

Licence S.T.S - Semestre 2

Algorithmes et Programmation Impérative 1

année universitaire 2013-2014



Licence Creative Commons 

Introduction

Un programme informatique fait ce que vous lui dites de faire, pas ce que vous voudriez qu'il fasse.

Pour comprendre l'informatique, il faut arriver à être aussi bête qu'un ordinateur. (G. Berry, leçon inaugurale au collège de France, 2009)

Ce document est le support du cours d'API1 (Algorithmes et Programmation Impérative 1) proposé au semestre 2 de la première année de licence Sciences, Technologies, Santé de l'université Lille 1, dans les parcours Aménagé, MASS, MIMP, PEIP et SPI.

Pour être suivi sans difficulté, ce cours suppose de la part des étudiants qu'ils aient quelques notions de programmation impérative : expressions, instructions élémentaires (affectation, impressions), instructions conditionnelles, itérations conditionnelles (boucle tant que) ou non (boucle pour), fonctions et procédures. Et tout cela de préférence dans le langage CAML. C'est normalement le cas de tout étudiant ayant suivi l'UE obligatoire InitProg (Initiation à la Programmation) proposée au S1.

Les étudiants qui le souhaitent pourront poursuivre l'étude de la programmation dans les profils MIMP et SPI au troisième semestre avec le cours API2 (Algorithmes et Programmation Impérative 2).

Présentation générale du cours

Le cours d'InitProg est principalement axé sur la présentation des structures de contrôle (séquence d'instructions, instructions conditionnelles, itérations) et sur la conception de fonctions et procédures paramétrées simples. Les données manipulées restent simples : nombres (entiers ou flottants), booléens, caractères. Seule exception les chaînes de caractères.

Le cours d'API1 porte principalement sur la manipulation de données plus élaborées :

1. n -uplets ;
2. enregistrements ;
3. tableaux ;

et présente quelques algorithmes classiques de recherche dans un tableau, et de tris.

Les fichiers y sont aussi abordés.

Autres lectures

Lire un polycopié de cours c'est bien. Accompagner sa lecture de la lecture d'autres ouvrages c'est encore mieux.

Voici quelques livres dont la lecture (même partielle) peut être profitable à tout étudiant :

Concepts et outils de programmation ([TAH92])

Apprentissage de la programmation avec OCAML ([CD04])

Développement d'applications avec OBJECTIVE CAML ([EC00])

Ces notes de cours doivent souvent leur inspiration à l'un ou l'autre de ces livres.

Conventions utilisées dans le polycopié

Les algorithmes de résolution de problème sont décrits en « pseudo-code » sous la forme

Algorithme 0.1 Calcul de $n!$

Entrée : un entier $n \geq 0$

Sortie : $n!$

$f \leftarrow 1$

pour i allant de 1 à n **faire**

$f \leftarrow f \times i$

fin pour

renvoyer f

La traduction d'un algorithme en CAML, ainsi que les programmes sont présentés sous la forme

```
let fact (n) =
  let f = ref 1
  in
    for i = 1 to n do
      f := !f * i
    done;
  !f
```

Les dialogues avec l'interprète du langage sont décrits sous la forme

```
# let a = 5 ;;
val a : int = 5
# fact (5) ;;
- : int = 120
```

Enfin les commandes effectuées dans un terminal sont présentées sous la forme

```
> ocamlc -c factorielle.ml
```

dans laquelle le symbole `>` désigne le prompt du terminal (qui peut avoir en pratique des formes diverses selon les systèmes d'exploitation et les configurations personnelles).

Lorsqu'une nouvelle structure syntaxique du langage est présentée pour la première fois, elle l'est sous la forme

Syntaxe : Forme générale de la déclaration d'un type

```
type nom_type = definition
```

Le site du cours

Le site du cours se trouve à l'adresse :

<http://www.fil.univ-lille1.fr/portail/ls2/api1>

Vous pouvez y accéder à l'aide de votre « téléphone élégant » (smartphone pour les anglophones) grâce au QR-code de la figure 1.



FIGURE 1 – QR-code du site du cours

Équipe pédagogique 2013-2014

Ahmad Aljundi, Guillaume Dubuisson, François Lemaire, Jean-Luc Levaire, Didier Mailliet, Nour-Eddine Oussous, Maude Pupin, Yosra Rekik, Éric Wegrzynowski, Léopold Weinberg.

Licence Creative Commons

Ce polycopié est soumis à la licence CREATIVE COMMONS ATTRIBUTION PAS D'UTILISATION COMMERCIALE PARTAGE À L'IDENTIQUE 3.0 FRANCE (<http://creativecommons.org/licenses/by-nc-sa/3.0/fr/>).

Chapitre 1

Retour sur les fonctions

Ce chapitre va préciser certains points sur la notion de fonction abordée dans le cours d'Init-
Prog.

Comme d'autres langages (le C par exemple), CAML ne fait aucune distinction entre fonction et procédure. Aucun élément syntaxique ne peut les différencier. Par conséquent, tout ce qui est dit dans ce chapitre s'applique aussi aux procédures puisqu'elles ne sont qu'une catégorie particulière de fonctions.

1.1 Fonctions à plusieurs paramètres

1.1.1 Schéma général du type d'une fonction

Toute fonction a pour vocation d'effectuer certaines opérations à partir de données passées en paramètres (ou en arguments) en vue de produire un résultat.

Par exemple, la fonction `sqrt` prédéfinie en CAML permet de calculer la racine carrée d'un nombre réel (positif¹).

```
# sqrt ;;  
- : float -> float = <fun>  
# sqrt (2.) ;;  
- : float = 1.41421356237309515
```

Le type `float → float` signalé par CAML signifie que la fonction prend comme paramètre un flottant et renvoie un flottant.

Toute fonction est caractérisée par le type t_1 de son paramètre et le type t_2 des valeurs qu'elle renvoie. On dit que le type de la fonction est

$$t_1 \rightarrow t_2,$$

que nous nommerons *schéma général du type d'une fonction*.

1.1.2 Une fonction à deux paramètres

Certaines fonctions demandent naturellement plusieurs paramètres. Par exemple, la fonction `add` qui, à deux entiers a et b , associe l'entier $a + b$

$$\begin{array}{ccc} \text{add} : & \mathbb{N} \times \mathbb{N} & \longrightarrow \mathbb{N} \\ & a, b & \longmapsto a + b \end{array}$$

1. On reviendra sur ce détail ultérieurement dans la section 1.4

demande deux entiers en paramètre.

Dans le cours d'InitProg, nous avons vu comment déclarer de telles fonctions en CAML. La fonction `add` peut être déclarée comme ceci :

```
let add (a,b) = a + b
```

Pour cette fonction ainsi déclarée, le schéma général du type d'une fonction donne $t_1 = \text{int} * \text{int}$, et $t_2 = \text{int}$. $t_1 = \text{int} * \text{int}$ signifie que le paramètre est un *couple* de deux entiers (nous reviendrons plus précisément sur la notion de couple dans la section suivante). Cela signifie que tout appel à la fonction `add` doit être fait sur un couple de deux entiers. Par exemple :

```
# add (1,2) ;;
- : int = 3
```

1.1.3 Une autre formulation de la même fonction

En CAML, il existe une autre façon de réaliser la même fonction. La voici

```
let add' a b = a + b
```

Quelles différences syntaxiques existe-t-il entre la déclaration de la fonction `add'` et celle de la fonction `add`? À droite du signe `=`, aucune! En revanche entre le nom de la fonction et le signe `=`, on constate que le couple `(a,b)` a été remplacé par deux paramètres `a` et `b` séparés par des espaces, et sans aucune parenthèses.

Pour appliquer cette fonction à deux entiers, on écrit tout simplement son nom suivi des deux entiers à ajouter, séparés par des espaces et sans parenthèses.

```
# add' 1 2 ;;
- : int = 3
```

Quelle différence de type, ces déclarations différentes entraînent-elles?

```
# add' ;;
- : int -> int -> int = <fun>
```

On peut constater que le type donné par l'interprète du langage pour la fonction `add'` est `int → int → int` et donc, selon le schéma général du type d'une fonction, on a $t_1 = \text{int}$, et $t_2 = \text{int} \rightarrow \text{int}$. Autrement dit, la fonction `add'` est une fonction qui appliquée à un `int` renvoie une valeur qui est de type `int → int`, c'est-à-dire une fonction. Ce point peut être vérifié en appliquant la fonction `add'` à un seul entier au lieu de deux :

```
# add' 1 ;;
- : int -> int = <fun>
```

Le résultat de cette application à un seul entier donne bien une fonction de type `int → int`.

Puisque l'expression `add' 1` est une fonction, on doit pouvoir l'appliquer à un entier, ce qu'on vérifie immédiatement :

```
# (add' 1) 2 ;;
- : int = 3
```


1.1.4 Un autre exemple

Considérons un autre exemple de fonction, avec trois paramètres cette fois-ci. La fonction que nous allons envisager est la fonction qui, à une chaîne de caractères **s**, un entier **debut** et un deuxième entier **long**, associe la chaîne de caractères **s'** constituée des **long** caractères consécutifs de **s** en débutant à l'indice **debut**. Nous nommerons cette fonction **sous_chaine**. Par exemple, la sous-chaîne de longueur 4 de la chaîne **s="timoleon"** débutant à l'indice 3 est **s'="oleo"**. Bien entendu, cette fonction n'est définie que si l'indice de début est compris entre 0 et $|s| - 1$ et que la longueur voulue ne soit pas trop grande.

$$\begin{aligned} \text{sous_chaine} : \text{Chaine} \times \mathbb{N} \times \mathbb{N} &\longrightarrow \text{Chaine} \\ s, \text{debut}, \text{long} &\longmapsto s' = s[\text{debut}..\text{debut} + \text{long} - 1] \\ \text{CU} : 0 \leq \text{debut} < |s| \text{ et } \text{debut} + \text{long} \leq |s| \end{aligned}$$

```
let sous_chaine (s,debut,long) =
  let s' = String.create long
  in
    for i = 0 to long - 1 do
      s'.[i] <- s.[debut + i]
    done ;
  s'
```

Cette réalisation de la fonction **sous_chaine** donne une fonction de type

$$\text{string} * \text{int} * \text{int} \rightarrow \text{string}.$$

Selon le schéma général du type d'une fonction, on a donc $t_1 = \text{string} * \text{int} * \text{int}$ et $t_2 = \text{string}$. t_1 est un *triplet* composé d'une chaîne de caractères et de deux entiers (nous reviendrons sur les triplets dans la section suivante). Cela signifie que cette fonction ne peut s'appliquer qu'à des triplets :

```
# sous_chaine ("timoleon",3,4) ;;
- : string = "oleo"
```

Comme pour la fonction **add**, il existe une autre façon de programmer cette fonction.

```
let sous_chaine' s debut long =
  let s' = String.create long
  in
    for i = 0 to long - 1 do
      s'.[i] <- s.[debut + i]
    done ;
  s'
```

Encore une fois la seule différence syntaxique entre les déclarations de ces deux fonctions réside dans la description des paramètres :

- des virgules séparant les paramètres le tout entre parenthèses pour la première;
- absence de virgules (remplacées par des espaces) et de paranthèses pour la seconde.

On vérifie que la fonction appliquée sur les mêmes données donne le même résultat :

```
# sous_chaine' "timoleon" 3 4 ;;
- : string = "oleo"
```

Mais le type de la fonction ainsi réalisée est devenu

$$\text{string} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{string}.$$

Pour cette fonction, on a $t_1 = \text{string}$ et $t_2 = \text{int} \rightarrow \text{int} \rightarrow \text{string}$. La fonction `sous_chaine'` peut donc s'appliquer à une chaîne de caractères seulement pour donner une fonction de type $\text{int} \rightarrow \text{int} \rightarrow \text{string}$. Vérifions-le :

```
# sous_chaine' "timoleon" ;;
- : int -> int -> string = <fun>
```

Et on peut même appliquer cette fonction à une chaîne et un entier seulement pour obtenir une fonction de type $\text{int} \rightarrow \text{string}$.

```
# sous_chaine' "timoleon" 3 ;;
- : int -> string = <fun>
```

Remarque : cette fonction `sous_chaine'` que nous venons de voir est prédéfinie en CAML dans le module `String` et se nomme `sub`. En cas de besoin, on peut faire appel à elle avec le nom pleinement qualifié `String.sub`.

1.1.5 Formes curryfiées, formes décurryfiées

Nous avons donc constaté qu'en CAML, lorsque nous avons à programmer une fonction à plusieurs paramètres, nous avons le choix entre deux possibilités

1. définir une fonction de type

$$t_1 * t_2 * \dots * t_n \rightarrow t,$$

2. ou bien, définir une fonction de type

$$t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t,$$

Les fonctions programmées selon le second schéma sont dites *curryfiées*, du nom du mathématicien Haskell Curry (1900-1982), et *décurryfiées* pour celles qui le sont selon le premier schéma.

Déclaration : La syntaxe générale de déclaration d'une fonction sous forme curryfiée n'utilise pas de parenthèses ni de virgules.

Syntaxe : Déclaration d'une fonction sous forme curryfiée

```
let fonction param1 ... paramn = expression
```

Tandis que celle d'une fonction sous forme décurryfiée nécessite parenthèses et virgules.

Syntaxe : Déclaration d'une fonction sous forme décurryfiée

```
let fonction (param1, ... ,paramn) = expression
```

Application : l'application d'une fonction décurryfiée s'effectue nécessairement sur la totalité des paramètres, séparés par des virgules le tout entouré de parenthèses.

Syntaxe : Application d'une fonction sous forme décurryfiée

```
fonction (param1, ... ,paramn)
```

En revanche, l'application d'une fonction curryfiée à ses paramètres ne nécessite ni parenthèses ni virgules.

Syntaxe : Application d'une fonction sous forme curryfiée

fonction param₁ ... param_n

L'application peut même être *partielle*.

Remarque : pour les fonctions à un seul paramètre, il n'y a aucune distinction entre forme curryfiée et forme décurryfiée.

1.1.6 Passage d'une forme à l'autre

La quasi-totalité des fonctions prédéfinies dans les diverses bibliothèques qui accompagnent le langage OBJECTIVE CAML sont déclarées sous forme curryfiée.

Il est facile de passer d'une fonction déclarée sous une forme à l'autre. Nous allons voir le principe général de passage d'une forme à l'autre.

Si f est sous forme curryfiée de $t_1 \rightarrow t_2 \rightarrow t_3$, on obtient une fonction f' sous forme décurryfiée de type $t_1 * t_2 \rightarrow t_3$ telle que

$$f \ x \ y = f' \ (x,y)$$

pour toutes les données x et y , en déclarant :

```
let f' (x,y) = f x y
```

Inversement, à partir d'une fonction f' sous forme décurryfiée, on obtient la fonction curryfiée équivalente en déclarant :

```
let f x y = f' (x,y)
```

1.1.7 Du bon usage des parenthèses

En fait la notation `(add' 1) 2` s'écrit plus simplement `add' 1 2` que nous avons utilisée plus haut. On pourrait aussi parenthéser complètement en écrivant `((add' 1) 2)`. Mais on ne peut pas parenthéser uniquement l'ensemble des paramètres

```
# add' (1 2) ;;
This expression is not a function, it cannot be applied
```

Le message d'erreur s'explique par le fait que la sous-expression `(1 2)` signifie appliquer la fonction `1` à `2`. Or `1` n'est pas une fonction.

En revanche, on peut parenthéser individuellement chacun des paramètres, ce qui peut s'avérer nécessaire dans le cas où un paramètre effectif est lui-même une expression composée.

```
# let add' x y = x + y ;;
val add' : int -> int -> int = <fun>
# add' -1 2 ;;
This expression has type int -> int -> int but is here used with type int
# add' (-1) 2 ;;
- : int = 1
```

Enfin, rappelons que lorsqu'une fonction est définie sous forme décurryfiée, il est nécessaire de parenthéser le n -uplet passé en paramètre. En cas d'oubli du parenthésage, une erreur de typage se produit qui interdit l'évaluation de l'expression.

```
# add 1,2 ;;
This expression has type int but is here used with type int * int
```

1.2 Couples, triplets, n -uplets

La section qui précède l'a montré, en CAML il est possible de travailler avec des couples, ou bien des triplets ou plus généralement des n -uplets de valeurs.

Ces couples, triplets ou n -uplets sont des structures de données simples qui permettent de rassembler plusieurs valeurs en une seule.


```
# 1,2 ;;
- : int * int = (1, 2)
# 1,1,1 ;;
- : int * int * int = (1, 1, 1)
# 3,1,4,1,6 ;;
- : int * int * int * int * int = (3, 1, 4, 1, 6)
# "timoleon",(sqrt 2.), 'a' ;;
- : string * float * char = ("timoleon", 1.41421356237309515, 'a')
# "timoleon",sqrt 2., 'a' ;;
- : string * float * char = ("timoleon", 1.41421356237309515, 'a')
```

1.2.1 Le type d'un n -uplet

Le type général d'un couple est $t_1 * t_2$, où t_1 est le type de la première composante et t_2 celui de la seconde. Le type général d'un triplet est $t_1 * t_2 * t_3$. Plus généralement, le type d'un n -uplets est

$$t_1 * t_2 * \dots * t_n,$$

les types t_1, t_2, \dots, t_n étant les types respectifs des première, deuxième, ..., n -ième composantes.

 CAML distingue un triplet de type $t_1 * t_2 * t_3$ d'un couple de type $(t_1 * t_2) * t_3$. Dans le premier cas, la première composante est de type t_1 , et dans le second cas de type $t_1 * t_2$.

```
# 1,sqrt 2.,true ;;
- : int * float * bool = (1, 1.41421356237309515, true)
# (1,sqrt 2.),true ;;
- : (int * float) * bool = ((1, 1.41421356237309515), true)
```

1.2.2 Valeurs d'un type n -uplet

On obtient un n -uplet en assemblant tout simplement plusieurs valeurs séparées par des virgules.

Remarque : observons dans les exemples donné au début de cette section, qu'aucune parenthèse ne figure dans les expressions données à évaluation, mais qu'en retour, l'interprète en a ajouté systématiquement. Les parenthèses sont optionnelles lorsqu'il n'y a pas ambiguïté. Dans certains contextes, en mettre ou non est indifférent, comme par exemple

```
# (1,2) ;;
- : int * int = (1, 2)
```

```
# 1,2 ;;
- : int * int = (1, 2)
```

En revanche, dans d'autres contextes, comme l'application d'une fonction à un *n*-uplet, elles sont indispensables. Voici avec l'exemple de la fonction **add** de la section 1.1.

```
# add (1,2) ;;
- : int = 3
# add 1,2 ;;
This expression has type int but is here used with type int * int
```

1.2.3 Comparaison de *n*-uplets

Deux couples sont égaux si et seulement si les deux composantes de ces couples sont égales entre elles. Plus généralement, deux *n*-uplets (x_1, x_2, \dots, x_n) et (y_1, y_2, \dots, y_n) sont égaux si et seulement si $x_i = y_i$ pour tout i compris entre 1 et n .

En CAML, la comparaison s'effectue à l'aide de l'opérateur usuel `=`.

```
# (1,2) = (1,1+1) ;;
- : bool = true
# (1,2) = (1,3) ;;
- : bool = false
```

Mais il faut prendre garde à bien parenthéser les deux *n*-uplets.

```
# 1,2 = 1,3 ;;
- : int * bool * int = (1, false, 3)
```

1.2.4 Variables de type *n*-uplet

```
# let s = 1,2 ;;
val s : int * int = (1, 2)
# add s ;;
- : int = 3
```

1.2.5 Fonction renvoyant un *n*-uplet

Une fonction ne peut renvoyer qu'une seule valeur à la fois. Si nous avons besoin de réaliser une fonction qui renvoie plusieurs valeurs au lieu d'une, une possibilité (parmi d'autres que nous verrons dans des chapitres ultérieurs) est de renvoyer un *n*-uplet.

Par exemple, la fonction

$$\begin{array}{ccc} \text{plus_et_moins} : & \mathbb{Z}^2 & \longrightarrow \mathbb{Z}^2 \\ & (n, p) & \longmapsto (n + p, n - p) \end{array}$$

peut être réalisée comme une fonction de type `int * int → int * int` en la codant comme il suit :

```
let plus_et_moins (n,p) =
  (n + p, n - p)
```

et voici un exemple d'application de cette fonction.

```
# plus_et_moins (1,2) ;;
- : int * int = (3, -1)
```

Le résultat d'une fonction renvoyant un n -uplet peut être utilisé comme argument d'une fonction admettant un n -uplet du même type en paramètre. En voici deux exemples :

```
# add (plus_et_moins (1,2)) ;;
- : int = 2
# plus_et_moins (plus_et_moins (1,2)) ;;
- : int * int = (2, 4)
```

Notons aussi une possibilité de récupérer dans n variables distinctes chacune des composantes d'une fonction renvoyant un n -uplet en une seule déclaration grâce à la possibilité de déclaration simultanée de variables à l'aide de n -uplets.

Syntaxe : Déclaration simultanée à l'aide de n -uplets

```
let  $id_1, id_2, \dots, id_n = expr_1, expr_2, \dots, expr_n$ 
```

Voici un exemple de déclaration simultanée :

```
# let x,y = 1, 2 ;;
val x : int = 1
val y : int = 2
```

Cette déclaration est complètement équivalente à

```
# let x = 1
  and y = 2 ;;
val x : int = 1
val y : int = 2
```

Les déclarations de n -uplets de variables s'avèrent utiles lorsqu'on doit attribuer à plusieurs variables les valeurs des composantes d'un n -uplet renvoyé par une fonction.

```
# let x,y = plus_et_moins (1,2) ;;
val x : int = 3
val y : int = -1
```

1.3 Filtrage

Il est fréquent qu'en programmation se fasse sentir la nécessité de distinguer plusieurs cas. Tous les langages de programmation proposent pour cela des expressions ou instructions conditionnelles (**if ... then ... else ...**). Ces expressions conditionnelles permettent de distinguer deux cas : celui où la condition est vraie, et le cas contraire. Lorsque il est nécessaire de distinguer plus de deux cas, il faut imbriquer les expressions conditionnelles.

1.3.1 Un premier exemple

À titre d'exemple, supposons que nous ayons à programmer une fonction de type `int → int` nulle pour presque tous les entiers, sauf pour les entiers 0, 1 et 2 pour lesquels elle vaut respectivement 1, -1 et 2. On la programme simplement avec des expressions conditionnelles imbriquées.

```

let f n =
  if n = 0 then
    1
  else if n = 1 then
    -1
  else if n = 2 then
    2
  else
    0

```

Cette imbrication peut s'avérer parfois un peu lourde et fastidieuse. C'est pourquoi certains langages de programmation offrent d'autres moyens d'exprimer ces distinctions de cas. Le langage CAML est un tel langage qui offre l'expression **match ... with ...**. Voici la fonction précédente redéfinie à l'aide d'une telle expression.

```

let f n =
  match n with
  | 0 -> 1
  | 1 -> -1
  | 2 -> 2
  | _ -> 0

```

Comment comprendre une telle expression ? Comment s'évalue-t-elle ?

Tout d'abord, syntaxiquement parlant, elle est constituée d'une ligne contenant les deux mots **match** et **with** entourant une expression qui est celle sur laquelle doit porter la distinction des cas. Dans le cas présent cette expression est l'entier **n** paramètre de la fonction dont on doit distinguer les différentes valeurs possibles. Après, viennent plusieurs lignes de même forme `| filtre -> expression`.

L'évaluation d'une telle expression se fait en cherchant, dans l'ordre dans lequel ils sont écrits, le premier filtre qui convient à la valeur de l'expression à filtrer. Dans notre exemple, les filtres étant des nombres entiers particuliers, si l'entier à filtrer est nul, le premier filtre convient, les autres sont ignorés et la valeur de l'expression est 1. Si en revanche il est égal à 1, le premier filtre ne convient pas, mais le second si, et en conséquence les deux filtres suivants ne sont pas considérés et la valeur de l'expression est -1.

Le dernier filtre, désigné par `_`, est nommé *filtre universel*. C'est un filtre qui convient à toute valeur. Il peut être considéré comme signifiant *dans tous les cas*.

1.3.2 Filtrage sur des *n*-uplets

Le filtrage est un très bon moyen d'accéder aux composantes d'un *n*-uplet.

Par exemple, lorsqu'on écrit

```

let premiere_composante (x,y) = x

```

on obtient une fonction qui renvoie la première composante d'un couple. On peut aussi l'écrire d'une autre façon en remplaçant le paramètre **(x,y)** qui met en évidence qu'il est un couple (ce qui est une forme de filtrage implicite), par un paramètre désigné par un identificateur unique, et en utilisant une expression de filtrage explicite. Voici cette autre façon :

```

let premiere_composante c =
  match c with

```

```
| x,_ -> x
```

Le filtre `x,_` appliqué par ce filtrage au paramètre `c` impose à `c` d'être un couple. Notons l'usage du motif universel `_` pour la seconde composante.

Remarque : cette fonction `premiere_composante` est prédéfinie en CAML et se nomme `fst` (pour first). Elle est de type `'a * 'b → 'a`. C'est donc une fonction polymorphe. Il existe aussi une fonction prédéfinie sur les couples nommée `snd` (pour second), de type `'a * 'b → 'b`, qui renvoie la seconde composante d'un couple. Il n'existe pas de fonctions prédéfinies pour l'accès aux composantes d'un triplet ou plus généralement d'un n -uplet.

Donnons un autre exemple de filtrage sur des triplets cette fois-ci. La fonction qui suit, de type `int * int * int → int * int * int`, calcule l'instant qui suit un horaire représenté par un triplet d'entiers heure, minute, seconde (`h,m,s`).

```
let seconde_suivante (h,m,s) =
  match h,m,s with
  | 23,59,59 -> 0,0,0
  | h,59,59   -> h + 1,0,0
  | h,m,59    -> h,m + 1,0
  | h,m,s     -> h,m,s + 1
```

1.3.3 Possibilité de « rassembler » les filtres

Lorsque plusieurs filtres s'associent à la même expression à évaluer, il est possible de rassembler ces filtres en n'écrivant qu'une seule fois l'expression à évaluer.

```
let f n =
  match n with
  | 0 | 1 | 2 -> 1
  | 3 | 4 | 5 -> 2
  | _ -> 0
```

Cette fonction associe l'entier 1 à 0, 1 et 2, l'entier 2 à 3, 4 et 5, et l'entier 0 à tous les autres.

1.3.4 L'expression `match ... with ...`

La syntaxe générale d'une expression de filtrage est donnée ci-dessous.

Syntaxe : Expression `match ... with ...`

```
match expression with
| filtre1 -> expr1
| filtre2 -> expr2
...
| filtren -> exprn
```

Dans cette syntaxe,

- *expression* est l'expression à filtrer ;
- *expr₁*, *expr₂*, ..., *expr_n* sont des expressions qui doivent toutes être du même type ;
- *filtre₁*, *filtre₂*, ..., *filtre_n* sont des *filtres* qui doivent eux aussi tous être de même type. *expr₁*, *expr₂*, ..., *expr_n* sont des expressions qui doivent toutes être du même type.

Un filtre peut être

- une constante littérale de n’importe quel type² ;
- un identificateur ;
- le symbole `_`, appelé motif universel ;
- un n -uplet constitué des éléments précédents.

Cette description des filtres n’est pas exhaustive, et pourra être complétée par la suite.

L’évaluation d’une expression de filtrage se fait en cherchant le premier filtre qui convient à l’expression filtrée. Ce filtre trouvé, l’expression qui lui est associée (à droite de la flèche `->`) est évaluée, et sa valeur est la valeur de l’expression **match ... with ...** entière.



Les filtres se doivent d’être exhaustifs, c’est-à-dire recouvrir tous les cas possibles pour l’expression à filtrer. La non exhaustivité du filtrage n’est pas une erreur pour le langage qui se contente d’un message d’avertissement (**Warning**), comme on peut le voir ci-dessous

```
# match 1,2 with
| 0,_ -> 1
| _,0 -> 2
| 1,2 -> 3 ;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(2, 1)
- : int = 3
```

mais c’est très certainement une mauvaise idée pour le programmeur que de laisser traîner de tels avertissements qui peuvent être le signe d’une mauvaise analyse ou d’un oubli. L’absence d’exhaustivité peut être source de déclenchement d’exception :

```
# match 1,2 with
| 0,_ -> 1
| _,0 -> 2 ;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(1, 1)
Exception: Match_failure ("", 23, -49).
```

L’usage du motif universel peut s’avérer utile pour assurer l’exhaustivité du filtrage.

```
# match 1,2 with
| 0,_ -> 1
| _,0 -> 2
| 1,2 -> 3
| _ -> 4 ;;
- : int = 3
```



Un filtre ne peut contenir deux fois le même identificateur. Par exemple, il serait tentant d’écrire l’expression suivante pour décider si un couple est constitué de deux composantes égales

```
match c with
| (x,x) -> true
| _ -> false
```

Mais, CAML interdit de tel filtre avec le message d’erreur

Variable x is bound several times in this matching.

2. à l’exception de type fonctionnel

1.4 Des exceptions pour les fonctions partiellement définies

Revenons sur la fonction prédéfinie `sqrt` évoquée dans la section 1.1. Cette fonction a pour type `float → float`. Cela signifie-t-il que cette fonction est définie pour tout nombre flottant ? Du point de vue mathématique, $\sqrt{-1}$ n'est certainement pas un nombre réel ! Observons ce qu'il en est en CAML.

```
# sqrt (-1.) ;;
- : float = nan
```

Nous remarquons donc qu'en CAML, $\sqrt{-1}$ a une valeur `nan`. Autrement dit en CAML, $\sqrt{-1}$ est bien défini et est une valeur de type `float`. Cette valeur `nan` signifie *Not A Number*, autrement dit *pas un nombre*. C'est une valeur particulière du type `float` pour désigner des nombres mal définis.

Cette valeur `nan` est une valeur comme les autres, et en particulier on peut faire des calculs avec elle.

```
# 1. +. nan ;;
- : float = nan
# 2. *. nan ;;
- : float = nan
# 1. /. nan ;;
- : float = nan
# (1. +. sqrt (-1.)) /. 2. ;;
- : float = nan
```

Par conséquent, si dans une expression arithmétique ne portant que sur des nombres flottants, une sous-expression a pour valeur `nan`, alors l'expression entière a pour valeur `nan`.

Par exemple, pour CAML, l'expression $\frac{1+\sqrt{-1}}{2}$ a une signification (la valeur `nan`), alors que pour le mathématicien elle n'en a pas (du moins s'il se restreint à travailler dans le domaine des nombres réels). Ainsi, s'il programme la fonction f qui à un réel x associe le réel $f(x) = \frac{1+\sqrt{x}}{2}$ en CAML de cette façon

```
let f x =
  (1. +. sqrt x) /. 2.
```

alors il ne pourra pas se rendre compte d'une utilisation de cette fonction ne respectant pas la contrainte $x \geq 0$.

Comment alors programmer la fonction f de sorte qu'elle ne renvoie aucune valeur lorsque $x < 0$?

Un programmeur débutant est tenté de programmer la fonction en écrivant quelque chose du genre

```
let f x =
  if x < 0. then
    "Erreur : nombre négatif"
  else
    (1. +. sqrt x) /. 2.
```

Malheureusement, cela est impossible en CAML car dans une expression conditionnelle, les deux expressions définissant les parties `then` et `else` doivent être de même type, ce qui n'est pas le cas ici (la première est de type `string` et la seconde de type `float`). Ajouter un `print_endline`

devant la chaîne n'y change rien car alors la première expression aurait le type `unit` qui n'est pas `float`.

La seule possibilité qui s'offre au programmeur est de *déclencher une exception*.

1.4.1 Qu'est-ce qu'une exception ?

Le mécanisme des *exceptions* est celui adopté dans la plupart des langages de programmation contemporains pour interrompre le déroulement de l'exécution d'un programme lorsque les conditions de son exécution ne sont pas remplies.

En CAML, les *exceptions* sont des valeurs d'un type particulier (`exn`) qui interrompent le déroulement d'un calcul. Il existe un certain nombre d'exceptions prédéfinies. En voici quelques exemples.

```
# 5 mod 0 ;;
Exception: Division_by_zero.
# let s = "timoleon"
  in
    s.[8] ;;
Exception: Invalid_argument "index_out_of_bounds".
# match 5 with
  | 0 -> 1 ;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 27, -24).
```

Comme on le voit, lorsqu'une exception est déclenchée (et si elle n'est pas récupérée³), CAML le signale par l'usage du mot **Exception**, toujours accompagné du nom de l'exception.

La première des exceptions prédéfinies rencontrées ci-dessus se nomme **Division_by_zero**. Elle est déclenchée par toute tentative de division entière par zéro⁴.

La deuxième exception se nomme **Invalid_argument**. Elle est toujours accompagnée d'une chaîne de caractères qui indique en quelques mots la raison de son déclenchement. Ici c'est **"index_out_of_bounds"** puisque que l'expression à évaluer cherche à accéder à un caractère dans une chaîne par un indice (index) trop grand (out of bounds).

Enfin la troisième exception se nomme **Match_failure**. Elle est déclenchée parce qu'aucun motif ne correspond à l'expression à filtrer. C'est un défaut d'exhaustivité du filtrage.

De manière générale, une exception interromp le déroulement de l'évaluation d'une expression ou de l'exécution d'une instruction. Voici une séquence d'instructions qui se déroule normalement, c'est-à-dire produit les affichages demandés, et donne la valeur `()`.

```
# print_string "5_mod_3=_";
  print_int (5 mod 3) ;
  print_newline () ;;
5 mod 3 = 2
- : unit = ()
```

Et maintenant voici la même séquence, mais appliquée à d'autres données, qui ne se déroule pas normalement jusqu'à son terme.

3. Nous verrons plus tard comment récupérer une exception déclenchée (cf chap. 6)

4. Seule la division entière par zéro déclenche une exception. La division des flottants par zéro ne déclenche pas d'exception, mais peut donner l'une ou l'autre des trois valeurs **infinity**, **neg_infinity**, **nan**, toutes trois de type `float`

```
# print_string "5_mod_0=_";
print_int (5 mod 0) ;
print_newline () ;;
5 mod 0 = Exception: Division_by_zero.
```

On peut noter dans cet exemple, que le premier affichage s'est effectué normalement, tandis que le second n'a pas pu être mené à son terme car l'évaluation de l'expression `5 mod 0` a déclenché l'exception `Division_by_zero` qui a interrompu le déroulement normal de la séquence, et par conséquent empêché le passage à la ligne attendu avec la dernière instruction de cette séquence.

1.4.2 Déclencher une exception

Il y a plusieurs façons de déclencher des exceptions en CAML. Nous n'en verrons qu'une seule qui déclenche toujours la même exception `Failure`.

La fonction prédéfinie `failwith` prend en argument une chaîne de caractères, et ne renvoie aucun résultat. Son évaluation n'aboutit qu'au *déclenchement d'une exception*. Son type est d'ailleurs

$$\text{string} \rightarrow 'a$$

le type `'a` signifiant un type quelconque, puisque cette fonction ne renvoie aucune valeur.

```
# failwith ;;
- : string -> 'a = <fun>
# failwith "ceci_est_un_essai" ;;
Exception: Failure "ceci_est_un_essai".
# failwith "" ;;
Exception: Failure "".
```

Comme on peut le voir dans le dialogue ci-dessus avec l'interprète, un appel à la fonction `failwith` déclenche une exception `Failure` accompagnée d'une chaîne de caractères. En particulier, un appel à la fonction `failwith` ne renvoie aucune valeur.

1.4.3 Déclencher une exception pour restreindre l'utilisation d'une fonction

La fonction `failwith` peut être très utile pour restreindre les valeurs possibles d'un type comme paramètres admissibles d'une fonction.

Voici un exemple d'utilisation de la fonction `failwith` pour restreindre la fonction prédéfinie `sqrt` aux seuls nombres positifs ou nuls :

```
let mon_sqrt x =
  if x < 0. then
    failwith "mon_sqrt:_argument_negatif"
  else
    sqrt x
```

et ce qui se passe lorsqu'on y fait appel avec un paramètre négatif :

```
# mon_sqrt (-1.) ;;
Exception: Failure "mon_sqrt:_argument_negatif".
```

Voici un autre exemple de déclenchement d'exception, pour restreindre la fonction `seconde_suivante` de type `int * int * int → int * int * int` aux seuls triplets d'entiers valides pour la représentation d'un horaire, c'est-à-dire aux triplets d'entiers compris entre 0 et 59.

```

let seconde_suivante (h,m,s) =
  if ((h < 0) || (m < 0) || (s < 0) ||
      (h > 23) || (m > 59) || (s > 59)) then
    failwith "seconde_suivante_:_horaire_incorrect"
  else
    match (h,m,s) with
    | 23,59,59 -> 0,0,0
    | h,59,59  -> h + 1,0,0
    | h,m,59   -> h,m + 1,0
    | h,m,s    -> h,m,s + 1

```

```

# seconde_suivante (-1,30,45) ;;
Exception: Failure "seconde_suivante_:_horaire_incorrect".
# seconde_suivante (1,60,45) ;;
Exception: Failure "seconde_suivante_:_horaire_incorrect".

```

1.4.4 Méthodologie

Durant tout le cours, lorsque dans la spécification d'une fonction il apparaîtra que pour certaines valeurs du type de ses paramètres la fonction n'est pas définie (contraintes d'utilisation), nous programmerons le déclenchement d'une exception pour ces valeurs à l'aide de la fonction `failwith`. La chaîne de caractères qu'on lui passera en paramètre débutera toujours par le nom de la fonction suivi de quelques mots précisant la raison du déclenchement de l'exception. Donc si pour une fonction `f` il est nécessaire de déclencher une exception pour une raison `y`, on trouvera dans le corps de la fonction l'expression

`failwith "f_:y".`

Un bémol à cette règle méthodologique de programmation : au cas où une exception prédéfinie serait naturellement déclenchée, nous nous permettrons de ne pas programmer son déclenchement dans la fonction. Par exemple, si on veut programmer la fonction `est_divisible_par` de type `int → int → bool`, qui vaut **true** si le premier entier est divisible par le second, et **false** dans le cas contraire, on se contentera d'écrire

```

let est_divisible_par n p = (n mod p = 0)

```

Une utilisation hors condition de cette fonction (`p = 0`) déclenche l'exception `Division_by_zero`

```

# est_divisible_par 12 0 ;;
Exception: Division_by_zero.

```

1.5 Exercices

1.5.1 Fonctions à plusieurs paramètres

Exercice 1-1 Parenthésage

Question 1 Pour les déclarations suivantes, indiquez lesquelles sont correctes ou non. Et pour les déclarations correctes indiquez le type.

```
let f1 (x) = x + 1
let f2 x   = x + 1
let (f3 x) = x + 1
```

Question 2 Pour les déclarations de fonctions correctes de la question précédente, indiquez quelles sont les applications correctes parmi les applications suivantes (en remplaçant f par une fonction correcte) :

```
f (1) ;;
f 1 ;;
(f 1) ;;
f -1 ;;
f (-1) ;;
f 1. ;;
```

Question 3 Parmi les déclarations suivantes, indiquez lesquelles sont correctes ou non. Et pour les déclarations correctes indiquez le type.

```
let f1 x y = x + y
let f2 (x) (y) = x + y
let f3 (x y) = x + y
let f4 (x,y) = x + y
let (f5 x y) = x + y
let f6 x,y = x + y
```

Exercice 1-2 Curryfication

En supposant qu'une fonction f écrite du langage CAML est de type $'a * 'b * 'c \rightarrow 'd$, écrivez en CAML une version curryfiée de cette fonction.

Exercice 1-3 Décurryfication

En supposant qu'une fonction f du langage CAML est de type $'a \rightarrow 'b \rightarrow 'c \rightarrow 'd$, écrivez en CAML une version décurryfiée de cette fonction.

1.5.2 Filtrage

Exercice 1-4 Ordre des clauses

On donne trois fonctions :

```
let f c = match c with
| (1,1) -> 2
| (1,_) -> 3
| (_,1) -> 4
| (_,_) -> 5
```

```
let g c = match c with
| (_,1) -> 4
| (1,_) -> 3
| (1,1) -> 2
| (_,_) -> 5
```

```
let h c = match c with
| (_,_) -> 5
| (_,1) -> 4
| (1,1) -> 2
| (1,_) -> 3
```

et on donne 4 couples :

```
let c1=(1,1)
```

```
let c2=(1,2)
```

```
let c3=(2,1)
```

```
let c4=(2,2)
```

Pour chacune des fonctions, calculez l'image de chaque couple.

Exercice 1-5 *Conversion*

Réalisez une fonction qui convertit un caractère représentant un chiffre en l'entier qui lui correspond, en utilisant un filtrage de motif.

1.5.3 Couples, triplets, n -uplets

Exercice 1-6 *Addition de couples d'entiers*

Dans cet exercice, on appelle somme de deux couples d'entiers (n_1, p_1) et (n_2, p_2) , le couple d'entiers en additionnant composante par composante les entiers des deux couples, autrement dit le couple $(n_1 + n_2, p_1 + p_2)$

Question 1 Réalisez une fonction qui à partir de deux couples d'entiers calcule la somme de ces deux couples. Vous en réaliserez une version curryfiée et une version decurryfiée. Et pour chacune de ces formes vous donnerez une version où les paramètres formels indiquent explicitement qu'ils sont des couples, et une autre version où ils sont de simples noms.

Enfin pour chacune des fonctions réalisées, vous indiquerez comment les utiliser pour calculer la somme de deux couples d'entiers.

Question 2 On veut maintenant réaliser une fonction qui calcule la somme de trois couples d'entiers. Réalisez-en plusieurs versions différentes.

Exercice 1-7 *Racines du trinôme*

Les racines réelles du trinôme $ax^2 + bx + c$ sont données par les formules

$$\begin{aligned} x_1 &= \frac{-b + \sqrt{\Delta}}{2a} \\ x_2 &= \frac{-b - \sqrt{\Delta}}{2a} \end{aligned} \tag{1.1}$$

où $\Delta = b^2 - 4ac$ est le discriminant.

Question 1

Écrivez une fonction nommée `racines` qui à partir de trois flottants renvoie le couple des deux racines réelles. Vous ne tiendrez pas compte des cas où $a = 0$ ou $\Delta < 0$.

Question 2 Testez votre fonction dans les cas suivants :

1. $a = 1, b = -1, c = -1$;
2. $a = 1, b = 0, c = -1$;
3. $a = 1, b = 1, c = -1$;
4. $a = 1, b = 2, c = 1$;
5. $a = 1, b = 1, c = 1$;
6. $a = 0, b = 1, c = 1$.

Expliquez les réponses données pour les deux derniers cas.

Exercice 1-8 *Dates*

Dans cet exercice on considère que les dates sont représentées par des triplets d'entiers j, m, a , où j désigne le numéro du jour dans le mois, m le numéro du mois et a l'année.

N'importe quel triplet d'entiers ne constitue pas une date valide. Voici quelques exemples de triplets ne représentant pas des dates valides.

- $-1, 1, 2010$
- $1, 50, 2010$
- $31, 4, 2010$
- $29, 2, 1010$

Question 1 Expliquez en quoi ces triplets ne sont pas des dates valides.

Le but de cet exercice est de créer une fonction nommée `est_date_valide` de type

`int * int * int → bool`

qui renvoie la valeur **true** si le triplet passé en paramètre représente une date valide, et la valeur **false** dans le cas contraire.

Question 2 Les années bissextiles, celles ayant 366 jours et non 365 pour les autres, sont reconnaissables par le fait qu'elles sont divisibles par 4, sauf celles divisibles par 100, avec toutefois une exception pour celles qui sont divisibles par 400. Par exemple,

- 2010 n'est pas une année bissextile puisque 2010 n'est pas divisible par 4 ;
- 2012 est une année bissextile puisque 2012 est divisible par 4 ;
- en revanche, 2100 n'est pas bissextile puisque divisible par 100 ;
- et 2000 l'est puisque divisible par 400.

Réalisez une fonction nommée `est_bissextile` de type `int → bool` qui teste si une année est bissextile.

Question 3 Réalisez une fonction nommée `est_mois_valide`, de type `int → bool`, qui teste si un entier représente bien un mois, autrement dit s'il est bien compris entre 1 et 12.

Question 4 Réalisez une fonction nommée `nb_jours` de type `int * int → int` qui donne le nombre de jours dans le mois m de l'année a , le couple d'entiers m, a étant passé en paramètre.

Question 5 Réalisez maintenant une fonction nommée `est_jour_valide` qui teste la validité du jour dans un triplet d'entiers pour représenter une date. Que doit-on fournir en paramètre ?

Question 6 Réalisez enfin la fonction `est_date_valide` signalée plus haut.

Question 7 Réalisez une fonction `lendemain` qui calcule la date du lendemain de celle passée en paramètre. On supposera que le triplet d'entiers passé en paramètre est bien une date valide.

Exercice 1-9 *fraction*

Dans cet exercice, on décide de représenter les nombres rationnels —c'est à dire les fractions $\frac{p}{q}$ où $q \neq 0$ — par des couples d'entiers. La première composante contient le numérateur. La seconde composante désigne le dénominateur. par exemple

```
# let c1=(3,4);;
val c1 : int*int = (3, 4)
# let c2=(7,8);;
val c2 : int*int = (7, 8)
# let c3=(6,8);;
val c3 : int*int = (6, 8)
```

les couples `c1`, `c2` et `c3` représente respectivement les fractions $\frac{3}{4}$, $\frac{7}{8}$ et $\frac{6}{8}$. Notez bien que la dernière n'est pas irréductible. L'opérateur d'égalité prédéfini risque donc de donner une réponse suprenante. Ainsi si on teste l'égalité entre `c1` et `c3` on obtient :


```
# c1=c3;;
-: bool = false
```

On souhaite donc redéfinir une fonction `sont_egaux`. Voici le code de quatre fonctions

```
let sont_egaux_1 (a,b) (c,d)= a*d = b*c;;
let sont_egaux_2 (a,b,c,d)= a*d = b*c;;
let sont_egaux_3 a b c d = a*d = b*c;;
let sont_egaux_4 ((a,b),(c,d))= a*d = b*c;;
```

Question 1 Donnez le type de chacune d'elle

Question 2 Parmi les fonctions données deux sont facilement utilisables lesquelles? Donnez un appel à chacune d'elle. Laquelle est curryfiée, laquelle est décurryfiée?

Question 3 Parmi les fonctions données deux sont plus difficilement utilisables lesquelles? Donnez un appel à chacune d'elle.

Question 4 Programmez la somme de deux fractions

Question 5 Programmez le produit de deux fractions

Question 6 Réalisez une fonction qui calcule la forme irréductible d'une fraction. (Considérez que vous disposez d'une fonction `pgcd` de type `int->int->int`)

1.5.4 Exceptions

Exercice 1-10 *Retour sur les racines du trinôme*

Question 1 Ajoutez un déclenchement d'exceptions pour les cas $a = 0$ et $\Delta < 0$.

Exercice 1-11 *Retour sur les dates*

Question 1 La définition des années bissextiles donnée dans l'exercice ?? est celle établie dans le calendrier grégorien en 1582. La fonction `est_bissextile` n'a donc aucun sens pour les années antérieures à 1582. Modifiez cette fonction de sorte qu'elle déclenche l'exception

`Failure "est_bissextile: _annee_hors_calendrier_gregorien".`

si une année antérieure à 1582 est passée en paramètre.

Question 2 Modifiez de même la fonction `nb_jours` de façon à ce qu'elle déclenche l'exception

`Failure "nb_jour: _mois_invalide".`

si l'entier représentant le mois n'est pas compris entre 1 et 12.

Question 3 Réalisez une fonction `creer_date` qui renvoie la date représentée par le triplet d'entiers passé en paramètre s'il est valide pour représenter une date, et déclenche l'exception

`Failure "creer_date: _triplet_d_entiers_invalide".`

sinon.

Chapitre 2

Les tableaux

Les *tableaux* sont des structures de données permettant de stocker et manipuler un nombre déterminé de données toutes de même type. Plusieurs structures de données permettent un tel stockage. Ce qui caractérise les tableaux, c'est que chacun de leurs éléments possède une *adresse* qui offre la possibilité d'un accès direct à cet élément dans la structure. Cette adresse est constituée d'un ou plusieurs indices. Lorsque l'accès à un élément s'effectue à l'aide d'un seul indice, on dit que le tableau est à *une dimension* ; s'il faut deux indices, le tableau est à *deux dimensions* ; etc ... Les tableaux à une dimension sont parfois aussi nommés *vecteurs*, et les tableaux à deux dimensions *matrices* (cf [EC00] et [CD04] par exemple). Dans ces notes de cours, nous emploierons indifféremment les mots « tableau » ou « vecteur » pour les tableaux à une dimension.

Comme nous le verrons, une autre caractéristique des tableaux est que ce sont des structures de données *mutables*.

2.1 Les vecteurs : tableaux à une dimension

Dans un tableau à une dimension, ou vecteur, chaque élément est repéré par un seul *indice*, qui est un nombre entier compris entre 0 (inclus) et la longueur du tableau exclue¹. Ainsi si t est un vecteur de dimension n , les indices de ses éléments sont les entiers de 0 à $n - 1$.

Formellement, on peut définir un tableau de n éléments de type A comme une application de l'intervalle $\llbracket 0, n - 1 \rrbracket$ des entiers compris entre 0 et $n - 1$ dans l'ensemble des éléments de type A .

La figure 2.1 donne une représentation usuelle d'un tableau t de longueur $n = 5$ sous forme de cases contigües contenant les éléments t_i pour les indices i compris entre 0 et $n - 1 = 4$.

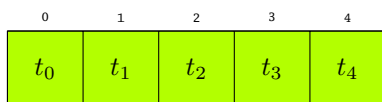


FIGURE 2.1 – Représentation usuelle d'un tableau à une dimension

En CAML, le type tableau est désigné par `array`. Comme les éléments contenus dans un tableau sont tous d'un même type, le type `array` est paramétré par le type commun de ces

1. Certains langages de programmation autorisent que les indices d'un tableau ne débutent pas par 0, ou bien même ne soient pas des entiers.

éléments. Ainsi un tableau d'entiers est de type `int array`, un tableau de flottants de type `float array`, un tableau de booléens de type `bool array`, etc ...

2.2 Création d'un tableau

2.2.1 Création par énumération littérale

Les tableaux peuvent être obtenus par une énumération littérale de leurs éléments successifs. Cette énumération doit être placée entre un crochet ouvrant `[` et un crochet fermant `]`. Les éléments de cette énumération sont séparés par des points-virgules `;`.


Exemple 2.1 :

```
# [| 1 ; 2 ; 3 |] ;;
- : int array = [|1; 2; 3|]
# [| 1. ; 2. ; 3. ; 4. |] ;;
- : float array = [|1.; 2.; 3.; 4.|]
# [| true ; true ; false ; true |] ;;
- : bool array = [|true; true; false; true|]
# [| 'a' ; '0' ; 'A' ; '?' |] ;;
- : char array = [|'a'; '0'; 'A'; '?'|]
# [| "timoleon" ; "cunegonde" |] ;;
- : string array = [|"timoleon"; "cunegonde"|]
```


Un tableau peut même être vide, c'est-à-dire ne contenir aucun élément.

```
# [|] ;;
- : 'a array = [|]
```

Et dans ce cas, le type des éléments du tableau ne peut pas être déterminé par l'énumération qui n'en contient aucun. Le type du tableau vide obtenu est alors `'a array`, le symbole `'a` désignant un type quelconque. On dit alors que le tableau vide a un type *polymorphe*.

 Rappelons que les éléments d'un tableau doivent tous avoir le même type. Dans l'exemple qui suit, un message d'erreur souligne la tentative de mettre une chaîne de caractères en deuxième élément d'un tableau dont le premier est un entier.

```
# [|1 ; "timoleon" |] ;;
This expression has type string but is here used with type int
```

 Si on place une virgule entre les éléments du tableau, on n'obtient pas d'erreur de syntaxe, mais tout au plus une erreur de type. En effet, le tableau ainsi réalisé est *syntactiquement* correct, mais la *signification* n'est pas celle souhaitée :

```
# [|1;2;3|] ;;
- : int array = [|1; 2; 3|]
# [| 1, 2, 3 |] ;;
- : (int * int * int) array = [| (1, 2, 3) |]
```

2.2.2 Filtrage

Une valeur de type tableau peut être filtrée.

Exemple 2.2 :

```
# match [|1;2|] with
| [| |] -> "VIDE"
| [| 1; _ |] -> "TAILLE_2_COMMENCE_PAR_1_"
| [| x; y |] -> "TAILLE_2_commence_par_"^string_of_int(x)
| _ -> "autre_cas";;
- : string = "TAILLE_2_COMMENCE_PAR_1_"
```

2.2.3 Création d'un tableau par sa longueur et une valeur par défaut

La fonction `Array.make` de type `int → 'a → 'a array` permet de créer des tableaux d'une longueur donnée par le premier argument et dont la valeur commune de tous les éléments est donnée par le second.

Exemple 2.3 :

```
# Array.make 10 0 ;;
- : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
# Array.make 5 (sin 1.) ;;
- : float array =
[|0.841470984807896505; 0.841470984807896505; 0.841470984807896505;
 0.841470984807896505; 0.841470984807896505|]
# Array.make 3 'A' ;;
- : char array = [|'A'; 'A'; 'A'|]
# Array.make 0 1 ;;
- : int array = [|]|
```

2.2.4 Création d'un tableau par sa longueur et une fonction de remplissage

La fonction `Array.init` de type `int → (int → 'a) → 'a array` permet de créer des tableaux d'une longueur donnée par le premier argument et dont les éléments s'expriment en fonction de l'indice par la fonction f donnée par le second argument. L'élément d'indice i du tableau ainsi créé a pour valeur $f(i)$.

Exemple 2.4 :

```
# let carre x = x*x in
  Array.init 10 carre ;;
- : int array = [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

2.3 Gestion du contenu d'un tableau

2.3.1 Nombre d'éléments d'un tableau

Le nombre d'éléments contenu dans un tableau peut être calculé par la fonction `Array.length` de type `'a array → int`.

Exemple 2.5 :

```
# Array.length [|1 ; 2; 3|] ;;
- : int = 3
# Array.length (Array.make 10 0) ;;
- : int = 10
```

2.3.2 Accès à un élément

L'accès à l'élément d'indice i d'un tableau t s'obtient avec une notation pointée, l'indice étant entre parenthèses².

Syntaxe : Accès à l'élément d'indice i d'un tableau t

$t.(i)$

Si le tableau t est de type `'a array`, alors l'expression $t.(i)$ est de type `'a`.

Exemple 2.6 :

```
# let t = [| 3 ; 1 ; 4 ; 1 ; 5 ; 9 ; 2|] ;;
val t : int array = [|3; 1; 4; 1; 5; 9; 2|]
# t.(0) ;;
- : int = 3
# t.(1) ;;
- : int = 1
# t.(6) ;;
- : int = 2
```



Bien évidemment, la valeur de l'indice i doit être comprise entre 0 et $n-1$ pour un tableau de longueur n , sous peine de rencontrer une exception `Invalid_argument "index_out_of_bounds"`.

```
# t.(7) ;;
Exception: Invalid_argument "index_out_of_bounds".
```

2.3.3 Modification d'un élément

Les tableaux sont des structures de données mutables. Cela signifie que l'on peut modifier la valeur de l'un quelconque de ses éléments. Pour cela, on utilise une instruction de la forme $t.(i) \leftarrow expr$, où $t.(i)$ est l'élément du tableau t à modifier, et $expr$ est une expression dont la valeur (de même type que les éléments de t) doit être attribuée à $t.(i)$.

2. Cette syntaxe est similaire à celle des chaînes de caractères, les crochets étant remplacés par des parenthèses.

Syntaxe : Instruction de modification de l'élément d'indice i d'un tableau t

```
 $t.(i) \leftarrow expr$ 
```


Exemple 2.7 :

Avec le tableau t défini dans l'exemple 2.2.6

```
# t.(2) <- 8 ;;
- : unit = ()
# t ;;
- : int array = [|3; 1; 8; 1; 5; 9; 2|]
```

Notons bien que le contenu du tableau t a été modifié sans que la variable t ait été déclarée mutable.

2.3.4 Mutabilité et partage de valeurs

 Parce que les tableaux sont des structures de données mutables, il faut prendre garde à certains pièges !

Exemple 2.8 :

En poursuivant l'exemple 2.7, on a

```
# let s = t ;;
val s : int array = [|3; 1; 8; 1; 5; 9; 2|]
# s.(2) <- 4 ;;
- : unit = ()
# s ;;
- : int array = [|3; 1; 4; 1; 5; 9; 2|]
# t ;;
- : int array = [|3; 1; 4; 1; 5; 9; 2|]
```

Pour éviter de tels effets, si on souhaite effectuer des modifications sur un tableau au départ identique à un autre tableau, sans que ce dernier ne subisse les mêmes modifications, il faut préalablement en faire une copie à l'aide de la fonction `Array.copy` de type `'a array → 'a array`.

Exemple 2.9 :

En poursuivant la session donnée à l'exemple 2.8

```
# let s = Array.copy t ;;
val s : int array = [|3; 1; 4; 1; 5; 9; 2|]
# s.(1) <- 2 ;;
- : unit = ()
# s ;;
- : int array = [|3; 2; 4; 1; 5; 9; 2|]
# t ;;
- : int array = [|3; 1; 4; 1; 5; 9; 2|]
```

Exemple 2.10 :

Pour terminer encore un exemple de partage de valeurs. Lorsqu'on utilise la fonction `Array.make`

avec pour second argument une donnée d'un type mutable, il faut prendre garde au partage de valeurs. Les chaînes de caractères sont des données mutables. Et donc, si on crée un tableau de chaînes de caractères, tous les éléments du tableau désignent la même chaîne. Et par conséquent toute modification de la chaîne contenue dans un élément du tableau entraîne une modification des chaînes désignées par tous les éléments du tableau.

```
# let tab_chaines = Array.make 5 "timoleon" ;;
val tab_chaines : string array =
  [|"timoleon"; "timoleon"; "timoleon"; "timoleon"; "timoleon"|]
# tab_chaines.(0).[1] <- 'u' ;;
- : unit = ()
# tab_chaines ;;
- : string array =
  [|"tumoleon"; "tumoleon"; "tumoleon"; "tumoleon"; "tumoleon"|]
```

Si on veut éviter ce partage de valeurs, on peut utiliser les fonction `String.copy` et `Array.init`.

```
# let tab_chaines =
  let f i = String.copy "timoleon" in
  Array.init 5 f ;;
val tab_chaines : string array =
  [|"timoleon"; "timoleon"; "timoleon"; "timoleon"; "timoleon"|]
# tab_chaines.(0).[1] <- 'u' ;;
- : unit = ()
# tab_chaines ;;
- : string array =
  [|"tumoleon"; "timoleon"; "timoleon"; "timoleon"; "timoleon"|]
```

2.3.5 Comparaison de tableaux

Deux tableaux sont égaux s'ils ont même longueur et si pour chaque indice, les éléments correspondants sont égaux.

Sur cette question d'égalité, en informatique, il est parfois nécessaire d'être plus précis en distinguant l'*égalité logique* de l'*égalité physique*. Deux tableaux sont *logiquement égaux* s'ils sont égaux au sens signifié plus haut (même longueur, et valeurs contenues égales). Ils sont dits *physiquement égaux* s'ils sont tous deux stockés au même endroit en mémoire.

L'opérateur usuel `=` permet de tester l'égalité logique.

Exemple 2.11 :

```
# [|1; 2; 3|] = [|1; 2; 3|] ;;
- : bool = true
# [|1; 2; 3|] = [|3; 2; 1|] ;;
- : bool = false
# [|1; 2; 3|] = [|1; 2|] ;;
- : bool = false
# let t = [|1; 2; 3|] in
  t = Array.copy t ;;
- : bool = true
```


Et l'opérateur `==` permet de tester l'égalité physique.

Exemple 2.12 :

```
# [|1; 2; 3|] == [|1; 2; 3|] ;;
- : bool = false
# let t = [|1; 2; 3|] in
  t == Array.copy t ;;
- : bool = false
# let t = [|1; 2; 3|] in
  let s = t in
    t == s ;;
- : bool = true
```

En fait deux tableaux physiquement égaux ne sont jamais qu'un seul et même tableau en mémoire qui peut éventuellement être désigné par deux noms de variables différentes, tandis que deux tableaux logiquement égaux peuvent être stockés à deux endroits différents de la mémoire.

C'est en particulier cette différence qui explique le partage de valeurs mentionnés dans la section précédente.

2.3.6 Vecteurs et chaînes de caractères

Il existe une ressemblance importante entre les vecteurs et les chaînes de caractères. D'une certaine façon, ces dernières peuvent être considérées comme des tableaux de caractères. Cette analogie est résumée dans le tableau 2.1.

	Vecteur t	Chaîne de caractères s
Type	'a array	string
Création	<code>Array.make n $expr$</code>	<code>String.make n $expr$</code>
Longueur	<code>Array.length</code>	<code>String.length</code>
Indices	de 0 à $n - 1$	de 0 à $n - 1$
Accès à un élément	$t.(i)$	$s.[i]$
Modification d'un élément	$t.(i) <- expr$	$s.[i] <- car$
Copie	<code>Array.copy t</code>	<code>String.copy s</code>

TABLE 2.1 – Comparaison vecteurs et chaînes de caractères

2.4 Tableaux en paramètres de fonctions

Cette section a pour but de bien mettre en garde l'apprenti programmeur des effets que peut avoir le passage d'un tableau en paramètre d'une fonction.

Prenons pour exemple, le problème du doublement des valeurs contenues dans un tableau d'entiers.

L'algorithme 2.1 pour réaliser cette tâche se décrit très simplement par une simple boucle pour dont l'indice parcourt l'ensemble des indices des éléments du tableau et dont chaque étape calcule la multiplication par deux et stocke le résultat.

Cependant, la spécification du problème et l'algorithme présenté laissent de côté la question de savoir s'il faut conserver intact le tableau original et produire un nouveau tableau avec les

Algorithme 2.1 Doublement des valeurs des éléments d'un tableau de nombres

pour chaque indice i du tableau **faire**
 $t'(i) \leftarrow 2t(i)$
fin pour

valeurs doublées, ou bien au contraire si le tableau d'origine doit être modifié pour contenir les valeurs doubles de celles de départ.

Dans le premier cas, on pourra implanter l'algorithme sous forme d'une fonction qui produit un nouveau tableau et laisse intact le tableau passé en paramètre. Le code d'une telle fonction est donné ci-dessous.

```
(*
  fonction double_elements : int array -> int array
  parametre
    t : int array = tableau dont on veut doubler les elts
  valeur renvoyee : int array = tableau dont les elts sont double de ceux de t
  CU : aucune
*)
let double_elements t =
  let n = Array.length t in
  let t' = Array.make n 0 in
  for i = 0 to n - 1 do
    t'.(i) <- 2 * t.(i)
  done ;
  t'
```

Et voici quelques exemples d'appels à cette fonction.

```
# double_elements ;;
- : int array -> int array = <fun>
# double_elements [|1; 2; 3|] ;;
- : int array = [|2; 4; 6|]
# let t = [|1; 2; 3|] ;;
val t : int array = [|1; 2; 3|]
# double_elements t ;;
- : int array = [|2; 4; 6|]
# t ;;
- : int array = [|1; 2; 3|]
# double_elements (double_elements t) ;;
- : int array = [|4; 8; 12|]
# t ;;
- : int array = [|1; 2; 3|]
```

Si au contraire, on ne désire pas produire un nouveau tableau, mais plutôt modifier le tableau passé en paramètre, c'est une fonction qui modifie son paramètre qu'il faut réaliser. Par conséquent, dans ce cas la fonction a un effet de bord (modification du paramètre), et c'est donc selon notre terminologie une procédure. En respectant notre convention, la valeur renvoyée d'une procédure est l'unique valeur de type `unit`. Une réalisation possible de cette procédure est donnée ci-dessous.

```
(*
  procedure doubler_elements : int array -> unit
  parametre
    t : int array = tableau dont on veut doubler les elts
  action : double les elts de t (t est modifie)
  CU : aucune
*)
let doubler_elements t =
  let n = Array.length t in
  for i = 0 to n - 1 do
    t.(i) <- 2 * t.(i)
  done
```

Et quelques exemples d'appels à cette procédure.

```
# doubler_elements ;;
- : int array -> unit = <fun>
# doubler_elements [|1; 2; 3|] ;;
- : unit = ()
# let t = [|1; 2; 3|] ;;
val t : int array = [|1; 2; 3|]
# doubler_elements t ;;
- : unit = ()
# t ;;
- : int array = [|2; 4; 6|]
# doubler_elements (doubler_elements t) ;;

This expression has type unit but is here used with type int array
# doubler_elements t ; doubler_elements t ;;
- : unit = ()
# t ;;
- : int array = [|8; 16; 24|]
```

2.5 Tableaux à plusieurs dimensions

Les tableaux à plusieurs dimensions sont tout simplement des tableaux dont les éléments sont eux-mêmes des tableaux.

2.5.1 Tableaux à deux dimensions

Un tableau à deux dimensions est un tableau dont les éléments sont des vecteurs. Le type général d'un tableau à deux dimensions est donc `'a array array`.

La figure 2.2 montre un tableau à deux dimensions : trois lignes et quatre colonnes dont les éléments sont des couples d'entiers. Du point de vue du langage CAML, le type de ce tableau est donc `(int * int) array array`.

Ce tableau peut être vu comme un tableau de trois vecteurs de dimension 4 (les trois lignes) ou bien de quatre vecteurs de dimension 3 (les quatre colonnes). Du premier point de vue, il s'agit du tableau déclaré en CAML par

```
# let tab2D_1 =
```

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)

FIGURE 2.2 – Un tableau à deux dimensions, rectangulaire

```

[|(0,0); (0,1); (0,2); (0,3)|];
 |(1,0); (1,1); (1,2); (1,3)|];
 |(2,0); (2,1); (2,2); (2,3)|]] ;;
val tab2D_1 : (int * int) array array =
  [| |(0, 0); (0, 1); (0, 2); (0, 3)|]; |(1, 0); (1, 1); (1, 2); (1, 3)|];
   |(2, 0); (2, 1); (2, 2); (2, 3)|]]]

```

et représenté à la figure 2.3.

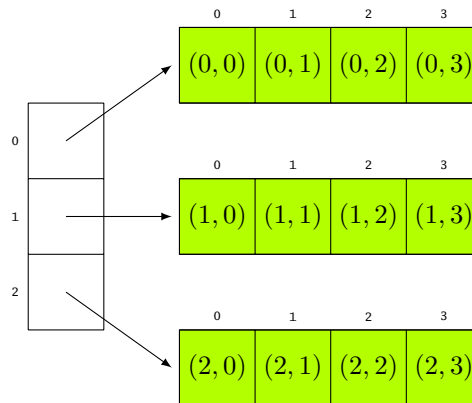


FIGURE 2.3 – Le tableau de la figure 2.2 vu comme un tableau de trois lignes

Et du second point de vue (tableau de quatre colonnes), la déclaration en CAML est

```

# let tab2D_2 =
  [| |(0,0); (1,0); (2,0)|];
   |(0,1); (1,1); (2,1)|];
   |(0,2); (1,2); (2,2)|];
   |(0,3); (1,3); (2,3)|]]] ;;
val tab2D_2 : (int * int) array array =
  [| |(0, 0); (1, 0); (2, 0)|]; |(0, 1); (1, 1); (2, 1)|];
   |(0, 2); (1, 2); (2, 2)|]; |(0, 3); (1, 3); (2, 3)|]]]

```

et correspond à celui représenté sur la figure 2.4.

Le point de vue à adopter n'est qu'une convention arbitraire. Dans ce poly nous adopterons le premier point de vue, c'est-à-dire que la représentation d'un tableau à deux dimensions sera à interpréter comme un tableau dont les éléments sont les tableaux représentés par les lignes.

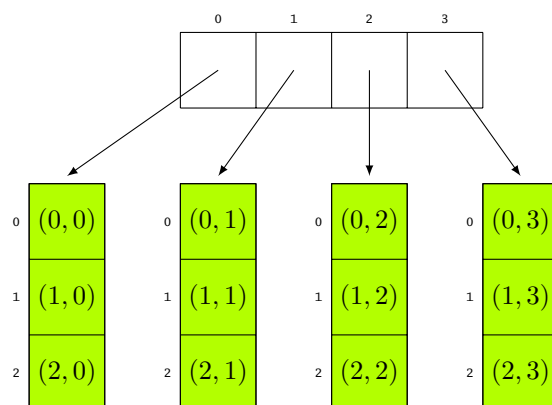


FIGURE 2.4 – Le tableau de la figure 2.2 vu comme un tableau de quatre colonnes

Un tableau rectangulaire comme celui de la figure 2.2 est donc un tableau de tableaux de tailles toutes identiques. Mais rien oblige qu'un tableau à deux dimensions soit constitué de tableaux tous de même longueur. Il est tout à fait possible de concevoir des tableaux dont les différentes lignes n'ont pas la même longueur comme celui représenté sur la figure 2.5, dont la définition en CAML est

```
# let tab2D_3 =
[|(0,0)|];
 |(1,0); (1,1); (1,2)|];
 |(2,0); (2,1); (2,2); (2,3); (2,4)|];
 |(3,0); (3,1); (3,2); (3,3); (3,4); (3,5); (3,6)|];|] ;;
val tab2D_3 : (int * int) array array =
[|(0, 0)|]; |(1, 0); (1, 1); (1, 2)|];
 |(2, 0); (2, 1); (2, 2); (2, 3); (2, 4)|];
 |(3, 0); (3, 1); (3, 2); (3, 3); (3, 4); (3, 5); (3, 6)|]]
```

	0	1	2	3	4	5	6
0	(0, 0)						
1	(1, 0)	(1, 1)	(1, 2)				
2	(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)		
3	(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)

FIGURE 2.5 – Un tableau à deux dimensions, triangulaire

2.5.2 Accès et modification d'un élément

L'accès à un élément d'un tableau à deux dimensions nécessite deux indices : un indice pour désigner la ligne, et un indice pour désigner la colonne. Bien entendu ces indices doivent satisfaire des contraintes de dimension habituelles.

Syntaxe : Accès à l'élément d'indices i et j d'un tableau t à deux dimensions

$t.(i).(j)$

Exemple 2.13 :

Avec les tableaux définis dans la section précédente, voici quelques accès et modifications.

```
# tab2D_1.(2).(1) ;;
- : int * int = (2, 1)
# tab2D_1.(3).(1) ;;
Exception: Invalid_argument "index_out_of_bounds".
# tab2D_1.(2).(4) ;;
Exception: Invalid_argument "index_out_of_bounds".
```

Si on ne donne qu'un seul indice pour un tableau à deux éléments, on obtient une ligne de ce tableau, c'est-à-dire un tableau.

Exemple 2.14 :

```
# tab2D_1.(0) ;;
- : (int * int) array = [| (0, 0); (0, 1); (0, 2); (0, 3) |]
```

Pour connaître le nombre de lignes d'un tableau à deux dimensions, il suffit d'appliquer la fonction `Array.length` au tableau.

Exemple 2.15 :

```
# Array.length tab2D_1 ;;
- : int = 3
# Array.length tab2D_3 ;;
- : int = 4
```

Et pour connaître le nombre de colonnes, on applique la même fonction sur une ligne quelconque pour un tableau rectangulaire, et sur la ligne désirée pour un tableau non rectangulaire.

Exemple 2.16 :

```
# Array.length tab2D_1.(0) ;;
- : int = 4
# Array.length tab2D_1.(2) ;;
- : int = 4
# Array.length tab2D_3.(0) ;;
- : int = 1
# Array.length tab2D_3.(1) ;;
- : int = 3
# Array.length tab2D_3.(2) ;;
- : int = 5
```

La modification d'un élément d'un tableau à deux dimensions se fait par une instruction analogue à celle d'un tableau à une dimension.

Syntaxe : Instruction de modification de l'élément d'indices i et j d'un tableau t à deux dimensions

```
t.(i).(j) <- expr
```

Exemple 2.17 :

```
# tab2D_1.(2).(1) <- (0,0) ;;
- : unit = ()
# tab2D_1.(2).(1) ;;
- : int * int = (0, 0)
```

Bien évidemment, comme les tableaux à deux dimensions sont des tableaux de tableaux, il est tout à fait possible de modifier une ligne à la fois.

```
# tab2D_1.(0) <- Array.make 2 (5,5) ;;
- : unit = ()
# tab2D_1 ;;
- : (int * int) array array =
[|(1, 5); (5, 5)|]; |(1, 0); (1, 1); (1, 2); (1, 3)|];
  |(2, 0); (0, 0); (2, 2); (2, 3)|]]
```

2.5.3 Création de tableaux à deux dimensions

S'il est tout à fait possible de créer des tableaux à deux dimensions par énumération de tous leurs éléments, énumérés ligne par ligne, comme on a pu le voir pour les tableaux `tab2D_1`, `tab2D_2` et `tab2D_3` dans la section 2.5.1, cette façon de procéder s'avère très vite fastidieuse, si ce n'est impossible, lorsque les dimensions sont grandes.

On peut alors recourir aux fonctions déjà présentées `Array.make` et `Array.init`.

Voici l'utilisation de ces deux fonctions pour construire les trois tableaux de la section 2.5.1.

Exemple 2.18 :

Le tableau `tab2D_1` peut être construit avec la fonction `Array.make`. Voici comment.

```
let tab2D_1 =
  let t = Array.make 3 [|] in
  for i = 0 to 2 do
    let ligne = Array.make 4 (0,0) in
    for j = 0 to 3 do
      ligne.(j) <- (i,j)
    done ;
    t.(i) <- ligne
  done ;
  t
```

La méthode consiste à construire le tableau qui contiendra les lignes, puis à construire une à une chacune des lignes, et à les affecter au tableau de lignes.

Le même tableau peut être construit avec la fonction `Array.init`. Voici comment.

```

let tab2D_1 =
  let elt i j = (i,j) in
  let ligne i = Array.init 4 (elt i) in
    Array.init 3 ligne

```

Admirez la concision de l'expression de construction de ce tableau. Elle repose sur la déclaration locale de deux fonctions :

1. la fonction **elt** qui construit le couple de ses deux paramètres ;
2. et la fonction **ligne** qui construit des tableaux de longueur 4 à l'aide de la fonction **elt i** obtenue par application partielle de la fonction **elt**.

Il existe une fonction permettant de construire plus facilement des tableaux rectangulaires que la fonction **Array.make**. C'est la fonction **Array.make_matrix** de type

$$\text{int} \rightarrow \text{int} \rightarrow 'a \rightarrow 'a \text{ array array}.$$

Exemple 2.19 :

```

let tab2D_1 =
let t = Array.make_matrix 3 4 (0,0) in
  for i = 0 to 2 do
    for j = 0 to 3 do
      t.(i).(j) <- (i,j)
    done
  done ;
t

```

Cependant cette fonction ne permet pas la création de tableaux non rectangulaires.

Exemple 2.20 :

Voici donc la construction du tableau triangulaire **tab2D_3** avec la fonction **Array.make** pour commencer.

```

let tab2D_3 =
  let t = Array.make 4 [[]] in
    for i = 0 to 3 do
      let ligne = Array.make (2 * i + 1) (0,0) in
        for j = 0 to 2 * i do
          ligne.(j) <- (i,j)
        done ;
      t.(i) <- ligne
    done ;
t

```

Et enfin avec la fonction **Array.init**.

```

let tab2D_3 =
  let elt i j = (i,j) in
  let ligne i = Array.init (2 * i + 1) (elt i) in
    Array.init 4 ligne

```


2.5.4 Partage de valeurs

Nous avons déjà évoqué cette difficulté du partage de valeurs que l'on peut rencontrer avec les tableaux (cf page 2.3.4). Voyons à nouveau un piège dans lequel il est très facile de tomber.

⚠ Au lieu de construire le tableau `tab2D_1` comme nous l'avons vu plus haut, on pourrait être tenté de construire un tableau de 3 lignes initialisées avec *des* tableaux de longueur 4 contenant le couple d'entiers (0,0), puis de remplir une à une toutes les cases du tableau. Voici cette idée exprimée en CAML.

```
let tab2D_1 =
  let t = Array.make 3 (Array.make 4 (0,0)) in
  for i = 0 to 2 do
    for j = 0 to 3 do
      t.(i).(j) <- (i,j)
    done
  done ;
  t
```

Ce code ne construit pas le tableau souhaité, mais un tableau rectangulaire contenant trois lignes identiques.

```
# tab2D_1 ;;
- : (int * int) array array =
[[|(2, 0); (2, 1); (2, 2); (2, 3)|]; |(2, 0); (2, 1); (2, 2); (2, 3)|];
 [| (2, 0); (2, 1); (2, 2); (2, 3)|]]
```

En fait, dans cette construction l'initialisation du tableau sous la forme

```
let t = Array.make 3 (Array.make 4 (0,0))
```

crée bien un tableau contenant trois lignes chacune étant un tableau de couples d'entiers de longueur 4, mais ces lignes ne sont qu'un seul et même tableau. Une représentation du tableau ainsi construit est représenté à la figure 2.6.

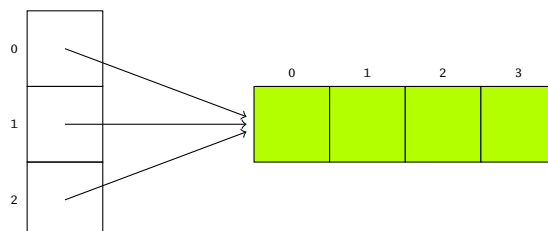


FIGURE 2.6 – Partage de valeurs

On peut même vérifier que les lignes du tableau `tab2D_1` sont bien physiquement égales à l'aide de l'opérateur de comparaison `==`.

```
# tab2D_1.(0) == tab2D_1.(1) ;;
- : bool = true
```

⚠ La fonction `Array.copy` appliquée à un tableau de tableaux, effectue une copie du tableau principal mais pas des tableaux désignés par les éléments de ce tableau principal. Pour effectuer véritablement une copie de l'ensemble du contenu d'un tableau à deux dimensions, il faut donc

recourir à une copie de tous les tableaux. Voici comment procéder à la copie d'un tableau à deux dimensions.

```
(**
  copie2D : 'a array array -> 'a array array
  copie2D t = copie des elements du tableau a deux dimensions t
  CU : aucune
*)
let copie2D t =
  let n = Array.length t in
  let t' = Array.make n [|] in
  for i = 0 to n - 1 do
    t'.(i) <- Array.copy t.(i)
  done ;
  t'
```

2.5.5 Encore plus de dimensions

De la même façon que les tableaux à deux dimensions sont des tableaux de tableaux, les tableaux à trois dimensions sont des tableaux de tableaux de tableaux, et il faut trois indices pour désigner un élément.

Le langage CAML n'impose aucune restriction sur le nombre de dimensions d'un tableau. Mais en programmation, il est assez rare d'avoir à représenter et manipuler des données dans un tableau ayant plus de trois dimensions.

2.6 Exemples d'utilisation des tableaux

2.6.1 Représentation de fonction sous forme de tableau

Le nombre de jours dans un mois (pour une année non bissextile) peut être décrit par un tableau de longueur 12.

```
let nb_jours = [| 31 ; 28 ; 31 ; 30 ;
                  31 ; 30 ; 31 ; 31 ;
                  30 ; 31 ; 30 ; 31 |]
```

Et à l'aide d'un tel tableau, il devient facile de définir une fonction `nb_jours` de type `int → int` qui donne le nombre de jours d'un mois dont le numéro est passé en paramètre :

```
let nb_jours m =
  let nb_jours = [| 31 ; 28 ; 31 ; 30 ;
                    31 ; 30 ; 31 ; 31 ;
                    30 ; 31 ; 30 ; 31 |]
  in
  if (m < 1) || (m > 12) then
    failwith ("nb_jours_:_^s^":_mois_incorrect")
  else
    nb_jours.(m-1)
```

2.6.2 Compter les caractères d'une chaîne

Nous avons déjà compté certaines choses simples dans le cours d'InitProg. Par exemple, nous avons compté le nombre de cartes sur un tas, ou bien le nombre de déplacements de cartes nécessaires pour accomplir une tâche donnée. Pour compter, nous avons utilisé un compteur, représenté par une variable mutable de type `int`, initialisée à zéro, et *incrémentée* à chaque fois que nous observons une occurrence de ce que nous avons à compter. Ce principe général est décrit dans l'algorithme 2.2.

Algorithme 2.2 Principe général d'un compteur simple

```

mise à zéro du compteur
pour chaque occurrence de l'évènement à compter faire
    incrémenter le compteur
fin pour
renvoyer la valeur du compteur
  
```

Mais comment compter beaucoup d'évènements différents ? Par exemple, comment compter le nombre d'occurrences de chacun des caractères d'une chaîne de caractères ? Devons nous prévoir 256 compteurs, un pour chaque caractère potentiel ?

Une solution élégante et simple consiste à utiliser un tableau de compteurs, dont la longueur est égale au nombre de compteurs dont nous avons besoin, c'est-à-dire au nombre d'évènements à compter. Dans notre exemple, un tableau de 256 compteurs. Tous ces compteurs sont initialisés à zéro à la création du tableau. Puis, la phase d'observation nous conduit à incrémenter tel ou tel compteur en fonction des évènements observés. L'algorithme 2.3 résume ce principe, et l'exemple 2.21 en est une mise en œuvre sur notre problème du calcul du nombre de chaque caractère dans une chaîne donnée.

Algorithme 2.3 Principe général de manipulations de compteurs multiples

```

initialiser à zéro un tableau de compteurs
pour chaque évènement observé faire
    incrémenter le compteur correspondant dans le tableau
fin pour
renvoyer le tableau de compteurs
  
```

Exemple 2.21 :

Supposons que `s` soit une chaîne de caractères dont nous souhaitons connaître le nombre d'occurrences de chaque caractère. En application du principe de l'algorithme 2.3, il suffit d'écrire

```

let compteurs = Array.make 256 0
and n = String.length s
in
  for i=0 to n - 1 do
    compteurs.(int_of_char s.[i]) <- compteurs.(int_of_char s.[i]) + 1
  done ;
  compteurs
  
```

Remarques :

- Rappelons encore une fois que les tableaux sont des structures de données mutables, et que la variable **compteurs** n'a pas à être déclarée mutable (pas de **ref** dans la déclaration).
- Dans ce code, à chaque étape de la boucle pour, l'indice du compteur à incrémenter est calculé deux fois : dans la partie gauche et dans la partie droite de l'affectation. Cela aurait pu être évité avec une variable locale.

2.6.3 Représentation d'images

Sous leur forme numérisée, les images sont des tableaux à deux dimensions dont les éléments sont des *pixels* (pour *picture element*), un pixel étant l'unité atomique d'une image représentant une nuance de couleur.

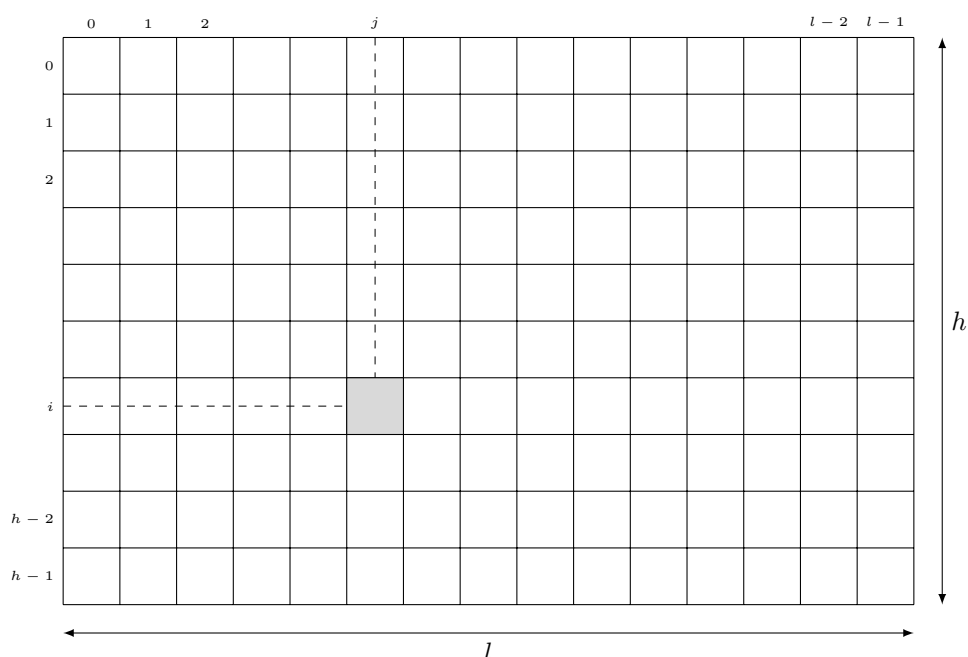


FIGURE 2.7 – Une image composée de $h \times l$ pixels, et le pixel de coordonnées (i, j)

La figure 2.7 donne une représentation de la décomposition d'une image en un tableau de $h \times l$ pixels, h étant la hauteur de l'image et l sa largeur, exprimées en nombre de pixels.

Le module **Graphics** livré avec toute distribution d'OBJECTIVE CAML définit un type

Graphics.color,

dont les valeurs représentent des nuances de couleur que peuvent prendre les pixels d'une image. Ces couleurs peuvent être définies grâce à la fonction

Graphics.rgb : int → int → int → Graphics.color,

qui renvoie une couleur définie par ses trois composantes Rouge, Verte et Bleue désignées par un entier compris entre 0 et 255.

Remarque : Toutes les lignes de code écrites dans cette section supposent que le module **Graphics** a été chargé dans l'interpréteur ou le compilateur (en ajoutant le fichier **graphics.cma** dans la commande de lancement de l'interpréteur (**ocaml**) ou de compilation (**ocamlc**)).

Commençons par définir les dimensions de l'image que nous souhaitons construire, et ouvrons une fenêtre graphique dans laquelle nous visualiserons les images.

```
(* dimensions de l'image a construire *)
let hauteur = 256
and largeur = 384

let _ =
  (* ouverture d'une fenetre graphique
    avec les dimensions de l'image *)
  Graphics.open_graph ("_"^(string_of_int largeur)^"x"^(string_of_int hauteur)) ;
  (* etablisement du texte du bandeau de cette fenetre *)
  Graphics.set_window_title "Mon_image"
```

Construisons maintenant un tableau de pixels tous noirs.

```
(* Un tableau de pixels noirs (R=0, V=0, B=0) *)
let tableau_noir =
  Array.make_matrix hauteur largeur (Graphics.rgb 0 0 0)
```

Avant de visualiser l'image correspondant à ce tableau, il faut convertir le tableau de pixels en une donnée du type `Graphics.image` à l'aide de la fonction

`Graphics.make_image : Graphics.color array array → Graphics.image.`

```
(* conversion du tableau de pixels
  en une image (type du module Graphics) *)
let image_noire = Graphics.make_image tableau_noir
```

Tout est prêt maintenant pour visualiser notre image noire. C'est la procédure

`Graphics.draw_image : Graphics.image → int → int → unit,`

l'instruction `Graphics.draw_image img x y` dessine l'image `img` en plaçant son coin inférieur gauche au point de coordonnées `x,y`.

```
let _ =
  (* dessin de l'image dans la fenetre graphique
    depuis le coin inferieur gauche *)
  Graphics.draw_image image_noire 0 0
```

Cette instruction produit l'image montrée sur la figure 2.8.

Si on préfère une image rouge :

```
(* Un tableau de pixels rouges (R=255, V=0, B=0) *)
let tableau_rouge =
  Array.make_matrix hauteur largeur (Graphics.rgb 255 0 0)

(* conversion du tableau de pixels
  en une image (type du module Graphics) *)
let image_rouge = Graphics.make_image tableau_rouge

let _ =
  (* dessin de l'image dans la fenetre graphique
    depuis le coin inferieur gauche *)
  Graphics.draw_image image_rouge 0 0
```

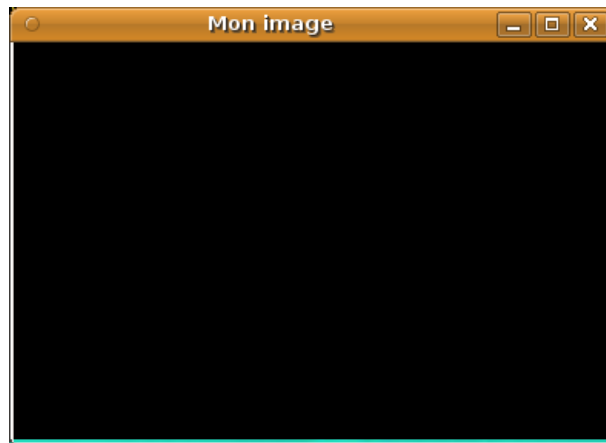


FIGURE 2.8 – Une image noire

ou bien un beau dégradé vertical de gris allant du noir au blanc, comme celui de la figure 2.9 :

```
(* Un tableau de pixels avec une nuance de gris par ligne *)
let tableau_degrade_vertical =
  let t = Array.make_matrix hauteur largeur (Graphics.rgb 0 0 0) in
  for i = 0 to hauteur - 1 do
    let gris = Graphics.rgb i i i in
    for j = 0 to largeur - 1 do
      t.(i).(j) <- gris
    done
  done ;
  t

let image_degrade_vertical = Graphics.make_image tableau_degrade_vertical

let _ = Graphics.draw_image image_degrade_vertical 0 0
```

2.6.4 Le triangle de Pascal

Le triangle de Pascal est un tableau triangulaire contenant les valeurs des coefficients binomiaux (appelés aussi combinaisons). À l'intersection de la ligne n et de la colonne p , on trouve le coefficient binomial

$$\binom{n}{p} = \frac{n!}{(n-p)!p!}.$$

Une relation bien connue relie un coefficient binomial à deux coefficients de la ligne qui précède :

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}.$$



FIGURE 2.9 – Un dégradé de gris

À partir de cette relation, il est facile de construire un tableau qui contient les lignes 0 à n du triangle de Pascal. C'est ce qui est fait dans le code qui suit.

```
(*
  fonction triangle_pascal : int -> int array array
  parametre
    n : int
  valeur renvoyee : int array array = tableau contenant les lignes 0 a n
    du triangle de Pascal
  CU : n >= 0
*)
let triangle_pascal n =
  let t = Array.make (n + 1) [|1|] in
  for i = 1 to n do
    let ligne = Array.make (i + 1) 1 in
    for j = 1 to i - 1 do
      ligne.(j) <- t.(i - 1).(j - 1) + t.(i - 1).(j)
    done ;
    t.(i) <- ligne
  done ;
  t
```

Et voici une procédure pour imprimer le triangle de Pascal (en fait cette procédure convient pour imprimer n'importe quel tableau à deux dimensions dont les éléments sont des nombres entiers).

```
(*
  procedure imprimer_tableau_2D : int array array -> unit
  parametre
    t : int array array
  action = imprime, ligne par ligne, les entiers contenus dans le tableau t
  CU : aucune
*)
```

0	1										
1	1	1									
2	1	2	1								
3	1	3	3	1							
4	1	4	6	4	1						
5	1	5	10	10	5	1					
6	1	6	15	20	15	6	1				
7	1	7	21	35	35	21	7	1			
8	1	8	28	56	70	56	28	8	1		
9	1	9	36	84	126	126	84	36	9	1	
10	1	10	45	120	210	252	210	120	45	10	1
	0	1	2	3	4	5	6	7	8	9	10

FIGURE 2.10 – Les 11 premières lignes du triangle de Pascal

```

let imprimer_tableau_2D t =
  let n = Array.length t in
    for i = 0 to n - 1 do
      for j = 0 to Array.length t.(i) - 1 do
        Printf.printf "%8d" t.(i).(j)
      done ;
      Printf.printf "\n"
    done

```

Cette procédure fait appel à la procédure `Printf.printf` avec le format `"%8d"` qui réserve un emplacement de longueur 8 pour imprimer des données de type `int`, ce qui est suffisant pour l'écriture de tous les nombres entiers positifs du type `int` tout en gardant (au moins) un espace entre eux. Cette réservation assure un alignement en colonnes des nombres imprimés.

```

# imprimer_tableau_2D (triangle_pascal 8) ;;
1
1      1
1      2      1
1      3      3      1
1      4      6      4      1
1      5      10     10     5      1
1      6      15     20     15     6      1
1      7      21     35     35     21     7      1
1      8      28     56     70     56     28     8      1
- : unit = ()

```


2.7 Le tableau `Sys.argv`

Le tableau `Sys.argv` est un tableau défini par le module `Sys`. C'est un tableau de chaînes de caractères dont la vocation est de contenir tous les arguments qui accompagnent la commande d'exécution du programme. Ce tableau n'est donc en général utilisé que pour les programmes exécutés après compilation (pour la compilation cf page 107).

À titre d'exemple, voici un programme qui affiche tous les arguments passés sur la ligne de commande,

```
for i = 0 to Array.length Sys.argv - 1 do
  print_endline Sys.argv.(i)
done
```

et ci-dessous quelques exemples d'exécution de ce programme (supposé être dans un fichier nommé `affiche_arguments.ml`) après compilation.

```
> ocamlc -o affiche_arguments affiche_arguments.ml
> ./affiche_arguments
./affiche_arguments
> ./affiche_arguments 1
./affiche_arguments
1
> ./affiche_arguments 1 2 3
./affiche_arguments
1
2
3
```

Comme cet exemple le montre, on peut remarquer que le premier argument (celui d'indice 0) est le nom de la commande elle-même.



Rappelons que le tableau `Sys.argv` est un tableau de chaînes de caractères. Les arguments récupérés sur la ligne de commande sont donc tous considérés comme des chaînes de caractères. Si on souhaite passer une donnée d'un autre type, il faut donc effectuer une conversion de type appropriée. Par exemple, si on souhaite passer un nombre entier, il faut faire appel à la fonction `int_of_string`.

2.8 Exercices

2.8.1 Tableaux à une dimension

Création de tableaux

Exercice 2-1 *Toutes sortes de tableaux*

Pour chacun des tableaux décrits ci-dessous, donnez leur type puis donnez deux expressions les construisant (autres que l'énumération littérale des éléments). Les exemples sont donnés pour des tableaux de longueur 10, mais vos expressions seront exprimées pour une longueur n quelconque.

1. `[|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]`;
2. `[|"1"; "3"; "5"; "7"; "9"; "11"; "13"; "15"; "17"; "19"|]`;
3. `[|9; 8; 7; 6; 5; 4; 3; 2; 1; 0|]`;

4. [|19; 17; 15; 13; 11; 9; 7; 5; 3; 1|];
5. [|0; 0; 1; 1; 2; 2; 3; 3; 4; 4|];
6. [| (0,0); (1,1); (2,2); (3,3); (4,4); (5,5); (6,6); (7,7); (8,8); (9,9) |];
7. [|0.; 1.; 1.4142; 1.7321; 2.; 2.2361; 2.4495; 2.6458; 2.8284; 3. |] (racine carrée);

Exercice 2-2 Tableaux de caractères ou chaînes de caractères

Question 1 Réalisez une fonction de conversion d'une chaîne de caractères en un tableau de caractères. Par exemple, la chaîne de caractères "timoleon" doit être convertie en le tableau [|'t'; 'i'; 'm'; 'o'; 'l'; 'e'; 'o'; 'n'|].

Question 2 Réalisez la fonction réciproque.

Compter

Exercice 2-3 Compter dans une chaîne de caractères

Indiquer comment compter les différents éléments suivants dans une chaîne de caractères **s**, en précisant à chaque fois la longueur du tableau de compteurs, et le lien entre les indices des compteurs et les différents éléments à compter.

Question 1 Compter les lettres majuscules.

Question 2 Compter les lettres majuscules et minuscules en ne les différenciant pas.

Question 3 Compter les lettres majuscules et minuscules en les différenciant.

Question 4 Compter les voyelles majuscules et minuscules sans les différencier.

Exercice 2-4 Anagrammes

Deux mots sont des *anagrammes* s'ils sont constitués des mêmes lettres. Par exemple, les mots **soigneur** et **guerison** sont des anagrammes (on dit aussi que **soigneur** est une anagramme de **guerison**). En particulier, deux mots anagrammes l'un de l'autre ont même longueur, et le nombre d'occurrences d'une lettre dans l'un est égal au nombre d'occurrences dans l'autre.

En vous appuyant sur cette remarque, réalisez un prédicat **sont_anagrammes** qui détermine si les deux chaînes de caractères passées en paramètre sont anagrammes ou non, en donnant la valeur booléenne **true** dans l'affirmative, et la valeur booléenne **false** dans le cas contraire.

Vous étudierez les différentes situations

1. tenir compte de tous les caractères;
2. ne tenir compte que des lettres (non accentuées) en assimilant majuscule et minuscule.

Max, min et autre moyenne

Exercice 2-5 Max, min, moyenne des éléments d'un tableau

Dans tout cet exercice **t** est un tableau de **n** flottants, **n** étant un entier strictement positif.

Question 1 Comment déterminer la valeur du plus grand des nombres contenus dans ce tableau? Comment déterminer son indice?

Question 2 En un seul parcours du tableau, déterminez les valeurs extrêmes (minimale et maximale)

contenues dans le tableau.

Question 3 Comment calculer la somme des éléments de t ?

Question 4 Comment calculer la moyenne des éléments de ce tableau ?

Pour un entier $i \in \llbracket 0, n \rrbracket$, on appelle *somme partielle de rang i* des éléments de t , la somme des éléments d'indice compris entre 0 et i , c'est-à-dire $\sum_{k=0}^i t_k$.

Question 5 Comment construire un tableau contenant les sommes partielles des éléments de t de rang 0 à $n - 1$?

Tableaux pour représenter des fonctions

Exercice 2-6 *Tabulation de la fonction sinus*

On veut tabuler la fonction sinus.

Question 1 Construisez un tableau `tab_sin` à une dimension de taille N qui contient les valeurs de la fonction `sin` sur les points de subdivision de l'intervalle $[0, 2\pi]$ en N parties égales. Plus précisément, à l'indice k , $0 \leq k < N$, on doit avoir

$$\text{tab_sin.}(k) = \sin\left(\frac{2k\pi}{N}\right).$$

En CAML, la fonction sinus est prédéfinie et se nomme `sin`. Son type est `float → float`. La constante π peut être obtenue à l'aide de la fonction `arctangent` par $\pi = 4 \arctan(1)$. La fonction `arctangent` en CAML se nomme `atan` et est de type `float → float`.

Question 2 À l'aide du tableau `tab_sin`, réalisez une fonction qui calcule le sinus d'un flottant quelconque, sans utiliser la fonction prédéfinie `sin`.

Le principe à suivre consiste à

1. tenir compte du fait que la fonction `sin` est impaire, i.e. $f(-x) = -f(x)$;
2. tenir compte du fait que la fonction `sin` est périodique de période 2π , ce qui permet de ramener le calcul de `sin x` pour un réel quelconque au calcul de `sin y` pour un réel $y \in [0, 2\pi]$ tel que $x \equiv y \pmod{2\pi}$;
3. chercher l'unique entier $k \in \llbracket 0, N - 1 \rrbracket$ tel que $y \in [\frac{2k\pi}{N}, \frac{2(k+1)\pi}{N}]$;
4. interpoler la valeur de `sin y` avec les valeurs `tab_sin(k)` et `tab_sin(k + 1)`.

Vous pourrez utiliser avec profit les fonctions prédéfinies

- `int_of_float : float → int`
- `floor : float → float`
- `abs_float : float → float` qui calcule la valeur absolue d'un flottant ;
- et `mod_float : float → float → float`. l'expression `mod_float x y` a pour valeur $x - n \times y$ où n est la partie entière de $\frac{x}{y}$.

`Sys.argv`

Exercice 2-7 *Somme de nombres*

Écrire un programme qui calcule la somme des nombres entiers passés en arguments sur la ligne de commande. Par exemple, en supposant ce programme nommé `somme`, voici une exécution possible de ce programme.

```
> somme 1 2 3
6
```

2.8.2 Tableaux à plusieurs dimensions

Exercice 2-8 Copie de tableaux à deux dimensions

Reprogrammez la fonction `copie2D` (cf page 42) en utilisant la fonction `Array.init` au lieu de `Array.make`.

Exercice 2-9 Tables de multiplication

On veut stocker en mémoire les 10 tables de multiplications (tables de multiplication par 1, 2, ..., 10, chaque table ayant 10 lignes).

Question 1 Proposez une structure de données pour le faire.

Question 2 En adoptant la structure de données choisie dans la question précédente, créez cette représentation des dix tables de multiplications.

Question 3 Comment obtenir la valeur de $n \times p$ à l'aide de votre structure lorsque n et p sont deux nombres entiers compris entre 1 et 10.

Question 4 Construisez une instruction qui permet un affichage des dix tables sous la forme :

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

Exercice 2-10 Un peu d'algèbre linéaire

Dans cet exercice on considère les vecteurs de l'espace vectoriel \mathbb{R}^n de dimension n sur \mathbb{R} , représentés en CAML par un tableau à une dimension de longueur n de flottants. Par exemple, avec $n = 3$, le vecteur $(1, -3, 2)$ est représenté par le tableau

`[|1.; -3.; 2.|]`.

Les matrices à n lignes et p colonnes sont représentées par des tableaux à deux dimensions de flottants. Par exemple, avec $n = 2$ et $p = 3$, la matrice

$$\begin{pmatrix} 1 & -2 & 3 \\ -4 & 5 & -6 \end{pmatrix}$$

est représentée par le tableau

`[| [|1.; -2.; 3.]; [-4.; 5.; -6.] |]`.

Question 1 Réalisez une fonction `add_vecteurs` qui calcule le vecteur somme des deux vecteurs passés en paramètre. Quelle contrainte d'utilisation pour cette fonction? Programmez votre fonction

de sorte qu'une exception soit déclenchée avec le message "add_vecteurs_:dimensions_incompatibles".

Question 2 Addition de deux matrices.

Question 3 Multiplication d'une matrice par un vecteur.

Question 4 Multiplication de deux matrices.

Exercice 2-11 Carrés magiques

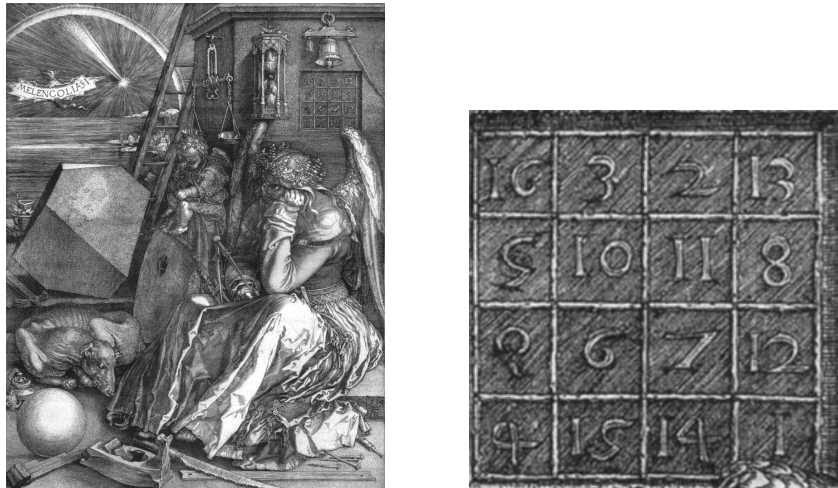


FIGURE 2.11 – La mélancolie de Dürer (1471-1528) et le carré magique en détail

Un carré magique est un carré découpé en $n \times n$ cases dans lesquelles sont placés tous les nombres entiers consécutifs de 1 à n^2 , de sorte que la somme des nombres d'une ligne, d'une colonne ou d'une diagonale quelle qu'elle soit, est toujours la même.

La figure 2.11 montre une célèbre gravure d'Albrecht Dürer, la Mélancolie, qui contient un carré magique 4×4 (agrandi à droite).

Question 1 Réalisez une fonction de vérification qu'un carré est magique.

Il est assez aisé de créer des carrés magiques d'ordre impair.

Algorithme 2.4 Construction d'un carré magique d'ordre impair

Entrée : n un entier impair

Sortie : un carré magique d'ordre n

- 1: Placer le 1 dans la case qui se trouve sous la case du milieu du carré.
 - 2: Décaler d'une case vers la droite puis d'une case vers le bas pour le 2, et ainsi de suite pour le 3, puis le 4, etc.
 - 3: Si une case est déjà occupée, il faut revenir au nombre précédent, ne pas décaler à droite, mais descendre de 2 cases à la place.
-

Précision : quand on arrive au bord du carré, on continue du côté opposé (en haut ou à gauche), un peu comme si le carré était torique.

Question 2 Réalisez une fonction de construction de carrés magiques d'ordre impair.

Chapitre 3

Recherche dans un tableau

3.1 Introduction

3.1.1 Tranche

On appelle *tranche de tableau*, la donnée d'un tableau t et de deux indices a et b .
On note cette tranche $t.(a..b)$.

Exemple 3.1 :

3	6	7	4	1	2	8	5
---	---	---	---	---	---	---	---

$t.(3..5)$



Lorsque $a > b$ la tranche $t.(a..b)$ est vide.

3.1.2 Appartenance

Lorsqu'on se donne un tableau t , un couple d'indice (a, b) , il est fréquent d'avoir à se demander si un élément x appartient, ou bien n'appartient pas à la tranche de tableau.

Plus précisément, il s'agit de déterminer s'il existe un indice i dans l'intervalle d'entier $\llbracket a, b \rrbracket$ tel que $t.(i) = x$, dans ce cas on note $x \in t.(a..b)$



Il s'agit d'une notation, qu'on peut utiliser dans les commentaires d'un programme, ou bien dans un texte pour décrire un algorithme. Mais, il ne s'agit pas de code CAML. On ne doit pas employer cette notation dans le code d'un programme.

Exemple 3.2 :

Si $t =$

3	6	7	4	1	2	8	5
---	---	---	---	---	---	---	---

, alors on peut affirmer que $8 \in t.(4..7)$ et $7 \in t.(0..5)$;

Quand pour tout i dans l'intervalle d'entier $\llbracket a, b \rrbracket$, on a $t.(i) \neq x$, on note $x \notin t.(a..b)$.

Exemple 3.3 :

En revanche avec le même tableau t que dans l'exemple 3.2, on peut affirmer que $9 \notin t.(0..7)$; $8 \notin t.(0..3)$

3.1.3 Les problèmes

t est un tableau, a et b sont deux indices, et x un élément, naturellement, on peut se poser plusieurs questions, par exemple :

- « L'élément x appartient-il oui ou non à $t.(a..b)$? » ;
- « l'élément x appartient-il oui ou non à $t.(a..b)$ et si oui à quel indice i peut-on trouver cet élément ? » ;
- « s'il est présent plusieurs fois, quel indice choisir » ;
- « l'élément x est-il présent plusieurs fois dans la tranche de tableau $t.(a..b)$? » ;
- « combien de fois l'élément x est-il présent dans le tableau $t.(a..b)$? ».

D'autre part, les méthodes mises en œuvre pour trouver la réponse peuvent être différentes, lorsque le tableau est quelconque, ou bien lorsque le tableau est trié.

3.1.4 Type d'une fonction de recherche

Le type d'une fonction de recherche dépend directement de la question qu'on se pose.

S'il s'agit de tester si un élément x appartient à la tranche, on pourra avoir comme type :
`'a array → int → int → 'a → bool`

Si on souhaite plutôt déterminer à quel indice on peut trouver l'élément, on peut utiliser le type `'a array → int → int → 'a → int`, et prévoir le déclenchement d'une exception en cas d'absence de l'élément recherché. Ou bien, on peut décider de renvoyer un couple `bool * int`, le booléen servant à indiquer si l'élément est présent, et l'entier sa position dans le cas où l'élément a été trouvé.

Enfin, si on souhaite savoir combien de fois l'élément x apparaît dans la tranche de tableau t , on peut utiliser le `'a array → int → int → 'a → int`

3.1.5 Objectifs

Dans ce chapitre, on présente uniquement des algorithmes de test d'appartenance, qui sont facilement adaptables pour trouver l'indice de la première occurrence de l'élément x dans la tranche $t.(a..b)$.

On fait de plus une analyse du nombre de comparaisons entre l'élément x et des éléments du tableaux.

3.2 Un premier algorithme : la recherche laborieuse

3.2.1 Principe

On parcourt complètement le tableau. Pour chaque élément du tableau, on teste l'égalité avec x . Dans ce cas, on a trouvé l'élément x recherché, on essaie de se souvenir de cette rencontre.

3.2.2 Algorithme

L'algorithme 3.1 formalise le principe de la recherche laborieuse.

Dans cet algorithme, les énoncés placés entre accolades `{` et `}` sont des commentaires. Ces commentaires ont ici pour but de justifier la validité de l'algorithme.

Algorithme 3.1 Algorithme de recherche séquentielle laborieuse.

Entrée : t un tableau, a et b deux indices, x un élément

Sortie : vrai si $x \in t.(a..b)$, faux sinon.

```

trouve := faux
pour  $k \leq b$  et  $t.(k) \neq x$  faire
    {trouve  $\iff x \in t.(a..k-1)$ }
    si  $t.(k) = x$  alors
        trouve := vrai
    fin si
    {trouve  $\iff x \in t.(a..k)$ }
fin pour
{trouve  $\iff x \in t.(a..b)$ }
renvoyer trouve

```

3.2.3 Coût

Si n est la longueur de la tranche, le nombre de comparaisons d'éléments du tableau $c(n)$ vaut dans tous les cas n .

3.3 La recherche séquentielle

3.3.1 Principe

Le principe est simple, on va parcourir les cases de la tranche de tableau dans l'ordre croissant des indices jusqu'à ce qu'on trouve l'élément x , ou bien jusqu'à ce qu'on arrive à la fin de la tranche, sans avoir trouvé l'élément x .

3.3.2 Vers l'algorithme

Plus précisément, soit k un entier de l'intervalle d'entier $\llbracket a, b \rrbracket$. On suppose qu'on a parcouru $t.(a..k-1)$, sans avoir rencontré l'élément cherché x . C'est à dire, $x \notin t.(a..k-1)$.

- Si $t.(k) = x$ alors la recherche est fructueuse.
- Sinon on peut affirmer que $t.(k) <> x$ et par conséquent $x \notin t.(a..k)$.
 - s'il existe un élément suivant, on procède de même...
 - sinon la recherche est infructueuse.

3.3.3 Algorithme

L'algorithme 3.2 précise le principe de la recherche séquentielle décrit ci-dessus.

3.3.4 Remarque importante

Les opérateurs en CAML sont séquentiels, c'est-à-dire la seconde opérande n'est évaluée que lorsque la première opérande ne suffit pas pour déterminer le résultat de l'opération. L'algorithme précédent utilise cette propriété, *mais il serait incorrect, si les opérateurs n'étaient pas séquentiels*. Toutefois, il est toujours possible de modifier cet algorithme pour obtenir une version qui reste valable même avec des opérateurs qui procèdent à l'évaluation totale.

L'algorithme 3.3 montre la recherche séquentielle lorsque les opérateurs booléens ne sont pas séquentiels.

Algorithme 3.2 Algorithme de recherche séquentielle.

Entrée : t un tableau, a et b deux indices, x un élément

Sortie : vrai si $x \in t.(a..b)$, faux sinon.

```

 $k := a$ 
 $\{x \notin t.(a..k - 1)\}$ 
tant que  $k \leq b$  et  $t.(k) \neq x$  faire
     $\{x \notin t.(a..k)\}$ 
    incrémenter  $k$ 
     $\{x \notin t.(a..k - 1)\}$ 
fin tant que
 $\{ (k > b \text{ et } x \notin t.(a..b)) \text{ ou bien } (k \leq b \text{ et } t.(k) = x) \}$ 
si  $k \leq b$  alors
    {recherche fructueuse et  $k$  est le plus petit indice de la tranche tel que  $t.(k) = x$ }
    renvoyer vrai
sinon
    {recherche infructueuse}
    renvoyer faux
fin si

```

Algorithme 3.3 Algorithme de recherche séquentielle modifié pour opérateurs booléens non séquentiels.

Entrée : t un tableau, a et b deux indices, x un élément

Sortie : vrai si $x \in t.(a..b)$, faux sinon.

```

si  $a \leq b$  alors
     $k := a$ 
     $\{x \notin t.(a..k - 1)\}$ 
    tant que  $k < b$  et  $t.(k) \neq x$  faire
         $\{x \notin t.(a..k)\}$ 
        incrémenter  $k$   $\{x \notin t.(a..k - 1)\}$ 
    fin tant que
     $\{ (k = b \text{ et } x \notin t.(a..b - 1)) \text{ ou bien } (k < b \text{ et } t.(k) = x) \}$ 
    si  $t.(k) = x$  alors
        {recherche fructueuse et  $k$  est le plus petit indice de la tranche tel que  $t.(k) = x$ }
        renvoyer vrai
    sinon
        {recherche infructueuse}
        renvoyer faux
    fin si
sinon
    {recherche infructueuse}
    renvoyer faux
fin si

```

3.3.5 Coût

On étudie le coût de la recherche séquentielle, en comptant le nombre de comparaisons $c(n)$ d'éléments du tableau, qu'on effectue lors d'une recherche dans une tranche $t.(a..b)$ de longueur n . Néanmoins ce nombre de comparaisons peut varier selon les valeurs contenues dans la tranche, et la valeur cherchée. C'est pourquoi, on va s'intéresser à certains cas particuliers comme le meilleur et le pire des cas. Ces deux situations particulières sont importantes car

- il est souvent bien plus facile de compter les comparaisons dans ces cas particulier que de déterminer le nombre de comparaisons effectuées en moyenne;
- elles permettent tout de même de procéder à un encadrement du comportement en moyenne.

Dans les deux algorithmes présentés précédemment, le meilleur des cas est celui, où on trouve en première position de la tranche $t.(a..b)$ l'élément x cherché, c'est-à-dire $x = t.(a)$.

Le pire des cas est atteint

- en cas de recherche infructueuse, c'est-à-dire lorsque l'élément $x \notin t.(a..b)$;
- ou bien lorsque l'élément x se trouve dans la dernière case de la tranche et pas avant, c'est à dire lorsque $x \notin t.(a..b - 1)$ et $x = t.(b)$.

algorithme	meilleur des cas	pire des cas
algorithme de recherche séquentielle (version opérateur booléen séquentiel)	1	n
algorithme de recherche séquentielle (version opérateur booléen non séquentiel)	2	n

3.4 La recherche séquentielle dans un tableau trié

Dans la suite de ce chapitre, on suppose que le type des éléments du tableau est muni d'une relation d'ordre *totale*, notée \leq .

On rappelle qu'une relation binaire est une relation d'ordre lorsqu'elle est *réflexive*, *anti-symétrique* et *transitive*.

réflexivité : pour tout x , on a $x \leq x$;

antisymétrie : pour tout (x, y) , si $x \leq y$ et si $y \leq x$ alors $x = y$;

transitivité : pour tout (x, y, z) , si $x \leq y$ et si $y \leq z$ alors $x \leq z$.

On dit qu'une relation d'ordre est totale lorsque pour tout (x, y) , on a $x \leq y$ ou bien $y \leq x$

En CAML, \leq est polymorphe. C'est-à-dire que le type de \leq est $'a \rightarrow 'a \rightarrow \text{bool}$ Pour les types `int`, `char`, `string`, `float` la "signification" de \leq est usuelle.

```
# 1<=3;;
- : bool = true
# 4<=3;;
- : bool = false
# "bleu" <= "rouge";;
- : bool = true
# 'a'<='e';;
- : bool = true
# 'Z'<='a';;
- : bool = true
# 1.0 <= 1.000001;;
- : bool = true
```

Mais, on ne peut comparer que des choses comparables, c'est à dire du même type.

```
# 1<="deux";;
Characters 2-8:
  1<="deux";;
    ^^^^^^
This expression has type string but is here used with type int
```

Enfin, la comparaison de valeurs fonctionnelles déclenche l'exception `Invalid_argument`

```
# let carre x = x*x ;;
val carre : int -> int = <fun>
# let cube x = x *(carre x) ;;
val cube : int -> int = <fun>
# carre <= cube;;
Exception: Invalid_argument "equal:_functional_value".
```

3.4.1 Tableau trié

On dit qu'une tranche de tableau $t.(a..b)$ est triée, si l'application de $\llbracket a, b \rrbracket$ définie par $i \mapsto t.(i)$ est croissante. c'est-à-dire pour tout i et j de $\llbracket a, b \rrbracket$ on a $i \leq j \implies t.(i) \leq t.(j)$.

Une caractérisation équivalente est

$$\forall i \in \llbracket a, b-1 \rrbracket \quad t.(i) \leq t.(i+1)$$

Exemple 3.4 :

voici un tableau d'entier trié :

0	5	5	7	12	21
---	---	---	---	----	----

3.4.2 Principe

En fait, on peut s'apercevoir qu'une recherche est infructueuse plus tôt. En effet lorsque la tranche est triée et lorsque $t.(i) > x$ alors pour tout j tel que $i \leq j \leq b$, on est sûr que $t.(j) \neq x$.

3.4.3 Algorithme

L'algorithme 3.4 montre la recherche séquentielle dans un tableau trié.

3.4.4 Coût

Le meilleur des cas, on trouve x tout de suite, où on s'aperçoit qu'il est inutile de continuer. Le pire des cas, on trouve x dans la dernière case de la tranche $t.(a..b)$, ou bien on s'aperçoit qu'il est inutile de continuer qu'à la dernière case.

algorithme	meilleur des cas	pire des cas
algorithme de recherche séquentielle dans un tableau trié	2	$n + 1$

Algorithme 3.4 Algorithme de recherche séquentielle dans un tableau trié.

Entrée : t un tableau trié, a et b deux indices, x un élément

Sortie : vrai si $x \in t.(a..b)$, faux sinon.

```

 $k := a$ 
 $\{x \notin t.(a..k-1)\}$ 
tant que  $k \leq b$  et  $t.(k) < x$  faire
     $\{x \notin t.(a..k)\}$ 
    incrémenter  $k$ 
     $\{x \notin t.(a..k-1)\}$ 
fin tant que
 $\{ (k > b \text{ et } x \notin t.(a..b)) \text{ ou bien } (k \leq b \text{ et } t.(k) \geq x) \}$ 
si  $k \leq b$  et  $t.(k) = x$  alors
    {recherche fructueuse et  $k$  est le plus petit indice de la tranche tel que  $t.(k) = x$ }
    renvoyer vrai
sinon
    {recherche infructueuse}
    renvoyer faux
fin si
  
```

3.5 La recherche dichotomique

3.5.1 Principe

Lorsqu'on fait une comparaison entre x et un élément du tableau trié $t.(m)$, on peut d'un coup éliminer de la tranche dans laquelle on effectue la recherche un gros morceau de cette tranche.

- lorsque $t.(m) < x$, on sait qu'il est inutile de regarder dans la tranche $t.(a..m)$ pour trouver x
- lorsque $t.(m) \geq x$, on sait que la première occurrence de x ne peut pas se trouver dans la tranche $t.(m+1..b)$

En choisissant, pour m l'indice du milieu de la tranche, cela permet d'éliminer (un peu plus de) la moitié des cases de la tranche. La longueur de la tranche, dans laquelle on doit effectuer la recherche, est donc divisée par 2.

Il ne reste plus qu'à recommencer... tant que la taille de la tranche restant à examiner est strictement supérieure à 1

3.5.2 Algorithme

La recherche dichotomique dans un tableau trié est décrite plus précisément dans l'algorithme 3.5.

3.5.3 Coût de la recherche dichotomique

Les figures 3.1 et 3.2 montrent sous forme d'arbres l'évolution des intervalles d'indice $[a, b]$ dans lesquels la recherche s'effectue par dichotomie pour des tableaux de longueur 8 et 10.

Une recherche s'effectue en partant de la racine de ces arbres et en suivant une branche vers le bas jusqu'à arriver à un intervalle ne contenant plus qu'un seul indice (en grisé sur les figures).

Le nombre de comparaisons de l'élément à rechercher avec des éléments du tableau est égal au nombre d'intervalles rencontrés en parcourant une telle branche.

Algorithme 3.5 Algorithme de recherche dichotomique dans un tableau trié.

Entrée : t un tableau trié, a et b deux indices, x un élément

Sortie : vrai si $x \in t.(a..b)$, faux sinon.

```

 $d := a$ 
 $f := b$ 
 $\{ \forall i \in \llbracket a, d-1 \rrbracket t.(i) < x \}$ 
 $\{ \forall i \in \llbracket f+1, b \rrbracket x \leq t.(i) \}$ 
tant que  $d < f$  faire
     $\{ \forall i \in \llbracket a, d-1 \rrbracket t.(i) < x \}$ 
     $\{ \forall i \in \llbracket f+1, b \rrbracket x \leq t.(i) \}$ 
     $m := (d + f)/2$  (division euclidienne)
    si  $t.(m) < x$  alors
         $d := m + 1$ 
    sinon
         $f := m$ 
    fin si
fin tant que
 $\{ f \leq d \text{ et } \}$ 
 $\{ \forall i \in \llbracket a, d-1 \rrbracket t.(i) < x \}$ 
 $\{ \forall i \in \llbracket f+1, b \rrbracket x \leq t.(i) \}$ 
si  $d = f$  et  $t.(d) = x$  alors
    {recherche fructueuse et  $d$  est le plus petit indice de la tranche tel que  $t.(d) = x$ }
    renvoyer vrai
sinon
    {recherche infructueuse}
    renvoyer faux
fin si

```

On peut donc conclure que pour un tableau de longueur 8, dans tous les cas, quatre comparaisons sont effectuées dans une recherche dichotomique, et pour un tableau de longueur 10 il en faut quatre ou cinq.

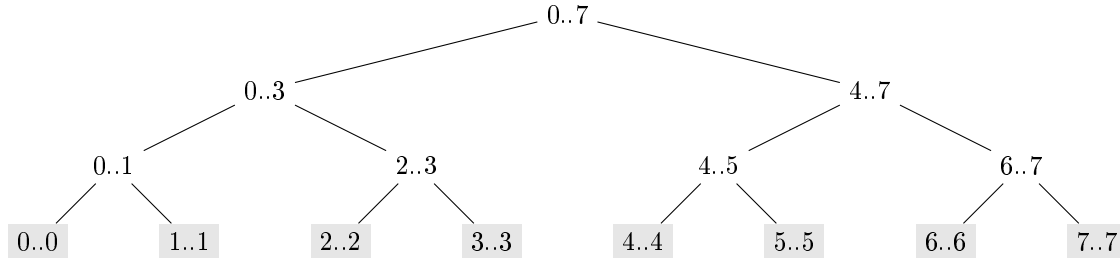


FIGURE 3.1 – Arbre des différents parcours pour une recherche dichotomique dans un tableau trié de longueur 8.

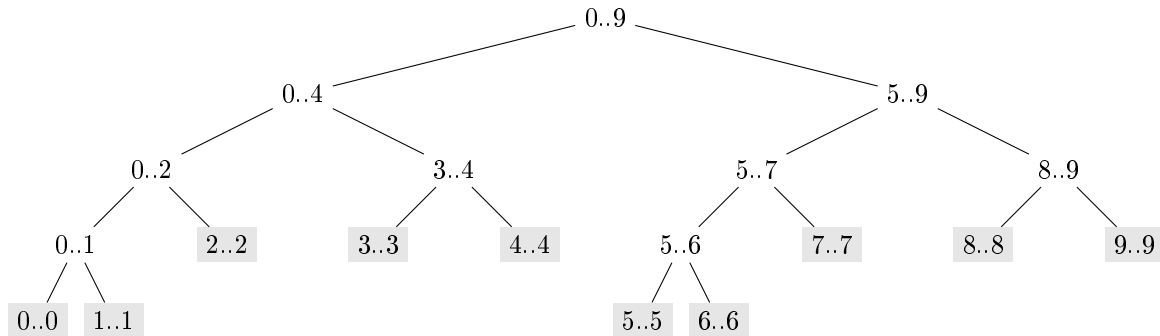


FIGURE 3.2 – Arbre des différents parcours pour une recherche dichotomique dans un tableau trié de longueur 10.

On peut montrer plus généralement que le nombre de comparaisons effectuée dans une recherche dichotomique dans un tableau trié de longueur n est compris entre $1 + \lfloor \log_2 n \rfloor$ et $1 + \lceil \log_2 n \rceil$, expressions dans lesquelles les notations $\lfloor x \rfloor$, $\lceil x \rceil$ et $\log_2 x$ désignent respectivement le plus grand entier inférieur ou égal à x , le plus petit entier supérieur ou égal à x , et le logarithme de base 2 de x (autrement dit $\frac{\ln x}{\ln 2}$).

On retiendra que l'ordre de grandeur du coût d'une recherche dichotomique est logarithmique en fonction de la taille du tableau.

3.6 Le code CAML

Les quatre algorithmes de recherche que nous avons vus dans ce qui précède sont implantés ici en CAML pour réaliser à chaque fois une fonction dont voici la spécification :

$$\begin{aligned}
&\text{recherche} : 'a \text{ array} \rightarrow \text{int} \rightarrow \text{int} \rightarrow 'a \rightarrow \text{bool} \\
&\text{recherche } t \ a \ b \ x \mapsto \begin{cases} \text{Vrai} & \text{si } x \in t(a..b) \\ \text{Faux} & \text{sinon} \end{cases} \\
&\text{CU} : 0 \leq a, b < \text{long}(t).
\end{aligned}$$

3.6.1 Recherche séquentielle laborieuse

```

let recherche_laborieuse t a b x =
  let trouve = ref false in
    for i = a to b do
      if t.(i) = x then
        trouve := true
    done;
  !trouve

```

3.6.2 Recherche séquentielle

```

let recherche t a b x =
  let i = ref a in
    while (!i <= b) && (t.(!i) <> x) do
      i := !i + 1
    done;
  !i <= b

```

```

let recherche t a b x =
  if a <= b then
    let i = ref a in
      while (!i < b) && (t.(!i) <> x) do
        i := !i + 1
      done;
    (!i < b) || ((!i = b) && (t.(!i) = x))
  else
    false

```

3.6.3 Recherche séquentielle dans tableau trié

```

let recherche_triee t a b x =
  let i = ref a in
    while (!i <= b) && (t.(!i) < x) do
      i := !i + 1
    done;
  (!i <= b) && (t.(!i) = x)

```


3.6.4 Recherche dichotomique dans un tableau trié

```

let recherche_dicho t a b x =
  if a <= b then
    let (d,f) = (ref a,ref b) in
      while (!d < !f) do
        let m = (!d + !f)/2 in
          if t.(m) < x then
            d := m + 1
          else
            f := m
        done;
      t.(!d) = x
  else
    false

```

3.7 Exercices

3.7.1 Recherche séquentielle

Exercice 3-1 *Recherches séquentielles renvoyant un indice*

Nous avons vu comment implanter les algorithmes de recherche séquentielle pour réaliser des fonctions de recherche renvoyant une valeur booléenne. Vous allez dans cet exercice réaliser des fonctions donnant l'indice de l'élément trouvé. Le type des fonctions à réaliser est donc

$'a \text{ array} \rightarrow \text{int} \rightarrow \text{int} \rightarrow 'a \rightarrow \text{int}.$

Question 1 Avant de vous lancer dans ces implantations, il faut préciser la spécification de ces fonctions de recherche.

1. Si l'élément recherché est présent plusieurs fois dans le tableau, quel indice renvoyer ?
2. Idem si cet élément n'est pas dans le tableau ?

Question 2 Réalisez une implantation de la recherche séquentielle dans un tableau non trié, puis dans un tableau trié, qui donne le plus petit indice d'un élément présent dans le tableau.

Question 3 Idem pour le plus grand indice.

Exercice 3-2 *Point trop n'en faut !*

Voici une implantation en CAML erronée de la recherche laborieuse vue en cours.

```

let recherche_laborieuse t a b x =
  let trouve = ref false
  in
    for i = a to b do
      if t.(i)=x then
        trouve:=true
      else
        trouve:=false
    done;

```

!trouve

Question 1 Pour chacun des deux appels suivants à cette fonction de recherche, présentez sous forme d'un tableau les valeurs successives que prennent les variables `i` et `trouve`, ainsi que la valeur renvoyée par la fonction.

1. `recherche_laborieuse [|3; 1; 2; 4; 1|] 0 4 1;`
2. `recherche_laborieuse [|3; 1; 2; 4; 5|] 0 4 1.`

Question 2 Quelle est l'erreur de programmation manifestement commise ?

3.7.2 Tableau trié

Exercice 3-3 *Tester si un tableau est trié*

Question 1 Quel algorithme peut-on utiliser pour tester si un tableau est trié ? Combien de comparaisons d'éléments du tableau faut-il faire ?

Question 2 Réalisez une fonction qui teste si un tableau est trié.

Question 3 Quel est le type de la fonction que vous avez réalisée ? Que signifie-t-il ?

3.7.3 Recherche dichotomique

Exercice 3-4 *Pour comprendre la recherche dichotomique*

Soit $t =$

0	2	4	6	8	10	12	14	16	18	20	22
---	---	---	---	---	----	----	----	----	----	----	----

 un tableau d'entiers.

Question 1 Donnez la suite des tranches de tableau parcourue lors de la recherche dichotomique de $x = 11$ dans le tableau t . Combien de comparaisons d'éléments du tableau sont-elles effectuées ?

Question 2 Même question avec $x = 12$.

Question 3 Même question avec $x = 10$.

Question 4 Dessinez l'arbre des recherches possibles pour une recherche dichotomique dans un tableau de longueur 12.

Exercice 3-5 *Recherche dichotomique renvoyant un indice*

Question 1 Réalisez une implantation de la recherche dichotomique qui renvoie un indice plutôt qu'une valeur booléenne.

Question 2 Dans le cas où l'élément recherché est présent plusieurs fois dans le tableau, quel est l'indice qui est renvoyé par votre fonction ?

Exercice 3-6 *Variante de la recherche dichotomique*

Qu'elle soit fructueuse ou non, la recherche dichotomique présentée en cours effectue toujours $\log_2 n$ comparaisons environ. Cela provient du fait que la comparaison effectuée à chaque étape de la recherche est $t.(m) < x$. Pourtant, si on s'intéressait au cas de l'égalité de l'élément $t.(m)$, milieu de la tranche courante, avec l'élément x recherché, on pourrait arrêter plus vite l'exploration du tableau.

Question 1 Décrivez un algorithme de recherche dichotomique qui tient compte de cette remarque.

Question 2 Dans le cas d'une recherche infructueuse combien de comparaisons sont-elles effectuées ?

et dans le cas d'une recherche fructueuse ?

Question 3 En CAML, la fonction `compare` de type `'a → 'a → int`, compare deux éléments de même type¹ et renvoie l'entier

1. -1 si le premier élément est strictement plus petit que le second ;
2. 0 si les deux éléments sont égaux ;
3. et 1 si le premier élément est strictement plus grand que le second.

En voici quelques exemples :

```
# compare 'a' 'b' ;;
- : int = -1
# compare 1 3 ;;
- : int = -1
# compare 1 1 ;;
- : int = 0
# compare "toto" "toto" ;;
- : int = 0
# compare 3.5 1. ;;
- : int = 1
# compare (1,3) (0,5) ;;
- : int = 1
```

Utilisez cette fonction pour réduire le nombre de comparaisons d'éléments du tableau.

1. Le type de ces deux éléments peut être quelconque mais pas une fonction.

Chapitre 4

Trier un tableau

4.1 Introduction

Réaliser un tri (ou un classement) est une opération relativement courante dans la vie quotidienne : trier les cartes d'un jeu, trier par ordre alphabétique les livres d'une bibliothèque, classer des concurrents selon leurs performances, etc... Gérer un agenda téléphonique est aussi une forme de tri, puisqu'un ordre (alphabétique selon le nom des individus) est appliqué, même si ce tri est fait petit à petit, au fur et à mesure de l'insertion de nouveaux numéros.

Un tri porte généralement sur un nombre assez important de données. En effet, lorsque l'on a peu de numéros de téléphone à gérer, on se contente souvent de les inscrire dans l'ordre où ils sont connus sur une feuille libre, mais, dès que leur nombre devient trop important¹, on ressent le besoin de les classer et donc de les trier dans un agenda, afin d'accéder plus rapidement à une information.

Effectuer un tri est souvent assez long et fastidieux (du moins quand les données initiales ne sont pas ou peu triées). Cependant, l'intérêt d'une telle opération, une fois réalisée, est de pouvoir facilement accéder aux différentes données en s'appuyant sur le critère du tri. C'est le cas en particulier de l'algorithme efficace de recherche dichotomique (cf 3.5).

Un tri est toujours réalisé selon un critère, particulier et arbitraire, portant sur ces données : une *relation d'ordre*, c'est-à-dire une relation binaire \mathcal{R} sur les éléments d'un ensemble E ayant les trois propriétés

- *réflexivité*, i.e. $\forall x \in E, x\mathcal{R}x$;
- *antisymétrie*, i.e. $\forall x, y \in E, x\mathcal{R}y$ et $y\mathcal{R}x$ entraînent $x = y$;
- *transitivité*, i.e. $\forall x, y, z \in E, x\mathcal{R}y$ et $y\mathcal{R}z$ entraînent $x\mathcal{R}z$.

Cette relation est dite *totale* si $\forall x, y \in E$, on a $x\mathcal{R}y$ ou $y\mathcal{R}x$. Elle est dite *partielle* dans le cas contraire.

Ainsi, on peut arbitrairement choisir de classer les livres d'une bibliothèque selon un ordre croissant ou décroissant : alphabétique par auteurs ou par titres ; ou numérique selon les dates de parutions, etc.

Du fait de leur fréquente utilisation, les tris ont été très étudiés en informatique et constituent des exercices classiques lors de l'initiation à la programmation, car ils représentent un excellent support pour l'étude de problèmes plus généraux. De nombreux algorithmes de tri existent, plus ou moins efficaces et plus ou moins faciles à mettre en œuvre.

1. Il est d'ailleurs difficile d'expliquer à partir de quand ce nombre est trop important, on retrouve ici le problème de savoir combien il faut de grains de sable pour en faire un tas...

4.2 Qu'est-ce que trier ?

On suppose dans ce chapitre que les éléments des tableaux considérés appartiennent à un ensemble E totalement ordonné par une relation notée \leq . Cet ensemble peut être par exemple l'ensemble des nombres entiers ou flottants muni de l'ordre numérique usuel, l'ensemble des caractères muni de l'ordre induit par le code ASCII, l'ensemble des chaînes de caractères muni de l'ordre lexicographique.

Trier un tableau c'est obtenir, à partir d'un tableau t , un tableau contenant les mêmes éléments mais rangés par ordre croissant

Du point de vue du traitement des données, cette définition n'est pas suffisante : doit-on construire un nouveau tableau et laisser le tableau t inchangé ? ou bien doit-on transformer le tableau de sorte qu'il soit trié ?

Le problème du tri d'un tableau peut donc se décliner sous ces deux aspects :

1. à partir d'un tableau t , construire un nouveau tableau t' trié de même longueur que t , contenant les mêmes éléments que t ;
2. transformer un tableau t en un tableau trié, sans globalement changer son contenu, mais en déplaçant les éléments au sein du tableau.

De nombreux algorithmes de tri ont été conçus. Parmi eux, on distingue les tris *comparatifs* qui opèrent par comparaison d'éléments du tableau (tri par sélection, tri par insertion, ...), d'autres tris qui opèrent par d'autres moyens (comme le tri par dénombrement par exemple).

4.3 Le tri par sélection

4.3.1 Sélection du minimum dans une tranche de tableau

L'algorithme 4.1 montre comment effectuer une recherche de l'indice du plus petit élément d'un tableau.

Algorithme 4.1 Algorithme de sélection du minimum dans une tranche de tableau

Entrée : t un tableau de longueur n , et a et b deux indices tels que $0 \leq a \leq b < n$.

Sortie : indice d'un élément minimal de la tranche $t(a..b)$

```

1: indice_min :=  $a$ 
2: pour  $i$  variant de  $a + 1$  à  $b$  faire
3:   si  $t(i) < t(\textit{indice\_min})$  alors
4:      $\textit{indice\_min} := i$ 
5:   fin si
6: fin pour
7: renvoyer  $\textit{indice\_min}$ 
```

4.3.2 L'algorithme de tri par sélection

Le principe de l'algorithme de tri par sélection consiste à construire petit à petit une tranche triée grandissante du tableau en

- sélectionnant à chaque étape le plus petit élément de la partie non triée
- et en l'échangeant avec l'élément du début de la tranche non triée.

Exemple 4.1 :

La table 4.1 montre l'évolution de la tranche triée au cours des différentes étapes du tri par sélection pour le tableau de longueur 8 :

		0	1	2	3	4	5	6	7
	$t =$	T	I	M	O	L	E	O	N

i	ind_min	t
0	5	E I M O L T O N
1	1	E I M O L T O N
2	4	E I L O M T O N
3	4	E I L M O T O N
4	7	E I L M N T O O
5	6	E I L M N O T O
6	7	E I L M N O O T

TABLE 4.1 – États successifs du tableau au cours du tri par sélection

Algorithme 4.2 Algorithme de tri par sélection du minimum

Entrée : t un tableau de longueur n .

Sortie : t un tableau trié de longueur n contenant les mêmes éléments.

- 1: **pour** i variant de 0 à $n - 2$ **faire**
- 2: $indice_min := \text{sélectionner_min}(t, i, n - 1)$
- 3: échanger $t(indice_min)$ et $t(i)$
- 4: {À ce stade, la tranche $t(0..i)$ est triée
et contient les $i + 1$ plus petits éléments de t }
- 5: **fin pour**
- 6: {Le tableau t est trié.}

4.3.3 Coût du tri par sélection

Notons $c_{\text{tri-select}}(n)$ le nombre de comparaisons d'éléments du tableau à effectuer pour trier un tableau de longueur n par le tri par sélection. Et notons $c_{\text{select-min}}(k)$ le nombre de comparaisons à effectuer pour sélectionner l'indice du minimum d'une tranche de tableau de longueur k selon l'algorithme 4.1.

Exemple 4.2 :

Commençons par examiner le nombre de comparaisons effectuées dans le tri du tableau de l'exemple 4.1.

En reprenant le tableau 4.1, voici le nombre de comparaisons d'éléments de t effectuées à chaque étape.

i	0	1	2	3	4	5	6
$c_{\text{select-min}}(8-i)$	7	6	5	4	3	2	1

Soit au total $7 + 6 + \dots + 1 = 28$ comparaisons.

L'examen de l'algorithme 4.2 montre que l'on a

$$c_{\text{tri-select}}(n) = \sum_{k=2}^n c_{\text{select-min}}(k),$$

et celui de l'algorithme 4.1 montre que

$$c_{\text{select-min}}(k) = k - 1.$$

Par conséquent, on a

$$\begin{aligned} c_{\text{tri-select}}(n) &= \sum_{k=2}^n (k - 1) \\ &= \frac{n(n-1)}{2}. \end{aligned}$$

Théorème 4.1 (Coût du tri par sélection) *Pour tout tableau de longueur n , le nombre de comparaisons d'éléments du tableau pour le trier par le tri par sélection est donné par*

$$c_{\text{tri-select}}(n) = \frac{n(n-1)}{2} \sim \frac{n^2}{2}.$$

4.4 Le tri par insertion

4.4.1 Insertion d'un élément dans une tranche triée

L'algorithme 4.3 montre comment insérer un élément dans un début de tableau trié.

Algorithme 4.3 Insertion d'un élément dans une tranche triée de tableau

Entrée : t un tableau de longueur n et $1 \leq i < n$ un indice tel que la tranche $t(0..i-1)$ soit triée

Sortie : insertion de l'élément $t(i)$ dans $t(0..i)$ de sorte que la tranche $t(0..i)$ soit triée

```

1:  $aux := t(i)$ 
2:  $k := i$ 
3: tant que  $(k \geq 1)$  et  $(aux < t(k-1))$  faire
4:    $t(k) := t(k-1)$ 
5:    $k := k - 1$ 
6: fin tant que
7:  $t(k) := aux$ 
8: {la tranche  $t(0..i)$  est triée}
```

Algorithme 4.4 Algorithme de tri par insertion**Entrée :** t un tableau de longueur n .**Sortie :** t un tableau trié de longueur n contenant les mêmes éléments.

- 1: **pour** i variant de 1 à $n - 1$ **faire**
- 2: insérer $t(i)$ dans $t(0..i)$
- 3: {À ce stade, la tranche $t(0..i)$ est triée}
- 4: **fin pour**

4.4.2 L'algorithme du tri par insertion

Le principe de l'algorithme du tri par insertion (cf algo 4.4) consiste à contruire une tranche triée de longueur croissante en insérant successivement le premier élément non considéré dans la tranche triée en construction.

Exemple 4.3 :

Le tableau 4.2 montre les états successifs d'un tableau au cours du tri par insertion. Le tableau à trier est le même que celui de l'exemple 4.1.

i	t
1	I T M O L E O N
2	I M T O L E O N
3	I M O T L E O N
4	I L M O T E O N
5	E I L M O T O N
6	E I L M O O T N
7	E I L M N O O T

TABLE 4.2 – États successifs du tableau au cours du tri par insertion

4.4.3 Coût du tri par insertion

Notons $c_{\text{tri-insert}}(n)$ le nombre de comparaisons d'éléments du tableau à effectuer pour trier un tableau de longueur n par le tri par insertion. Et notons $c_{\text{insérer}}(k)$ le nombre de comparaisons à effectuer pour insérer un élément dans une tranche triée de tableau de longueur k selon l'algorithme 4.3.

Exemple 4.4 :

Commençons par examiner le nombre de comparaisons effectuées dans le tri du tableau de l'exemple 4.3.

En reprenant le tableau 4.2, voici le nombre de comparaisons d'éléments de t effectuées à chaque étape.

i	1	2	3	4	5	6	7
$c_{\text{insérer}}(i+1)$	1	2	2	4	5	2	4

Soit au total $1 + 2 + \dots + 2 + 4 = 20$ comparaisons.

Théorème 4.2 (Coût du tri par insertion) *Pour tout tableau de longueur n , le nombre de comparaisons d'éléments du tableau pour le trier par le tri par insertion est encadré par*

$$n - 1 \leq c_{\text{tri-select}}(n) \leq \frac{n(n-1)}{2} \sim \frac{n^2}{2}.$$

La minoration (meilleur des cas) est atteinte pour un tableau trié. Et la majoration (pire des cas) est atteinte pour un tableau trié dans l'ordre inverse.

On peut prouver que le coût moyen du tri par insertion est équivalent à $\frac{n^2}{4}$.

4.5 Le tri par dénombrement

Le tri qui va être présenté ici diffère des tris précédents car il n'opère pas par comparaison des éléments du tableau à trier. Il consiste essentiellement à compter le nombre d'occurrences de chacun des éléments du tableau.

Cet algorithme nécessite que soit connu et fixé par avance les éléments susceptibles d'appartenir au tableau, et que le nombre de ces éléments soit relativement limité. Il n'est pas adapté au tri de tableaux de nombres (entiers ou flottants), ou bien de chaînes de caractères, si aucune restriction sur ces nombres ou ces chaînes pouvant figurer dans le tableau. En revanche, il convient très bien dans le cas où le nombre des valeurs possibles pour les éléments est limité. Par exemple, un tableau ne pouvant contenir que des entiers de 1 à 10, ou bien un tableau ne pouvant contenir que les lettres de l'alphabet latin (caractères compris entre A et Z).

Désignons par m le nombre des valeurs possibles contenues dans un tel tableau ($m = \text{card}(E)$).

Exemple 4.5 :

Prenons pour exemple l'ensemble des lettres de l'alphabet latin.

$$E = \{A, B, \dots, Z\}$$

$$m = 26.$$

On veut trier le tableau de lettres

$t =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
T	I	M	O	L	E	O	N	E	S	T	U	N	H	O	M	M	E	P	O	L	I	T	I	Q	U	E	G	R	E	C

.

On commence par compter le nombre d'occurrences dans t de chacune des 26 lettres de l'alphabet. Pour cela on initialise un tableau de 26 compteurs (un par lettre).

$cpt =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

.

Puis en parcourant séquentiellement le tableau t , on compte chacune des 26 lettres. On obtient

$cpt =$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	0	1	0	5	0	1	1	3	0	0	2	3	2	4	1	1	1	1	3	2	0	0	0	0	0

.

Enfin on transforme ce tableau en remplaçant les effectifs en effectifs cumulés.

$$cpt = \begin{array}{c} 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19 \quad 20 \quad 21 \quad 22 \quad 23 \quad 24 \quad 25 \\ \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{6} \boxed{6} \boxed{7} \boxed{8} \boxed{11} \boxed{11} \boxed{11} \boxed{13} \boxed{16} \boxed{18} \boxed{22} \boxed{23} \boxed{24} \boxed{25} \boxed{26} \boxed{29} \boxed{31} \boxed{31} \boxed{31} \boxed{31} \boxed{31} \boxed{31} \end{array}.$$

Ceci fait on construit en parcourant séquentiellement le tableau t , on place l'élément $t(i)$ de ce tableau dans un tableau t' à l'indice donné par le tableau cpt que l'on aura préalablement décrémenté.

$$t' = \begin{array}{c} 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19 \quad 20 \quad 21 \quad 22 \quad 23 \quad 24 \quad 25 \quad 26 \quad 27 \quad 28 \quad 29 \quad 30 \\ \boxed{C} \boxed{E} \boxed{E} \boxed{E} \boxed{E} \boxed{E} \boxed{G} \boxed{H} \boxed{I} \boxed{I} \boxed{I} \boxed{L} \boxed{L} \boxed{M} \boxed{M} \boxed{M} \boxed{N} \boxed{N} \boxed{O} \boxed{O} \boxed{O} \boxed{O} \boxed{P} \boxed{Q} \boxed{R} \boxed{S} \boxed{T} \boxed{T} \boxed{T} \boxed{U} \boxed{U} \end{array}.$$

Algorithme 4.5 Algorithme de tri par dénombrement

Entrée : t un tableau de longueur n dont les éléments sont pris dans un ensemble qui en contient m .

Sortie : t' un tableau trié de longueur n contenant les mêmes éléments.

- 1: {Compter les éléments de t }
 - 2: initialiser à 0 un tableau cpt de compteurs de longueur m
 - 3: **pour** chaque élément e de t **faire**
 - 4: incrémenter le compteur qui lui correspond dans le tableau cpt
 - 5: **fin pour**
 - 6: {Transformer cpt pour obtenir les effectifs croissants}
 - 7: **pour** k variant de 1 à $m - 1$ **faire**
 - 8: $cpt(k) := cpt(k - 1) + cpt(k)$
 - 9: **fin pour**
 - 10: Soit t' un tableau de longueur n
 - 11: **pour** i variant de 0 à $n - 1$ **faire**
 - 12: Soit k le rang de $t(i)$
 - 13: décrémenter $cpt(k)$
 - 14: $t'(cpt(k)) := t(i)$
 - 15: **fin pour**
-

4.6 Implantations en CAML

Dans ce qui suit, sont données les implantations en CAML des fonctions construisant une version triée du tableau passé en paramètre. Toutes ces fonctions vérifient la même spécification que nous donnons ci-dessous, et qui ne sera pas appelée à chaque fois.

```
(*
  fonction tri_XXX : 'a array -> 'a array
  parametre
    t : 'a array
  valeur renvoyee : 'a array = une version triee du tableau t
  CU : aucune
*)
```

Il est très facile d'adapter l'implantation de ces fonctions pour en faire des procédures de tri qui transforment le tableau passé en paramètre en un tableau trié. La spécification commune de toutes ces procédures est alors

```
(*
  procedure trier_XXX : 'a array -> unit
  parametre
    t : 'a array
    action : trie le tableau t
    CU : aucune
*)
```

4.6.1 Tri par sélection

Voici une implantation en CAML d'une fonction de tri suivant l'algorithme de tri par sélection présenté page 71. Elle est précédée d'une implantation d'une fonction de sélection de l'indice du minimum d'une tranche de tableau.

```
(* fonction indice_minimum : 'a array -> int -> int -> int
  parametres
    t : 'a array
    a : int
    b : int
    valeur renvoyee : int = indice du plus petit element de
      la tranche t(a..b)
    CU :  $0 \leq a \leq b < n$ 
*)
let indice_minimum t a b =
  let ind_min = ref a in
  for i = a + 1 to b do
    if t.(i) < t.(!ind_min) then
      ind_min := i
  done ;
  !ind_min
```

```
let tri_selection t =
  let t' = Array.copy t
  and n = Array.length t
  in
  for i = 0 to n - 2 do
    (* recherche du plus petit element de la tranche t'.(i..n-1) *)
    let ind_min = indice_minimum t' i (n - 1) in
    (* echanger t'.(i) et t'.(ind_min) si necessaire *)
    if i <> ind_min then begin
      let aux = t'.(i)
      in
      t'.(i) <- t'.(ind_min) ;
      t'.(ind_min) <- aux
    end ;
```

```
done ;
t'
```

4.6.2 Tri par insertion

Voici une implantation en CAML d'une fonction de tri suivant l'algorithme de tri par insertion présenté page 73. Cette fonction utilise la procédure **inserer** que voici

```
(*
  procedure inserer : 'a array -> int -> unit
  parametres
    t : 'a array
    i : int
  action : insere l'element t(i) a sa place dans
  la tranche t(0..i) de sorte qu'elle soit triee
  CU :  $1 \leq i < n$  et  $t(0..i-1)$  triee
*)
let inserer t i =
  let k = ref i
  and aux = t.(i) in
  while (!k >= 1) && (aux < t.(!k - 1)) do
    t.(!k) <- t.(!k - 1) ;
    k := !k - 1
  done ;
  t.(!k) <- aux
```

```
let tri_insertion t =
  let t' = Array.copy t
  and n = Array.length t in
  for i = 1 to n - 1 do
    inserer t' i
  done ;
  t'
```

On peut aussi déclarer la fonction **inserer** localement dans la fonction **tri_insertion**. Il n'est alors plus nécessaire de passer le tableau en paramètre de la fonction **inserer** si on la déclare sous la portée de la déclaration de la variable locale **t'**.

```
let tri_insertion t =
  let t' = Array.copy t
  and n = Array.length t in
  let inserer i =
    let k = ref i
    and aux = t'.(i) in
    while (!k >= 1) && (aux < t'.(!k - 1)) do
      t'.(!k) <- t'.(!k - 1) ;
      k := !k - 1
    done ;
    t'.(!k) <- aux
  in
```

```

in
  for i = 1 to n - 1 do
    insérer i
  done ;
  t'

```

4.6.3 Tri par dénombrement

Voici une implantation en CAML d'une fonction de tri suivant l'algorithme de tri par dénombrement présenté page 75. Conformément à ce qui a été dit cette fonction nécessite une fonction de conversion des éléments à trier en indices entiers :

`element_en_indice : 'a → int`

```

let tri_denombrement t =
  let n = Array.length t in
  let cpt = Array.make m 0
  and t' = Array.make n t.(0) in
    for i = 0 to n - 1 do
      let k = element_en_indice t.(i) in
        cpt.(k) <- cpt.(k) + 1
    done ;
    for k = 1 to m - 1 do
      cpt.(k) <- cpt.(k-1) + cpt.(k)
    done ;
    for i = 0 to n - 1 do
      let k = element_en_indice t.(i) in
        cpt.(k) <- cpt.(k) - 1 ;
        t'.(cpt.(k)) <- t.(i)
    done ;
  t'

```

4.7 À consulter sur le Web

Deux sites intéressants pour illustrer les tris :

1. Sur le site *Interstices*

http://interstices.info/jcms/c_6973/les-algorithmes-de-tri

(consulté le 15/01/2012)

2. Plusieurs algorithmes de tri chorégraphiés

tri par sélection <http://www.youtube.com/watch?v=Ns4TPTC8whw>

tri par insertion <http://www.youtube.com/watch?v=R0a1U379l3U>

(consulté le 15/01/2012)

4.8 Exercices

Exercice 4-1 Variante du tri par sélection

Nous avons présenté le tri par sélection du plus petit élément de la tranche restant à trier. Il est possible aussi de faire un tri par sélection du plus grand élément.

Question 1 Donnez l'algorithme de tri par sélection du plus grand élément.

Question 2 Implantez cet algorithme pour réaliser une procédure qui trie par cette méthode le tableau passé en paramètre.

Exercice 4-2 Tri à bulle

L'algorithme 4.6 est un algorithme de tri dénommé *tri à bulles* qui est une certaine forme de tri par sélection du minimum.

Algorithme 4.6 Algorithme du tri à bulles

Entrée : t un tableau de longueur n .

Sortie : t un tableau trié de longueur n contenant les mêmes éléments.

```

1: pour  $i$  variant de 0 à  $n - 2$  faire
2:   {mettre le plus petit élément de la tranche  $t(i..n - 1)$  en position  $i$ }
3:   pour  $j$  variant de  $n - 1$  à  $i + 1$  en décroissant faire
4:     si  $t(j) < t(j - 1)$  alors
5:       échanger  $t(j)$  et  $t(j - 1)$ 
6:     fin si
7:   fin pour
8:   {La tranche  $t(0..i)$  est triée et ses éléments sont inférieurs ou égaux
   à tous les autres éléments de  $t$ .}
9: fin pour
10: {la tranche  $t(0..n - 1)$  est triée.}
```

Question 1 Donnez les états successifs du tableau à la fin de chaque étape de la boucle pour située des lignes 3 à 7 lorsque $i = 0$ et le tableau à trier est

$$t = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline T & I & M & O & L & E & O & N \\ \hline \end{array}$$

Question 2 Même question pour la fin de chaque étape de la boucle pour située des lignes 1 à 9.

Question 3 Combien de comparaisons d'éléments du tableau sont-elles effectuées lors du tri d'un tableau de longueur n ?

Question 4 Si lors d'une étape de la boucle pour principale, aucun échange n'est effectué dans la boucle pour interne, c'est que le tableau est trié. Il est donc inutile de poursuivre la boucle externe.

Décrivez un algorithme qui tient compte de cette remarque.

Question 5 Combien de comparaisons d'éléments du tableau sont-elles effectuées lors du tri d'un tableau de longueur n avec cette variante du tri à bulles ? Décrivez le meilleur et le pire des cas.

Question 6 Réalisez une procédure qui trie le tableau passé en paramètre selon cet algorithme.

Exercice 4-3

On modifie l'algorithme de tri par insertion en effectuant la boucle dans l'ordre décroissant des indices :

Entrée : t un tableau de longueur n .

Sortie : t un tableau trié de longueur n contenant les mêmes éléments.

- 1: **pour** i variant de $n - 1$ à 1 en décroissant **faire**
- 2: insérer $t(i)$ dans $t(0..i)$
- 3: **fin pour**

Question 1 Quel est le tableau que l'on obtient lorsque cet algorithme est appliqué à

$$t = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline \text{T} & \text{I} & \text{M} & \text{O} & \text{L} & \text{E} & \text{O} & \text{N} \\ \hline \end{array} ?$$

Question 2 Qu'obtient-on si on applique cet algorithme sur un tableau trié dans l'ordre croissant ? décroissant ?

Exercice 4-4 *Trier les caractères d'une chaîne de caractères*

Réalisez avec l'algorithme de votre choix une fonction `trie_chaine` de type `string → string` qui construit la chaîne de caractères obtenues en triant les caractères de la chaîne passée en paramètre.

Exercice 4-5 *Passage de l'opérateur de comparaison en paramètre*

Les algorithmes de tris sont exposés en utilisant la relation \leq entre éléments du tableau. Les implantations en CAML peuvent se faire en utilisant l'opérateur polymorphe `<=` qui, rappelons-le, est de type

$$'a \rightarrow 'a \rightarrow \text{bool}.$$

Dans certaines situations, nous pouvons être amenés à ne pas utiliser cet opérateur prédéfini, mais un autre opérateur défini par nos soins.

Question 1 Réalisez une fonction de tri qui, outre le tableau à trier, prend aussi la fonction de comparaison en paramètre. Le type de cette fonction est donc

$$('a \rightarrow 'a \rightarrow \text{bool}) \rightarrow 'a \text{ array} \rightarrow 'a \text{ array}^2.$$

De cette façon le tri d'un tableau d'entiers dans l'ordre croissant s'obtient par l'appel suivant

```
# tri (<=) [|3;1;4;1;5;9;2|] ;;
- : int array = [|1; 1; 2; 3; 4; 5; 9|]
```

et le tri dans l'ordre décroissant s'obtient par

```
# tri (>=) [|3;1;4;1;5;9;2|] ;;
- : int array = [|9; 5; 4; 3; 2; 1; 1|]
```

Question 2 Soit t un tableau de couples dont la première composante est une chaîne de caractères, et la seconde un nombre entier. On veut trier le tableau par valeurs croissantes des premières composantes, et en cas d'égalité par valeurs croissantes des secondes composantes.

Donnez une expression permettant de le faire.

2. ou bien $'a \text{ array} \rightarrow ('a \rightarrow 'a \rightarrow \text{bool}) \rightarrow 'a \text{ array}$

Chapitre 5

Enregistrements

5.1 Besoin des enregistrements

La nécessité de manipuler des données composées de plusieurs autres données nous a conduit à introduire la notion de n -uplet (cf chapitre 1 page 1.2) et la notion de tableau (cf chapitre 2 page 2.1).

Les tableaux fournissent une structure simple permettant un accès facile et immédiat à chacune des composantes grâce à la notation indicielle, mais obligent toutes les composantes à être du même type. Il est par exemple impossible (et pas naturel) d'utiliser un tableau pour représenter un étudiant avec son NIP, ses noms et prénoms, ainsi que sa note en informatique.

Les n -uplets permettent l'agrégation de données de types hétérogènes. Mais l'accès à une composante ne peut se faire qu'à l'aide d'une opération de filtrage qui doit faire apparaître toutes les composantes du n -uplets. De plus l'ordre dans lequel ces composantes apparaissent est important. Par exemple, si on veut une structure pour représenter les NIP, nom, prénom et note en informatique d'un étudiant, il est possible de le faire avec des quadruplets de types différents :

- (11009999, "Raymond", "Calbuth", 10.) de type `(int * string * string * float)`;
- (10., "Calbuth", "Raymond", 11009999) de type `(float * string * string * int)`;
- ("Calbuth", "Raymond", 11009999, 10.) de type `(string * string * int * float)`;
- et bien d'autres possibilités encore ...

Cette multiplicité de représentations liés à l'ordre impose au programmeur de se souvenir de l'ordre dans lequel les composantes sont énumérées dans le n -uplet, ordre qui le plus souvent est tout à fait arbitraire : le nom doit-il être placé avant ou après le prénom ? De plus l'accès à l'une ou l'autre des composantes ne peut se faire que par filtrage avec une connaissance précise de l'ordre des composantes dans le n -uplet. Par exemple, si on considère la première représentation de nos étudiants, l'accès au nom d'un étudiant `e` se fait avec l'expression

```
match e with  
| (_,_,n,_) -> n
```

qui reconnaitsons-le est un peu lourde.

Les *enregistrements* sont donc des structures de données qui allient à la fois la possibilité de regrouper des données de type hétérogènes, comme avec les n -uplets, et l'accès facile à ces données, comme avec les tableaux, mais à la différence de ceux-ci non pas par des indices, mais par des noms de composantes.

Un *enregistrement* est donc la donnée d'un nombre fini de *composantes* (ou encore *champs*) pouvant être de types hétérogènes, chacune des composantes étant *nommées*. L'accès à une

composante d'un enregistrement ne s'effectue plus par filtrage en connaissant la position de cette composante comme dans un n -uplet, mais par son nom.

Exemple 5.1 :

Nous avons déjà eu l'occasion de représenter des dates par des triplets (cf exercice 8 page 8). Nous pouvons aussi représenter les dates par des enregistrements à trois champs :

1. **quantieme** : entier ;
2. **mois** : entier ;
3. **annee** : entier.

Exemple 5.2 :

Représentons le NIP, le nom, le prénom, la date de naissance et la note obtenue en informatique d'un étudiant par un enregistrement à cinq champs :

1. **nip** : entier ;
2. **nom** : chaîne de caractères ;
3. **prénom** : chaîne de caractères ;
4. **date de naissance** : date ;
5. **note d'informatique** : flottant.

Cet exemple montre un enregistrement dont l'un des champs (**date**) est lui même un enregistrement.

Bon nombre de langages de programmation offre la possibilité de définir et manipuler des enregistrements. Dans les sections qui suivent nous présentons ce qu'il en est en CAML.

5.2 Déclaration d'un type enregistrement

Les enregistrements utilisant des noms pour leurs composantes, il est nécessaire d'en faire une déclaration préalable par la déclaration d'un type.

C'est pour nous la première occasion de rencontrer une nouvelle sorte de phrase en CAML : la *déclaration de type*. Une déclaration de type en CAML s'écrit en commençant par le mot-clé **type** suivi du nom donné au type déclaré suivi de l'expression de sa définition.

Syntaxe : Forme générale de la déclaration d'un type

```
type nom_type = définition
```

La déclaration d'un type enregistrement suit ce schéma, l'expression de définition étant une énumération des composantes accompagnées de leur type séparées par des points-virgules, le tout étant entre accolades.

Syntaxe : Déclaration d'un type enregistrement

```
type nom_type =
{ nom1 : t1 ;
  nom2 : t2 ;
  .
  .
  .
  nomn : tn
}
```

où $n \geq 1$ est le nombre de composantes, les nom_i sont les noms (respectant les mêmes règles que les identificateurs de variables) pour désigner les différentes composantes et les t_i des types quelconques.

Exemple 5.3 :

Voici en CAML la déclaration du type **date** :

```
# type date =
{ quantieme : int ; mois : int ; annee : int } ;;
type date = { quantieme : int; mois : int; annee : int; }
```

Comme on peut le voir, dans sa réponse l'interpréteur ne fait que répéter la déclaration du type pour signaler qu'il l'a bien comprise. (Il rajoute même un point-virgule après le dernier champ, ce qui montre qu'il est possible d'en mettre un à cette place.)

Exemple 5.4 :

La note obtenue en informatique est en fait un ensemble de 4 notes et d'un bonus. Nous déclarons le type **notes_apil** par un enregistrement à cinq champs :

```
type notes_apil =
{ td : float ;
  ds1 : float ;
  ds2 : float ;
  tp : float ;
  bonus : float
}
```

Exemple 5.5 :

Déclarons maintenant le type **etudiant** comme un enregistrement à cinq champs :

```
type etudiant =
{ nip : int ;
  nom : string ;
  prenom : string ;
  date_naiss : date ;
  notes_apil : notes_apil
}
```

Cet exemple montre qu'un champ d'un enregistrement peut-être lui-même d'un type enregistrement. C'est le cas de deux d'entre eux : **date_naiss** et **notes_apil**. On peut même noter qu'un identificateur de champ peut porter le même nom qu'un identificateur de type.

Bien entendu, pour être valide, cette déclaration de type doit impérativement être précédée par la déclaration des types **date** et **notes_apil**.

5.3 Création d'un enregistrement

À partir du moment où un type enregistrement est déclaré, il devient possible de créer des enregistrements de ce type. L'expression permettant de contruire une valeur d'un type enregistrement est tout simplement l'énumération des noms de champs accompagnés de leurs valeurs, séparés par des points-virgules, le tout encadré par des accolades.

Syntaxe : Création d'une valeur enregistrement

```
{ nom1 = expr1 ;
  nom2 = expr2 ;
  .
  .
  .
  nomn = exprn
}
```

où les $expr_i$ sont des expressions dont la valeur est du type t_i du champ correspondant. Cette syntaxe est donc tout à fait semblable à celle de la définition d'un type enregistrement : on remplace simplement les deux points (:) par le symbole =, et les type t_i par des expressions du même type.

Exemple 5.6 :

Voici la création d'une valeur du type **date** avec le 14 juillet 1789.

```
# {quantieme = 14; mois = 7 ; annee = 1789} ;;
- : date = {quantieme = 14; mois = 7; annee = 1789}
```

L'indication de type fourni par l'interpréteur montre qu'il a reconnu dans notre expression une valeur de type **date**. Ce sont bien évidemment les noms des champs qui ont permis cette reconnaissance.

Bien entendu, comme pour tout autre type de valeurs, cette valeur peut être attribuée à une variable (locale ou non) ou à un paramètre de fonction.

```
# let prise_bastille1 = {quantieme = 14; mois = 7; annee = 1789} ;;
val prise_bastille1 : date = {quantieme = 14; mois = 7; annee = 1789}
```

Dans la création d'une valeur enregistrement, il n'est pas indispensable que l'ordre des composantes soit le même que celui de la définition du type.

Exemple 5.7 :

Création de la même date avec un ordre différent des composantes qui serait celui utilisé par nos voisins d'outre manche.

```
# let prise_bastille2 = {annee=1789; mois = 7; quantieme = 14} ;;
val prise_bastille2 : date = {quantieme = 14; mois = 7; annee = 1789}
```



En revanche, il est indispensable de mentionner la totalité des composantes lors de la création.

Exemple 5.8 :

```
# let prise_bastille3 = {annee=1789} ;;
Some record field labels are undefined: quantieme mois
```

Le retour de l'interpréteur signale que l'expression donnant la valeur à la variable **prise_bastille3** a été refusée parce qu'il manque deux champs. En conséquence, cette variable n'est pas déclarée.

```
# prise_bastille3 ;;
Unbound value prise_bastille3
```

Exemple 5.9 :

Et voici maintenant comment construire une valeur de type **etudiant** dont certains champs sont eux-mêmes des enregistrements.

```
# let etudiant1 = {
  nip = 11009999 ;
  nom = "Calbuth" ;
  prenom = "Raymond" ;
  date_naiss = {quantieme = 1 ; mois = 1 ; annee = 1984} ;
  notes_apil = {td = 10. ; ds1 = 10. ; ds2 = 8. ; tp = 10. ; bonus = 0.}
} ;;
val etudiant1 : etudiant =
{nip = 11009999; nom = "Calbuth"; prenom = "Raymond";
 date_naiss = {quantieme = 1; mois = 1; annee = 1984};
 notes_apil = {td = 10.; ds1 = 10.; ds2 = 8.; tp = 10.; bonus = 0.}}
```

⚠ S'il est clair que deux champs d'un même type enregistrement ne peuvent pas porter le même nom, qu'en est-il de deux champs de deux types enregistrements différents ?

La réponse diffère selon les langages de programmation. C'est tout à fait possible pour des langages comme ADA, C ou PASCAL. C'est possible aussi en CAML, cependant la dernière déclaration du type enregistrement masquera le nom du champ partagé avec le type enregistrement précédemment déclaré, et interdira donc toute création d'un enregistrement du premier type déclaré.

```
# type aaa = {a : int ; b : string} ;;
type aaa = { a : int; b : string; }
# type ccc = {c : int ; b : string} ;;
type ccc = { c : int; b : string; }
# {a=12; b="timoleon"} ;;
The record field label b belongs to the type ccc
but is here mixed with labels of type aaa
```

Il est donc prudent, si ce n'est indispensable, d'éviter des noms de champs identiques dans deux types enregistrement. Ces conflits de noms peuvent être évités grâce à la programmation modulaire qui sera abordée dans le cours d'API2 en deuxième année.

5.4 Accès aux composantes

L'accès aux divers champs d'un enregistrement peut se faire par filtrage, comme pour les n -uplets, ou bien en exploitant le nommage des champs par une notation pointée.

5.4.1 Filtrage

Le filtrage est l'un des deux moyens d'accéder à une composante d'un enregistrement. Les filtres permettant de filtrer des enregistrements suivent la même syntaxe que celle de la construction des enregistrements, à l'exception des deux points suivants :

1. l'expression à droite du symbole = qui suit le nom d'un champ doit être un filtre (cf page 16);
2. tous les champs ne doivent pas nécessairement être précisés (il en faut cependant au moins un).

Exemple 5.10 :

On peut tester si une date est celle du premier jour de l'an.

```
let est_nouvelle_annee d =
  match d with
  | {quantieme = 1 ; mois = 1 ; annee = _} -> true
  | _ -> false
```

Le premier filtre est bien évidemment un filtre pour les valeurs du type date. Il est satisfait si le quantième vaut 1, ainsi que le mois, et si l'année est quelconque (filtre universel `_`).

```
# est_nouvelle_annee prise_bastille1 ;;
- : bool = false
# est_nouvelle_annee {quantieme = 1 ; mois = 1 ; annee = 1984} ;;
- : bool = true
```

Comme indiqué plus haut, un filtre d'enregistrement ne doit pas préciser tous les champs. Certains d'entre eux peuvent être omis.

Exemple 5.11 :

Voici une autre formulation du filtrage de la fonction définie dans l'exemple 5.10.

```
let est_nouvelle_annee d =
  match d with
  | {quantieme = 1 ; mois = 1} -> true
  | _ -> false
```

Le champ `annee` qui était filtré par le motif universel dans l'exemple précédent, ne figure plus ici.

Exemple 5.12 :

Pour terminer, voici comment filtrer les étudiants nés une certaine année.

```
(*
  fonction : ne_en : etudiant -> int -> bool
  parametres :
    e : etudiant
    a : int
  valeur renvoyee : bool =
    Vrai si l'etudiant e est ne l'annee a
    Faux sinon
  CU : aucune
*)
let ne_en e a =
  match e with
  | {date_naiss = {annee = a'}} -> (a=a')
```

```
# ne_en etudiant1 1985 ;;
- : bool = false
# ne_en etudiant1 1984 ;;
- : bool = true
```

5.4.2 Notation pointée

La seconde façon d'accéder aux composantes d'un enregistrement consiste à utiliser leurs noms.

Pour accéder à la composante *nom* d'un enregistrement *e*, on écrit *e.n*, cette expression ayant le type donné au champ *n* lors de la déclaration du type enregistrement.

Syntaxe : Notation pointée pour accéder à une composante *n* d'un enregistrement *e*

e.n

Exemple 5.13 :

Voici l'accès aux trois champs de la variable définie dans l'exemple ??

```
# prise_bastille1.quantieme ;;
- : int = 14
# prise_bastille1.mois ;;
- : int = 7
# prise_bastille1.annee ;;
- : int = 1789
```

Exemple 5.14 :

Idem avec la variable définie dans l'exemple 5.9.

```
# etudiant1.nom ;;
- : string = "Calbuth"
# etudiant1.date_naiss ;;
- : date = {quantieme = 1; mois = 1; annee = 1984}

# etudiant1.date_naiss.annee ;;
- : int = 1984
```

5.5 Enregistrement à champs mutables

Avec les déclarations des types enregistrement que nous avons fait jusqu'à présent, les composantes d'un enregistrement ne sont pas mutables, autrement dit on ne peut pas modifier leur valeur.

⚠ On pourrait être tenté d'utiliser la même instruction que pour changer la valeur d'un élément tableau (ou d'un caractère d'une chaîne), mais cela n'est pas possible.

Exemple 5.15 :

Si la note de ds2 d'informatique de notre étudiant Raymond Calbuth change (suite à une erreur de notation) et devient 10 au lieu de 8, on ne peut pas changer le champ adéquat avec l'instruction <-.

```
# etudiant1.notes_api1.ds2 <- 10. ;;
The record field label ds2 is not mutable
```

Le message d'erreur obtenu indique clairement que le champ `ds2` (du type `notes_apil`) n'est pas mutable.

On pourrait s'en sortir en construisant une nouvelle valeur de type `etudiant` dont tous les champs sont identiques sauf pour le `ds2`. Mais cela serait vraiment très lourd, et ne correspondrait pas à l'idée de mutabilité.

Si on désire permettre à certains champs d'un enregistrement de changer de valeur, il faut que ces champs soient déclarés mutables lors de la déclaration du type. Pour ce faire on utilise le mot-clé **mutable** placé devant le nom des champs qui doivent l'être.

Syntaxe : Déclaration d'un champ mutable dans un enregistrement

```
...
mutable nom : t ;
...
```

Exemple 5.16 :

A priori les cinq composantes d'un enregistrement de type `notes_apil` sont susceptibles de pouvoir changer. Par conséquent les cinq champs sont mutables.

```
type notes_apil =
{ mutable td : float ;
  mutable ds1 : float ;
  mutable ds2 : float ;
  mutable tp : float ;
  mutable bonus : float
}
```

En revanche, les cinq champs du type `etudiant` n'ont aucune raison d'être mutables : on ne change pas de NIP, ni de nom, ni de prénom, ni de date de naissance. Pour le champ `notes_apil`, il n'a pas à changer non plus. Ce sont les cinq champs qui le composent qui sont mutables. Autrement dit la déclaration du type `etudiant` reste identique à celle donnée dans l'exemple 5.5, mais elle doit bien entendu être faite après celle du type `notes_apil` avec champs mutables.

La création d'un enregistrement avec champs mutables se fait exactement de la même façon qu'un enregistrement à champs non mutables.

Exemple 5.17 :

```
# let etudiant1 = {
  nip = 11009999 ;
  nom = "Calbuth" ;
  prenom = "Raymond" ;
  date_naiss = {quantieme = 1 ; mois = 1 ; annee = 1984} ;
  notes_apil = {td = 10. ; ds1 = 10. ; ds2 = 8. ; tp = 10. ; bonus = 0.}
} ;;

val etudiant1 : etudiant =
{nip = 11009999; nom = "Calbuth"; prenom = "Raymond";
 date_naiss = {quantieme = 1; mois = 1; annee = 1984};
 notes_apil = {td = 10.; ds1 = 10.; ds2 = 8.; tp = 10.; bonus = 0.}}
```


Dans la réponse de l'interpréteur, on ne voit aucune mention que tel ou tel champ est mutable ou non. Cette réponse est absolument identique à celle obtenue dans l'exemple 5.9.

Exemple 5.18 :

```
# etudiant1 ;;
- : etudiant =
{nip = 11009999; nom = "Calbuth"; prenom = "Raymond";
 date_naiss = {quantieme = 1; mois = 1; annee = 1984};
 notes_apil = {td = 10.; ds1 = 10.; ds2 = 10.; tp = 10.; bonus = 0.}}
```

5.5.1 Retour sur les variables mutables

Les variables mutables que nous utilisons déjà ont un type enregistrement comme nous pouvons nous en convaincre en analysant les indications de l'interpréteur.

Exemple 5.19 :

```
# let n = ref 0 ;;
val n : int ref = {contents = 0}
# let x = ref 1.0 ;;
val x : float ref = {contents = 1.}
# let c = ref 'A' ;;
val c : char ref = {contents = 'A'}
```

Le type de la variable **n** est **int ref** et sa valeur **{contents = 0}**. Le type de la variable **x** est **float ref** et sa valeur **{contents = 1.}**. Le type de la variable **c** est **char ref** et sa valeur **{contents = 'A'}**.

Les variables mutables sont d'un type enregistrement nommé **ref** et paramétré par un autre type (**int**, **float**, **char**, ...). Ces enregistrement n'ont qu'un seul champ nommé **contents**.

Exemple 5.20 :

```
# n.contents ;;
- : int = 0
# x.contents ;;
- : float = 1.
# c.contents ;;
- : char = 'A'
```

Ce champ **contents** est mutable.

Exemple 5.21 :

```
n.contents <- 2 ;;
- : unit = ()
# n ;;
- : int ref = {contents = 2}
```

```
# x.contents <- 2. *. x.contents ;;
- : unit = ()
# x ;;
- : float ref = {contents = 2.}
```

Les valeurs mutables sont donc des enregistrements à un seul champ mutable nommé **contents**. Le programmeur peut définir ses propres types de valeurs mutables. Voici par exemple comment on peut (re)programmer le type **ref** de CAML.

```
type 'a reference = {mutable contenu : 'a}


let reference y = {contenu = y}

let ( ! ) x = x.contenu

let ( := ) x y = x.contenu <- y
```

Et voici un exemple d'utilisation :

```
# let n = reference 0 ;;
val n : int reference = {contenu = 0}
# let x = reference 1.0 ;;
val x : float reference = {contenu = 1.}
# let c = reference 'A' ;;
val c : char reference = {contenu = 'A'}
# !n ;;
- : int = 0
# n := !n + 1;;
- : unit = ()
# n ;;
- : int reference = {contenu = 1}
# c := char_of_int (int_of_char !c + 1) ;;
- : unit = ()
# c ;;
- : char reference = {contenu = 'B'}
```


 Après avoir défini le type **reference** et les opérateurs de déréférencement (!) et d'affectation (:=), il n'est plus possible de les utiliser sur les variables de type **ref**.

```
# let p = ref 1 ;;
val p : int ref = {contents = 1}
# !p ;;
This expression has type int ref but is here used with type 'a reference
# p := 2 ;;
This expression has type int ref but is here used with type 'a reference
```

5.6 Un exemple complet

```
type complexe = {re : float ; im : float}
```

```
(* quelques constantes complexes *)
let zero = {re = 0. ; im = 0.}
let un = {re = 1. ; im = 0.}
let i = {re = 0. ; im = 1.}
```

 Le choix du nom de la constante complexe i n'est peut être pas très judicieux. Il interdit en effet toute utilisation ultérieure de ce nom pour désigner d'autres valeurs, en particulier en tant qu'indice d'une boucle pour.

On reviendra sur ce point dans un chapitre sur la programmation modulaire du cours d'API2 en deuxième année.

```
(* Conversions reel -> complexe *)
let complexe_of_float x = {re = x ; im = 0.}

let complexe_of_alg x y = {re = x ; im = y}
```

Et maintenant les fonctions donnant le module au carré et le module d'un nombre complexe $z = x + iy$ (le nom **module** étant un mot-clé du langage, il n'est pas possible de donner ce nom à la fonction que nous voulons réaliser, aussi l'avons-nous nommée **norme**) :

$$|z|^2 = x^2 + y^2$$

$$|z| = \sqrt{x^2 + y^2}$$

```
let norme_carree z =
  z.re *. z.re +. z.im *. z.im
let norme z =
  sqrt (norme_carree z)
```

Enfin les opérations arithmétiques de base :

$$z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$$

$$z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2)$$

$$z_1 \times z_2 = (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + x_2 y_1)$$

$$z^{-1} = \frac{\bar{z}}{|z|^2}$$

$$\frac{z_1}{z_2} = z_1 \times z_2^{-1}$$

```
(* operations arithmetiques *)
let add z1 z2 =
  {re = z1.re +. z2.re ;
   im = z1.im +. z2.im}

let sub z1 z2 =
  {re = z1.re -. z2.re ;
   im = z1.im -. z2.im}
```

```

let mult z1 z2 =
  {re = z1.re *. z2.re -. z1.im *. z2.im ;
   im = z1.re *. z2.im +. z1.im *. z2.re}

let inv z =
  let r2 = norme_carree z in
  if r2 = 0. then
    failwith "inv_:_complexe_nul"
  else
    {re = z.re /. r2 ;
     im = -. z.im /.r2}

let div z1 z2 =
  mult z1 (inv z2)

```

5.7 Exercices

Exercice 5-1

Dans cet exercice le type `toto` est défini par

```
type toto = {a : int ; b : bool}
```

Question 1 Déclarez une variable `var` de type `toto` dont le champ `a` vaut 1 et le champ `b` vaut `false`.

Question 2 Quelle expression permet d'obtenir la valeur du champ `a` de la variable `var`? Donnez deux solutions.

Dans la suite, on suppose que `t` est un tableau de `toto` (`toto array`).

Question 3 Comment avoir la valeur du champ `b` de l'élément d'indice `i` du tableau `t`?

Question 4 Calculez l'entier `n` égal à la somme des champs `a` de tous les enregistrements du tableau `t` dont le champ `b` a la valeur `true`.

Exercice 5-2 *Fiche d'élèves*

Dans sa classe, un professeur des écoles, fabrique des fiches comportant deux informations, l'une est le prénom de l'élève, l'autre est la taille de celui-ci. On considère qu'il n'y a pas deux élèves avec le même prénom... La taille est exprimée grâce à un nombre réel en mètre.

Question 1 Déclarez le type `fiche` en CAML.

Question 2 Zoé mesure 1m12 et Arthur mesure 1m05. Déclarez en CAML les variables `zoe` et `arthur` contenant les fiches de ces deux élèves.

Question 3 La fiche d'un élève est dans la variable `b`. Donnez une expression donnant le prénom de cet élève. Donnez une expression donnant sa taille.

Question 4 Une variable nommée `classe` de type `fiche array` contient l'ensemble des fiches d'une classe. Construisez une expression pour calculer la taille moyenne des élèves de cette classe.

Question 5 Pendant l'année scolaire, les enfants grandissent. Comment aurait-il fallu déclarer le

type `fiche` pour tenir compte de ce fait ?

Question 6 En supposant que la fiche `zoe` de Zoé soit de ce nouveau type `fiche`, donnez une instruction permettant de mettre à jour la fiche `zoe` sachant que Zoe a grandi d'un centimètre.

Question 7 On dit que deux élèves sont de tailles voisines, si leurs tailles diffèrent d'au plus deux centimètres. Réaliser le prédicat `sont_de_tailles_voisines`.

Exercice 5-3 *Gestion de notes*

Pour gérer les notes des étudiants suivant le semestre 2 de la licence, on définit le type suivant :

```
type etudiant = {
  nip : int ;
  nom : string ;
  prenom : string ;
  notes : float array ;
  mutable semestre_valide : bool
}
```

Le champ `notes` de ce type est un tableau de 6 nombres (`float`) correspondant aux notes des 6 UE que tous les étudiants suivent ce semestre. Ces 6 UE sont les mêmes pour tous les étudiants et sont, dans l'ordre croissant des indices du tableau de notes, APII, Maths, Physique, TEC, Anglais, Sport.

Au début du semestre 2 de licence, on suppose qu'une variable nommée `etudiants_s2` de type `etudiant array` a été créée et contient l'ensemble des étudiants de ce semestre. Bien entendu aucune note n'a encore été attribuée (elles sont toutes égales à la note fictive `-1`), et aucun étudiant n'a validé son semestre. À titre d'exemple, voici la fiche de l'étudiant Raymond Calbuth en début de semestre.

```
{ nip = 11009999 ;
  nom = "Calbuth" ;
  prenom = "Raymond" ;
  notes = [| -1.0 ; -1.0; -1.0; -1.0 ; -1.0; -1.0 |] ;
  semestre_valide = false
}
```

Question 1 Réalisez une procédure qui à partir d'un tableau de notes (`float array`) et une UE passés en paramètre modifie chacune des fiches des étudiants en y reportant sa note. Vous pouvez faire l'hypothèse que les notes contenues dans le tableau de notes, et les étudiants dans le tableau `etudiants_s2` sont dans le même ordre.

Question 2 On suppose que les notes de toutes les UE ont été reportées, et que tous les étudiants ont eu une note dans toutes les UE (il ne reste plus aucune note fictive `-1`).

Modifiez le champ `semestre_valide` des étudiants contenu dans `etudiants_s2` en lui attribuant la valeur `true` si la moyenne des 6 UE de l'étudiant est supérieure ou égale à 10.

Question 3 Réalisez maintenant une procédure non paramétrée qui affiche la liste des étudiants ayant validé leur semestre rangés par ordre alphabétique des noms, et en cas d'égalité de nom par ordre alphabétique des prénoms, et en cas d'égalité de nom et de prénom (et oui cela arrive!) par ordre numérique du NIP. L'affichage se fera à raison d'un étudiant par ligne sous la forme

```
NIP  NOM  PRENOM
```

Attention, les fiches d'étudiants dans le tableau `etudiants_s2` ne sont pas nécessairement rangées dans l'ordre. De plus votre procédure ne doit pas modifier ce tableau (il faut donc en trier une copie).

Chapitre 6

Les fichiers

6.1 Besoin des fichiers

Les chapitres précédents ont permis de découvrir comment on peut représenter des données complexes. Cependant, pour l’instant, rien – à part éventuellement la redirection des sorties – ne permet de stocker les données complexes que le programme a produit. Et surtout rien ne permet une réutilisation facile des données produites.

Les fichiers vont permettre de conserver des données et de les réutiliser facilement.

Les données sont écrites les unes à la suite des autres. C’est pourquoi, on parle de fichier *séquentiel*. L’ordre dans lequel sont rangées les données est donc très important.

6.2 Entrées et Sorties

En CAML, comme dans beaucoup d’autres langages, on désigne par *entrées* les données lues dans un fichier et qui sont traitées par le programme, et par *sorties* les données qui ont été produites et qui sont écrites dans un fichier. Cette distinction est assez marquée en CAML. Il existe en effet deux types nommés respectivement `in_channel` respectivement `out_channel` qui permettent respectivement

- de représenter un canal de lecture depuis un fichier ;
- de représenter un canal d’écriture vers un fichier.

Par conséquent, il existe des procédures et des fonctions spécifiques de manipulation de fichiers pour la lecture et d’autres fonctions et procédures pour l’écriture.

Les variables de type `in_channel` et `out_channel` contiennent les informations qui permettent de décrire l’état des fichiers. L’utilisateur n’a pas besoin de savoir comment ces données sont effectivement représentées. C’est pourquoi le type est *abstrait*. Cela se voit lorsqu’on utilise une valeur de ce type. L’interprète caml répond quelque chose du genre :

```
|| val fi : in_channel = <abstr>
```

6.3 Les trois phases de traitements sur les fichiers

Une lecture ou une écriture dans un fichier se déroule en trois phases :

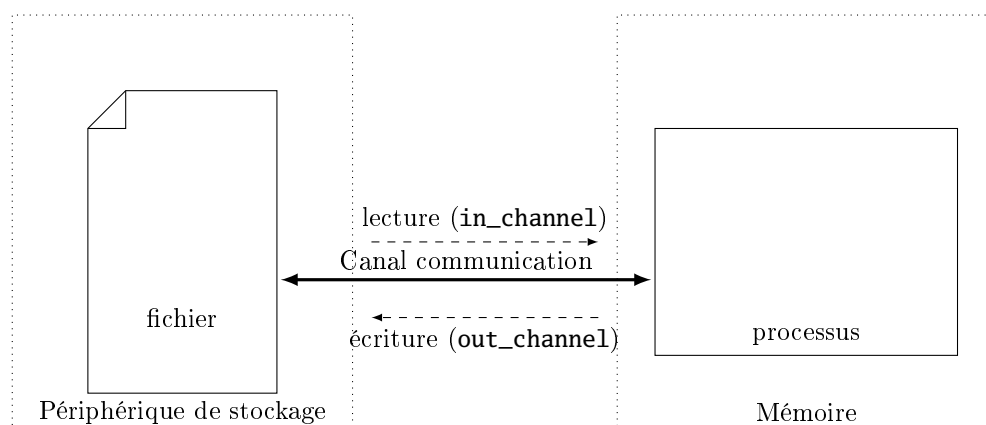


FIGURE 6.1 – Canal de communication entre processus et fichier

Lecture	Écriture
<code>open_in</code>	<code>open_out</code>
<code>input_value</code>	<code>output_value</code>
<code>input_line</code>	<code>output_string</code>
<code>input_char</code>	<code>output_char</code>
<code>input_byte</code>	<code>output_byte</code>
<code>close_in</code>	<code>close_out</code>

TABLE 6.1 – Fonctions et procédures de lecture et écriture

1. On *ouvre* le fichier. Dans cette phase, on doit indiquer le nom du fichier. CAML renvoie une variable de type `in_channel` ou `out_channel` selon que le fichier est ouvert en lecture ou en écriture;
2. on *lit* ou on *écrit* le fichier;
3. on *ferme* le fichier.

6.4 Ouverture d'un fichier

Deux fonctions permettent d'ouvrir des fichiers. Elles sont toutes les deux paramétrées par le nom du fichier.

```
# open_out;;
- : string -> out_channel = <fun>
# open_in;;
- : string -> in_channel = <fun>
#
```

Pour ouvrir un fichier, il suffit de passer son nom (complété éventuellement par le chemin d'accès) en paramètre.

par exemple :

Exemple 6.1 :


```
let source = open_in "titi.txt";;
let destination = open_out "toto.txt";;
```

6.5 Lecture et écriture

Il existe plusieurs fonctions de lecture et d'écriture. On choisit celle qu'on doit utiliser en fonction des besoins. Toutefois il est déconseillé de panacher les appels. Par exemple, on doit éviter de faire un `input_byte` après un `input_value`.

Les fonctions de lecture et d'écriture ont *des effets de bord*, c'est-à-dire qu'elles modifient l'environnement. Deux appels consécutifs à la fonction `input_byte` avec le même paramètre, ne donneront pas nécessairement le même résultat.

Exemple 6.2 :

On considère le fichier `titi.txt` dont la représentation du contenu par un afficheur hexadécimal est la suivante :

```
00000000  41 50 49 31 0a 0a                                |API1..|
00000006
```

```
# let source = open_in "titi.txt";;
val source : in_channel = <abstr>
# input_byte source;;
- : int = 65
# input_byte source;;
- : int = 80
# close_in source;;
- : unit = ()
#
```

On note que $(41)_{16} = (65)_{10}$ et $(50)_{16} = (80)_{10}$.

6.5.1 Lecture ou écriture d'un octet

On rappelle que *octet* en anglais se dit *byte* qu'il ne faut pas confondre avec *bit* qui est la contraction de *binary digit* et qui désigne une valeur 0 ou 1.

les fonctions `output_byte` et `input_byte` permettent de lire et d'écrire des données dans un fichier binaire. On dit parfois que ces fonctions sont de *bas niveau* car elles ne permettent de manipuler que des données brutes sous forme octet, ce qui est au plus proche de la définition d'un fichier en tant que suite finie d'octets. On peut donc manipuler toute sorte de fichiers grâce à ces fonctions. Mais par exemple lire un tableau de nombre réels en n'utilisant que `input_byte` est loin d'être immédiat !

- la fonction `input_byte` a pour type `in_channel → int`;
- la fonction `output_byte` a pour type `out_channel → int → unit`.

Lors de l'écriture, si l'entier passé en paramètre est en dehors de l'intervalle $0..255$, alors le paramètre est réduit modulo 256.

Exemple 6.3 :

```
# let destination = "tyty.txt";;
val destination : string = "tyty.txt"
# let fd = open_out destination;;
val fd : out_channel = <abstr>
# output_byte fd 513;;
- : unit = ()
# close_out fd;;
- : unit = ()
# let source = open_in "tyty.txt";;
val source : in_channel = <abstr>
# input_byte source;;
- : int = 1
# close_in source;;
- : unit = ()
```

6.5.2 Lecture ou écriture d'un caractère

Il s'agit de fonctions un peu plus évoluées, qui sont principalement destinées au traitement de fichiers texte. Toutefois, on préfère généralement utiliser des fonctions d'un peu plus au niveau encore, qui permettent de lire et d'écrire une ligne entière d'un fichier texte.

- la fonction `input_char` a pour type `in_channel → char`;
- la fonction `output_char` a pour type `out_channel → char → unit`.

On pourrait croire que la lecture d'un caractère est juste la composition de `char_of_int` et de `input_byte`. Ce n'est toutefois pas exact. En effet cela dépend du codage utilisé. Dans certains codages, comme UTF-8, il existe des caractères qui sont codés sur plusieurs octets.

Exemple 6.4 :

```
# let nom_sortie = "contrepeterie.txt";;
val nom_sortie : string = "contrepeterie.txt"
# let fo = open_out nom_sortie;;
val fo : out_channel = <abstr>
# let texte="La_gardienne_du_campus_dessine";;
val texte : string = "La_gardienne_du_campus_dessine"
# for i =0 to (String.length(texte))-1 do
  output_char fo texte.[i]
done;;
- : unit = ()
# close_out fo;;
- : unit = ()
```

Dans la console, on peut regarder le contenu du fichier avec un afficheur hexadécimal :

```
~/API1/Poly$ hexdump -C contrepeterie.txt
00000000  4c 61 20 67 61 72 64 69  65 6e 6e 65 20 64 75 20  |La gardienne du |
00000010  63 61 6d 70 75 73 20 64  65 73 73 69 6e 65       |campus dessine|
0000001e
~/API1/Poly$
```

6.5.3 Lecture ou écriture d'une chaîne de caractères

La fonction `input_line` permet de lire une ligne d'un coup dans un fichier texte. La marque de fin de ligne est consommée dans le fichier, mais ne fait pas partie de la chaîne renvoyée par la fonction de lecture. Cette fonction est très pratique car la représentation de la marque de fin de ligne varie d'un système d'exploitation à l'autre. Cette fonction permet donc d'écrire un code portable d'un système à l'autre, sans avoir à se préoccuper de ce genre de détail.

Pour écrire dans le fichier, il existe la fonction `output_string`. Si on souhaite terminer une ligne, il suffit d'envoyer une marque de fin de ligne. (cela peut se coder par `"\n"`)

- la fonction `input_line` a pour type `in_channel → string`;
- la fonction `output_string` a pour type `out_channel → string → unit`.

Exemple 6.5 :

Voici le contenu du fichier `texte_exemple`

```
Voici
un
texte
de
quelques lignes
pour illustrer les
fonctions d'entrees
sortie en Caml
```

Voici une petite procédure qui affiche les 5 premières lignes de d'un fichier dont le nom est passé en paramètre. Une contrainte d'utilisation de cette fonction est que le fichier contient au moins cinq lignes.

```
# let afficheur_numerote s =
  let fi = open_in s in
  for i = 1 to 5 do
    let s = input_line fi in
    print_endline (string_of_int(i)^":"^s)
  done;
  close_in fi;;
val afficheur_numerote : string -> unit = <fun>
# afficheur_numerote "texte_exemple";;
1:Voici
2:un
3:texte
4:de
5:quelques lignes
- : unit = ()
#
```

6.5.4 Lecture d'un élément

Les fonctions `input_value` et `output_value` sont les fonctions d'entrées et sorties de plus haut niveau. Elles permettent l'écriture et la lecture de toute sorte de données. Toutefois dans les implantations actuelles, la fonction de lecture a pour inconvénient de ne pas être sûre dans le sens suivant : si on essaie de lire dans un fichier un élément d'un type et que le fichier contient un

élément d'un autre type, tout peut arriver. Le programme peut continuer avec une valeur fantaisiste, ou bien s'arrêter brutalement suite à une erreur système... Il est donc de la responsabilité du programmeur de savoir ce qu'il va lire dans le fichier.

- la fonction `input_value` a pour type `in_channel → 'a`;
- la fonction `output_value` a pour type `out_channel → 'a → unit`.

Le type du résultat de `input_value` pose un problème. Un élément ne peut pas avoir un type arbitraire. Il faut donc que lors de la lecture, le programmeur impose une contrainte de type. On peut par exemple déclarer préalablement une variable mutable avec le type de donnée qu'on souhaite lire, puis faire la lecture en affectant le résultat à la variable mutable.

Exemple 6.6 :

```
# let toto = [| 3; 1 ; 4; 1; 5; 9 |];;
val toto : int array = [|3; 1; 4; 1; 5; 9|]
# let canalsortie = open_out "exemple";;
val canalsortie : out_channel = <abstr>
# let toto = [| 3; 1 ; 4; 1; 5; 9 |];;
val toto : int array = [|3; 1; 4; 1; 5; 9|]
# output_value canalsortie toto;;
- : unit = ()
# close_out canalsortie;;
- : unit = ()
# let titi = ref [| 1 |];;
val titi : int array ref = {contents = [|1|]}
# let canalentree = open_in "exemple";;
val canalentree : in_channel = <abstr>
# titi := input_value canalentree;;
- : unit = ()
# close_in canalentree;;
- : unit = ()
# !titi;;
- : int array = [|3; 1; 4; 1; 5; 9|]
#
```

La manière dont les données sont représentées dans le fichier dépend de la version de CAML. Cela a au moins deux conséquences.

- le format n'est pas aussi simple que du texte, cela peut handicaper l'interopérabilité;
- deux versions différentes ne produiront peut être pas le même fichier, et ainsi des données produites par un programme compilé pour la version 3.10 ne seront peut être pas lisibles par le même programme compilé avec la version 3.11.

6.6 Fin de fichier

Il n'y a pas de fonction qui permette de tester facilement, si on a atteint la fin de fichier lors d'une lecture. Si on essaie de lire un élément dans un fichier et qu'on a préalablement déjà atteint la fin du fichier une exception `End_of_file` se produit. Dans le paragraphe suivant, on montre comment on peut récupérer une exception.

Exemple 6.7 :

```
#let fo= open_out "tutu";;
val fo : out_channel = <abstr>
# output_byte fo 33;;
- : unit = ()
# close_out fo;;
- : unit = ()
# let fi= open_in "tutu";;
val fi : in_channel = <abstr>
# input_byte fi;;
- : int = 33
# input_byte fi;;
Exception: End_of_file.
# close_in fi;;
- : unit = ()
```

6.7 Retour sur les exceptions

On sait déclencher des exceptions. Si on laisse l'exception se propager, le programme s'arrête brutalement. Mais il est parfois utile de tenter un calcul et dans le cas où ce calcul échoue prévoir une solution de rechange (un plan B?). Caml permet de faire cela grâce à l'expression de traitement exceptionnel **try ... with ...**. La syntaxe ressemble un peu à celle du **match**. Toutefois les filtres sont remplacés par des filtres d'exceptions.

La syntaxe générale d'une expression avec traitement exceptionnel est donnée ci-dessous.

Syntaxe : Expression **try ... with ...**

```
try expression with
  | exn1 -> expr1
  | exn2 -> expr2
  ...
  | exnn -> exprn
```

Dans cette syntaxe,

- *expression* est l'expression à évaluer et qui peut déclencher une exception;
- *expr*₁, *expr*₂, ..., *expr*_{*n*} sont des expressions qui doivent toutes être du même type que l'*expression*;
- *exn*₁, *exn*₂, ..., *exn*_{*n*} sont des *filtres d'exception*.

Un filtre d'exception commence par un constructeur d'exception suivi d'un filtre habituel.

Exemple 6.8 :

```
# let racine x =
  if x<0 then
    failwith "racine_: _negatif"
  else
    int_of_float (sqrt (float_of_int x));;
val racine : int -> int = <fun>
# let fonction x =
  try
```

```

    print_int (1/(racine x));
    print_newline()
  with
    | Division_by_zero -> print_string "division_par_0\n"
    | Failure s -> print_endline s;;
val fonction : int -> unit = <fun>
# 1/racine 0;;
Exception: Division_by_zero.
# 1/racine (-1);;
Exception: Failure "racine_:negatif".
# 1/racine 1;;
- : int = 1
# fonction 0;;
division par 0
- : unit = ()
# fonction (-1);;
racine : negatif
- : unit = ()
# fonction 1;;
1
- : unit = ()

```

C'est de cette manière qu'on va éviter qu'un programme s'arrête brutalement lorsque la fin de fichier est atteinte.

Exemple 6.9 :

```

let afficheur_numerote s =
  let fi = open_in s in
  let i = ref 1 in
  try
    while true do
      let s = input_line fi in
      print_endline (string_of_int(!i)^":"^s);
      i := !i + 1
    done
  with
  | End_of_file ->
    ();
    close_in fi

```

6.8 Forcer les écritures en suspens

L'instruction `flush` permet de demander que soient effectuées les opérations d'écriture qui étaient en suspens pour un canal de sortie passé en paramètre.

L'instruction `flush_all` permet de faire cela sur tous les fichiers ouverts en sortie. Cette instruction ignore les erreurs éventuelles.

6.9 Trois fichiers particuliers

Il existe trois fichiers particuliers :

- l'entrée standard ;
- la sortie standard ;
- la sortie erreur.

La première, nommée **stdin**, sert d'entrée par défaut, par conséquent elle est de type **in_channel**. Il n'est pas nécessaire de l'ouvrir, on doit considérer qu'elle est toujours ouverte par défaut ;

La seconde, nommée **stdout**, – de type **out_channel** – est la sortie par défaut ;

La troisième, nommée **stderr**, – de type **out_channel** également – est la sortie utilisée pour afficher les message d'erreur.

Exemple 6.10 :

```
# output_string stdout "\nBlabla\n";;

Blabla
- : unit = ()
# let a=input_line stdin ;;
fjdkfj
val a : string = "fjdkfj"
# output_string stderr "\nBlabla\n";;
- : unit = ()
# flush stderr;;

Blabla
- : unit = ()
```

6.10 Quelques messages

Dans cette section on liste en vrac quelques messages exceptionnels qui peuvent se produire. Ces messages sont assez courant mais cette liste n'est pas exhaustive.

Fichier inexistant

```
let source = open_in "titi.txt";;
# Exception: Sys_error "titi.txt:_No_such_file_or_directory".
```

lecture sur un descripteur de fichier fermé.

```
(* ... *)
# close_in canalentree;;
- : unit = ()
# titi := input_value canalentree;;
Exception: Sys_error "Bad_file_descriptor".
```

lecture dans fichier dont la totalité des données a déjà été lue.

```
(* ... *)
titi := input_byte canalentree;;
Exception: End_of_file.
```

6.11 Exercices

6.11.1 Entrées et sorties de bas niveau, fichiers d'octets

Exercice 6-1 *Fichier d'octets*

On dispose d'un fichier d'octets nommé **note.data**. On suppose que le contenu du fichier est une suite d'octets non nuls, terminée par un octet 0.

Question 1 Déclarez une variable nommée **source** dont la valeur est un canal ouvert vers ce fichier.

Question 2 Donnez une expression CAML qui permet de calculer la somme de tous les octets lus dans le canal **source** jusqu'au premier octet nul rencontré.

Question 3 La lecture terminée que doit on faire ensuite ?

Question 4 On ne suppose plus que le fichier se termine par 0, mais on suppose qu'on connaît sa longueur : 1024 octets. Transformez votre boucle de lecture pour tenir compte de ce fait.

Question 5 On suppose maintenant qu'il n'y a pas d'octet signalant la fin, et qu'on ne connaît pas la longueur du fichier, comment faire pour calculer la somme de tous les octets du fichier ?

Exercice 6-2 *Numéroter les lignes d'un fichier texte*

Réalisez une fonction qui fait la copie d'un fichier texte dans un autre en rajoutant au début le numéro de chaque ligne. Cette fonction devra être du type

string → **string** → **unit**,

la chaîne passée en premier paramètre désignant le fichier à recopier, et celle donnée en second paramètre désignant le fichier à produire. Voici un exemple de ce qui est attendu.

Fichier de départ :

La cigale et la fourmi

La cigale ayant chanté
tout l'été
se trouva fort dépourvue
quand la bise fut venue

Fichier recopié :

1 La cigale et la fourmi
2
3 La cigale ayant chanté
4 tout l'été
5 se trouva fort dépourvue
6 quand la bise fut venue

Exercice 6-3 *Avancez masqué*

Voici une méthode pour chiffrer un fichier. On dispose d'un fichier d'octets nommé **masque.data** et d'un fichier **message.txt**. Alice et Bob sont les seuls à connaître le fichier **masque.data**. Alice souhaite envoyer à Bob le fichier **message.txt** mais elle sait que le canal de communication qu'elle va utiliser n'est pas sûr. C'est à dire que Charlie peut écouter. Elle va donc transformer son fichier en un fichier nommé **message_chiffre.data**, de la manière suivante :

- elle lit un octet *lu* dans le fichier **message.txt** ;
- elle lit un octet *msq* dans le fichier **masque.data** ;
- elle calcule le ou exclusif bit à bit entre les deux octets ; par exemple si $lu = (01000001)_2$ et si $msq = (10100110)_2$ alors $lu \oplus msq = (11100111)_2$. (remarque importante il y a un **opérateur** prédéfini en CAML nommé **lxor** qui fait le travail.) ;
- elle écrit le résultat dans **message_chiffre.data** ;
- et elle recommence jusqu'à ce qu'elle atteigne la fin du fichier **message.txt** ;
- lorsqu'elle atteint la fin du fichier **masque.data** elle recommence au début. Pour cela, on peut utiliser la fonction **seek_in**, de type

`in_channel → int → unit,`
`seek_in entree n` positionne le pointeur de lecture dans le canal `entree` sur l'octet numéro `n`, étant entendu que la numérotation débute à 0.

Question 1 Ecrire une procédure nommée **vigenere** paramétrée par 3 chaînes supposées distinctes qui sont respectivement le nom du fichier de message, le nom du fichier de masque et le nom du fichier qui va contenir le message chiffré fabriqué par la procédure. le texte de Blaise de Vigenere sur le site de la bibliothèque nationale de France :

<http://gallica.bnf.fr/ark:/12148/bpt6k73371g.planchecontact.f1>

Question 2 Alice veut envoyer à Bob un fichier nommé `poly.pdf`, elle décide de chiffrer le fichier avec le masque contenu dans le fichier `masque.dat` dont dispose aussi Bob, et de produire le fichier `poly_chiffre.dat` quelle va envoyer à Bob. Quelle commande doit elle taper ?

Question 3 Bob a reçu le fichier chiffré `poly_crypte.dat` que doit il faire pour retrouver le fichier initial? Remarques : le xor bit à bit
 - est associatif ce qui signifie

$$\forall x y z \quad (x \oplus y) \oplus z = x \oplus (y \oplus z)$$

- admet un élément neutre qui est 0 ce qui signifie

$$\forall x \quad (x \oplus 0) = 0 \oplus x = x$$

- tout élément et son propre symétrique,

$$\forall x \quad (x \oplus x) = 0$$

6.11.2 Entrées et sorties de haut niveau, fichiers de lignes de texte, fichiers contenant des structures

Exercice 6-4 *mot croisé*

Un mot croisé est présenté comme un fichier de ligne de texte.

- Sur la première ligne, il y a le nombre l de lignes de la grille;
- sur la seconde ligne, il y a le nombre c de colonnes de la grille;
- sur la troisième ligne, il y a le nombre n de cases noires;
- sur les $2n$ lignes suivantes, on a le numéro de ligne puis le numéro de colonne juste en dessous des cases noires;
- sur les l lignes suivantes, on a les définitions des mots horizontaux; il peut y avoir plusieurs mots sur une même ligne dans ce cas les définitions sont séparées par des barres verticales | (on suppose qu'aucune ne contient ce caractère);
- sur les c lignes suivantes, on a les définitions des mots verticaux; il peut y avoir plusieurs mots sur une même colonne dans ce cas les définitions sont séparées par des barres verticales | (on suppose qu'aucune ne contient ce caractère);
- sur les l lignes suivantes, on a les mots horizontaux, s'il y en a plusieurs sur la ligne on les sépare par des barres verticales;
- sur les c lignes suivantes, on a les mots verticaux, s'il y en a plusieurs sur la colonne on les sépare par des barres verticales.

par exemple :

3

4

```

2
2
4
3
3
un célèbre langage de programmation
un outil qui permet d'écrire des programmes et de les tester...
possessif|un célèbre langage de programmation popularisé par Kernighan et Ritchie.
démonstratif
un célèbre langage de programmation
note
cinquante| dernière lettre de la machine abstraite catégorique
caml
edi
ta|c
cet
ada
mi
l|c

```

Question 1 Déclarez les types nécessaires à la représentation des informations d'une grille de mot croisé.

Question 2 Réalisez une procédure paramétrée par un élément de type `mot_croise` et un canal de sortie et qui envoie en mode texte dans ce canal l'élément de type `mot_croise` passé en paramètre.

Question 3 Réalisez la fonction (à effet de bord) qui fait la lecture en mode texte d'un mot croisé et renvoie le mot croisé lu.

Question 4 On s'autorise maintenant d'utiliser d'autre format que le texte. Le langage `caml` fournit des instructions de haut niveau pour lire et écrire dans un fichier une variable de type `mot_croise`. En supposant que la variable `mini` contient un mot croisé, donnez l'instruction qui permet de l'écrire dans le canal de sortie `f`.

Annexe A

La compilation

A.1 Qu'est-ce que la compilation ?

En informatique, la *compilation* désigne le processus de transformation d'un programme écrit dans un langage lisible par un humain en un programme exécutable par une machine.

De manière plus générale, il s'agit de traduire un programme écrit dans un langage source en un programme écrit dans un langage cible. Dans notre cas, le langage source est le langage CAML, le langage cible peut-être soit le langage d'instructions d'une machine virtuelle (bytecode), soit le langage d'instructions du processeur de la machine physique (code natif).

Les étapes d'une telle transformation sont représentées à la figure A.1.

Nous n'entrerons pas dans le détail de ces étapes, nous nous intéressons seulement à la manière de produire le résultat, c'est-à-dire de produire un programme objet. Sachez que tous les programmes que vous utilisez sur un ordinateur sont le résultat d'une compilation : les navigateurs, les éditeurs de texte, les gestionnaires de fenêtres, les systèmes d'exploitation, ... et les compilateurs eux-mêmes.

A.2 Compiler un programme en OBJECTIVE CAML

Le terme *compilateur* désigne un programme qui réalise une compilation : il reçoit en argument le nom d'un fichier contenant un programme à *compiler*, transforme ce programme, et produit comme résultat un fichier contenant le programme en langage cible. Il existe ainsi au moins autant de compilateurs qu'il y a de langages de programmation.

Nous utiliserons le compilateur `ocamlc` : `ocamlc` produit du bytecode, c'est à dire des instructions compréhensibles par une machine virtuelle (`ocamlrun` en l'occurrence). L'intérêt de produire du bytecode est qu'il est exécutable sur toute machine/processeur disposant de la machine virtuelle, et ce de manière transparente. Nous allons illustrer l'utilisation du compilateur `ocamlc` avec le programme d'affichage de la table de multiplication par 7.

1. créer un fichier `table_mult_7.ml` contenant l'instruction suivante :

```
let table = 7
in
  for i = 1 to 10 do
```

```
Printf.printf "%2d_x_%2d=_%3d\n" i table (i * table)
done
```

2. dans un terminal, appeler le compilateur en donnant comme argument le nom du fichier à compiler. L'option `-o` permet de spécifier le nom du programme exécutable à générer (`table_mult_7` ici, sinon ce sera `a.out` par défaut) :

```
[levoire@zywiec tp-compil] ocamlc table_mult_7.ml -o table_mult_7
```

Si il n'y a pas d'erreur dans le fichier à compiler, le compilateur vous rend la main sans rien afficher. Il aura cependant créé trois fichiers, `table_mult_7.cmi`, `table_mult_7.cmo` et `table_mult_7`. Un listage de votre répertoire de travail permet de le confirmer :

```
[levoire@zywiec tp-compil] ls -l table_mult_7*
-rwxr-xr-x. 1 levoire lock 46287 fevr.  4 10:45 table_mult_7
-rw-r--r--. 1 levoire lock   224 fevr.  4 10:45 table_mult_7.cmi
-rw-r--r--. 1 levoire lock   403 fevr.  4 10:45 table_mult_7.cmo
-rw-r--r--. 1 levoire lock   107 fevr.  4 10:35 table_mult_7.ml
```

Les deux premiers fichiers `table_mult_7.cmi` et `table_mult_7.cmo` sont des résultats intermédiaires de la compilation, et nous les utiliserons plus tard avec la compilation séparée. Le fichier exécutable qui nous intéresse est le fichier `table_mult_7`.

3. lancez l'exécution du programme `table_mult_7` depuis ce même terminal :

```
[levoire@zywiec tp-compil] ./table_mult_7
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
6 x 7 = 42
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70
```

Notez que sous Linux, il vous faudra toujours ajouter les caractères `./` devant les noms de vos programmes quand vous souhaitez les exécuter.

Remarque concernant les systèmes Windows La compilation d'un fichier source CAML sous Windows s'effectue avec la même commande que sous Linux. Elle produit aussi trois fichiers de même nom. La différence réside dans l'exécution de l'exécutable produit : elle se fait en invoquant explicitement la machine virtuelle `ocamlrun`.

```
C:\Documents and Settings\Raymond Calbuth> ocamlrun table_mult_7
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
```

```
6 x 7 = 42
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70
```

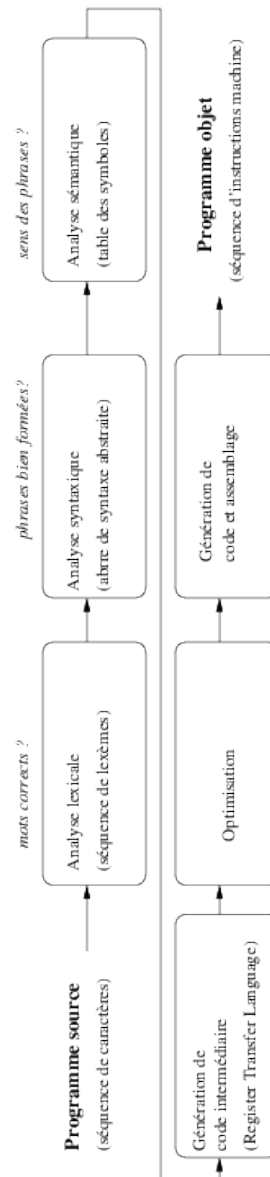


FIGURE A.1 – Les étapes d’une compilation.

Annexe B

Éléments du langage étudiés dans ce cours

B.1 Éléments syntaxiques du cours d'InitProg

B.1.1 Commentaires

Syntaxe : Commentaires en CAML

```
(* ceci est un commentaire *)
```

B.1.2 Expressions et instructions

Syntaxe : Séquence d'instructions

```
< instruction1 >;  
< instruction2 >;  
...  
< instructionn >
```

Syntaxe : Bloc d'instructions

```
begin  
  (* sequence d'instructions *)  
end
```

B.1.3 Expressions conditionnelles

Syntaxe : Expression conditionnelle

```
if < condition > then  
  < expression1 >  
else  
  < expression2 >
```

B.1.4 Instructions conditionnelles

Syntaxe : Instruction conditionnelle complète

```
if < condition > then
  (* bloc d'instructions *)
else
  (* bloc d'instructions *)
```

Syntaxe : Instruction conditionnelle simple

```
if < condition > then
  (* bloc d'instructions *)
```

B.1.5 Instructions conditionnellement répétées

Syntaxe : Boucle tant que

```
while < condition > do
  (* sequence d'instructions *)
done
```

B.1.6 Constantes, variables

Syntaxe : Déclaration d'une variable

```
let < identificateur > = < expression >
```

Syntaxe : Déclaration simultanée de plusieurs variables

```
let < identificateur1 > = < expression1 >
and < identificateur2 > = < expression2 >
...
and < identificateurn > = < expressionn >
```

Syntaxe : Déclaration de variables locales à une expression

```
let < identificateur1 > = < expression1 >
and < identificateur2 > = < expression2 >
...
in
  < expression >
```

Syntaxe : Déclaration d'une variable mutable

```
let < identificateur > = ref < expression >
```

Syntaxe : Affectation d'une valeur à une variable mutable

```
< identificateur > := < expression >
```


B.1.7 La boucle pour

Syntaxe : Boucle pour

```
for i = a to b do
  (* sequence d'instructions *)
done
```

Syntaxe : Boucle pour à indice décroissant

```
for i = a downto b do
  (* sequence d'instructions *)
done
```

B.1.8 Les fonctions

Syntaxe : Déclaration d'une fonction

```
let <nom> (<liste parametres>) =
  (* expression simple ou sequence d'instructions
    definissant la fonction *)
```

B.1.9 Caractères et chaînes de caractères

Syntaxe : Accès au i -ème caractère d'une chaîne s

```
s.[i]
```

Syntaxe : Modifier la valeur du i -ème caractère d'une chaîne s

```
s.[i] <- (* une expression de type char *)
```

B.2 Éléments syntaxiques du cours d'API1

B.2.1 Les fonctions

Syntaxe : Déclaration d'une fonction sous forme curryfiée

```
let fonction param1 ... paramn = expression
```

Syntaxe : Déclaration d'une fonction sous forme decurryfiée

```
let fonction (param1, ... ,paramn) = expression
```

Syntaxe : Application d'une fonction sous forme curryfiée

```
fonction param1 ... paramn
```

Syntaxe : Application d'une fonction sous forme décurryfiée

```
fonction (param1, ... ,paramn)
```

B.2.2 Le filtrage

Syntaxe : Expression **match** ... **with** ...

```
match expression with
  | filtre1 -> expr1
  | filtre2 -> expr2
  ...
  | filtren -> exprn
```

B.2.3 Les n-uplets

Syntaxe : Déclaration simultanée à l'aide de *n*-uplets

```
let id1, id2, ..., idn = expr1, expr2, ..., exprn
```

B.2.4 Les tableaux

Syntaxe : Accès à l'élément d'indice *i* d'un tableau *t*

```
t.(i)
```

Syntaxe : Instruction de modification de l'élément d'indice *i* d'un tableau *t*

```
t.(i) <- expr
```

Syntaxe : Accès à l'élément d'indices *i* et *j* d'un tableau *t* à deux dimensions

```
t.(i).(j)
```

Syntaxe : Instruction de modification de l'élément d'indices *i* et *j* d'un tableau *t* à deux dimensions

```
t.(i).(j) <- expr
```

B.2.5 Les enregistrements

Syntaxe : Notation pointée pour accéder à une composante *n* d'un enregistrement *e*

```
e.n
```

Syntaxe : Déclaration d'un champ mutable dans un enregistrement

```
...
mutable nom : t ;
...
```

B.2.6 Les exceptions

Syntaxe : Expression **try** ... **with** ...

```
try expression with
  | exn1 -> expr1
  | exn2 -> expr2
  ...
  | exnn -> exprn
```

B.3 Mots-clés du langage

Liste des mots-clés (ou encore mots réservés) du langage rencontrés dans ce cours.

and, begin, do, done, downto, else, end, false, for, if, in, let, match, mutable, open, ref, then, to, try, true, type, while, with.

Rappel Les mots-clés du langage ne peuvent pas servir d'identificateurs de variables.

B.4 Fonctions prédéfinies du langage

B.4.1 Fonctions de conversion

- `int_of_float : float → int`
`int_of_float x` convertit le flottant `x` en un entier.
- `float_of_int : int → float`
`float_of_int n` convertit l'entier `n` en un flottant.
- `int_of_char : char → int`
`int_of_char c` convertit le caractère `c` en un entier (son code ASCII).
- `char_of_int : int → char`
`char_of_int n` convertit l'entier `n` en le caractère de code `n`.
CU : déclenche l'exception `Invalid_argument "char_of_int"` si `n` n'est pas un entier compris entre 0 et 255.
- `int_of_string : string → int`
`int_of_string s` convertit la chaîne `s` en un entier.
CU : déclenche l'exception `Failure "int_of_string"` si `s` n'est pas l'écriture d'un entier.
- `string_of_int : int → string`
`string_of_int n` convertit l'entier `n` en une chaîne de caractères : son écriture décimale.
- `float_of_string : string → float`
`float_of_string s` convertit la chaîne `s` en un flottant.
CU : déclenche l'exception `Failure "float_of_string"` si `s` n'est pas l'écriture d'un flottant.
- `string_of_float : float → string`
`string_of_float x` convertit le flottant `x` en une chaîne de caractères : son écriture décimale.

B.4.2 Impressions

- `print_int : int → unit`
`print_int n` imprime l'entier `n`.
- `print_float : float → unit`
`print_float x` imprime le flottant `f`.
- `print_char : char → unit`
`print_char c` imprime le caractère `c`.
- `print_string : string → unit`
`print_string s` imprime la chaîne de caractères `s`.
- `print_endline : string → unit`
`print_endline s` imprime la chaîne de caractères `s` et passe à la ligne.
- `print_newline : unit → unit`
`print_newline ()` imprime un passage à la ligne.

B.4.3 Du module String

- `String.length : string → int`
`String.length s` donne la longueur de la chaîne `s`.
- `String.create : int → string`
`String.create n` crée une nouvelle chaîne de longueur `n`.
- `String.copy : string → string`
`String.copy s` crée une copie de la chaîne `s`.
- `String.sub : string → string`
`String.sub s deb long` crée une chaîne de longueur `long` constituée des caractères de la chaîne `s` à partir de l'indice `deb`.

B.4.4 Du module Array

- `Array.length : 'a array → int`
`Array.length t` donne la longueur du tableau `t`.
- `Array.make : int → 'a → 'a array`
`Array.make n a` crée un nouveau tableau de longueur `n` rempli avec la valeur de `a`.
- `Array.init : int → (int → 'a) → 'a array`
`Array.init n f` crée un nouveau tableau de longueur `n` rempli avec les valeurs prises par `f i`.
- `Array.make_matrix : int → int → 'a → 'a array array`
`Array.make_matrix n p a` crée un nouveau tableau rectangulaire de dimensions `n` sur `p` rempli avec la valeur de `a`.
- `Array.copy : 'a array → 'a array`
`Array.copy t` crée une copie du tableau `t`.

B.4.5 Du module Printf

- `Printf.printf`
`Printf.printf <format> <expr1> ...` imprime sur la sortie standard les valeurs des expressions `<expr1> ...` selon le format indiqué par `<format>`.
 - `%d` : permet d'insérer un nombre entier.
 - `%f` : permet d'insérer un nombre flottant.
 - `%B` : permet d'insérer un booléen.

- %c : permet d'insérer un caractère.
- %s : permet d'insérer une chaîne de caractères.

B.4.6 Sur les fichiers

- `open_in : string → in_channel`
`open_in s` ouvre un canal de lecture vers le fichier désigné par la chaîne `s` et renvoie ce canal. **CU** : déclenche l'exception `Sys_error` si le fichier désigné par `s` n'existe pas.
- `open_in_bin : string → in_channel`
`open_in_bin s` idem que `open_in` mais pour des fichiers binaires.
- `open_out : string → out_channel`
`open_out s` ouvre un canal d'écriture vers le fichier désigné par la chaîne `s` et renvoie ce canal. Si ce fichier n'existe pas il est créé, sinon il est réinitialisé (perte du contenu initial!).
- `open_out_bin : string → out_channel`
`open_out_bin s` idem que `open_in` mais pour des fichiers binaires.
- `close_in : in_channel → unit`
`close_in c` ferme le canal de lecture `c`.
- `close_out : out_channel → unit`
`close_out c` ferme le canal d'écriture `c`.
- `input_byte : in_channel → int`
`input_byte c` donne l'octet courant lu via le canal `c`.
CU : déclenche l'exception `End_of_file` si plus aucun octet ne peut être lu dans le fichier sur lequel le canal est ouvert.
- `input_char : in_channel → char`
`input_char c` donne le caractère courant lu via le canal `c`.
CU : déclenche l'exception `End_of_file` si plus aucun caractère ne peut être lu dans le fichier sur lequel le canal est ouvert.
- `input_line : in_channel → string`
`input_line c` donne la ligne courante lue via le canal `c`.
CU : déclenche l'exception `End_of_file` si plus aucune ligne ne peut être lue dans le fichier sur lequel le canal est ouvert.
- `input_value : in_channel → 'a`
`input_value c` donne la valeur de type `'a` courante lue via le canal `c`. Le type `'a` de la valeur doit être spécifié par le programmeur.
CU : déclenche l'exception `End_of_file` si plus aucune valeur ne peut être lue dans le fichier sur lequel le canal est ouvert.
- `output_byte : out_channel → int → unit`
`output_byte c n` écrit l'octet `n` via le canal `c`.
- `output_char : out_channel → char → unit`
`output_char c n` écrit le caractère `n` via le canal `c`.
- `output_string : out_channel → string → unit`
`output_string c s` écrit le caractère `s` via le canal `c`.
- `output_value : out_channel → 'a → unit`
`output_value c a` écrit la valeur `a` via le canal `c`.
- `flush : out_channel → unit`
`flush s` force la réalisation des écritures en suspens du canal `s`.
- `flush_all : unit → unit`
`flush_all ()` force la réalisation des écritures en suspens dans tous les canaux.

B.5 Directives

#quit : pour quitter l'interprète.

#use <nom fichier> : pour charger un fichier contenant des déclarations et expressions à évaluer.

#load <module> : pour charger un module.

open <module> : pour se dispenser de préfixer les variables d'un module par son nom.

Annexe C

Messages d'erreurs de l'interprète du langage

C.1 Messages rencontrés en InitProg

Syntax error message d'erreur lorsqu'une phrase (expression ou déclaration) est malformée.

```
# 1 + ;;  
Syntax error  
# let = 3+1 ;;  
Syntax error
```

This expression has type ...but is here used with type ... message d'erreur lorsqu'une expression n'est pas typable par inadéquation des types de certaines sous-expressions.

```
# 1 + 2. ;;  
This expression has type float but is here used with type int  
# print_string (1);;  
This expression has type int but is here used with type string
```

Unbound value ... message d'erreur lorsque dans l'évaluation d'une expression une variable non préalablement déclarée intervient.

```
# a + 1;;  
Unbound value a
```

Unbound constructor ... message d'erreur lorsque dans une expression intervient un constructeur non déclaré. En CAML, les constructeurs sont désignés par des identificateurs débutant par une lettre majuscule.

```
# init_tas(2,TT);;  
Unbound constructor TT  
# let a = 1 in a*a + A ;;  
Unbound constructor A
```

This function is applied to too many arguments, maybe you forgot a ';' message d'erreur lorsqu'on précise d'avantage de paramètres qu'il n'est nécessaire.

```
# let f(x) = x + 1;;
val f : int -> int = <fun>
# f(1)(2);;
This function is applied to too many arguments, maybe you forgot a ';
```

This expression is not a function, it cannot be applied message d'erreur lorsqu'on essaie d'utiliser une valeur comme une fonction...

```
# let a = 3 ;;
val a : int = 3
# a(4);;
This expression is not a function, it cannot be applied
```

C.2 Messages rencontrés en API1

Unbound record field label ... message d'erreur rencontré lorsque dans une expression un identificateur de composante d'un enregistrement non préalablement déclaré est utilisé.

```
# {timoleon = 2} ;;
Unbound record field label timoleon
```

Some record field labels are undefined : ... message d'erreur rencontré dans une expression d'un type enregistrement dans laquelle certains champs ne sont pas définis.

```
# let prise_bastille = {annee=1789} ;;
Some record field labels are undefined: quantieme mois
```

Variable ... is bound several times in this matching message d'erreur rencontré quand on utilise plusieurs fois une même variable dans motif de filtre.

```
# match (1,2) with
| (x,x) -> true
| _      -> false;;
Variable x is bound several times in this matching
```


Bibliographie

- [CD04] Valérie Ménissier-Morain Catherine Dubois. *Apprentissage de la programmation avec OCaml*. Hermès, 2004.
- [EC00] Bruno Pagano Emmanuel Chailloux, Pascal Manoury. *Développement d'applications avec Objective Caml*. O'Reilly, 2000. Disponible en ligne à l'adresse <http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/>.
- [TAH92] Véronique Donzeau-Gouge Viguié Thérèse Accart Hardin. *Concepts et outils de programmation. Le style fonctionnel, le style impératif avec CAML et ADA*. Interéditions, 1992.

Liste des algorithmes

0.1	Calcul de $n!$	4
2.1	Doublement des valeurs des éléments d'un tableau de nombres	34
2.2	Principe général d'un compteur simple	43
2.3	Principe général de manipulations de compteurs multiples	43
2.4	Construction d'un carré magique d'ordre impair	53
3.1	Algorithme de recherche séquentielle laborieuse.	57
3.2	Algorithme de recherche séquentielle.	58
3.3	Algorithme de recherche séquentielle modifié pour opérateurs booléens non séquentiels.	58
3.4	Algorithme de recherche séquentielle dans un tableau trié.	61
3.5	Algorithme de recherche dichotomique dans un tableau trié.	62
4.1	Algorithme de sélection du minimum dans une tranche de tableau	70
4.2	Algorithme de tri par sélection du minimum	71
4.3	Insertion d'un élément dans une tranche triée de tableau	72
4.4	Algorithme de tri par insertion	73
4.5	Algorithme de tri par dénombrement	75
4.6	Algorithme du tri à bulles	79

Table des figures

1	QR-code du site du cours	5
2.1	Représentation usuelle d'un tableau à une dimension	27
2.2	Un tableau à deux dimensions, rectangulaire	36
2.3	Le tableau de la figure 2.2 vu comme un tableau de trois lignes	36
2.4	Le tableau de la figure 2.2 vu comme un tableau de quatre colonnes	37
2.5	Un tableau à deux dimensions, triangulaire	37
2.6	Partage de valeurs	41
2.7	Une image composée de $h \times l$ pixels, et le pixel de coordonnées (i, j)	44
2.8	Une image noire	46
2.9	Un dégradé de gris	47
2.10	Les 11 premières lignes du triangle de Pascal	48
2.11	La mélancolie de Dürer (1471-1528) et le carré magique en détail	53
3.1	Arbre des différents parcours pour une recherche dichotomique dans un tableau trié de longueur 8.	63
3.2	Arbre des différents parcours pour une recherche dichotomique dans un tableau trié de longueur 10.	63
6.1	Canal de communication entre processus et fichier	96
A.1	Les étapes d'une compilation.	110

Table des matières

1	Retour sur les fonctions	7
1.1	Fonctions à plusieurs paramètres	7
1.1.1	Schéma général du type d'une fonction	7
1.1.2	Une fonction à deux paramètres	7
1.1.3	Une autre formulation de la même fonction	8
1.1.4	Un autre exemple	9
1.1.5	Formes curryfiées, formes décurryfiées	10
1.1.6	Passage d'une forme à l'autre	11
1.1.7	Du bon usage des parenthèses	11
1.2	Couples, triplets, n -uplets	12
1.2.1	Le type d'un n -uplet	12
1.2.2	Valeurs d'un type n -uplet	12
1.2.3	Comparaison de n -uplets	13
1.2.4	Variables de type n -uplet	13
1.2.5	Fonction renvoyant un n -uplet	13
1.3	Filtrage	14
1.3.1	Un premier exemple	14
1.3.2	Filtrage sur des n -uplets	15
1.3.3	Possibilité de « rassembler » les filtres	16
1.3.4	L'expression match ... with	16
1.4	Des exceptions pour les fonctions partiellement définies	18
1.4.1	Qu'est-ce qu'une exception ?	19
1.4.2	Déclencher une exception	20
1.4.3	Déclencher une exception pour restreindre l'utilisation d'une fonction	20
1.4.4	Méthodologie	21
1.5	Exercices	21
1.5.1	Fonctions à plusieurs paramètres	21
1.5.2	Filtrage	22
1.5.3	Couples, triplets, n -uplets	23
1.5.4	Exceptions	25
2	Les tableaux	27
2.1	Les vecteurs : tableaux à une dimension	27
2.2	Création d'un tableau	28
2.2.1	Création par énumération littérale	28
2.2.2	Filtrage	29
2.2.3	Création d'un tableau par sa longueur et une valeur par défaut	29

2.2.4	Création d'un tableau par sa longueur et une fonction de remplissage . . .	29
2.3	Gestion du contenu d'un tableau	30
2.3.1	Nombre d'éléments d'un tableau	30
2.3.2	Accès à un élément	30
2.3.3	Modification d'un élément	30
2.3.4	Mutabilité et partage de valeurs	31
2.3.5	Comparaison de tableaux	32
2.3.6	Vecteurs et chaînes de caractères	33
2.4	Tableaux en paramètres de fonctions	33
2.5	Tableaux à plusieurs dimensions	35
2.5.1	Tableaux à deux dimensions	35
2.5.2	Accès et modification d'un élément	38
2.5.3	Création de tableaux à deux dimensions	39
2.5.4	Partage de valeurs	41
2.5.5	Encore plus de dimensions	42
2.6	Exemples d'utilisation des tableaux	42
2.6.1	Représentation de fonction sous forme de tableau	42
2.6.2	Compter les caractères d'une chaîne	43
2.6.3	Représentation d'images	44
2.6.4	Le triangle de Pascal	46
2.7	Le tableau <code>Sys.argv</code>	49
2.8	Exercices	49
2.8.1	Tableaux à une dimension	49
2.8.2	Tableaux à plusieurs dimensions	52
3	Recherche dans un tableau	55
3.1	Introduction	55
3.1.1	Tranche	55
3.1.2	Appartenance	55
3.1.3	Les problèmes	56
3.1.4	Type d'une fonction de recherche	56
3.1.5	Objectifs	56
3.2	Un premier algorithme : la recherche laborieuse	56
3.2.1	Principe	56
3.2.2	Algorithme	56
3.2.3	Coût	57
3.3	La recherche séquentielle	57
3.3.1	Principe	57
3.3.2	Vers l'algorithme	57
3.3.3	Algorithme	57
3.3.4	Remarque importante	57
3.3.5	Coût	59
3.4	La recherche séquentielle dans un tableau trié	59
3.4.1	Tableau trié	60
3.4.2	Principe	60
3.4.3	Algorithme	60
3.4.4	Coût	60
3.5	La recherche dichotomique	61
3.5.1	Principe	61

3.5.2	Algorithme	61
3.5.3	Coût de la recherche dichotomique	61
3.6	Le code CAML	63
3.6.1	Recherche séquentielle laborieuse	64
3.6.2	Recherche séquentielle	64
3.6.3	Recherche séquentielle dans tableau trié	64
3.6.4	Recherche dichotomique dans un tableau trié	65
3.7	Exercices	65
3.7.1	Recherche séquentielle	65
3.7.2	Tableau trié	66
3.7.3	Recherche dichotomique	66
4	Trier un tableau	69
4.1	Introduction	69
4.2	Qu'est-ce que trier ?	70
4.3	Le tri par sélection	70
4.3.1	Sélection du minimum dans une tranche de tableau	70
4.3.2	L'algorithme de tri par sélection	70
4.3.3	Coût du tri par sélection	71
4.4	Le tri par insertion	72
4.4.1	Insertion d'un élément dans une tranche triée	72
4.4.2	L'algorithme du tri par insertion	73
4.4.3	Coût du tri par insertion	73
4.5	Le tri par dénombrement	74
4.6	Implantations en CAML	75
4.6.1	Tri par sélection	76
4.6.2	Tri par insertion	77
4.6.3	Tri par dénombrement	78
4.7	À consulter sur le Web	78
4.8	Exercices	79
5	Enregistrements	81
5.1	Besoin des enregistrements	81
5.2	Déclaration d'un type enregistrement	82
5.3	Création d'un enregistrement	83
5.4	Accès aux composantes	85
5.4.1	Filtrage	85
5.4.2	Notation pointée	87
5.5	Enregistrement à champs mutables	87
5.5.1	Retour sur les variables mutables	89
5.6	Un exemple complet	90
5.7	Exercices	92
6	Les fichiers	95
6.1	Besoin des fichiers	95
6.2	Entrées et Sorties	95
6.3	Les trois phases de traitements sur les fichiers	95
6.4	Ouverture d'un fichier	96
6.5	Lecture et écriture	97

6.5.1	Lecture ou écriture d'un octet	97
6.5.2	Lecture ou écriture d'un caractère	98
6.5.3	Lecture ou écriture d'une chaîne de caractères	99
6.5.4	Lecture d'un élément	99
6.6	Fin de fichier	100
6.7	Retour sur les exceptions	101
6.8	Forcer les écritures en suspens	102
6.9	Trois fichiers particuliers	103
6.10	Quelques messages	103
6.11	Exercices	104
6.11.1	Entrées et sorties de bas niveau, fichiers d'octets	104
6.11.2	Entrées et sorties de haut niveau, fichiers de lignes de texte, fichiers contenant des structures	105
A	La compilation	107
A.1	Qu'est-ce que la compilation?	107
A.2	Compiler un programme en OBJECTIVE CAML	107
B	Éléments du langage étudiés dans ce cours	111
B.1	Éléments syntaxiques du cours d'InitProg	111
B.1.1	Commentaires	111
B.1.2	Expressions et instructions	111
B.1.3	Expressions conditionnelles	111
B.1.4	Instructions conditionnelles	112
B.1.5	Instructions conditionnellement répétées	112
B.1.6	Constantes, variables	112
B.1.7	La boucle pour	113
B.1.8	Les fonctions	113
B.1.9	Caractères et chaînes de caractères	113
B.2	Éléments syntaxiques du cours d'API1	113
B.2.1	Les fonctions	113
B.2.2	Le filtrage	114
B.2.3	Les n-uplets	114
B.2.4	Les tableaux	114
B.2.5	Les enregistrements	114
B.2.6	Les exceptions	115
B.3	Mots-clés du langage	115
B.4	Fonctions prédéfinies du langage	115
B.4.1	Fonctions de conversion	115
B.4.2	Impressions	116
B.4.3	Du module String	116
B.4.4	Du module Array	116
B.4.5	Du module Printf	116
B.4.6	Sur les fichiers	117
B.5	Directives	118

<i>TABLE DES MATIÈRES</i>	131
---------------------------	-----

C Messages d'erreurs de l'interprète du langage	119
C.1 Messages rencontrés en InitProg	119
C.2 Messages rencontrés en API1	120