# let tete = function
  | Nil -> failwith "pas_de_tête_pour_la_liste_vide"
  | Cons (a, q) -> a;;
tete : 'a Liste -> 'a = <fun>

# let queue = function
  | Nil -> failwith "pas_de_tête_pour_la_liste_vide"
  | Cons (a, q) -> q;;
queue : 'a Liste -> 'a Liste = <fun>

(* Illustration d'un test d'égalité *)

# Cons (1, Cons (2, Nil)) = ajout 1 (Cons (2, Nil));;
- : bool = true
```

---

**Remarque :** en Caml, les fonctions tête et queue sont souvent escamotées par les possibilités de filtrage de ce langage, mais il ne faut pas oublier la structure de données sous-jacente.

## 2.2. PROGRAMMES FONDAMENTAUX SUR LES LISTES

Revoyons, dans le cadre du type Liste ci-dessus, comment redéfinir les fonctions qui à une liste associent respectivement sa longueur, son maximum, son n-ième élément.

---

```
(* Longueur, maximum, n-ième terme, concaténation *)

# let l = Cons (4, Cons(9, Cons(3, Nil)));;
l : int Liste = Cons (4, Cons (9, Cons (3, Nil)))

# let rec longueur = function
  | Nil -> 0
  | Cons (a, q) -> 1 + longueur q;;
longueur : 'a Liste -> int = <fun>

# longueur l;;
- : int = 3

# let rec maximum = function
  | Nil -> failwith "pas_de_maximum_pour_la_liste_vide"
  | Cons (a, Nil) -> a
  | Cons (a, q) -> max a (maximum q);;
maximum : 'a Liste -> 'a = <fun>

# maximum l;;
- : int = 9

# let rec nieme_terme n l = match l with
  | Nil -> failwith "pas_de_nieme_terme"
  | Cons (a, q) -> match n with
    | 1 -> a
    | _ -> nieme_terme (n - 1) q;;
nieme_terme : int -> 'a Liste -> 'a = <fun>

# for i = 1 to 4 do print_int (nieme_terme i l); print_newline (); done;;
4
9
3
Uncaught exception: Failure "pas_de_nieme_terme"

# let rec concat l l' = match l with
  | Nil -> l'
  | Cons (a, q) -> Cons (a, concat q l');;
concat : 'a Liste -> 'a Liste -> 'a Liste = <fun>

# let l' = Cons(1, Cons(7, Cons(10, Nil)));;
l' : int Liste = Cons (1, Cons (7, Cons (10, Nil)))

# concat l l';;
- : int Liste =
  Cons (4, Cons (9, Cons (3, Cons (1, Cons (7, Cons (10, Nil))))))
```

---

## 3. PILES

## 3.1. MODÈLE DE DONNÉES, IMPLÉMENTATION

Une *pile* est un modèle de données, dite structure LIFO (last in, first out), dont la signature est donnée par les constructeurs `Crée_pile_vider` et `Empiler`, les fonctions de sélection sont `Sommet` et `Dépile`, le prédicat `Est_vider` (et éventuellement `Egalite`).

On ne peut ajouter ou supprimer des éléments qu'au sommet de la pile.

**Remarque :** on ne peut dépiler un élément que si on a déjà dépilé les éléments qui sont « au-dessus ». Cela se produit en sens inverse, et c'est un peu ce que nous avons fait pour l'image miroir.

Le modèle pile est déjà implémenté dans Caml, sous le nom `stack` :

---

(\* Le type `stack` donné par Caml \*)

```
# #open "stack";;
```

```
# let l = new ();;
l : 'a t = <abstr>
```

```
# pop l;;
Uncaught exception: Empty
```

```
# push 3 l;;
- : unit = ()
```

```
# push 4 l;;
- : unit = ()
```

```
# l;;
- : int t = <abstr>
```

```
# pop l;;
- : int = 4
```

```
# l;;
- : int t = <abstr>
```

```
# pop l;;
- : int = 3
```

---

(\* On dispose aussi des fonctions `clear`, `length`, mais pas de `top` \*)

**Remarque :** `push` et `pop` agissent par effets de bord sur la pile passée en argument. La structure de pile est ici implémentée dans un style impératif.

Comme exemples de piles, on peut citer l'historique dans un navigateur ou dans un logiciel de retouche d'images, les expressions parenthésées, comme les expressions algébriques infixes (où chaque opérateur est entre ses arguments).

**Remarque :** le modèle de données abstrait pile est isomorphe à celui des listes (mais les implémentations en Caml diffèrent, puisque l'une est persistante tandis que l'autre est mutable).

Pour implémenter la structure de données abstraite de pile, on peut penser à

---

(\* Représentation par une liste \*)

(\* Par "abréviation" \*)

```
# type 'a pile = 'a list;;
Type pile defined.
```

(\* Par structure inductive \*)

```
# type 'a pile = Pile_Vide | Empile of 'a * 'a pile;;
Type pile defined.
```

```
(* Par un type mutable *)
```

```
# type 'a pile = {mutable contenu : 'a list}
Type pile defined.
```

```
(* Représentation par un vecteur *)
```

```
# type 'a pile = { contenu : 'a vect; mutable sommet : int };;
Type pile defined.
```

---

Nous allons utiliser l'implémentation par un type de contenu une liste mutable :

---

```
(* Structure de données pile *)
```

```
# type 'a pile = {mutable contenu : 'a list };;
Type pile defined.
```

```
# let pile_vide () = {contenu = [] };;
pile_vide : unit -> 'a pile = <fun>
```

```
# let est_vide pile = pile.contenu = [];;
est_vide : 'a pile -> bool = <fun>
```

```
# est_vide {contenu = [5; 6] };;
- : bool = false
```

```
# est_vide (pile_vide ());;
- : bool = true
```

```
# let empile a pile = pile.contenu <- a :: pile.contenu;;
empile : 'a -> 'a pile -> unit = <fun>
```

```
# let pile = pile_vide ();;
pile : 'a pile = {contenu = []}
```

```
# empile 3 pile;;
- : unit = ()
```

```
# empile 9 pile;;
- : unit = ()
```

```
# pile;;
- : int pile = {contenu = [9; 3]}
```

```
# let depile pile = match pile.contenu with
  | [] -> failwith "Pile_vide"
  | a :: q -> pile.contenu <- q;
  a;;
depile : 'a pile -> 'a = <fun>
```

```
# depile pile;; (* On renvoie le sommet de notre pile *)
- : int = 9
```

```
# pile;; (* depile a agi par effet de bord sur notre pile *)
```



```

- : int pile = {contenu = [3]}

# depile pile;;
- : int = 3

# pile;;
- : int pile = {contenu = []}

# depile pile;;
Uncaught exception: Failure "Pile_vide"

# pile;;
- : int pile = {contenu = []}

```

---

### 3.2. PILE DE RÉCURSIVITÉ

Les appels et retours d'une fonction récursive ou même de plusieurs sont gérés par une pile d'exécution. On peut suivre ce processus grâce à la fonction `trace`.

```

# let rec fibo = function
    | 0 | 1 -> 1
    | n -> fibo (n - 2) + fibo (n - 1);;
fibo : int -> int = <fun>

# trace "fibo";;
The function fibo is now traced.
- : unit = ()

# fibo 4;;
fibo <- 4
fibo <- 3
fibo <- 2
fibo <- 1
fibo -> 1
fibo <- 0
fibo -> 1
fibo -> 2
fibo <- 1
fibo -> 1
fibo -> 3
fibo <- 2
fibo <- 1
fibo -> 1
fibo <- 0
fibo -> 1
fibo -> 2
fibo -> 5
- : int = 5

```

---

### 3.3. EXPRESSIONS ALGÈBRIQUES POSTFIXÉES

Nous sommes habitués à la représentation infixe des expressions algébriques :

$$(5.2 - ((4.1 + 8.9) \times (5.2/9.3))) \times (3.3 - 6)$$

On peut définir inductivement la notion d'expression algébrique.

(\* Définition d'un type expression algébrique \*)

```

# type exp_alg =
    | Var of float

```

```

| Plus of exp_alg * exp_alg
| Moins of exp_alg * exp_alg
| Mult of exp_alg * exp_alg
| Divise of exp_alg * exp_alg;;
Type exp_alg defined.

```

```

# let exemple = Mult
    (Moins
      (
        Var 5.2,
        Mult
          (
            Plus (Var 4.1, Var 8.9),
            Divise (Var 5.2, Var 9.3)
          )
        ),
      Moins (Var 3.3, Var 6.)
    )
);;
exemple : exp_alg =
Mult
  (Moins (Var 5.2, Mult (Plus (Var 4.1, Var 8.9), Divise (Var 5.2, Var 9.3))),
    Moins (Var 3.3, Var 6.0))

```

La représentation la plus naturelle de cette structure inductive est arborescente (nous reviendrons sur les arbres ultérieurement). Pour effectuer une représentation linéaire d'un arbre, on peut le parcourir en profondeur, en mettant l'étiquette d'un nœud quand on y arrive la première fois<sup>2</sup> (écriture *préfixe* ou *préfixée*) ou la troisième fois (forme *postfixe*, ou *postfixée*, ou *polonaise inverse*). Ces écritures ont l'avantage de ne pas nécessiter de parenthèses, contrairement à l'écriture infixe.

## 4. FILES

### 4.1. LA STRUCTURE DE DONNÉES ABSTRAITE

Il existe une structure proche des piles, celle de *file*, fondée sur le principe FIFO (first in, first out).

La signature est constituée des constructeurs Créé\_File\_Vide et Enfile, les fonctions de sélections sont Défile, le prédicat est Est\_vider (et éventuellement Egalité et Est\_Pleine).

Lorsqu'un élément est enfile, il prend place en queue de file. L'élément défilé est toujours celui en tête de file.

Cette structure de file modélise bien la notion de file d'attente, que l'on retrouve par exemple dans le tampon d'un clavier, de la liste des fichiers envoyés à une imprimante, une file d'attente à un guichet, etc.

On pourrait implémenter les files à l'aide d'une unique liste, mais Défile (ou insérer si on la prend à l'envers) prendrait un temps linéaire, ce qui n'est pas raisonnable. Nous allons proposer des réalisations impérative et persistante des files, plus efficaces.

### 4.2. STRUCTURE IMPÉRATIVE DE FILE

On peut réaliser la structure impérative de file à l'aide d'un tableau : il faut d'abord se fixer une taille  $N$  maximale pour nos files, puis représenter une file par la donnée d'un tableau  $F$  de taille  $N + 1$ , et de deux indices  $tete(F)$  et  $queue(F)$ , donnant respectivement la tête et la queue de la file. En fait, on prend une structure circulaire de tableau (après l'indice  $N$ , on repasse à l'indice 0) afin de permettre un nombre quelconque d'opérations sur la file (tant que sa taille ne dépasse pas  $N$ ).

Le *successeur* d'un entier  $n \in \llbracket 0, N \rrbracket$  sera  $n + 1$  si  $n \neq N$ , et 0 si  $n = N$ .

Quand  $tete(F) = queue(F)$ , la file est vide. Au départ,  $tete(F) = queue(F) = 0$ . Quand on enfile un élément,  $queue(F)$  passe à son successeur. Quand on défile un élément,  $tete(F)$  passe à son successeur.

En raison de la structure circulaire de tableau, il se peut fort bien que  $tete(F) > queue(F)$ , même pour une file non vide.

Il y a débordement négatif de la file quand on veut défiler une file vide.

Il y a débordement positif quand on enfile une file de taille  $N$ , *i.e.* quand  $tete(F)$  est le successeur de  $queue(F)$ .

2. C'est comme cela qu'on l'a rentré dans l'exemple ci-dessus.

Bien sûr, le coût de chaque opération est en  $O(1)$ .

#### 4.3. STRUCTURE PERSISTANTE DE FILE

L'idée, pour réaliser une structure persistante de file, consiste à s'aider de deux listes `file_bas` et `file_haut`, de telle sorte que la file corresponde à la concaténée de `file_bas` avec l'*inverse* de `file_haut` : ainsi, `Enfile` a toujours un coût constant, et `Défile` a un coût constant, sauf dans le cas exceptionnel où `file_haut` est vide. Dans ce dernier cas, on effectuera une opération `Déversement`, qui transférera tous les éléments de `file_bas` dans `file_haut` (dans le bon ordre). Cette opération est certes coûteuse, mais elle ne se produit que rarement, de sorte que la complexité de `Défile` reste acceptable.

#### 4.4. FILES DE PRIORITÉ

La notion de file de priorité généralise à la fois celles de pile et de file : une file de priorité consiste en la donnée d'un ensemble d'éléments, chacun d'eux ayant une priorité associée : la signature comprend les constructeurs `Créer_File_de_Priorité_Vide`, `Insère`, le sélecteur `Supprime_Max`, et les prédicats `Est_File_de_Priorité_Vide` et `Egalité`.

`Supprime_Max` renvoie l'élément de plus grande priorité (ou l'un d'entre eux si plusieurs ont une priorité maximale), et le supprime de la file passée en argument.

Pour les piles (resp. les files), la priorité est la date d'insertion (resp. le temps écoulé depuis l'insertion).

La notion de file de priorité permet de trier : il suffit de prendre pour priorité d'un élément sa valeur.

Ainsi, si une file de priorité offre des opérations d'ajout et de retrait de coûts logarithmiques (ce qui est possible), alors on en déduit un tri en  $O(n \log(n))$  : c'est ce qu'on appelle le *tri par tas*.

## 5. ARBRES

## 5.1. DÉFINITION INDUCTIVE DU TYPE ARBRE BINAIRE

Le modèle de données arbre binaire (étiqueté et enraciné) peut être résumé par la formule suivante :

$$\text{Arbre} = \text{nil} + \text{Arbre} \times \text{Element} \times \text{Arbre}$$

ce qui se lit ainsi : un arbre est vide, ou se présente sous la forme d'un sommet  $R$  étiqueté par un élément, duquel partent éventuellement des arcs gauche et droite, reliant chacun  $R$  à un sous-arbre (gauche ou droite).

Un arbre peut être représenté par des *nœuds* (ou *sommets*) reliés par des *arcs* (ou *flèches*, ou *arêtes*). Étant donné une flèche, on a un *fil* (gauche ou droit) et son *père*. La *racine* est le seul sommet sans père. Un sommet sans fils est une *feuille*. Un nœud qui n'est pas une feuille est dit *intérieur*.

Les *descendants* d'un nœud sont ses fils, et les descendants de ses fils. Les *ancêtres* d'un nœud sont les nœuds trouvés sur l'unique chemin entre ce nœud et la racine. Les descendants d'un nœud  $N$  sont donc exactement les nœuds qui admettent  $N$  pour ancêtre.

Un arbre binaire est dit *complet* si chaque nœud intérieur a exactement deux fils.

**Remarque :** les arbres ainsi définis sont usuellement représentés racine en haut. Les arcs d'un arbre peuvent naturellement être orientés, puisque l'arbre est enraciné, mais ce n'est pas une nécessité.

**Remarque :** la structure inductive de cette notion d'arbre binaire fait que nous donnerons des preuves par induction structurelle. De même, nous définirons en Caml une fonction sur les arbres de manière inductive.

**Remarque :** il existe des définitions plus générales de la notion d'arbre : on peut ne pas imposer qu'un sommet soit distingué en tant que racine, on peut enlever la condition qu'un père ait au plus deux fils. Cette notion étendue permet alors par exemple de représenter les arbres généalogiques ou phylogénétiques.

**Remarque :** on peut même voir la notion d'arbre comme cas particulier de celle de graphe : un arbre est un graphe non orienté, acyclique (sans cycle) et connexe (d'un seul tenant : on peut toujours relier deux sommets par un chemin). D'ailleurs, si on enlève la condition de connexité, cela définit une *forêt*.

La notion d'arbre est fondamentale en informatique théorique. On rencontre notamment les arbres représentant des expressions arithmétiques ou des propositions logiques, ou les arbres de décision. De plus, la notion d'arbre binaire généralise considérablement celle de liste : on peut voir les listes comme les arbres binaires pour lesquels chaque fils gauche est une feuille. Cependant, ces deux structures sont différentes dans leur approche (linéaire pour les listes, pas pour les arbres), et dans leurs applications.

La *profondeur* (ou *hauteur*) d'un nœud est la longueur du chemin (en nombre d'arêtes) depuis la racine jusqu'à ce nœud. La *hauteur* d'un arbre est la plus grande des hauteurs de ses sommets.

On définit les *bords gauche* (resp. *droit*) d'un arbre comme le plus long chemin depuis la racine en ne suivant que les fils gauches (resp. droits)

On mesure souvent un arbre par sa *taille*, *i.e.* le nombre de ses sommets, ou par le nombre de ses feuilles.

## 5.2. PARCOURS D'UN ARBRE

On distingue deux types de parcours des nœuds d'un arbre binaire :

- le *parcours en largeur* commence à la racine, puis explore ses éventuels fils (les sommets de hauteur 1), puis les fils de ces fils (les sommets de hauteur 2), etc.
- le *parcours en profondeur* commence à la racine, puis explore en profondeur son sous-arbre gauche, puis son sous-arbre droit.

Il y a trois types principaux de traitements lors d'un parcours en profondeur : préfixe (lorsqu'on passe à gauche d'un sommet), infixé (lorsqu'on passe dessous), et postfixé (lorsqu'on passe à droite).

## 5.3. ARBRES BINAIRES DE RECHERCHE

Un *arbre binaire de recherche* (en abrégé, un ABR) est un arbre binaire enraciné étiqueté tel que tout nœud ait une étiquette

- plus grande que les étiquettes des nœuds de son sous-arbre gauche.
- plus petite que les étiquettes des nœuds de son sous-arbre droit.

**Remarque :** pour un ABR, on parle plus souvent de *clés* que d'étiquettes.

**Remarque :** un ABR n'est donc rien d'autre qu'un arbre dont le traitement infixé en profondeur des nœuds fournit une suite strictement croissante. Il est donc très facile de trouver le minimum et le maximum d'un ABR, ou encore de chercher un élément dans un arbre.

La complexité des différentes opérations est (dans le pire des cas) en  $h(a)$ , donc logarithmique en la taille de  $a$  dans le cas où l'arbre est équilibré.

## 5.4. STRUCTURE DE TAS

On appelle *tas (binaire)* tout arbre binaire tel que :

- (1) (arbre binaire parfait) tous les niveaux sont complets, à l'exception possible du dernier, que l'on remplit de gauche à droite
- (2) (propriété de tas) la clé de chaque nœud est supérieure ou égale à celles de ses (éventuels) fils.

Ainsi, grâce à la première condition, un tas de taille  $n$  a pour hauteur  $\lceil \log_2(n) \rceil$ .

**Insertion** – L'insertion dans un tas a un coût logarithmique, en procédant ainsi :

- On place l'élément  $x$  à insérer à la première position libre permettant de respecter les deux premières propriétés des tas (concrètement, si le dernier niveau n'est pas rempli, on place l'élément à insérer juste à droite de la dernière feuille de ce niveau, et sinon, on l'insère en tant que fils gauche de la feuille du bord gauche).
- Ensuite, on effectue tant qu'on le peut une *percolation vers le haut*, *i.e.* si  $x$  est plus grand que son père  $p$ , on échange  $x$  et  $p$  : il y a au plus  $h$  échanges, d'où la complexité logarithmique.

**SupprimeMax** – Pour réaliser `Supprime_Max`, on procède ainsi :

- On renvoie la racine, et on la remplace par le dernier élément  $x$  du tas (*i.e.* le plus à droite du dernier niveau).
- Tant que  $x$  est inférieur à l'un de ses fils, on échange  $x$  et un tel fils (procédé de *percolation vers le bas*), et on obtient ainsi un tas.

Pour la même raison que précédemment, le coût est logarithmique.

Il est clair que la structure de tas permet une réalisation de la structure persistante de file de priorité, et qu'elle fournit donc un algorithme de tri (fondé sur les comparaisons) en temps quasi-linéaire  $n \log(n)$ .

## 6. DICTIONNAIRES

La structure de données *dictionnaire* consiste en la donnée d'un ensemble de valeurs, référencées par des *clés*. Les données associées à une clé sont dites *satellites*.

Nous voulons insérer des éléments, supprimer des éléments, ou rechercher la présence d'un élément. On rajoute parfois la modification d'une valeur. En général, l'ensemble des clés est totalement ordonné : nous nous plaçons dans cette situation.

On définit donc les constructeurs `Créé_Dictionnaire_Vide`, `Insère`, le sélecteur `Supprime` et le prédicat `Est_Présent`.

## 6.1. RÉALISATION D'UN DICTIONNAIRE À L'AIDE D'UN TABLEAU

On peut toujours réaliser la structure impérative de dictionnaire à l'aide d'un tableau si on connaît à l'avance un majorant pour le nombre de termes du dictionnaire, chaque élément du tableau consistant la donnée d'une clé et des les données satellites. L'insertion a un coût constant, mais la recherche un coût linéaire (dans le pire des cas).

On peut aussi imaginer, si l'ensemble des clés possibles est de cardinal  $N$ , construire un tableau de taille  $N$  et, quitte à faire une correspondance bijective entre les clés et les entiers de 0 à  $N - 1$ , remplir les cases correspondant à des clés effectivement utilisées. Cependant, cette approche est inutilement coûteuse en espace si en pratique le nombre de clés effectivement utilisées est très petit par rapport à  $N$ .

## 6.2. RÉALISATION D'UN DICTIONNAIRE À L'AIDE D'UN ABR

Dans le cas (qui est le plus fréquemment rencontré) où l'ensemble des clés est totalement ordonné, on peut réaliser la structure persistante de dictionnaire à l'aide d'un arbre binaire de recherche, en étiquetant par les clés. On obtient ainsi une bien meilleure complexité que précédemment.

## 6.3. RÉALISATION D'UN DICTIONNAIRE À L'AIDE D'UNE TABLE DE HACHAGE

La notion de table de hachage se fonde sur l'idée que l'indice d'un terme du dictionnaire n'est pas nécessairement sa clé, il peut aussi être une *fonction* de cette clé. L'intérêt réside en ce que cette fonction n'est pas nécessairement injective, et donc que le nombre d'indices peut être bien inférieur au nombre de clés possibles, mais beaucoup plus proche du nombre de clés réellement utilisées.

Par exemple, le nombre de pseudos possibles des différents utilisateurs d'un forum est très grand (si par exemple on se limite à des mots de passe d'au plus 20 caractères construits sur un alphabet de 26 lettres, on obtient bien plus que  $10^{20}$  mots de passe possibles), mais en pratique, le nombre réel d'utilisateurs d'un forum est beaucoup plus petit.

L'idée consiste donc à introduire une fonction dite de *hachage*

$$h : \llbracket 1, N \rrbracket \rightarrow \llbracket 1, m \rrbracket,$$

où  $m \ll N$ .

On stocke ensuite les données dans une *table de hachage*, qui est un tableau de taille  $m$ .

Un problème évident se pose, celui de la *collision* : puisque  $m \ll N$ ,  $h$  n'est pas injective, et deux clés distinctes peuvent avoir même image par  $h$ . On peut résoudre les collisions par *chaînage* : dans le tableau de taille  $m$ , on peut stocker en un indice  $k$  la liste chaînée des clés  $c$ , effectivement utilisées, telles que  $h(c) = k$ .

Bien sûr, si on n'a pas de chance, la table de hachage ne sera remplie qu'en quelques indices, et le chaînage provoquera des recherches coûteuses (de la longueur des listes dans le pire des cas). Il faut donc bien choisir sa table de hachage, selon les heuristiques suivantes :

- (1) Une fonction de hachage doit bien « mixer » (d'où son nom expressif). Si  $p$  et  $q$  sont proches, alors  $h(p)$  et  $h(q)$  sont éloignés.
- (2) Une fonction de hachage doit bien répartir. Tous les entiers de 1 à  $m$  doivent avoir environ  $N/m$  antécédents par  $h$ .
- (3) Une fonction de hachage doit être rapide.

Une façon classique de hacher consiste à transformer la clé en un entier naturel, puis à utiliser l'arithmétique modulaire (à un entier, on associe son reste dans la division euclidienne par un entier fixé  $p$ ).

**Remarque :** les fonctions de hachage sont aussi utiles pour vérifier l'intégrité d'un message ou d'un fichier transmis à un tiers (somme de contrôle, *i.e.* les fameux *checksum* en anglais).

## Cinquième partie

### Exercices





## Premiers pas en Caml

## Exercice 1 (Simple calcul)

Calculer  $\sqrt{1 + \sqrt{17}}$ .

## Exercice 2 (Utilisation d'identificateurs locaux)

- 1 Calculer  $\frac{\cos(\ln(3)) + \sin(\ln(2))}{\cos^3(\ln(3)) - \sin^2(\ln(2))}$ .
- 2 Calculer  $12^{(\ln(12)^{\ln(\ln(12))})}$  par déclarations emboîtées.

## Exercice 3 (Autour de fonction)

- 1 Définir la fonction tangente hyperbolique.
- 2 Définir l'opération d'addition des entiers sous forme curryfiée avec fonction.

## Exercice 4 (Recherche de type)

- 1 Donner le type de 2, 4, de 2., 2, et de (3, ('a', 4.5), 3).
- 2 Donner le type des fonctions `let evaluer_en_2 f = f 2` et `evaluer x f = f x`

## Exercice 5 (Structure de groupe produit)

Proposer une fonction curryfiée qui à deux lois de composition interne associe la loi produit.

## Exercice 6 (Ordre lexicographique)

Proposer une fonction permettant de comparer deux couples d'entiers pour l'ordre lexicographique.

## Exercice 7 (Filtrage)

- 1 Programmer le `et` logique et l'implication sous forme curryfiée par filtrage.
- 2 Reconstruire la structure de contrôle conditionnelle (`if ... then ... else`) par filtrage.

Exercice 8 (Taux d'accroissement)

0

Définir une fonction taux de type  $(\text{float} \rightarrow \text{float}) \rightarrow \text{float} \rightarrow \text{float} \rightarrow \text{float}$  associant le taux d'accroissement de  $f$  entre  $x$  et  $y$ .

Exercice 9 (Composition)

2

1 Définir l'opération de composition  $\circ$ , que l'on peut mettre sous forme infixe avec la directive `#infix "o"` (il faut mettre le dièse).

2 Profitez-en pour redéfinir la fonction qui à  $f$  associe  $x \mapsto f(x + 2) + 2$ .

Exercice 10 (Curryfication)

1

Définir des fonctions de curryfication et de décurryfication.

Exercice 11 (Détermination compliquée d'un type)

4

Donner le type des fonctions `let p x y = y x x` et `let p' y = p p y`.

## Premiers pas en récursivité et itération

### Exercice 1 (Références)

Que va produire la séquence suivante ?

```
let a = "cou";;  
let b = a ^ a;;  
a.[1] <- 'r';;  
b;;
```

### Exercice 2 (Réécriture de fonctions simples)

Réécrire les fonctions `hd` `tl` `@` `mem` `index`.

### Exercice 3 (Somme de termes)

- 1 Écrire une fonction de type `int vect -> int` renvoyant la somme des termes d'un vecteur d'entiers.
- 2 Écrire une fonction de type `int list -> int` renvoyant la somme des termes d'une liste d'entiers.

### Exercice 4 (Fonction puissance)

Écrire une fonction puissance curryfiée prenant en arguments deux entiers naturels  $a$  et  $b$ , et renvoyant  $a^b$  :

- 1 De manière impérative.
- 2 De manière récursive.

### Exercice 5 (Accès à un terme d'une liste)

Écrire une fonction `acces` telle que `acces liste i` renvoie le  $i$ -ème terme de liste.

### Exercice 6 (Enlever le dernier liste d'une liste)

Écrire une fonction qui ôte le dernier terme d'une liste.

### Exercice 7 (Éviter les répétitions)

Écrire une fonction qui supprime les répétitions consécutives dans une liste.

### Exercice 8 (Ajout de zéros)

- 1 Écrire la fonction qui ajoute un zéro entre deux termes d'une liste.
- 2 Écrire la fonction qui ajoute un zéro entre les deux premiers termes d'une liste, puis deux entre le deuxième et le troisième, etc. (elle laisse invariant une liste d'au plus un terme).

### Exercice 9 (Recherche dans un tableau)

- 1 Écrire une fonction recherche de type `'a vect -> 'a -> bool` qui détermine la présence d'un élément dans un tableau.
- 2 Écrire à nouveau une telle fonction qui rattrape l'exception `Invalid_argument "vect_item"` et une exception trouve que vous aurez préalablement définie.

## Itération

### Exercice 1 (Maximum d'un tableau)

- 1 Définir une fonction `max_tableau` prenant en argument un tableau d'éléments comparables pour `<`, renvoyant son plus grand élément.
- 2 Prouver sa correction.
- 3 Abstraire pour généraliser cet algorithme.

### Exercice 2 (Plus petit diviseur premier)

Écrire une fonction qui à un entier  $n \geq 2$  associe son plus petit diviseur premier. Prouver sa terminaison et sa correction.

---

```

let f n =
  let x = ref n and y = ref n in
    while not (!y = 0) do
      x := !x + 2;
      y := !y - 1;
    done;
  !x;;

```

---

### Exercice 3 (Preuve d'un programme itératif)

- 1 Que fait la fonction  $f$ ? Prouvez-le.
- 2 Que se passe-t-il si on remplace la ligne `x := !x + 2` par `x := !x * 2`? Quel invariant de boucle donneriez-vous dans ce cas?

### Exercice 4 (Bézout itératif)

- 1 Donner (et prouver la correction et terminaison d')une fonction impérative permettant le calcul du pgcd de deux entiers.
- 2 Donner (et prouver la correction et terminaison d')une fonction impérative donnant un couple de Bézout pour deux entiers.

### Exercice 5 (Algorithme de Hörner)

1 Écrire une procédure `evaluation_simple` de type `float vect -> float -> float` permettant d'évaluer en un réel  $x$  un polynôme  $p$  (à coefficients réels) dont on a entré les coefficients sous forme de tableau (l'indice de l'élément correspondant à l'indice du monôme).

2 Trouver un algorithme d'évaluation d'un polynôme en un point, utilisant moins d'opérations que le premier. C'est (peut-être) la méthode de Hörner.

---

```
let myst x y = let m = ref 0 and a = ref x and b = ref y in
  while !b > 0 do
    if !b mod 2 = 1 then m := !m + !a;
    a := !a * 2;
    b := !b / 2;
  done;
  !m;;
```

---

### Exercice 6 (Une fonction mystérieuse)

Que fait la fonction `myst`? Prouvez-le.

## Terminaison et correction des fonctions récursives

## Exercice 1 (Ordre produit)

0

Montrer que l'ordre produit sur  $\mathbb{N}^2$ , défini, pour tous  $(m, n), (m', n') \in \mathbb{N}^2$ , par

$$((m, n) \preceq (m', n')) \Leftrightarrow ((m \leq m') \wedge (n \leq n'))$$

est bien fondé.

```
# let rec morris = fun
  | 0 _ -> 1
  | m n -> morris (m - 1) (morris m n);;
```

## Exercice 2 (Fonction de Morris)

0

La fonction morris ci-dessus termine-t-elle sur  $\mathbb{N}^2$  ?

```
# let rec ackermann = fun
  | 0 p -> p + 1
  | n 0 -> ackermann (n - 1) 1
  | n p -> ackermann (n - 1) (ackermann n (p - 1));;
```

## Exercice 3 (Fonction d'Ackermann)

1

- 1 Montrer que la fonction ackermann ci-dessus termine sur  $\mathbb{N}^2$ .
- 2 Calculer ackermann 1 p, ackermann 2 p et ackermann 3 p.
- 3 Montrer que pour tout  $(n, p) \in \mathbb{N}^2$ , ackermann n p > p.

```
# let rec mccarthy = function
  | n when n > 100 -> n - 10
  | n -> mccarthy (mccarthy (n + 11));;
```

## Exercice 4 (Fonction de McCarthy)

2

Montrer que la fonction mccarthy termine sur  $\mathbb{Z}$  et calculer sa valeur en tout entier.





## Réversivité terminale

### Exercice 1 (Réversivité croisée)

Soit  $(u_n)$  et  $(v_n)$  les suites définies par  $u_0 = a, v_0 = b \in \mathbb{R}_+^*$  ( $a$  et  $b$  sont préalablement déclarés) et, pour tout  $n \in \mathbb{N}$  :

$$u_{n+1} = \frac{u_n + v_n}{2}, \quad v_{n+1} = \sqrt{u_n v_n}.$$

Définir ces suites par réversivité croisée.

### Exercice 2 (Suite récurrente version terminale)

On suppose disposer d'une fonction  $f$  de type `float -> float`. On fixe un réel  $a$ , et on considère la suite  $(u_n)$  de terme initial  $u_0 = a$  et d'itératrice  $f$ .

Écrire une fonction réversive terminale permettant le calcul de tout terme de cette suite.

### Exercice 3 (Somme des termes d'une liste)

Écrire une fonction réversive terminale qui calcule la somme des termes d'une liste d'entiers.

### Exercice 4 (Opérations classiques sur les listes chaînées)

Proposer des versions réversives terminales de l'inversion d'une suite, de la concaténation de deux suites.

### Exercice 5 (Aplatissement de listes)

**1** Écrire une fonction `aplatir : 'a list list -> 'a list` qui à une liste de listes associe leur concaténée.

**2** Proposer une version réversive terminale de l'aplatissement.



## Induction structurelle

### Exercice 1 (Fonctions sur des ensembles inductifs)

- 1 Écrire la fonction `pred` donnant le prédécesseur d'un objet de type entier (type défini dans le poly).
- 2 Généraliser l'exemple de la somme de deux nombres (le type nombre est défini dans le poly) par abstraction de l'opération.
- 3 Écrire une fonction permettant le calcul du nombre de feuilles d'un arbre binaire donné.
- 4 Écrire la fonction `arbre_fibo` de type `int -> arbre_binaire` envoyant l'entier  $n$  sur l'arbre binaire des appels à la suite de Fibonacci dans la programmation maladroite de cette suite.

### Exercice 2 (Définition d'un type calcul)

Définir un type calcul sous forme d'arbres binaires, comprenant les entiers, les lois d'addition et de multiplication. Définir une fonction qui à un tel calcul associe sa valeur.



## Complexité

## Exercice 1 (Tours de Hanoï)

1

On dispose de  $n$  rondelles  $r_1, \dots, r_n$ , de diamètres  $1, \dots, n$ , évidées en leur centre, et de trois piquets, sur lesquels on peut enfiler les rondelles. Au départ, les rondelles sont toutes empilées sur le premier piquet, selon la taille de leurs diamètres (la plus large en bas). On cherche à déplacer ces rondelles du premier vers le troisième piquet, en respectant les lois suivantes :

- chaque étape consiste en un déplacement d'une unique rondelle vers une autre ;
- on ne peut pas poser une rondelle sur une rondelle plus petite.

C'est le fameux problème des tours de Hanoï (ou des tours de Babel à Plovdiv, une charmante ville bulgare).

- 1 Proposer une procédure résolvant ce problème.
- 2 Évaluer la complexité de votre algorithme en nombre de déplacements.

---

```
# let cherche elt t = let i = ref 0 and trouve = ref false in
  while (!i < vect_length t & !trouve = false) do
    if elt = t.(!i) then trouve := true else i := !i + 1
  done;
  !trouve;;
cherche : 'a -> 'a vect -> bool = <fun>

# cherche 3 [| 5; 6; 9|];;
- : bool = false
```

---

## Exercice 2 (Recherche dans un tableau)

2

1 On effectue encore une recherche d'un élément dans un tableau, en supposant cette fois-ci que les éléments du tableau de taille  $n$  sont des entiers de 1 à  $p$ , pouvant apparaître plusieurs fois, avec une probabilité uniforme. Calculer la complexité moyenne de l'algorithme.

2 On revient au cas général, et on suppose le tableau rangé dans l'ordre croissant. Proposer un algorithme de recherche dans ce tableau, de complexité logarithmique.

## Exercice 3 (Somme de relations de comparaison)

2

Montrer que si  $c_n = O(n^\alpha)$ , alors  $\sum_{1 \leq k \leq n} c_k = O(n^{\alpha+1})$ .

Vous vous trouvez face à un mur infini, sur lequel se trouve une unique porte permettant de passer de l'autre côté. Vous ne savez pas si cette porte se situe à votre gauche ou à votre droite. De plus, vous ne pouvez la trouver que si vous vous retrouvez en face d'elle. Proposer une stratégie de complexité linéaire pour trouver cette porte.

```
# let rec u1 = function
  | 0 -> 1.0
  | n -> let s = ref 0.0 in
        for k = 1 to n do
          s := !s +. u1 (n - k) /. float_of_int k
        done;
        !s;;
u1 : int -> float = <fun>

# u1 3;;
- : float = 2.3333333333333333

# let u2 n = let v = make_vect (n + 1) 0.0 in
  v.(0) <- 1.0;
  for p = 1 to n do
    for k = 1 to p do
      v.(p) <- v.(p) +. v.(p - k) /. float_of_int k
    done
  done;
  v.(n);;
u2 : int -> float = <fun>

# u2 3;;
- : float = 2.3333333333333333
```

On veut calculer les termes de la suite  $(u_n)$  définie par :

$$u_0 = 1, \quad \forall n \in \mathbb{N}^*, u_n = \frac{u_{n-1}}{1} + \frac{u_{n-2}}{2} + \dots + \frac{u_0}{n}.$$

Les deux programmes ci-dessus calculent  $u_n$  : comparer leurs complexités temporelles et spatiales.

## Diviser pour régner

### Exercice 1 (Suite de Fibonacci, encore et toujours)

2

En utilisant une stratégie diviser pour régner, montrer comment obtenir le terme d'indice  $n$  de la suite de Fibonacci en complexité logarithmique.

**Indication :** on pourra utiliser la multiplication matricielle.

### Exercice 2 (Multiplication polynomiale de Knuth)

2

On cherche à implémenter la multiplication rapide de Knuth (ou de Karatsuba). On représente les polynômes d'entiers comme des tableaux d'entiers, l'indice correspondant à la puissance de l'indéterminée.

**1** Définir l'addition polynomiale (curryfiée, comme les fonctions suivantes).

**2** Définir la multiplication par un scalaire d'un polynôme.

**3** Définir la soustraction de polynômes.

**4** Définir l'addition de trois polynômes, la soustraction de deux polynômes à un troisième.

**5** Définir une fonction décalage de type  $\text{int} \rightarrow \text{int vect} \rightarrow \text{int vect}$  tel que  $\text{decalage } n \ p$  soit le polynôme  $X^n p(X)$ .

**6** Pour simplifier la programmation, on supposera que les tableaux utilisés auront pour taille une même puissance de deux.

Définir la multiplication polynomiale grâce à l'algorithme de Knuth.

**7** Comment utiliser cette multiplication pour effectuer la multiplication d'entiers en base  $b$  « arbitrairement grands », représentés par des tableaux d'entiers entre 0 et  $b - 1$  ?





## Tris

### Exercice 1 (Recherche dichotomique)

1

Proposer une fonction de recherche d'un élément dans un tableau trié, fondée sur la stratégie diviser pour régner, en complexité logarithmique en nombre de comparaisons.

### Exercice 2 (Insertion dans une liste triée)

1

- 1 Proposer une fonction insérant (à la bonne place) un élément dans un vecteur trié.
- 2 Même question pour une liste chaînée.

### Exercice 3 (Fusion de listes triées)

1

- 1 Proposer une fonction fusionnant deux vecteurs triés.
- 2 Même question pour les listes chaînées.

### Exercice 4 (Tri par insertion)

1

Le tri par insertion est le tri du joueur de cartes (on prend les cartes une par une, en l'insérant au bon endroit).

- 1 Proposer un tri par insertion pour les vecteurs.
- 2 Même question pour les listes chaînées.

### Exercice 5 (Tri par sélection)

1

Le tri par sélection consiste à chercher le plus petit élément, puis le suivant, etc.

- 1 Proposer un tri par sélection pour les vecteurs.
- 2 Même question pour les listes chaînées.

### Exercice 6 (Tri bulle)

0

L'algorithme du tri bulle proposé dans le cours effectue un nombre constant de comparaisons : comment améliorer l'algorithme afin qu'il en effectue moins en moyenne ?

Exercice 7 (Tri par insertion en mieux)

0

L'algorithme du tri par insertion pour les vecteurs suit à la lettre la démarche de cette méthode de tri. En programmant par effets de bord, proposer un algorithme de tri par insertion plus efficace.

Exercice 8 (Stabilité des algorithmes de tri)

2

Un algorithme de tri est dit *stable* s'il n'échange jamais des termes égaux en des indices distincts. Parmi les algorithmes proposés, lesquels sont-ils stables? Comment rendre tout algorithme de tri stable?

## Listes

---

```
# let rec image_miroir_bof = function
  | [] -> []
  | a :: q -> image_miroir_bof q @ [a];;
image_miroir_bof : 'a list -> 'a list = <fun>

# image_miroir_bof [1; 4; 9; 13];;
- : int list = [13; 9; 4; 1]
```

---

### Exercice 1 (Image miroir)

1

La fonction `image_miroir_bof` ci-dessus renverse une liste.

- 1 Calculer sa complexité en nombre de conses, la complexité de `l1 @ l2` étant  $|l1|$ .
- 2 Proposer un algorithme de complexité linéaire effectuant l'image miroir.

### Exercice 2 (Versions itératives des programmes usuels sur les listes)

2

En utilisant le type `Liste` défini dans le cours, proposer des versions impératives des algorithmes fondamentaux sur les listes.

### Exercice 3 (Insertion et suppression)

2

On travaillera avec le type `Liste` défini dans le cours.

- 1 Proposer des fonctions d'insertion et de suppression d'un terme dans une liste à une position donnée.
- 2 Proposer une fonction d'insertion d'un élément dans une liste triée.

**1** Définir une fonction `map_bis`, de type  $(\text{'a} \rightarrow \text{'b}) \rightarrow \text{'a list} \rightarrow \text{'b list}$  telle que `map_bis f liste` renvoie la liste des applications de `f` à `liste`. Cette fonction, sous le nom `map`, est une primitive Caml.

**2** La fonctionnelle `it_list`, de type  $(\text{'a} \rightarrow \text{'b} \rightarrow \text{'a}) \rightarrow \text{'a} \rightarrow \text{'b list} \rightarrow \text{'a}$ , est définie informellement par

`it_list f b [e1; e2; ... ; en] = (f (... (f (f b e1) e2) ... ) en)`

**i** Programmer cette fonctionnelle.

**ii** Utiliser cette fonctionnelle pour définir une fonction qui à une liste d'entiers associe la somme de ses termes. Faire de même pour le produit.

**3** La fonctionnelle `list_it`, de type  $(\text{'a} \rightarrow \text{'b} \rightarrow \text{'b}) \rightarrow \text{'a list} \rightarrow \text{'b} \rightarrow \text{'b}$  est définie informellement par

`list_it f [e1; e2; ... ; en] b = f e1 (f e2 (... (f en b) ...))`

**i** Programmer cette fonctionnelle.

Utiliser cette fonctionnelle pour

**ii** définir des fonctions `somme` et `produit` comme ci-dessus.

**iii** programmer la concaténation.

**iv** programmer `map_bis`.

Si  $L = (a_1, \dots, a_n)$  est une liste, alors, pour tous  $i$  et  $j$  tels que  $1 \leq i \leq j \leq n$ ,  $(a_k)_{k \in \llbracket i, j \rrbracket}$  est une *sous-liste* de  $L$ . Nous disons aussi que la liste vide  $\varepsilon$  est sous-liste de toute liste.

Une *sous-séquence* de la liste  $L$  est une liste formée en éliminant zéro ou plusieurs éléments de  $L$ . Les éléments restants, qui forment la sous-séquence, doivent apparaître dans le même ordre que  $L$ , mais les éléments de la sous-séquence ne sont pas nécessairement consécutifs dans  $L$ .

Un *préfixe* d'une liste est une sous-liste commençant par la tête de la liste (*i.e.*  $i = 1$  ci-dessus).

Un *suffixe* d'une liste est une sous-liste qui se termine à la fin de la liste (*i.e.*  $j = n$  ci-dessus).

**1** Programmer des prédicats permettant de tester si une liste est une sous-liste, un préfixe, un suffixe, une sous-séquence de la seconde liste passée en argument (on travaillera avec les listes chaînées de Caml).

**2** Dénombrer le nombre maximal de préfixes, suffixes, sous-listes, sous-séquences d'une liste de longueur  $n$ .

**3** Décrire les listes commutant pour la concaténation.

**4** On travaille avec des listes dont les termes appartiennent à un alphabet  $\mathcal{A}$  de cardinal  $N$ .

**i** Dénombrer le nombre de listes de taille  $n$ .

**ii** Dénombrer le nombre de listes de tailles respectives 6 et 10 commutant pour la concaténation.

**5** On considère une liste de listes d'éléments de  $\mathcal{A}$ . On vous donne une telle liste, mais on a omis toutes les parenthèses permettant de distinguer les éléments de cette liste.

**i** Dénombrer les listes possibles n'admettant pas la liste vide pour élément.

**ii** Dénombrer les listes possibles de taille  $n$ .

## Piles et files

## Exercice 1 (Rotation d'une pile)

0

La rotation d'une pile consiste à prendre la première assiette, et à la mettre tout en bas. Programmer cette rotation pour l'implémentation détaillée dans le cours.

## Exercice 2 (EAP pour une loi associative)

2

On considère une loi de composition interne associative  $\star$  sur un ensemble  $E$ . Soit  $a, b, c, d \in E$ . Proposer des expressions algébriques permettant le calcul de  $a \star b \star c \star d$ .

---

```
# type operateur = Plus | Moins | Mult | Divise ;;
Type operateur defined.
```

```
# type Lexeme = Var of float | Op of operateur ;;
Type Lexeme defined.
```

---

## Exercice 3 (Expressions algébriques postfixées)

1

On utilise la structure de données pile donnée en cours. On cherche à évaluer une expression algébrique postfixée (EAP). Pour cela, nous la verrons comme une liste de lexèmes (voir ci-dessus). Proposer une évaluation d'EAP (syntaxiquement correcte) à l'aide d'une pile.

## Exercice 4 (Bon parenthésage)

1

À l'aide d'une pile, vérifier le bon parenthésage (en un sens évident) d'un mot constitué de parenthèses uniquement.

## Exercice 5 (Pile et expression préfixée)

2

Comment utiliseriez-vous une pile pour évaluer des expressions préfixées ?

## Exercice 6 (Expressions algébriques)

2

- 1 Programmer une fonction permettant de passer d'une expression algébrique infixée à une postfixée.
- 2 Programmer une fonction réciproque.

Exercice 7 (D'autres représentations des piles)

3

- 1 Implémenter les autres structures de données représentant le type de données pile vues en cours. On programmera en particulier le test d'égalité.
- 2 Reprendre les exercices précédents avec ces structures.

Exercice 8 (Représentations des files)

3

Implémenter les files (versions mutable et persistante).

## Arbres

### Exercice 1 (Résultats théoriques sur les arbres)

0

- 1 Montrer que tout arbre non vide a exactement un sommet de plus que d'arêtes.
- 2 Montrer que tout arbre binaire complet non vide a exactement une feuille de plus que de nœuds intérieurs.
- 3 Pour un arbre de hauteur  $h$ , quelles sont les valeurs possibles pour sa taille ?

### Exercice 2 (Implémentation des arbres)

1

Proposer un type implémentant les arbres binaires, et programmer les fonctions usuelles définies dans le cours.  
On pourra aussi s'amuser à définir une procédure de représentation des arbres.

### Exercice 3 (Implémentation des ABR)

1

Proposer une fonction testant si un arbre binaire est un ABR, et programmer les fonctions recherche, successeur et prédécesseur.

### Exercice 4 (Divers ABR pour un même ensemble de clés)

3

Dessiner des arbres binaires de recherche de hauteur 2, 3, 4, 5 et 6 pour le même ensemble de clés  $\{1, 4, 5, 10, 16, 17, 21\}$ .

### Exercice 5 (Parcours d'un ABR)

3

On suppose que les entiers de 1 à 1000 sont disposés dans un ABR, et on souhaite retrouver le nombre 363. Parmi les séquences suivantes, lesquelles pourraient et ne pourraient pas être la séquence de nœuds parcourus ?

- 1 2, 252, 401, 398, 330, 344, 397, 363.
- 2 924, 220, 911, 244, 898, 258, 362, 363.
- 3 925, 202, 911, 240, 912, 245, 363.
- 4 2, 399, 387, 219, 266, 382, 381, 278, 363.
- 5 935, 278, 347, 621, 299, 392, 358, 363.

### Exercice 6 (Bee trees)

3

Une abeille mâle est produite de manière asexuée à partir d'une abeille femelle. Par contre, une abeille femelle a deux parents : un mâle et une femelle.

- 1 Représenter l'arbre généalogique d'une abeille mâle en remontant jusqu'à la quatrième génération.
- 2 Combien une abeille mâle a-t-elle d'ancêtres de niveau 1 ? de niveau 2 ? Montrer que ce nombre d'ancêtres constitue une suite de Fibonacci.

### Exercice 7 (Arbres binaires équilibrés)

3

On appelle ici arbre équilibré un arbre binaire tel que, en tout nœud, la hauteur des sous-arbres gauche et droit diffère d'au plus un.

- 1 Donner plusieurs exemples possibles d'arbres équilibrés de 10 nœuds.
- 2 Quelles sont les hauteurs minimale et maximale d'un arbre équilibré de 20 nœuds ?
- 3 Écrire un algorithme récursif qui vérifie qu'un arbre binaire quelconque est bien équilibré. Quel est le parcours de l'arbre induit par cet algorithme ?



Sixième partie

Solution des exercices



## Solution de premiers pas en Caml

---

```

# sqrt (1. +. sqrt 17.);;
- : float = 2.26342784856

# (a +. b) /. (a ** 3. -. b ** 2.) where a = cos (log 3.) and b = sin (log 2.);;
- : float = -3.48143281984
#let x = 12. in
    let y = log x in
        let z = log y in
            x ** (y ** z);; (* x ** y ** z donnerait le bon résultat *)
- : float = 295.982482437

# ((x ** (y ** z) where z = log y) where y = log x) where x = 12.;;
(* Noter la nécessité du parenthésage pour les where imbriqués *)
- : float = 295.982482437

# let tanh x = (aux -. 1.) /. (aux +. 1.) where aux = exp (2. *. x);;
tanh : float -> float = <fun>

# tanh 1.;;
- : float = 0.761594155956

# let add = function x -> function y -> x + y;;
add : int -> int -> int = <fun>

# 2, 4;;
- : int * int = 2, 4

# 2., 2;;
- : float * int = 2.0, 2

# (3, ('a', 4.5), 3);;
- : int * (char * float) * int = 3, ('a', 4.5), 3

# let evaluate_en_2 f = f 2;;
evaluate_en_2 : (int -> 'a) -> 'a = <fun>

# let evaluate x f = f x;;
evaluate : 'a -> ('a -> 'b) -> 'b = <fun>

# let prod loi1 loi2 (x1,x2) (y1,y2) = (loi1 x1 y1, loi2 x2 y2);;
prod :
('a -> 'b -> 'c) -> ('d -> 'e -> 'f) -> 'a * 'd -> 'b * 'e -> 'c * 'f =
<fun>

# let ordre (x,y) (x',y') = (x < x') || ((x = x') && (y < y'));;
ordre : 'a * 'b -> 'a * 'b -> bool = <fun>

```

```

# let et_logique a b = match a with
  | false -> false
  | - -> b;;
et_logique : bool -> bool -> bool = <fun>

# let implication = fun
  | true false -> false
  | - - -> true;;
implication : bool -> bool -> bool = <fun>

# let if_then_else b v v' = match b with
  | true -> v
  | - -> v';;
if_then_else : bool -> 'a -> 'a -> 'a = <fun>

# if_then_else (0=1) 1 2;;
- : int = 2

# let taux f x y = (f(y) -. f(x))/.(y -. x);;
taux : (float -> float) -> float -> float -> float = <fun>

# let o f g x = f (g x);;
o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# #infix "o";;

# let decale_de_2 f = (add 2) o f o (add 2);;
decale_de_2 : (int -> int) -> int -> int = <fun>

# let triple x = 3 * x;;
triple : int -> int = <fun>

# let h = decale_de_2 triple;;
h : int -> int = <fun>

# h 2;;
- : int = 14

# let curry f x y = f (x, y);;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

# let uncurry f (x,y) = f x y;;
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>

# let p x y = y x x;;
p : 'a -> ('a -> 'a -> 'b) -> 'b = <fun>

# let p' y = p p y;;
p' :
  (('a -> ('a -> 'a -> 'b) -> 'b) -> ('a -> ('a -> 'a -> 'b) -> 'b) -> 'c) ->
  'c = <fun>

```

---

## Solution de premiers pas en récursivité et itération

---

```

# let a = "cou";;
a : string = "cou"

# let b = a ^ a;;
b : string = "coucou"

# a.[1] <- 'r';;
- : unit = ()

# b;;
- : string = "coucou"

# let tete = function
  | [] -> failwith "Pas_de_tête_pour_la_liste_vide"
  | a :: _ -> a;;
tete : 'a list -> 'a = <fun>

# let queue = function
  | [] -> failwith "Pas_de_queue_pour_la_liste_vide"
  | _ :: q -> q;;
queue : 'a list -> 'a list = <fun>

# tete [1; 2; 4];;
- : int = 1

# queue [1; 2; 4];;
- : int list = [2; 4]

# let rec concat l1 l2 = match l1 with
  | [] -> l2
  | a :: q -> a :: concat q l2;;
concat : 'a list -> 'a list -> 'a list = <fun>

# concat [1; 2; 4] [3; 4; 5];;
- : int list = [1; 2; 4; 3; 4; 5]

# let rec membre elt l = match l with
  | [] -> false
  | a :: q -> (elt = a) or (membre elt q);;
membre : 'a -> 'a list -> bool = <fun>

# membre 3 [5; 6; 3];;
- : bool = true

# membre 7 [5; 6; 3];;
- : bool = false

# let rec indice elt l = match l with
  | [] -> failwith "Element_non_dans_la_liste"

```

```

    | a :: q -> if elt = a then 1 else indice elt q + 1;;
indice : 'a -> 'a list -> int = <fun>

# indice 3 [5; 6; 3];;
- : int = 3

# indice 7 [5; 6; 3];;
Uncaught exception: Failure "Element_non_dans_la_liste"

# let somme_vect t = let s = ref 0 in
    for i = 0 to vect_length t - 1 do s := !s + t.(i) done;
    !s;;
somme_vect : int vect -> int = <fun>

# somme_vect [| 3; 5; 7|];;
- : int = 15

# let rec somme_liste = function
    | [] -> 0
    | a :: q -> a + somme_liste q;;
somme_liste : int list -> int = <fun>

# somme_liste [3; 6; 9];;
- : int = 18

(* accès au i-ème terme d'une liste *)
# let rec acces liste i = match i with
    | 1 -> hd liste
    | _ -> acces (tl liste) (i-1);;
acces : 'a list -> int -> 'a = <fun>

# acces [6; 7; 8; 9] 4;;
- : int = 9

# let rec ampute = function
    | [] -> failwith "on_n'ampute_pas_un_manchet_cul-de-jatte"
    | [a] -> []
    | a :: q -> a :: ampute q;;
ampute : 'a list -> 'a list = <fun>

# ampute [3; 7; 9; 11];;
- : int list = [3; 7; 9]

# let puissance_imperative a b = let p = ref 1 in for i = 1 to b do p := !p * a done; !p;;
puissance_imperative : int -> int -> int = <fun>

# puissance_imperative 2 10;;
- : int = 1024

# let rec puissance a = function
    | 0 -> 1
    | b -> a * puissance a (b - 1);;
puissance : int -> int -> int = <fun>

# puissance 2 10;;
- : int = 1024

# let rec pasderepet = function
    | [] -> []

```

```

    | [a] -> [a]
    | a :: q -> if a = hd q then pasderepet q else a :: pasderepet q;;
pasderepet : 'a list -> 'a list = <fun>

# pasderepet [1; 3; 3; 3; 5; 5; 10; 10; 10];;
- : int list = [1; 3; 5; 10]

# let rec ajoutsimple = function
    | [] -> []
    | [a] -> [a]
    | a :: q -> a :: 0 :: ajoutsimple q;;
ajoutsimple : int list -> int list = <fun>

# ajoutsimple [4];;
- : int list = [4]

# ajoutsimple [4; 8; 9; 2];;
- : int list = [4; 0; 8; 0; 9; 0; 2]

(* Fonction auxiliaire *)

# let rec ajout l n a = match n with
    | 0 -> l
    | _ -> a :: ajout l (n-1) a;;
ajout : 'a list -> int -> 'a -> 'a list = <fun>

# let rec ajoutde0 l = match l with
    | [] -> failwith "pas_d'ajout_pour_la_liste_vide"
    | [a] -> [a]
    | a :: q -> a :: (ajout (ajoutde0 q) (list_length l - 1) 0);;
ajoutde0 : int list -> int list = <fun>

# let bonajout l = rev (ajoutde0 (rev l));;
bonajout : int list -> int list = <fun>

# bonajout [1; 3; 4; 7; 23];;
- : int list = [1; 0; 3; 0; 0; 4; 0; 0; 0; 7; 0; 0; 0; 0; 23]

# let recherche t elt =
    let b = ref false in
    let i = ref 0 in
    while !i < vect_length t & !b = false do
        if t.(!i) = elt then b := true; i := !i + 1
    done;
    !b;;
recherche : 'a vect -> 'a -> bool = <fun>

# recherche [[1; 2; 3]] 4;;
- : bool = false

# exception trouve;;
Exception trouve defined.

# let recherche' t elt =
    try
        for i = 0 to vect_length t - 1 do
            if t.(i) = elt then raise trouve;
        done;
    false;

```

```

        with
recherche' : 'a vect -> 'a -> bool = <fun>
        trouve -> true;;

# recherche' [[1; 2; 3]] 4;;
- : bool = false

# let recherche' t elt =
    try
        for i = 0 to vect_length t do
            if t.(i) = elt then raise trouve; done;
        false;
    with
        | trouve -> true
        | Invalid_argument _ -> false;;
recherche' : 'a vect -> 'a -> bool = <fun>

# recherche' [[1; 2; 3]] 4;;
- : bool = false

# let recherche'' t elt =
    try
        let i = ref 0 in
            while true do
                if t.(!i) = elt then raise trouve; i := !i + 1 done;
            false;
    with
        | trouve -> true
        | Invalid_argument _ -> false;;
recherche'' : 'a vect -> 'a -> bool = <fun>

# recherche'' [[1; 2; 3]] 4;;
- : bool = false

# recherche'' [[1; 2; 3]] 2;;
- : bool = true

# exception trouve' of int;;
Exception trouve' defined.

# let indice t elt =
    try
        for i = 0 to vect_length t do
            if t.(i) = elt then raise (trouve' i) done;
        with
            | trouve' i -> print_string "trouvé_en_position_";
                print_int i; print_newline ();
            | Invalid_argument _ -> print_string "élément_non_trouvé";
                print_newline ();;
indice : 'a vect -> 'a -> unit = <fun>

# indice [[1; 2; 3]] 2;;
trouvé en position 1
- : unit = ()

```

---



## Solution d'itération

---

```

# let max_tableau t = let m = ref t.(0) in
  for i = 1 to vect_length t - 1 do
    if !m < t.(i) then m := t.(i); done;
    (* invariant après la i-ème itération : !m vaut max t.(0) ... t.(i) *)
  !m;;
max_tableau : 'a vect -> 'a = <fun>

# max_tableau [|4; -5; 10; 2|];;
- : int = 10

# let max_tableau ordre t = let m = ref t.(0) in
  for i = 1 to vect_length t - 1 do
    if ordre !m t.(i) then m := t.(i); done;
  !m;;
max_tableau : ('a -> 'a -> bool) -> 'a vect -> 'a = <fun>

# let plus_petit n = let m = ref 2 in
  while n mod !m <> 0 do m := !m + 1 done;
  !m;;
plus_petit : int -> int = <fun>

# plus_petit 65;;
- : int = 5

# let f n =
  let x = ref n and y = ref n in
    while not (!y = 0) do
      x := !x + 2;
      y := !y - 1;
    done;
  (* invariant : x + 2 y = 3n *)
  !x;;
f : int -> int = <fun>

# let fmod n =
  let x = ref n and y = ref n in
    while not (!y = 0) do
      x := !x * 2;
      y := !y - 1;
    done;
  (* invariant : !x*2^(!y)=n*2^n *)
  !x;;
fmod : int -> int = <fun>

# fmod 3;;
- : int = 24

# let pgcd a b = let m = ref a and n = ref b in
  while !n <> 0 do let c = !n in (n := !m mod !n; m := c;) done;

```

```

    !m;;
pgcd : int -> int -> int = <fun>

# pgcd 10 15;;
- : int = 5

# let cmod a b q = (fst(!a) - q * fst (!b),snd(!a) - q * snd (!b));;
cmod : (int * int) ref -> (int * int) ref -> int -> int * int = <fun>

# let bezout a b = let r = ref (a,b) and uv0 = ref (1,0) and uv1 = ref (0,1) in
    while snd(!r) < 0 do
    let q = fst(!r) / snd(!r) and c = !uv1 in
        (uv1 := cmod uv0 uv1 q; uv0 := c);
    r := (snd(!r),fst(!r) mod snd(!r));
    done;
    (* invariant : fst !uv0 * a + snd !uv0 * b = fst !r
    et
    fst !uv1 * a + snd !uv1 * b = snd !r *)
    !uv0;;
bezout : int -> int -> int * int = <fun>

# bezout 113 15;;
- : int * int = 2, -15

# let evaluate p x = let v = ref 0. and xp = ref 1. in
    for i = 0 to vect_length p - 1 do
        v := !v +. p.(i) *. !xp;
        xp := x *. !xp;
    done;
    (* invariant après le passage en boucle pour l'indice i :
    !v = somme pour k allant de 0 à i de p.k * x ^ k et !xp = x ^ (i + 1) *)
    !v;;
evaluate : float vect -> float -> float = <fun>

# evaluate [| 3.; 2.; 1. |] 2.;;
- : float = 11.0

# let horner p x = let m = (vect_length p) - 1 in let v = ref p.(m) in
    for i = m - 1 downto 0 do
        v := !v *. x +. p.(i)
    done;
    (* invariant après la i-ème itération :
    !v = somme pour k allant de 0 à i de p.(m - k) * x ^ (i - k) *)
    !v;;
horner : float vect -> float -> float = <fun>

# horner [| 3.; 2.; 1. |] 2.;;
- : float = 11.0

# let myst x y = let m = ref 0 and a = ref x and b = ref y in
    while !b > 0 do
        if !b mod 2 = 1 then m := !m + !a;
        a := !a * 2;
        b := !b / 2;
    done;
    (* invariant : !m + !a * !b = x * y *)
    !m;;
myst : int -> int -> int = <fun>

```

```
# myst 5 9;;  
- : int = 45  
  
# let g n = let p = ref 2 in  
             while (n mod !p != 0) do p := !p + 1 done;  
             !p;;  
g : int -> int = <fun>  
  
# g 113;;  
- : int = 113
```

---



## Solution de terminaison et correction des fonctions récursives

## Exercice 1 (Ordre produit)

0

Première méthode : soit  $(m_0, n_0)$  un élément donné de  $\mathbb{N}^2$ . L'ensemble  $\{(m, n) \in \mathbb{N}^2, (m, n) \preceq (m_0, n_0)\}$  étant fini (de cardinal  $(m_0 + 1)(n_0 + 1)$ ), il n'existe pas de suite strictement décroissante d'éléments de  $\mathbb{N}^2$  pour cet ordre, qui est donc bien fondé.

Deuxième méthode : soit  $\Omega$  une partie non vide de  $\mathbb{N}^2$ . On pose  $m_0 = \min\{m \in \mathbb{N}, \exists n \in \mathbb{N}, (m, n) \in \Omega\}$  (qui existe car  $\Omega$  est non vide et toute partie non vide de  $\mathbb{N}$  admet un minimum). On pose alors  $n_0 = \min\{n \in \mathbb{N}, (m_0, n) \in \Omega\}$ . L'élément  $(m_0, n_0)$  est alors minimal dans  $\Omega$  : en effet si  $(m, n) \in \Omega$  vérifie  $(m, n) \preceq (m_0, n_0)$ , alors  $m \leq m_0$  par définition de  $\preceq$ , et  $m_0 \leq m$  par définition de  $m_0$  :  $m = m_0$ . Il vient alors, par un raisonnement analogue,  $n = n_0$ .

Troisième méthode : supposons l'existence d'une suite infinie  $((m_p, n_p))_{p \in \mathbb{N}}$  strictement décroissante d'éléments de  $\mathbb{N}^2$ . L'application

$$\begin{aligned} \Phi : \quad \mathbb{N}^2 &\rightarrow \mathbb{N} \\ (m, n) &\mapsto m + n \end{aligned}$$

étant strictement croissante, la suite  $(\Phi(m_p, n_p))_{p \in \mathbb{N}}$  d'entiers naturels est strictement décroissante, ce qui est absurde.

## Exercice 2 (Fonction de Morris)

0

La fonction morris ne termine pas, puisque pour le calcul de morris 1 1 par exemple, qui n'est pas un cas de base, la fonction s'appelle d'abord pour le même argument.

On note  $A$  la fonction ackermann, en version non curryfiée.

**1** Cette fonction termine, comme le montre un raisonnement par induction bien fondée, en prenant l'ordre lexicographique sur  $\mathbb{N}^2$ . En effet, elle termine dès que l'un des deux arguments est nul, et, si on fixe  $(m_0, n_0) \in (\mathbb{N}^*)^2$ , et que l'on suppose que cette fonction termine pour tout couple  $(m, n) \in \mathbb{N}^2$  tel que  $(m, n) \prec (m_0, n_0)$ , alors  $A(m_0, n_0)$  appelle  $A$  pour le couple  $(m_0, (n_0 - 1)) \prec (m_0, n_0)$ , puis pour un couple  $(m_0 - 1, n'_0)$  (pour un certain entier  $n'_0$ ), strictement inférieur à  $(m_0, n_0)$ .

**2** Pour tout entier naturel non nul  $p$ ,

$$A(1, p) = A(0, (A(1, (p - 1)))) = A(1, (p - 1)) + 1.$$

Comme  $A(1, 0) = A(0, 1) = 2$ , on obtient  $A(1, p) = p + 2$  par une récurrence immédiate.

On observe déjà que  $A(2, 0) = 3$ , puis que pour tout  $p \in \mathbb{N}^*$ ,  $A(2, p) = A(2, (p - 1)) + 2$ , donc  $A(2, p) = 3 + 2p$  pour tout  $p \in \mathbb{N}$  par une récurrence immédiate.

Par un calcul simple,  $A(3, 0) = 5$ , et pour tout  $p \in \mathbb{N}^*$ ,  $A(3, p) = 2A(3, (p - 1)) + 3$ , d'où  $A(3, p) + 3 = 2^{p+3}$  par une récurrence immédiate.

**3** On raisonne par induction bien fondée sur  $\mathbb{N}^2$  muni de l'ordre lexicographique. Le résultat est vrai pour tout couple d'entiers naturels dont l'un est nul, et, si on fixe  $(m, n) \in (\mathbb{N}^*)^2$ , et que l'on suppose le résultat prouvé pour tous les éléments qui lui sont strictement inférieurs, alors  $A(n, (p - 1)) \geq p$  par hypothèse d'induction, puis, toujours par hypothèse d'induction :  $A(n, p) > A(n, (p - 1)) \geq p$ , donc  $A(n, p) > p$ .

Le résultat est donc bien prouvé.

On note  $M$  la fonction mccarthy, en version non curryfiée.

La fonction mccarthy termine clairement pour tout entier  $n \geq 101$ , et elle y vaut  $n - 10$ . Montrons que pour tout entier  $n \leq 100$ ,  $M(n) = 91$ . On raisonne par l'absurde, en supposant l'existence d'un entier  $n \leq 100$  tel que  $M(n)$  ne termine pas ou diffère de 91. On choisit le plus grand entier  $N \leq 100$  tel que  $M(N)$  ne termine pas ou diffère de 91.

Si  $N + 11 > 100$ , alors  $M(N + 11) = N + 1$ , puis  $M(N) = M(M(N + 11)) = M(N + 1)$ , ce qui impose  $N = 100$  par maximalité de  $N$ , or  $M(100) = 91$ , ce qui est exclu.

Si  $N + 11 \leq 100$ , alors  $M(N + 11) = 91$ , donc  $M(N)$  termine et vaut  $M(91)$ . Or un calcul simple montre que  $M(91) = 91$  (car  $M(n) = M(n + 1)$  pour tout  $n \in \llbracket 91, 100 \rrbracket$ ), ce qui est encore une fois absurde.

## Solution de récursivité terminale

---

```

# let a = 1. and b = 3.;;
a : float = 1.0
b : float = 3.0

# let rec u = function
  | 0 -> a
  | n -> (u (n - 1) +. v (n - 1)) /. 2.
and
v = function
  | 0 -> b
  | n -> sqrt (u (n - 1) *. v (n - 1));;
u : int -> float = <fun>
v : int -> float = <fun>

# u 3;;
- : float = 1.86361756099

# v 3;;
- : float = 1.8636160055

# let rec iter a f = function
  | 0 -> a
  | n -> iter (f a) f (n - 1);;
iter : 'a -> ('a -> 'a) -> int -> 'a = <fun>

# iter 3 (function x -> 2 * x + 1) 2;;
- : int = 15

# let f x = 3. *. x +. 2.;;
f : float -> float = <fun>

# let u = iter a f;;
u : int -> float = <fun>

# u 2;;
- : float = 17.0

# let somme = let rec somme_aux accu = function
  | [] -> accu
  | a :: q -> somme_aux (a + accu) q
in somme_aux 0;;
somme : int list -> int = <fun>

# somme [1; 2; 3; 4; 5];;
- : int = 15

# let reverse = let rec reverse_temp l l' = match l' with
  | [] -> l
  | a :: q -> reverse_temp (a :: l) q

```

```

in reverse_temp [];;
reverse : 'a list -> 'a list = <fun>

# reverse [1; 2; 3; 4; 5];;
- : int list = [5; 4; 3; 2; 1]

# let rec concat_rev l = function
  | [] -> l
  | a :: q -> concat_rev (a :: l) q;;
concat_rev : 'a list -> 'a list -> 'a list = <fun>

# concat_rev [1; 2; 3] [4; 5; 6];;
- : int list = [6; 5; 4; 1; 2; 3]

# let concat l l' = concat_rev l' (rev l);;
concat : 'a list -> 'a list -> 'a list = <fun>

# concat [1; 2; 3] [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]

# let concat2 =
  let rec concat_temp2 = fun
    | [] [] l -> l
    | accu (a :: q) l -> concat_temp2 (a :: accu) q l
    | (a :: q) [] l -> concat_temp2 q [] (a :: l)
  in concat_temp2 [];;
concat2 : 'a list -> 'a list -> 'a list = <fun>
# concat2 [1; 2; 3] [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]

# let rec aplatir = function
  | [] -> []
  | a :: q -> concat a (aplatir q);;
aplatir : 'a list list -> 'a list = <fun>

# aplatir [[1; 2; 3; 4]; []; [5; 6]; []];;
- : int list = [1; 2; 3; 4; 5; 6]

# let aplatir_term = let rec aplatir_term_temp accu = function
  | [] -> accu
  | a :: q -> aplatir_term_temp (concat accu a) q
  in aplatir_term_temp [];;
aplatir_term : 'a list list -> 'a list = <fun>

# aplatir_term [[1; 2; 3; 4]; []; [5; 6]; []];;
- : int list = [1; 2; 3; 4; 5; 6]

```

---



## Induction structurelle

```

# let pred = function
  | Zero -> failwith "0_n'a_pas_de_predecesseur"
  | succ x -> x;;
pred : entier -> entier = <fun>

# pred (succ (succ (succ Zero)));;
- : entier = succ (succ Zero)

# let operation f = fun
  | (Entier a) (Entier b) -> Entier (fst f a b)
  | (Reel a) (Entier b) -> Reel (snd f a (float_of_int b))
  | (Entier a) (Reel b) -> Reel (snd f (float_of_int a) b)
  | (Reel a) (Reel b) -> Reel (snd f a b);;
operation :
(int -> int -> int) * (float -> float -> float) ->
nombre -> nombre -> nombre = <fun>

# let prod x y = x * y;;
prod : int -> int -> int = <fun>

# let prodbis x y = x *. y;;
prodbis : float -> float -> float = <fun>

# operation (prod,prodbis) (Entier 3) (Reel 5.5);;
- : nombre = Reel 16.5

# let rec nombrefeuilles = function
  | Feuille n -> 1
  | Noeud (a, branche1, branche2) -> nombrefeuilles branche1 + nombrefeuilles branche2
nombrefeuilles : arbre_binaire -> int = <fun>

# let valeurracine = function
  | Feuille n -> n
  | Noeud (a, branche1, branche2) -> a;;
valeurracine : arbre_binaire -> int = <fun>

# let rec arbre_fibo = function
  | 0 | 1 -> (Feuille 1)
  | n -> Noeud (valeurracine (arbre_fibo (n - 2)) + valeurracine (arbre_fibo (n - 1)),
    arbre_fibo (n - 2), arbre_fibo (n - 1));;
arbre_fibo : int -> arbre_binaire = <fun>

# arbre_fibo 4;;
- : arbre_binaire =
Noeud
(5, Noeud (2, Feuille 1, Feuille 1),
Noeud (3, Feuille 1, Noeud (2, Feuille 1, Feuille 1)))

# nombrefeuilles (arbre_fibo 5);;

```

```
- : int = 8

# type oper = | addition | multiplication ;;
Type oper defined.

# type calcul = | Valeur of int | Fonc of oper * calcul * calcul;;
Type calcul defined.

# let rec compute = function
  | Valeur n -> n
  | Fonc (addition, souscalcul1, souscalcul2) -> compute souscalcul1 +
    compute souscalcul2
  | Fonc (multiplication, souscalcul1, souscalcul2) ->
    compute souscalcul1 * compute souscalcul2;;
compute : calcul -> int = <fun>

# compute (Fonc( addition, (Valeur 12312), (Valeur 35453))));
- : int = 47765

# compute (Fonc( addition, (Valeur 12), (Fonc (multiplication, Valeur 2, Valeur 35))));
- : int = 82
```

---

## Solution de complexité



## Solution de diviser pour régner

Exercice 1 (Suite de Fibonacci, encore et toujours)

2

Il suffit d'observer qu'on obtient les termes de la suite de Fibonacci en calculant les puissances de  $M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ , car, pour tout entier naturel  $n$ ,

$$\begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix} = M \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}.$$

En utilisant l'exponentiation rapide, on obtient bien une complexité logarithmique.

(\* Multiplication de Knuth \*)

(\* Addition des polynômes \*)

```
# let rec add_poly a b = let p = vect_length a and q = vect_length b in
  if p < q then add_poly b a else let s = make_vect p 0 in
    for i = 0 to (q - 1) do
      s.(i) <- a.(i) + b.(i)
    done;
    for i = q to (p - 1) do
      s.(i) <- a.(i)
    done;
  s;;
add_poly : int vect -> int vect -> int vect = <fun>
```

```
# add_poly [| 1; 3; -4|] [|3; 6; 4|];;
- : int vect = [|4; 9; 0|]
```

(\* Multiplication par d'un polynôme par un entier \*)

```
# let multi lambda a = let p = vect_length a in
  let m = make_vect p 0 in
    for i = 0 to (p - 1) do m.(i) <- lambda * a.(i) done;
  m;;
multi : int -> int vect -> int vect = <fun>
```

(\* Soustraction, en utilisant la multiplication par -1 \*)

```
# let mun = multi (-1);;
mun : int vect -> int vect = <fun>
```

```
# mun [|1; 8|];;
- : int vect = [| -1; -8|]
```

```
# let subs a b = add_poly a (mun b);;
subs : int vect -> int vect -> int vect = <fun>
```

```

(* Addition de trois polynômes *)

# let add3 a b c = add_poly (add_poly a b) c;;
add3 : int vect -> int vect -> int vect -> int vect = <fun>

(* Soustraction de deux polynômes à un troisième *)

# let sub2 a b c = subs (subs a b) c;;
sub2 : int vect -> int vect -> int vect -> int vect = <fun>

(* Décalage *)

# let decale a n = let p = vect_length a in
  let d = make_vect (p + n) 0 in
    for i = 0 to (p - 1) do
      d.(i + n) <- a.(i)
    done;
  d;;

# let complete a n = let p = vect_length a in
  let d = make_vect (p + n) 0 in
    for i = 0 to (p - 1) do
      d.(i) <- a.(i)
    done;
  d;;
decale : int vect -> int -> int vect = <fun>

(* Multiplication de Knuth : attention, ceci ne fonctionne que
si ces deux tableaux ont pour même taille une puissance de deux
Cela peut s'arranger en complétant nos tableaux par des 0 *)

# let rec mult_poly a b = let m = vect_length a in match m with
| 1 -> [|a.(0) * b.(0)|]
| _ -> let n = m / 2 in
  let p0 = sub_vect a 0 n and p1 = sub_vect a n n
  and q0 = sub_vect b 0 n and q1 = sub_vect b n n in
  let prod0 = mult_poly p0 q0 and prod1 = mult_poly p1 q1
  and prod01 = mult_poly (add_poly p0 p1) (add_poly q0 q1) in
  add3 prod0 (decale prod1 (2 * n)) (decale (sub2 prod01 prod0 prod1) n);;
mult_poly : int vect -> int vect -> int vect = <fun>

# mult_poly [|1; -1|] [|1; 1|];;
- : int vect = [|1; 0; -1|]

# mult_poly [|1; 2; 1; 0|] [|1; 3; 3; 1|];;
- : int vect = [|1; 5; 10; 10; 5; 1; 0|]

```

---

## Solution de tris

### Exercice 1 (Tri bulle)

0

L'algorithme du tri bulle proposé dans le cours effectue un nombre constant de comparaisons : comment améliorer l'algorithme afin qu'il en effectue moins en moyenne ?

### Exercice 2 (Tri par insertion)

0

L'algorithme du tri par insertion pour les vecteurs suit à la lettre la démarche de cette méthode de tri. En programmant par effets de bord, proposer un algorithme de tri par insertion plus efficace.

### Exercice 3 (Tris pour les listes)

1

- 1 Proposer un algorithme de fusion de deux listes ordonnées.
- 2 Proposer un algorithme de recherche dichotomique dans une liste chaînée triée.
- 3 Faire de même pour les tris par insertion, par sélection, et à bulles pour les listes chaînées.
- 4 Programmer les tris fusion et rapide pour les listes chaînées.

### Exercice 4 (Stabilité des algorithmes de tri)

2

Un algorithme de tri est dit *stable* s'il n'échange jamais des termes égaux en des indices distincts. Parmi les algorithmes proposés, lesquels sont-ils stables ? Comment rendre tout algorithme de tri stable ?





## Solution de listes

```
(* Image miroir *)
```

```
# let rec image_miroir_bof = function
  | [] -> []
  | a :: q -> image_miroir_bof q @ [a];;
image_miroir_bof : 'a list -> 'a list = <fun>

# image_miroir_bof [1; 4; 9; 13];;
- : int list = [13; 9; 4; 1]

# let rec miroir l (accu : int list) = match l with
  | [] -> accu
  | a :: q -> miroir q (a :: accu);;
miroir : int list -> int list -> int list = <fun>

# let image_miroir l = miroir l [];;
image_miroir : int list -> int list = <fun>

# trace "miroir";;
The function miroir is now traced.
- : unit = ()
```

```
# image_miroir [1; 3; 8; 9];;
miroir <- [1; 3; 8; 9]
miroir -> <fun>
miroir* <- []
miroir <- [3; 8; 9]
miroir -> <fun>
miroir* <- [1]
miroir <- [8; 9]
miroir -> <fun>
miroir* <- [3; 1]
miroir <- [9]
miroir -> <fun>
miroir* <- [8; 3; 1]
miroir <- []
miroir -> <fun>
miroir* <- [9; 8; 3; 1]
miroir* -> [9; 8; 3; 1]
miroir* -> [9; 8; 3; 1]
miroir* -> [9; 8; 3; 1]
miroir* -> [9; 8; 3; 1]
miroir* -> [9; 8; 3; 1]
miroir* -> [9; 8; 3; 1]
- : int list = [9; 8; 3; 1]
```

```
(* Définition d'un type liste *)
```

```
# type 'a Liste = Nil | Cons of 'a * ('a Liste);;
Type Liste defined.
```

```
(* Sémantique des fonctions de manipulation d'une liste *)
```

```
(* Prédicat testant si une liste est vide *)
```

```
# let est_vide = function
  | Nil -> true
  | _ -> false ;;
est_vide : 'a Liste -> bool = <fun>
```

```
# est_vide (Cons (1, Nil));;
- : bool = false
```

```
(* Constructeur d'ajout en tête de liste *)
```

```
# let ajout a l = Cons (a, l);;
ajout : 'a -> 'a Liste -> 'a Liste = <fun>
```

```
# ajout 3.8 Nil;;
- : float Liste = Cons (3.8, Nil)
```

```
(* Fonctions de sélection *)
```

```
# let tete = function
  | Nil -> failwith "pas_de_tête_pour_la_liste_vide"
  | Cons (a, q) -> a;;
tete : 'a Liste -> 'a = <fun>
```

```
# let queue = function
  | Nil -> failwith "pas_de_tête_pour_la_liste_vide"
  | Cons (a, q) -> q;;
queue : 'a Liste -> 'a Liste = <fun>
```

```
# let longueur_iter l = let temp = ref l and i = ref 0 in
  while !temp <> Nil do
    i := !i + 1;
    temp := queue !temp;
  done;
  !i;;
longueur_iter : 'a Liste -> int = <fun>
```

```
# longueur_iter (Cons (1, (Cons (3, Cons( 8, Nil)))));;
- : int = 3
```

```
(* Insertion et suppression *)
```

```
# let rec insertion elt pos l = match (l, pos) with
  | (_, 1) -> elt :: l
  | ([], _) -> failwith "Mauvaise_position_d'insertion"
  | (a :: q, _) -> a :: (insertion elt (pos - 1) q);;
```

```
# insertion 4 2 [1; 2; 3; 8];;
- : int list = [1; 4; 2; 3; 8]
```

```
# let rec suppression pos l = match (l, pos) with
  | ([], _) -> failwith "Rien_à_supprimer_dans_cette_liste_à_cette_position"
  | (a :: q, 1) -> q
  | (a :: q, _) -> a :: (suppression (pos - 1) q);;
insertion : 'a -> int -> 'a list -> 'a list = <fun>
```

```

suppression : int -> 'a list -> 'a list = <fun>

# suppression 4 [1; 4; 9; 6; 12];;
- : int list = [1; 4; 9; 12]

(* Programmes itératifs sur les listes *)

# let rec map_bis f = function
  | [] -> []
  | a :: q -> f a :: (map_bis f q);;
map_bis : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map_bis (function x -> x * x) [1; 4; 9];;
- : int list = [1; 16; 81]

# let rec itlist f b = function
  | [] -> b
  | a :: q -> itlist f (f b a) q;;
itlist : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# let somme = itlist (prefix +) 0;;
somme : int list -> int = <fun>

# somme [1; 4; 9];;
- : int = 14

# let produit = itlist (prefix *) 1;;
produit : int list -> int = <fun>

# produit [2; 4; 5];;
- : int = 40

# let rec listit f l b = match l with
  | [] -> b
  | a :: q -> f a (listit f q b);;
listit : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

# let somme_bis l = listit (prefix +) l 0;;
somme_bis : int list -> int = <fun>

# somme_bis [1; 4; 9];;
- : int = 14

# let produit_bis l = listit (prefix *) l 1;;
produit_bis : int list -> int = <fun>

# produit_bis [2; 4; 5];;
- : int = 40

# let conse a l = a :: l;;
conse : 'a -> 'a list -> 'a list = <fun>

# let concat = listit conse;;
concat : '_a list -> '_a list -> '_a list = <fun>

# concat [1; 3; 5] [2; 4; 6];;
- : int list = [1; 3; 5; 2; 4; 6]

# let map_ter f l = let g a q = f a :: q in listit g l [];;

```

```
map_ter : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
# map_ter (function x -> x * x) [2; 5; 9];;  
- : int list = [4; 25; 81]
```

---

## Solution de piles et files

---

```

(* Rotation d'une pile *)

# let transfert pile1 pile2 = while not est_vide pile1 do empile (depile pile1) pile2 done;;
  transfert : 'a pile -> 'a pile -> unit = <fun>

# let rotation pile = let a = depile pile and aux = pile_vide () in
  transfert pile aux;
  empile a pile;
  transfert aux pile;;
  rotation : 'a pile -> unit = <fun>

# let p = {contenu = [1; 2; 3; 4; 5]};;
  p : int pile = {contenu = [1; 2; 3; 4; 5]}

# rotation p;;
- : unit = ()

# p;;
- : int pile = {contenu = [2; 3; 4; 5; 1]}

(* Expressions algébriques postfixées *)

# type operateur = Plus | Moins | Mult | Divise;;
  Type operateur defined.

# type Lexeme = Var of float | Op of operateur;;
  Type Lexeme defined.

# let valeur ope (a,b) = match ope with
  | Plus -> a +. b
  | Moins -> a -. b
  | Mult -> a *. b
  | Divise -> a /. b;;
  valeur : operateur -> float * float -> float = <fun>

# let action_sur_pile op pile = match op with
  | Var c -> empile c pile;
  | Op ope -> empile (valeur ope (depile pile, depile pile)) pile;;
  action_sur_pile : Lexeme -> float pile -> unit = <fun>

# let pile = pile_vide ();; empile 3. pile;; empile 2. pile;;
  pile : 'a pile = {contenu = []}
- : unit = ()
- : unit = ()

# action_sur_pile (Op Mult) pile;;
- : unit = ()

# pile;;

```

```

- : float pile = {contenu = [6.0]}

# let rec eval pile = function
  | [] -> ()
  | a :: q -> action_sur_pile a pile; eval pile q;;
eval : float pile -> Lexeme list -> unit = <fun>

# let evaluate_EAP l = let temp = pile_vide () in eval temp l; depile temp;;
evaluate_EAP : Lexeme list -> float = <fun>

# evaluate_EAP [Var 1.; Var 2.; Op Divise];;
- : float = 0.5

# evaluate_EAP [Var 3.5; Var 4.5; Op Moins];;
- : float = -1.0

# evaluate_EAP
  [Var 1.; Var 2.; Op Divise; Var 2.; Var 3.4; Var 4.2; Op Plus; Op Mult; Op Mult];;
- : float = 7.6

(* Bon parenthésage *)

# let depile_sans_effet pile = match pile.contenu with
  | [] -> failwith "Pile_vide"
  | a :: q -> pile.contenu <- q;;
depile_sans_effet : 'a pile -> unit = <fun>

# let rec bon_parenthesage mot = let temp = pile_vide () in
  for i = 0 to (string_length mot - 1) do
    if mot.[i] = '(' then empile 1 temp else depile_sans_effet temp;
  done;
  est_vide temp;;
bon_parenthesage : string -> bool = <fun>

# bon_parenthesage "(((())())";;
- : bool = false

# bon_parenthesage "(()())";;
Uncaught exception: Failure "Pile_vide"

# bon_parenthesage "((())())";;
- : bool = true

```

---