

Licence S.T.S - Semestre 1  
**Initiation à la programmation**  
année universitaire 2013-2014



Licence Creative Commons 



# Introduction

*[...] l'informatique est la discipline qui s'applique à étudier et à définir des opérations, et où l'on opère des transformations sur des combinaisons de symboles très généraux représentant diverses informations en de nouvelles combinaisons de symboles, elles-mêmes porteuses d'informations.*

G. IFRAH, Histoire Universelle des Chiffres, coll. Bouquins, t. II, p. 725.

Ce document est le support du cours d'initiation à la programmation (InitProg) proposé au semestre 1 de la première année de licence Sciences et Technologies de l'université de Lille 1, dans les parcours MASS, MIMP, PC, PEIP et SPI.

Les étudiants peuvent poursuivre l'étude de la programmation dans les deux cours

- Algorithmes et Programmation Impérative 1 (API1), proposé au semestre 2 de première année dans les profils MASS, MIMP, PEIP et SPI;
- et Algorithmes et Programmation Impérative 2 (API2), proposé au semestre 3 en deuxième année de licence dans les mentions Génie civil, Génie méca, Informatique, Maths, Mécanique et MASS.

## La programmation

Un algorithme est la description d'un enchaînement de calculs et de tâches élémentaires en vue de la réalisation d'un traitement automatique de données. Ces données doivent être représentées par des structures appropriées. L'algorithme peut ensuite être codé dans un langage de programmation pour donner un programme. La programmation regroupe ces activités : algorithmique, structuration de données et codage, les deux premières étant certainement les plus importantes.

Contrairement à ce que pourrait croire un novice, la tâche principale de la conception d'un programme n'est pas l'écriture de celui-ci dans un langage informatique donné.

La réalisation d'un programme se décompose en effet en plusieurs phases. De manière simplifiée, on présente ainsi la genèse d'un programme :

1. établissement d'un cahier des charges : le problème posé;
2. analyse du problème et décomposition de celui-ci en sous-problèmes plus ou moins indépendants et plus simples à résoudre;
3. choix de structures de données pour représenter les objets du problème;
4. mise en œuvre des différents algorithmes à utiliser pour résoudre les sous-problèmes;
5. codage des algorithmes et création du programme dans un langage de programmation donné;
6. tests (et corrections) et validation auprès des utilisateurs.

Bien souvent, dans la pratique, ces étapes ne sont pas aussi nettement découpées. Les quatre premières sont certainement les plus importantes et on peut remarquer qu'elles sont indépendantes du langage de programmation choisi, qui n'intervient normalement qu'à la cinquième

étape. La dernière étape est, elle aussi, très importante et ce serait une grave erreur de penser qu'elle ne fait pas partie du cycle de création d'un programme.

Pour un débutant en informatique, étudiant de surcroît, le cahier des charges consistera souvent en un énoncé d'exercice ou de problème proposé par un enseignant. Une première tâche, et elle est souvent plus difficile qu'on ne le croit, sera de comprendre cet énoncé ! Il faudra ensuite réfléchir à la résolution de ce problème et, ensuite seulement, le coder dans un langage. On peut donc le constater, une bonne partie du travail doit se faire avec un crayon et un papier ! Une erreur (méthodologique) des débutants est souvent, une fois l'énoncé lu (ou plus souvent simplement survolé), de se ruer sur leur clavier et de commencer à taper dessus (on ne sait d'ailleurs trop quoi) et seulement alors de commencer à réfléchir à la résolution du problème.

## Objectif du cours

Il s'agit de poser les notions de base en programmation impérative.

- Expressions et instructions.
- Constantes et variables, affectation. Variables locales.
- Types de données simples : nombres entiers ou non entiers, booléens, caractères et chaînes de caractères.
- Expressions et instructions conditionnelles.
- Répétition conditionnelle d'instructions (boucle tant que). Répétition non conditionnelle d'instructions (boucle pour).
- Fonctions et procédures. Paramètres formels, paramètres effectifs.

Dans les deux cours qui suivent (API1 et API2) sont abordées les notions

- tableaux ;
- enregistrements ;
- fichiers ;
- récursivité ;
- structures de données dynamiques ;
- programmation modulaire ;
- algorithmes de recherche et de tri ;
- complexité des algorithmes.

## Le choix du langage

Il existe de très nombreux langages impératifs : C, PASCAL, ADA, PYTHON, CAML. . .

Pendant quelques années nous avons utilisé le langage PASCAL. Mais l'expérience a montré que la lourdeur de la syntaxe de ce langage, ainsi que certaines de ses ambiguïtés, présentaient de réelles difficultés pour des étudiants débutant en programmation. Ce langage nécessitant une phase d'édition, puis une phase de compilation avant toute exécution, il devient très vite fastidieux de tester la moindre idée d'algorithme. Pour ces raisons nous avons décidé de changer de langage.

Nous nous sommes alors tournés vers CAML. Même si à la base ce langage est avant tout un langage fonctionnel, il offre aussi la possibilité de programmer dans un style impératif. Il dispose de plusieurs modes d'exécution :

- un mode interactif
- un mode compilé<sup>1</sup> qui permet de produire des exécutables autonomes.

---

1. en réalité, il existe deux modes compilés : une compilation en code-octet qui doit ensuite être interprété par une machine virtuelle : et une compilation en code natif pour une architecture donnée.

Dans le cours d'InitProg, le mode interactif est le seul mode abordé. La compilation pour produire des programmes exécutables autonomes sera abordée dans le cours d'APII.

CAML est un langage fonctionnel dans lequel les fonctions sont des valeurs du même ordre que toute autre valeur. En particulier, une fonction peut produire une autre fonction. Par exemple, la fonction **plus** définie ci-dessous est une fonction qui permet d'additionner deux nombres entiers.

```
# let plus a b = a + b ;;
val plus : int -> int -> int = <fun>
```

On peut le vérifier en appliquant la fonction **plus** aux deux entiers 2 et 5.

```
# plus 2 5 ;;
- : int = 7
```

Mais on peut aussi appliquer cette fonction à un seul entier, et la valeur produite par cette application donne une fonction qui, comme le montre le type indiqué par l'interprète du langage est une fonction qui à un entier associe un autre entier.

```
# let plus2 = plus 2 ;;
val plus2 : int -> int = <fun>
```

L'application de la fonction **plus2** à un entier donne cet entier augmenté de 2.

```
# plus2 5 ;;
- : int = 7
```

Mathématiquement parlant, la fonction **plus** est donc une fonction dont l'ensemble de départ est l'ensemble des entiers  $\mathbb{Z}$  et celui d'arrivée est l'ensemble des fonctions de  $\mathbb{Z}$  dans  $\mathbb{Z}$ .

$$\text{plus} : \mathbb{Z} \longrightarrow (\mathbb{Z} \rightarrow \mathbb{Z}).$$

Une autre façon de considérer la fonction qui à deux entiers associe la somme de ces deux entiers consiste à la définir comme une fonction dont l'ensemble de départ est l'ensemble des couples d'entiers  $\mathbb{Z} \times \mathbb{Z}$  et celui d'arrivée est  $\mathbb{Z}$ .

$$\text{add} : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}.$$

Les deux fonctions **plus** et **add** sont d'une certaine manière deux facettes de la même fonction : on dit que la première en est la version *curryfiée* et la seconde la version *décurryfiée*.

En CAML, on peut écrire très simplement la forme *décurryfiée*.

```
# let add (a,b) = a + b ;;
val add : int * int -> int = <fun>
```

L'application de cette fonction à deux entiers donne le résultat attendu.

```
# add (2,5) ;;
- : int = 7
```

Mais l'application à un seul entier n'est plus possible

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

puisque l'interprète du langage attend qu'un couple d'entiers soit fourni à la fonction **add**.

En programmation, il est fréquent d'avoir à définir des fonctions ou procédures à deux (ou plus) paramètres. Dans la plupart des langages de programmation (C, ADA, PASCAL...),

lorsqu'une telle fonction a été définie, il n'est pas possible de l'appliquer avec un nombre d'arguments inférieur au nombre de paramètres précisé dans sa définition. Nous avons donc choisi, pour ce cours d'InitProg, de programmer toutes les fonctions ou procédures à plusieurs paramètres que nous rencontrerons sous leur forme décurryfiée<sup>2</sup>. Cela a le double avantage

- d'être syntaxiquement proche de ce qui se fait dans d'autres langages de programmation ;
- de ne pas permettre l'application partielle d'une fonction qui fournirait des valeurs fonctionnelles qui pourrait désorienter le programmeur débutant.

## Organisation du polycopié

Ce polycopié est découpé de manière assez classique en introduction (que vous lisez actuellement), chapitres, annexes, et diverses tables (des figures, des matières, ...).

Les chapitres recouvrent les objectifs du programme d'InitProg. Ils sont tous suivis de quelques exercices. Durant les séances hebdomadaires, les enseignants pourront compléter la liste de ces exercices par d'autres.

Une annexe importante est celle présentant les fonctions du module **Cartes**. Ce module définit un environnement de travail qui permet de déplacer des cartes situées sur quatre tas d'un tas vers un autre. Cet environnement est utilisé durant les premières semaines du cours pour introduire progressivement les différentes structures de contrôle en programmation impérative, ainsi que les notions de fonctions et procédures. Le report en annexe de la présentation de ce module est délibéré afin de distinguer les objectifs du cours (présentés dans les divers chapitres) des moyens pour les atteindre.

Ce polycopié ne contient pas les sujets de TP qui seront proposés au fur et à mesure par l'intermédiaire du semainier du portail des UE d'informatique (<http://www.fil.univ-lille1.fr/portail/l1s1/initprog/>) ou du site du cours sur Moodle (<http://moodle.univ-lille1.fr/>). Bien entendu, les TP font partie intégrante du cours et sont indispensables à une bonne compréhension de ce cours.

## 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
```

---

2. Les étudiants qui poursuivront leurs études en informatique découvriront dans un cours de 3ème année consacré à la programmation fonctionnelle les avantages des fonctions curryfiées.

```

in
  for i = 1 to n do
    f := !f * i
  done;
  !f

```

Enfin, les dialogues avec l'interprète du langage sont décrits sous la forme

```

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

```

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

**Syntaxe :** Déclaration d'une variable

```
let < identificateur > = < expression >
```

## Le site du cours

Le site du cours se trouve à l'adresse :

<http://www.fil.univ-lille1.fr/portail/ls1/initprog>

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

Pierre Allegraud, Fabrice Aubert, Alexandre Bilger, Julien Bosman, François Dervaux, Alexandre Feugas, Michel Fryziel, Sophie Jacquin, Laetitia Jourdan, Céline Kuttler, Jean-Luc Levaire, Didier Mailliet, Nour-Eddine Oussous, Yosra Rekik, Romain Rouvoy, Mikaël Salson, Alexandre Sedoglavic, Bayrem Tounsi, Cristian Versari, Christophe Vroland, Éric Wegrzynowski, Léopold Weinberg, Hassina Zeglache.

## 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

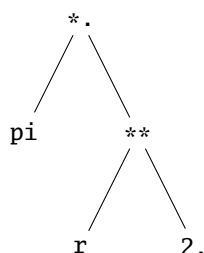
## Expressions et instructions

### 1.1 Expressions

Une expression est une manière de représenter un calcul. Par exemple, l'aire d'un disque de rayon  $r$  est donnée par l'expression :

$$\pi r^2$$

Cette expression est composée de plusieurs sous-expressions. Certaines sous-expressions ne peuvent plus être décomposées, elles sont *atomiques*. Il s'agit soit de constantes, soit de variables, soit de fonctions ou d'opérateurs.



Dans les formules les grandeurs possèdent ce que les physiciens appellent des *dimensions*, en informatique on parle plutôt de *types*.

Lorsqu'on écrit de manière traditionnelle une expression, un certain nombre de conventions d'écriture nous permet d'omettre certains signes d'opérations. Ici, par exemple, les signes de multiplications sont implicites. Lorsqu'on souhaite traduire cette expression de manière informatique, il faudra savoir que des règles strictes devront être appliquées, l'ensemble de ces règles forme *la syntaxe* du langage, et il faudra absolument les respecter pour espérer obtenir un résultat.

### 1.2 Le logiciel OBJECTIVE CAML

Le langage informatique que nous utiliserons pour ce cours est CAML. On peut utiliser ce langage dans différents modes.

- En mode *interprété* : dans ce mode le programmeur dialogue (en CAML) avec un *interprète* du langage. Plus précisément, il rédige des phrases qu'il communique à l'interprète, et celui-ci vérifie qu'elles sont bien rédigées et donne en retour une réponse au programmeur. Le mode interprété permet au programmeur d'évaluer immédiatement et pas à pas ses idées.

- En mode *compilé* : dans ce mode, le programmeur rédige (avec un éditeur de textes) l'intégralité de son programme, puis le soumet à un *compilateur* du langage dont le rôle est de produire un programme exécutable. Si le programme est correctement rédigé (aucune erreur de syntaxe, cohérence des types dans les expressions, . . .), il ne reste plus au programmeur qu'à lancer l'exécution de son programme et vérifier la conformité de cette exécution au cahier des charges initial.

Dans le cours d'InitProg, nous n'utiliserons que le mode interprété. La compilation sera abordée dans le cours APII au semestre suivant.

CAML est un langage développé par l'INRIA (Institut National de Recherche en Informatique et Automatique) que l'on peut télécharger à l'adresse <http://caml.inria.fr/>. On en trouve différentes implantations et celle que nous utiliserons est OBJECTIVE CAML. OBJECTIVE CAML est un logiciel libre qui peut être téléchargé depuis l'adresse précédente ainsi que le manuel d'utilisation<sup>1</sup>. Il en existe des versions pour les plateformes les plus courantes : Linux, Microsoft Windows, MacOS X. Dans ce polycopié, ainsi que dans les salles de TP, nous utiliserons une version pour Linux.

Pour débiter une session de dialogue avec l'interprète d'OBJECTIVE CAML, on tape la commande `ocaml` dans un terminal.

```
> ocaml
      Objective Caml version 3.10.2
```

```
#
```

Le symbole dièse (#) est l'invite de l'interprète : c'est le signe qu'il est prêt à vous écouter et à exécuter la commande que vous lui donnerez.

Avant de découvrir les premières expressions et instructions du langage CAML, voyons celle qui permet de quitter l'interprète. C'est la directive<sup>2</sup> `#quit`.

```
# #quit ;;
>
```

## 1.3 Premières expressions en CAML

### 1.3.1 Expressions numériques

La syntaxe des expressions numériques est très proche de la notation mathématique usuelle. Ci-dessous, l'expression en CAML pour l'expression

$$\frac{4 \times (-3 + 5)}{2}, \quad (1.1)$$

accompagnée de la réponse fournie par l'interprète.

```
# (4 * (-3 + 5)) / 2 ;;
- : int = 4
```

Analysons ce point de dialogue avec l'interprète.

1. Il existe aussi un excellent ouvrage d'introduction : *Développement d'applications avec OBJECTIVE CAML* (cf [EC00])

2. En CAML, les directives ne sont pas à proprement parler des expressions ou instructions. Ce sont plutôt des commandes qui modifient le comportement de l'interprète. Les directives débutent toutes par le symbole dièse.

1. Tout d'abord l'expression  $(4 * (-3 + 5)) / 2$ . Elle est très proche de la notation mathématique usuelle, les nombres sont notés comme d'habitude, l'addition et la division aussi, la multiplication est notée `*`, et on utilise des parenthèses pour marquer ici la priorité de certaines opérations sur d'autres.
2. L'expression est terminée par un double point-virgule `;;`. C'est une situation générale dans tout dialogue avec l'interprète, le double point-virgule marque la fin d'une phrase que l'on souhaite communiquer à l'évaluation. Ce marqueur mis, il ne reste plus qu'à le faire suivre d'un « retour-chariot » (touche **Entrée** du clavier) pour soumettre la phrase à l'interprète.
3. Si l'expression ne contient aucune erreur de syntaxe, et est typable<sup>3</sup>, alors l'interprète procède à son *évaluation* et en donne le résultat. Dans le cas présent, l'expression est parfaitement bien formée (aucune erreur de syntaxe), elle est correcte du point de vue des types mis en jeux, et l'interprète nous indique que cette expression a une valeur de type numérique entier (`int`) et que sa valeur est l'entier 4.

Précisons tout de suite qu'une expression ne doit pas nécessairement être écrite sur une seule ligne. On peut écrire une expression sur autant de lignes que l'on veut. C'est le marqueur de fin `;;` qui indique à l'interprète que la rédaction de l'expression est achevée. Voici la même expression rédigée sur plusieurs lignes.

```
# (
  4
  *
  (
    -3
    +
    5
  )
)
/
2
;;
- : int = 4
```

L'opération de division sur les nombres entiers fournit le quotient dans la division euclidienne. Par exemple  $7/3$  vaut 2 et non pas environ 2.3333. Le reste de la division est obtenu avec l'opérateur **mod**.

```
# 7 / 3 ;;
- : int = 2
# 7 mod 3 ;;
- : int = 1
# 3 * 2 + 1 ;;
- : int = 7
```

Bien entendu, on peut calculer des nombres non entiers que nous nommerons *nombres flottants* ou simplement *flottants*<sup>4</sup>. En CAML, ces nombres sont tous marqués d'un séparateur décimal, le point, qu'ils soient ou non dotés de chiffres après la virgule.

3. Cette notion de typage sera abordée petit à petit dans ce cours et les suivants.

4. Ces nombres correspondent approximativement aux nombres *décimaux*.

```
# 2. ;;
- : float = 2.
# 2.333 ;;
- : float = 2.333
```

Comme on peut le voir sur ces deux exemples, le type de ces nombres est `float`, pour nombre à « virgule flottante ».

⚠ En CAML, il faut distinguer les opérateurs arithmétiques portant sur des nombres entiers, de ceux portant sur des nombres non entiers. Les opérateurs pour les quatre opérations usuelles (addition, soustraction, multiplication et division) portant sur les nombres de type `float` sont identiques à ceux portant sur les nombres entiers, mais suivis d'un point. Voici l'expression arithmétique (1.1) ci-dessus, évaluée avec des nombres de type `float`.

```
# (4. *. (-. 3. +. 5.)) /. 2. ;;
- : float = 4.
```

CAML refuse systématiquement toute expression numérique où sont mélangées des nombres de nature différentes.

```
# 1 + 2. ;;
This expression has type float but is here used with type int
# 1 +. 2. ;;
This expression has type int but is here used with type float
```

Les deux fonctions `int_of_float` et `float_of_int` assurent la conversion d'un nombre d'un type à l'autre.

```
# int_of_float (-2.7) ;;
- : int = -2
# int_of_float (2.7) ;;
- : int = 2
# float_of_int (2);;
- : float = 2.
# 1 + int_of_float (2.) ;;
- : int = 3
# 1. +. float_of_int (2) ;;
- : float = 3.
```

Il existe de très nombreuses autres fonctions. En voici quelques unes :

1. sur les entiers : **abs** (valeur absolue d'un nombre entier).
2. sur les non entiers : **abs\_float** (valeur absolue d'un flottant), **sqrt** (racine carrée d'un flottant), **exp** (exponentielle de base  $e$ ), **log** (logarithme népérien), **cos**, **sin**, **tan**, et bien d'autres encore.

### 1.3.2 Expressions booléennes

En programmation, les expressions booléennes sont nécessaires pour effectuer des calculs ou exécuter des instructions dépendant de la réalisation ou non de certaines conditions (cf chapitres 2 et 3).

Les *booléens* forment un ensemble à deux valeurs seulement, désignées en CAML par **true** (vrai) et **false** (faux). Le type des booléens est désigné par `bool`.

Les *expressions booléennes* sont les expressions qui ont pour valeur l'un des deux booléens. Elles apparaissent naturellement dans les comparaisons.

```
# true ;;
- : bool = true
# false ;;
- : bool = false
# 1 = 5-4 ;;
- : bool = true
# 1 < 5-4 ;;
- : bool = false
# 1 >= 5-4 ;;
- : bool = true
```

Les opérateurs logiques usuels sont **&&** (et), **||** (ou) et **not** (négation).

```
# (1 = 1) && (1 = 0) ;;
- : bool = false
# (1 = 1) || (1 = 0) ;;
- : bool = true
# not (1 = 0) ;;
- : bool = true
```

Le chapitre 2 donne plus de précisions sur les opérateurs logiques et les expressions booléennes, ainsi que sur les expressions et instructions booléennes.

### 1.3.3 Caractères et chaînes de caractères

Les *caractères* forment un type nommé **char** en CAML qui comprend 256 valeurs. Parmi ces valeurs on trouve toutes les lettres de l'alphabet latin dans leur version minuscule et majuscule, ainsi que les dix chiffres, et des symboles de ponctuation.

Une expression littérale de type **char** est obtenue en plaçant le caractère voulu entre apostrophes.

```
# 'A';;
- : char = 'A'
# 'a';;
- : char = 'a'
# '1';;
- : char = '1'
# '?';;
- : char = '?'
```

Les *chaînes de caractères* sont des suites finies de caractères placés les uns à la suite des autres. Elles peuvent ne contenir aucun caractère : chaîne vide, et elles peuvent en contenir jusqu'à  $2^{24} - 6$ , soit plus de 16 millions de caractères. En CAML, elles sont mises entre guillemets et leur type est désigné par **string**.

```
# "Aa1?" ;;
- : string = "Aa1?"
# "";;
- : string = ""
```

L'opérateur de *concaténation* qui construit une chaîne en mettant bout à bout deux chaînes est désigné par **^**.

```
# "InitProg" ^ "_c'est_vachement_bien_!";
- : string = "InitProg_c'est_vachement_bien_!"
```

Le chapitre 7 est consacré à une présentation plus détaillée des caractères et chaînes de caractères.

## 1.4 Instructions

En CAML, une *instruction* est une expression dont la valeur est de type `unit`<sup>5</sup>. Ce type ne contient qu'une seule valeur notée `()`. Par conséquent, cela signifie que toute instruction a pour valeur `()`, que cette valeur est connue avant même que l'expression ne soit évaluée, et donc que ce qui importe n'est pas tant la valeur de l'instruction, que l'effet de bord qu'elle produit. Et d'ailleurs, plutôt que de parler d'évaluation de l'expression, on dit plutôt *exécution de l'instruction*.

Les *effets de bord* produits par l'exécution d'une instruction peuvent être de plusieurs natures :

- modification de la valeur de l'environnement courant de travail, par exemple en modifiant la valeur d'une variable mutable (cf chapitre 4);
- communication du programme avec le monde extérieur, par exemple par lecture ou écriture d'une valeur.

Voici la plus simple (et la plus inutile) des instructions.

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

L'exécution de cette instruction ne provoque aucun effet de bord (ni modification de variable, ni lecture, ni écriture). Seule sa valeur est fournie par l'interprète.

### 1.4.1 Instructions d'impression

Plus intéressantes sont les instructions pour *imprimer* ou écrire des valeurs. Elles se distinguent selon le type des valeurs à imprimer. En voici quatre illustrées chacune par un exemple.

```
# print_int (8) ;;
8- : unit = ()
# print_float (8.) ;;
8.- : unit = ()
# print_string ("8") ;;
8- : unit = ()
# print_char ('8') ;;
8- : unit = ()
```

Il faut bien comprendre la réponse de l'interprète à l'exécution de ces instructions d'impression. Dans chacun des cas, on peut voir l'impression de la valeur données à ces instructions, immédiatement suivie par la valeur de l'instruction qui est immuablement `- : unit = ()`.

On pourrait préférer qu'un saut de ligne distingue la valeur imprimée de la valeur de l'instruction. La fonction `print_endline` effectue un saut de ligne immédiatement après avoir imprimé ce qu'on lui demande. Mais cette fonction ne permet l'impression que des chaînes de caractères.

---

5. Cette définition n'est pas tout à fait exacte, mais nous nous en contenterons pour le cours de ce semestre. Nous verrons dans le cours qui suit qu'il existe des instructions dont la valeur est d'un autre type.

```
# print_endline ("Timoleon_apprend_a_programmer_en_Caml") ;;
Timoleon apprend a programmer en Caml
- : unit = ()
```

Une autre instruction provoque des sauts de ligne : l'instruction `print_newline ()`.

```
# print_newline () ;;
- : unit = ()
```

Comme on peut le voir sur cet exemple, cette instruction ne prend aucun paramètre. Elle se contente d'imprimer un saut de ligne, et si rien auparavant n'a été imprimé, comme c'est le cas ici, on obtient une ligne vide. Cette instruction suit en général d'autres instructions d'impression, et nécessite d'exécuter une *séquence d'instructions*. Cette notion de séquence d'instructions est présentée dans la section 1.4.3.

### 1.4.2 Instructions du module Cartes

Le module `Cartes` est une extension du langage CAML qui permet de manipuler des tas de cartes à jouer. Ce module est présenté dans l'annexe A.

Deux instructions y sont définies :

1. l'instruction `init_tas(n,s)` qui initialise le tas numéro `n` avec une pile de cartes décrite par la chaîne de caractères `s` ;
2. et l'instruction `deplacer_sommet(n,p)` qui déplace une carte du tas numéro `n` vers le tas numéro `p`.

Par exemple, en supposant qu'initialement aucun tas ne contienne de cartes, alors le dialogue<sup>6</sup> qui suit donne le tas tel qu'on peut le voir sur la figure 1.1.

```
# init_tas (1,"TCP") ;;
- : unit = ()
```

Comme on peut le constater sur la figure 1.1, l'effet de bord provoqué par cette instruction est une initialisation du tas numéro 1 avec trois cartes de couleurs successives ♣, ♥ et ♠.

En poursuivant le dialogue avec l'interprète, on déplace la carte au sommet du tas numéro 1 vers le tas numéro 2 et la situation atteinte est celle montrée sur la figure 1.2.

```
# deplacer_sommet (1,2) ;;
- : unit = ()
```

L'effet de bord de l'instruction `deplacer_sommet (1,2)` est la transformation de l'état des cartes.

### 1.4.3 Séquence d'instructions

Supposons qu'initialement seul le premier tas de cartes n'est pas vide, et que ce tas contient trois cartes de couleur ♣, ♥, ♠ dans cet ordre de bas en haut (cf figure 1.1). Supposons que nous voulions vider le tas 1 en plaçant une carte sur chacun des trois autres tas (cf 1.3).

Cela peut se faire simplement en déplaçant une à une les cartes du tas 1 vers les autres tas comme le montre l'algorithme 1.1.

En CAML, cet algorithme se traduit simplement par trois instructions `deplacer_sommet`.

Si on utilise la boucle de dialogue avec l'interprète du langage, on atteint la situation voulue avec le dialogue :

---

6. Ce dialogue suppose aussi que le module `Cartes` soit préalablement chargé.

FIGURE 1.1 – Le tas 1 après l'instruction `init_tas (1,"TCP")`


---

**Algorithme 1.1** Mettre les trois cartes du tas 1 vers les tas 2, 3 et 4

---

- 1: déplacer le sommet du tas 1 vers le tas 2
  - 2: déplacer le sommet du tas 1 vers le tas 3
  - 3: déplacer le sommet du tas 1 vers le tas 4
- 

```
# déplacer_sommet(1,2);
- : unit = ()
# déplacer_sommet(1,3);
- : unit = ()
# déplacer_sommet(1,4);
- : unit = ()
```

Dans ce dialogue, chacune des trois instructions est donnée à exécuter les unes après les autres, et l'interprète donne le résultat de son travail après chacune d'elles (la mention `- : unit = ()`).

On peut aussi regrouper ces trois instructions en une seule :

```
# déplacer_sommet(1,2);
  déplacer_sommet(1,3);
  déplacer_sommet(1,4);
- : unit = ()
```

L'usage du simple point-virgule pour séparer les instructions permet de les transmettre à l'interprète en une seule fois, et celui-ci ne donne qu'une seule réponse. En un seul ordre, nous avons commandé l'exécution de trois instructions.

Cette construction est très fréquente en programmation impérative, et on la nomme *séquence d'instructions*. En CAML, une séquence d'instructions s'obtient en séparant les instructions par un simple point-virgule. Dans une séquence d'instructions, le nombre d'instructions n'est pas



FIGURE 1.2 – Situation obtenue après l'instruction `deplacer_sommet (1,2)`

limité.

#### Syntaxe : Séquence d'instructions

```
< instruction1 >;
< instruction2 >;
...
< instructionn >
```

Lorsqu'une séquence d'instructions est exécutée, toutes les instructions qui la composent sont exécutées les unes après les autres dans l'ordre de leur apparition dans la séquence. La valeur fournie d'une séquence est la valeur de la dernière instruction, c'est-à-dire `()` en général.

Voici un autre exemple d'utilisation de la séquence d'instructions pour produire une affichage d'une ligne composée de plusieurs éléments de types distincts.

```
# print_string "(4*_(-3+_5))/_2=_";
print_int ((4 * (-3 + 5)) / 2) ;
print_newline () ;;
(4 * (-3 + 5)) / 2 = 4
- : unit = ()
```

On verra qu'il est parfois nécessaire de parenthéser<sup>7</sup> une séquence d'instructions pour éviter certaines ambiguïtés (cf page 28). En CAML cela peut se faire avec des parenthèses ouvrante et fermante, ou bien avec les deux mots-clés du langage **begin** et **end**. Dans la suite nous désignerons par *bloc d'instructions* toute séquence placée entre ces deux mots.

7. Par *parenthéser*, il faut entendre le fait de placer une expression entre un symbole d'ouverture et un symbole de fermeture, symboles qui peuvent être effectivement une parenthèse ouvrante `(` et une parenthèse fermante `)`, ou tout autre couple de symboles convenus comme `{` et `}`, ou bien `[` et `]`, ou bien encore **begin** et **end**.



FIGURE 1.3 – Situation finale souhaitée

**Syntaxe : Bloc d'instructions**

```
begin
  (* sequence d'instructions *)
end
```

Ce parenthésage n'a que pour seule vocation à éviter des ambiguïtés syntaxiques, et ne change rien à l'exécution de la séquence qu'il contient.

```
# begin
  print_string("(4*_(-3+_5))_/_2_=");
  print_int((4 * (-3 + 5)) / 2) ;
  print_newline ()
end ;;
(4 * (-3 + 5)) / 2 = 4
- : unit = ()
```

## 1.5 Exercices

### 1.5.1 Expressions numériques

#### Exercice 1-1

Évaluations d'expressions arithmétiques

Pour chacune des expressions arithmétiques qui suivent, évaluez-les avec OBJECTIVE CAML en arithmétique sur les entiers (`int`) et en arithmétique sur les flottants (`float`).

1.  $(3 + 4) \times 5$ .
2.  $3 + (4 \times 5)$ .
3.  $3 + 4 \times 5$ .
4.  $125 \times (12 - \frac{204}{6})$ .
5.  $125 \times (12 - \frac{204}{5})$ .

Pour la dernière expression, expliquez la différence sensible entre la valeur obtenue en arithmétique entière, et celle en arithmétique flottante.

### 1.5.2 Instructions

#### Exercice 1-2

Impressions de valeurs

**Question 1** Utilisez l'une des instructions `print_...` pour imprimer la valeur des expressions de l'exercice

**Question 2** Écrivez une instruction qui imprime une expression arithmétique, puis le signe `=`, puis la valeur de l'expression arithmétique et enfin un passage à la ligne. Voici un exemple, dans lequel l'instruction a été effacée.

```
# (* instruction effacee *)
(3+4)*5 = 35
- : unit = ()
```

Trouvez une séquence d'instructions qui réalise cela, puis cherchez une seule instruction.



## Chapitre 2

# Conditions, expressions et instructions conditionnelles

### 2.1 Le problème

- État initial : Tas 1 un trèfle ou un pique les autres tas vides.
- État final : tas 1 et 4 vides et le tas 2 contient le trèfle ou rien , et le tas 3 contient le pique ou rien.

#### 2.1.1 Algorithme

La résolution du problème nécessite une étude de cas :

---

**Algorithme 2.1** Solution du problème

---

```
si la carte au sommet du tas 1 est un trèfle alors
    mettre la carte sur le tas 2
sinon
    mettre la carte sur le tas 3
fin si
```

---

### 2.2 Expressions booléennes

#### 2.2.1 Booléens

On appelle *booléen*<sup>1</sup> l'une des deux valeurs de vérité *vrai* ou *faux*. On désigne en abrégé ces deux valeurs par les deux lettres *V* et *F*.

Une *expression booléenne* est une expression qui prend une valeur booléenne, autrement dit l'une ou l'autre des deux valeurs *V* ou *F*.

Les expressions booléennes servent à exprimer des conditions. Elles peuvent être

- *simples*
- ou *composées*

---

1. du nom du mathématicien anglais du XIXème siècle George Boole.

### 2.2.2 Expressions simples

Les expressions booléennes *simples*

"la carte au sommet du tas 1 est un trèfle"

### 2.2.3 Expressions composées

#### L'opérateur ou

L'expression  $E = \text{"la carte au sommet du tas 1 est un trèfle ou un carreau"}$  peut se décomposer en deux expressions simples

1.  $E_1 = \text{"la carte au sommet du tas 1 est un trèfle"}$ ,
2.  $E_2 = \text{"la carte au sommet du tas 1 est un carreau"}$

le lien les unissant étant le *connecteur* ou *opérateur* logique **ou**

$$E = E_1 \text{ ou } E_2$$

L'expression  $E$  est donc *composée*.

L'expression  $E$  ne prend la valeur  $V$  que si au moins l'une des deux expressions  $E_1$ ,  $E_2$  est vraie.

On résume cette définition de l'opérateur **ou** par la table de vérité représentée par le tableau 2.1

$E_1$	$E_2$	$E_1 \text{ ou } E_2$
$V$	$V$	$V$
$V$	$F$	$V$
$F$	$V$	$V$
$F$	$F$	$F$

TABLE 2.1 – Table de vérité de l'opérateur **ou**

#### L'opérateur et

L'expression  $E = \text{"la carte au sommet du tas 1 est un trèfle et sa valeur est supérieure à celle de la carte située au sommet du tas 2"}$  peut se décomposer en deux expressions simples

1.  $E_1 = \text{"la carte au sommet du tas 1 est un trèfle"}$ ,
2.  $E_2 = \text{"la carte au sommet du tas 1 a une valeur supérieure ou égale à celle du tas 2"}$

le lien les unissant étant le *connecteur* ou *opérateur* logique **et**

$$E = E_1 \text{ et } E_2$$

L'expression  $E$  est donc *composée*.

L'expression  $E$  ne prend la valeur  $V$  que si les deux expressions  $E_1$ ,  $E_2$  sont vraies.

On résume cette définition de l'opérateur **et** par la table de vérité représentée par le tableau 2.2.

$E_1$	$E_2$	$E_1$ <b>et</b> $E_2$
$V$	$V$	$V$
$V$	$F$	$F$
$F$	$V$	$F$
$F$	$F$	$F$

TABLE 2.2 – Table de vérité de l'opérateur **et****L'opérateur non**

L'expression  $E = \text{"la carte au sommet du tas 1 n'est pas un trèfle"}$  peut se décomposer en une expressions plus simple

1.  $E_1 = \text{"la carte au sommet du tas 1 est un trèfle"}$ ,  
que l'on nie.

$$E = \mathbf{non}(E_1)$$

L'expression  $E$  est donc *composée*.

La valeur de l'expression  $E$  est opposée à celle de  $E_1$ .

On résume cette définition de l'opérateur **non** par la table de vérité représentée par le tableau 2.3.

$E_1$	$\mathbf{non}(E_1)$
$V$	$F$
$F$	$V$

TABLE 2.3 – Table de vérité de l'opérateur **non****2.2.4 Propriétés des opérateurs**

$a$ ,  $b$  et  $c$  sont trois expressions booléennes.

Double négation	$\mathbf{non}(\mathbf{non}(a)) = a$
Élément neutre du <b>ou</b>	$a \text{ ou } F = a$
Élément neutre du <b>et</b>	$a \text{ et } V = a$
Élément absorbant du <b>ou</b>	$a \text{ ou } V = V$
Élément absorbant du <b>et</b>	$a \text{ et } F = F$
Idempotence du <b>ou</b>	$a \text{ ou } a = a$
Idempotence du <b>et</b>	$a \text{ et } a = a$
Tautologie	$a \text{ ou } \mathbf{non}(a) = V$
Contradiction	$a \text{ et } \mathbf{non}(a) = F$
Commutativité du <b>ou</b>	$a \text{ ou } b = b \text{ ou } a$
Commutativité du <b>et</b>	$a \text{ et } b = b \text{ et } a$
Associativité du <b>ou</b>	$(a \text{ ou } b) \text{ ou } c = a \text{ ou } (b \text{ ou } c)$
Associativité du <b>et</b>	$(a \text{ et } b) \text{ et } c = a \text{ et } (b \text{ et } c)$
Distributivité du <b>et</b> par rapport au <b>ou</b>	$a \text{ et } (b \text{ ou } c) = (a \text{ et } b) \text{ ou } (a \text{ et } c)$
Distributivité du <b>ou</b> par rapport au <b>et</b>	$a \text{ ou } (b \text{ et } c) = (a \text{ ou } b) \text{ et } (a \text{ ou } c)$
Loi de Morgan (1)	$\mathbf{non}(a \text{ ou } b) = \mathbf{non}(a) \text{ et } \mathbf{non}(b)$
Loi de Morgan (2)	$\mathbf{non}(a \text{ et } b) = \mathbf{non}(a) \text{ ou } \mathbf{non}(b)$

## 2.3 En CAML

### 2.3.1 Booléens

En CAML, les deux valeurs booléennes définissent le type `bool`.

<i>vrai</i>	<code>true</code>
<i>faux</i>	<code>false</code>

TABLE 2.4 – Booléens en CAML

### 2.3.2 Expressions booléennes

**Les expressions booléennes simples** peuvent s'exprimer en CAML par

- des valeurs littérales `true` ou `false`,
- une variable booléenne (voir plus tard),
- une expression décrivant une relation,
- un appel à une fonction à valeurs booléennes.

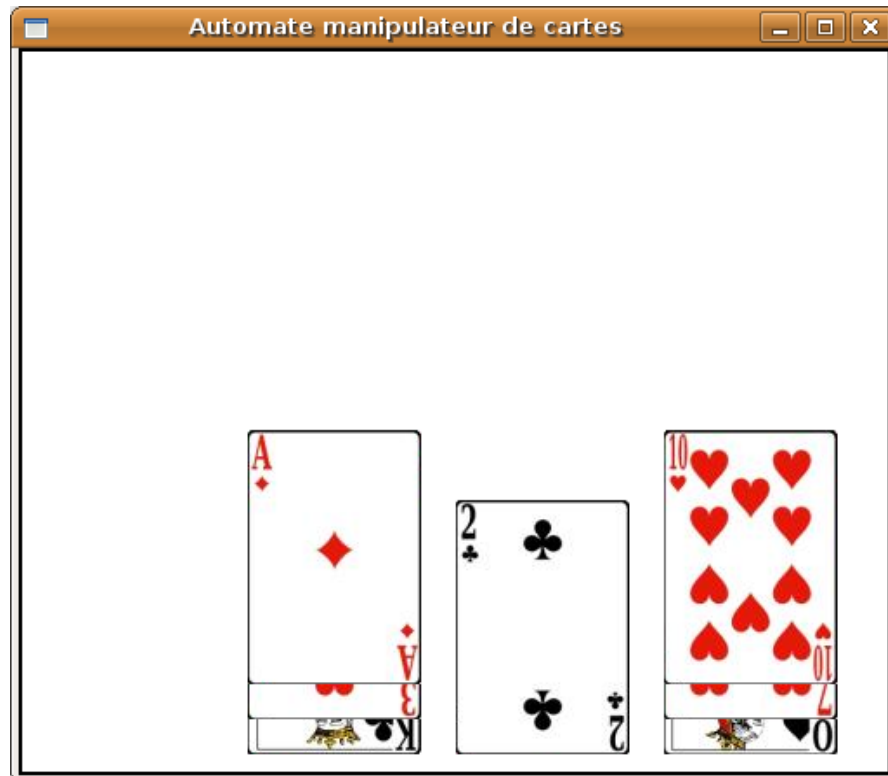


FIGURE 2.1 – Un état des tas de cartes

**Exemple 1 :**

Tous les exemples donnés par la suite s'appuient sur une configuration des tas donnée à la figure 2.1.



```
# true ;;
- : bool = true
# false ;;
- : bool = false
# tas_vide(1);;
- : bool = true
# tas_non_vide(1);;
- : bool = false
# couleur_sommet(2) = PIQUE;;
- : bool = false
# couleur_sommet(2) = CARREAU;;
- : bool = true
# sommet_trefle(3);;
- : bool = true
```

Les expressions composées se traduisent avec les opérateurs logiques `&&`, `||` et `not`.

Opérateur logique	et	ou	non
En CAML	<code>&amp;&amp;</code>	<code>  </code>	<code>not</code>

TABLE 2.5 – Opérateurs logiques en CAML

Exemple 2 :

```
# true && true;;
- : bool = true
# true && false;;
- : bool = false
# true || false;;
- : bool = true
# false || false;;
- : bool = false
# not (true);;
- : bool = false
# not (false);;
- : bool = true
# sommet_carreau(2) && sommet_trefle(3) ;;
- : bool = true
# sommet_carreau(2) && not (sommet_coeur(3)) ;;
- : bool = true
# tas_non_vide(1) || sommet_coeur(4);;
- : bool = true
```

### 2.3.3 Attention !

Les opérateurs `&&` et `||` n'agissent que sur des booléens !

On pourrait être tenté, pour exprimer que la carte au sommet du tas 1 est un trèfle ou un carreau, d'écrire

```
couleur_sommet(1) = TREFLE || CARREAU,
```

mais cela est incorrect car `TREFLE` et `CARREAU` ne sont pas des booléens, mais des couleurs. Il faut donc écrire

```
couleur_sommet(1) = TREFLE || couleur_sommet(1) = CARREAU
```

même si cela semble bien lourd.

### 2.3.4 Remarque sur les opérateurs `&&` et `||`

En logique booléenne, les opérateurs **et** et **ou** sont commutatifs

$$\begin{aligned} a \text{ et } b &= b \text{ et } a \\ a \text{ ou } b &= b \text{ ou } a \end{aligned}$$

En CAML, ces opérateurs ne sont pas commutatifs : les arguments sont évalués séquentiellement de gauche à droite.

#### Exemple 3 :

En CAML, les expressions

```
tas_non_vide(1) && sommet_trefle(1)
sommet_trefle(1) && tas_non_vide(1)
```

et

ne sont pas équivalentes :

- l'évaluation de la première expression ne déclenche jamais d'exception, car si le tas 1 est vide, `sommet_trefle(1)` n'est pas évalué, puisque la valeur de l'expression complète est alors déterminée

```
# tas_non_vide(1) && sommet_trefle(1);;
- : bool = false
```

- en revanche l'évaluation de la seconde expression déclenche une exception si le tas 1 est vide

```
# sommet_trefle(1) && tas_non_vide(1) ;;
Exception: LesExceptions.Tas_Vide 1.
```

## 2.4 Expressions conditionnelles

Les expressions conditionnelles sont des expressions qui déterminent l'expression à évaluer en fonction de la validité d'une condition.

**Syntaxe :** Expression conditionnelle

```
if < condition > then
  < expression1 >
else
  < expression2 >
```

#### Exemple 4 :

Dans le contexte indiqué par la figure 2.1, on a

```
# if sommet_carreau(2) then
  0
else
  1 ;;
- : int = 0
# if sommet_trefle(2) then
  0
else
  1 ;;
- : int = 1
```

## 2.5 Instructions conditionnelles

Les instructions conditionnelles sont des expressions conditionnelles dont la valeur est du type `unit`. On les utilise pour agir en fonction d'un contexte.

### Instruction conditionnelle complète

L'instruction conditionnelle complète (ou alternative) permet de mener une action si une condition est satisfaite, et une autre dans le cas contraire.

**Syntaxe :** Instruction conditionnelle complète

```
if < condition > then
  (* bloc d'instructions *)
else
  (* bloc d'instructions *)
```

### Exemple 5 :

Traduction en CAML de l'algorithme décrit à la section 2.1.1.

```
if sommet_trefle(1) then begin
  deplacer_sommet(1,2)
end else begin
  deplacer_sommet(1,3)
end
```

La condition est ici exprimée par l'expression booléenne (simple) `sommet_trefle(1)`, le premier bloc d'instructions (celui qui suit le mot réservé **then**) est

```
begin
  deplacer_sommet(1,2)
end
```

et le second bloc (qui suit le mot réservé **else**)

```
begin
  deplacer_sommet(1,3)
end
```

Notez qu'il n'y a pas de ; après le **end** qui précède le **else**. En revanche, si cette instruction conditionnelle est suivie d'une autre, il faudra ajouter un ; après le **end** du second bloc (**end {if};**).

**Instruction conditionnelle simple** Il arrive qu'une action ait besoin d'être menée si une condition est satisfaite, mais qu'aucune ne soit nécessaire sinon. C'est alors une instruction conditionnelle simple. En CAML, elle s'écrit sans la partie **else**.

**Syntaxe :** Instruction conditionnelle simple

```
if < condition > then
  (* bloc d'instructions *)
```

**Exemple 6 :**

Déplacer la carte située au sommet du tas 1 si c'est un trèfle sinon ne rien faire.

```
if sommet_trefle(1) then begin
  deplacer_sommet(1,2)
end
```

### 2.5.1 Remarque sur le parenthésage

La syntaxe donnée ci-dessus pour l'instruction conditionnelle mentionne qu'après le **then** et le **else** doivent figurer des blocs d'instructions, c'est-à-dire des séquences parenthésées par **begin** et **end**.

En réalité, il n'est pas nécessaire de mettre le parenthésage lorsque la séquence ne comprend qu'une seule instruction (comme pour les expressions conditionnelles). Ainsi on peut écrire

```
if sommet_trefle(1) then
  deplacer_sommet(1,2)
else
  deplacer_sommet(1,3)
```

⚠ En revanche, dès que la séquence comprend plusieurs instructions, le parenthésage devient nécessaire dans la partie **else** pour une instruction conditionnelle complète, et dans la partie **then** pour une instruction conditionnelle simple.

En effet, l'instruction

```
if sommet_trefle(1) then begin
  deplacer_sommet(1,2);
  deplacer_sommet(2,3)
end
```

ne peut pas être remplacée par

```
if sommet_trefle(1) then
  deplacer_sommet(1,2);
  deplacer_sommet(2,3)
```

qui est une séquence de deux instructions :

1. une instruction conditionnelle simple,
2. suivie d'une instruction de déplacement non soumise à condition.

## 2.6 Méthodologie

Voici quelques principes méthodologiques concernant la conception d'instructions conditionnelles.

1. Étude des cas intervenant dans le problème
  - commencer par distinguer les différents cas,
  - vérifier que ces cas sont bien exhaustifs (on n'oublie aucune situation)
  - vérifier que ces cas ne sont pas redondants (cas exclusifs)
  - donner un exemple de comportement attendu de l'instruction pour chacun des cas
2. Pour chacun des cas
  - établir un test (expression booléenne) permettant de distinguer le cas
  - déterminer la séquence d'instructions à exécuter dans ce cas
3. Construire un jeu de tests qui permet de s'assurer de la validité du programme : ce jeu doit comprendre au moins un test pour chacun des cas envisagés.

### Exemple

Considérons une situation initiale où les tas numérotés de 1 à 3 contiennent un nombre quelconque non nul de cartes, et le tas 4 est vide.

```
init_tas(1, "(T+K+C+P)[T+K+C+P]");
init_tas(2, "(T+K+C+P)[T+K+C+P]");
init_tas(3, "(T+K+C+P)[T+K+C+P]");
init_tas(4, "");
```

L'objectif à atteindre est de déplacer la carte de plus grande valeur au sommet d'un des trois tas vers le tas 4.

Résoudre ce problème revient à repérer quel tas possède en son sommet la carte de plus grande valeur.

**Étude des cas** Il y a évidemment trois possibilités : c'est le tas 1 ou le tas 2 ou le tas 3. Comment les distinguer ?

- c'est le tas 1, si la carte au sommet du tas 1 est supérieure ou égale à celle du tas 2, **et** est supérieure ou égale à celle du tas 3.
- c'est le tas 2, si la carte au sommet du tas 2 est supérieure ou égale à celle du tas 1, **et** est supérieure ou égale à celle du tas 3.
- c'est le tas 3, si la carte au sommet du tas 3 est supérieure ou égale à celle du tas 1, **et** est supérieure ou égale à celle du tas 2.

Autrement dit c'est le tas  $i$  si la condition `superieur(i,j) && superieur(i,k)` (dans laquelle  $j$  et  $k$  désignent les deux numéros de tas autres que  $i$ ) est satisfaite. Les conditions distinguant les trois cas ne sont pas exclusives (penser aux cas d'égalité).

## 2.7 Exercices

### Exercice 2-1

En utilisant les tables de vérité, prouvez quelques propriétés des opérateurs logiques **et**, **ou** et **non**.

### Exercice 2-2 *Ou exclusif*

Si  $a$  et  $b$  sont deux expressions booléennes, le **ou-exclusif** de ces deux expressions, noté  $a \oplus b$ , est une nouvelle expression booléenne qui est vraie si et seulement si une seule des deux expressions  $a$  ou  $b$  est vraie. La table de vérité du **ou-exclusif** est donnée

$a$	$b$	$a \oplus b$
$V$	$V$	$F$
$V$	$F$	$V$
$F$	$V$	$V$
$F$	$F$	$F$

TABLE 2.6 – Table de vérité de l'opérateur **ou-exclusif**

Écrivez l'expression  $a \oplus b$  à l'aide des opérateurs **et**, **ou** et **non**.

### Exercice 2-3

**Question 1** Exprimez le fait que  $[a, b]$  et  $[c, d]$  sont des intervalles disjoints, attention au cas des intervalles vides, par exemple si  $a = 15$  et  $b = -5$ .

**Question 2** Exprimez le fait que  $[a, b]$  et  $[c, d]$  sont des intervalles qui se recouvrent partiellement de deux manières :

1. en utilisant la solution de la question précédente (c'est très simple).
2. directement (c'est compliqué!).

### Exercice 2-4 égalité et valeur booléenne

**Question 1** Soit  $a$  une expression booléenne. Réaliser la table de vérité des expressions :

1.  $a = V$
2.  $a = F$

### Exercice 2-5

#### Question 1

Écrivez l'instruction ci-dessous sans utiliser l'opérateur **non** ni d'opérateurs de comparaison.

```

si non  $a$  alors
  instr1
sinon
  instr2
fin si

```

#### Question 2

Écrivez l'instruction ci-dessous sans utiliser l'opérateur **et**

```
si aetb alors
  instr1
fin si
```

**Question 3**

Écrivez l'instruction ci-dessous sans utiliser l'opérateur **ou**

```
si aoub alors
  instr1
fin si
```

**Exercice 2-6**

Expliquez pourquoi le programme

```
if superieur(1,2) && superieur(1,3) then begin
  deplacer_sommet(1,4)
end;
if superieur(2,1) && superieur(2,3) then begin
  deplacer_sommet(2,4)
end;
if superieur(3,1) && superieur(3,2) then begin
  deplacer_sommet(3,4)
end ;;
```

n'est pas correct pour le problème posé dans la section 2.6

**Exercice 2-7**

Reprenez le problème de la section 2.6 en admettant qu'un ou plusieurs des trois premiers tas peut être vide. Si les trois tas sont vides, on ne déplace aucune carte.





## Chapitre 3

# Instructions conditionnellement répétées

### 3.1 Un problème

- État initial : Tas 1 un nombre quelconque de cartes, les autres tas vides.
- État final : tas 1, 3 et 4 vides, le tas 2 contient toutes les cartes du tas 1 initial

### 3.2 Algorithme

La résolution du problème nécessite de déplacer (une à une) toutes les cartes du tas 1 vers le tas 2. Ce qu'on peut décrire par

---

**Algorithme 3.1** Solution du problème

---

```
tant que le tas 1 n'est pas vide faire  
    déplacer la carte au sommet du tas 1 vers le tas 2  
fin tant que
```

---

### 3.3 Sortie d'une boucle **tant que**

Une boucle **tant que** se termine lorsque sa condition n'est plus satisfaite. Par conséquent à la sortie de la boucle, on est assuré que la condition est fausse

```
tant que condition faire  
    actions  
fin tant que  
{condition est fausse}
```

### 3.4 La boucle **tant que** en CAML

Comme bien des langages de programmation, CAML permet d'écrire des instructions conditionnellement répétées. La syntaxe de l'instruction **tant que** est décrite de manière générale ci-dessous.

**Syntaxe :** Boucle tant que

```
while < condition > do
  (* sequence d'instructions *)
done
```

**Exemple** traduction en CAML de l'algorithme décrit à la section 3.2.

```
while tas_non_vide(1) do
  deplacer_sommet(1,2)
done
```

Lorsqu'elle sera exécutée, cette instruction testera si le tas 1 est non vide. Si ce n'est pas le cas (c'est-à-dire si le tas 1 est vide) alors l'instruction est terminée, sinon (si le tas 1 n'est pas vide), une carte sera déplacée du tas numéro 1 vers le tas numéro 2. Ce déplacement effectué, la vacuité du tas numéro 1 sera à nouveau testée, et selon le cas l'instruction est terminée ou un déplacement est effectué. Et on poursuit ainsi tant que le tas 1 n'est pas vide.

### 3.5 Méthodologie

La conception d'une boucle **tant que** nécessite plusieurs points d'attention :

- la situation avant d'entrer dans la boucle (pré-condition) est-elle celle souhaitée ?  
Par exemple, l'instruction qui suit peut être source d'erreur.

```
while sommet_trefle(1) do
  deplacer_sommet(1,2)
done
```

En effet, la condition du **tant que** porte sur la couleur de la carte au sommet du tas 1. Pour cela on utilise la fonction **sommet\_trefle**. Mais pour être utilisée, celle-ci est nécessaire que le tas numéro 1 ne soit pas vide.

- la situation à la sortie de la boucle (post-condition) est-elle bien celle souhaitée ?
- la boucle ne risque-t-elle pas d'être infinie (problème de l'arrêt) ?

Par exemple, l'instruction qui peut conduire à une répétition infinie de déplacement d'une carte du tas 1 vers le tas 2, puis d'un retour de la carte déplacée vers son tas d'origine. En effet si le tas 1 n'est pas vide au moment d'exécuter cette instruction, la condition du **tant que** est satisfaite et l'action est exécutée, action qui aboutit à la même situation que celle de départ : le tas 1 ne sera jamais vide.

```
while tas_non_vide(1) do
  deplacer_sommet(1,2);
  deplacer_sommet(2,1)
done
```

## 3.6 Exercices

### Exercice 3-1

Décrivez les relations satisfaites entre  $a$ ,  $b$  et  $c$  à l'issue des boucles qui suivent

1.   **tant que**  $a = b$  **faire**  
      actions  
      **fin tant que**
2.   **tant que**  $a \leq b$  **faire**  
      actions  
      **fin tant que**
3.   **tant que**  $(a \neq b)$  ou  $(a > c)$  **faire**  
      actions  
      **fin tant que**

### Exercice 3-2

Les exercices de manipulation de cartes.



# Chapitre 4

## Constantes, variables

### 4.1 Types de données

En programmation, les données manipulées peuvent être de natures différentes : nombres entiers, nombres réels, chaînes de caractères, booléens, . . . . Ces différentes natures de données sont appelées *types de données*.

Le langage CAML dispose d'un certain nombre de types prédéfinis.

#### 4.1.1 Les booléens

Le type `bool` est le type des expressions booléennes permettant d'écrire les conditions des instructions **if** et **while**. Les données de ce type ne peuvent prendre que deux valeurs.

**Nom :** `bool`

**Ensemble des valeurs :** `true`, `false`

**Littéraux :** `true` et `false`

**Opérateurs :** `&&`, `||`, `not`

**Exemple 1 :**

```
# true;;
- : bool = true
# not(true);;
- : bool = false
# true || false;;
- : bool = true
```

#### 4.1.2 Les nombres entiers

Le type `int` est le type des expressions numériques entières.

**Nom :** `int`

**Ensemble des valeurs :** le type `int` est limité aux entiers appartenant à un intervalle  $\llbracket e_{min}, e_{max} \rrbracket$ , et il y a donc un plus petit et un plus grand entier. Cet intervalle dépend de l'architecture du processeur de l'ordinateur utilisé. Sur des architectures 32 bits, on a

$$\llbracket e_{min}, e_{max} \rrbracket = \llbracket -2^{30}, 2^{30} - 1 \rrbracket = \llbracket -1073741824, 1073741823 \rrbracket.$$

Sur des architectures 64 bits, on a

$$\llbracket e_{min}, e_{max} \rrbracket = \llbracket -2^{62}, 2^{62} - 1 \rrbracket = \llbracket -4611686018427387904, 4611686018427387903 \rrbracket.$$

**Littéraux :** les entiers dans leur forme écrite usuelle. Par exemple : 24.

**Constantes prédéfinies :** les constantes `min_int` et `max_int` donnent les valeurs du plus petit ( $e_{min}$ ) et du plus grand ( $e_{max}$ ) entier.

**Opérateurs arithmétiques :** +, -, \*, /, `mod`

**Opérateurs de comparaison :** =, <>, <, <=, >, >=

**Remarque :** Les calculs arithmétiques sur les nombres de type `int` s'effectue modulo  $e_{max} - e_{min} + 1$ .

**Exemple 2 :**

La session ci-dessous est effectuée sur une machine 32 bits.

```
# (1+2)*3;;
- : int = 9
# 9 / 2;;
- : int = 4
# 9 mod 5;;
- : int = 4
# max_int;;
- : int = 1073741823
# min_int;;
- : int = -1073741824
# max_int + 1;;
- : int = -1073741824
# min_int - 1;;
- : int = 1073741823
```

### 4.1.3 Les nombres flottants

**Nom :** `float`

### 4.1.4 Types définis par le module `Cartes`

Le module `Cartes` définit deux types `couleur` et `numero_tas`<sup>1</sup>.

**Nom :** `couleur`

**Ensemble des valeurs :** Les données de ce type ne peuvent prendre que quatre valeurs notées : CARREAU, TREFLE, PIQUE et COEUR.

**Nom :** `numero_tas`

**Ensemble des valeurs :** Les données de ce type ne peuvent prendre que quatre valeurs : 1, 2, 3 et 4.

---

1. Le nom complet de ces types est en fait `Cartes.couleur` et `Cartes.numero_tas`. Dans le texte du poly nous n'employons que la notation raccourcie.

## 4.2 Variables

### 4.2.1 Notion de variable

Les variables servent à nommer des valeurs.

Elles sont caractérisées par leur nom ou *identificateur*, leur *type* et leur *valeur*.

### 4.2.2 Déclaration simple de variables

La déclaration des variables se fait à l'aide du mot réservé (ou mot-clé) **let**.

**Syntaxe :** Déclaration d'une variable

```
let < identificateur > = < expression >
```

où <*identificateur*> est le nom de la variable, et <*expression*> l'expression qui définit sa valeur.

#### Exemple 3 :

Voici la déclaration d'une variable nommée **a**, de type **int** et de valeur **12**.

```
# let a = 12;;
val a : int = 12
# a;;
- : int = 12
# 2 * a;;
- : int = 24
```

L'expression qui définit la valeur d'une variable peut faire référence à d'autres variables à condition que celles-ci aient été préalablement déclarées.

#### Exemple 4 :

À la suite de la déclaration de la variable **a** qui précède, on peut déclarer une nouvelle variable **b** comme ci-dessous

```
# let b = 2*a + 1;;
val b : int = 25
```

### 4.2.3 Identificateurs de variables

Les variables sont nommées par un identificateur. En CAML, les identificateurs de variables doivent obligatoirement commencer par une lettre minuscule ou un blanc souligné (\_) et peuvent être complétés par n'importe quelle quantité de lettres minuscules ou majuscules non accentuées, ou chiffres, ou blancs soulignés.

#### Exemple 5 :

Le tableau ci-dessous montre quelques exemples d'identificateurs de variables corrects et incorrects.

Identificateurs corrects	Identificateurs incorrects
<b>a</b>	<b>A</b>
<b>a1</b>	<b>1a</b>
<b>a_b</b>	<b>a-b</b>
<b>_a</b>	<b>-a</b>
<b>deplacer_sommet</b>	<b>deplacer-sommet</b>

En plus de ces règles, certains mots ne peuvent pas être utilisés comme identificateurs : ce sont les mots réservés ou mots-clés du langage (cf annexe C.2).

#### 4.2.4 Environnement

L'ensemble des variables définies dans un contexte de calcul s'appelle un *environnement*.

Lorsqu'on déclare une variable, on augmente l'environnement courant de cette nouvelle variable. Ainsi si dans un certain environnement  $E_0$  on déclare la variable

```
# let a = 12 ;;
val a : int = 12
```

l'environnement qui suit cette déclaration est augmenté d'une variable nommée **a** et de valeur 12, et l'environnement est maintenant

$$E_1 = \langle a, 12 \rangle : E_0.$$

Après la déclaration

```
# let b = 5 ;;
val b : int = 5
```

l'environnement est

$$E_2 = \langle b, 5 \rangle : E_1 = \langle b, 5 \rangle : \langle a, 12 \rangle : E_0.$$

On voit donc que la déclaration de variables peut augmenter l'environnement. Mais rien ne peut faire diminuer l'environnement : une variable déclarée le reste pour toute la durée de la session.

#### 4.2.5 Évaluation d'une expression

Lorsqu'une expression contient une ou plusieurs variables, pour chacune de ces variables on cherche dans l'environnement courant la variable de même nom la plus récemment déclarée, et on la remplace par la valeur correspondante.

Par exemple dans l'environnement  $E_2$  précédemment décrit, l'expression **a\*b** est transformée en **12\*5** qui finalement donne la valeur 60.

$$a*b \Rightarrow 12*5 \Rightarrow 60.$$

#### 4.2.6 Masquage d'une variable

Nous l'avons vu un environnement ne peut qu'augmenter. Que se passe-t-il lorsque nous déclarons une variable dont le nom est identique à celui d'une variable déjà déclarée? Que se passe-t-il si on déclare une nouvelle variable nommée **a** dans l'environnement  $E_2$ ?

Et bien, rien de neuf. Une nouvelle variable est ajoutée à l'environnement. Par exemple, si dans l'environnement  $E_2$  on fait la déclaration

```
# let a = 9 ;;
val a : int = 9
```

l'environnement devient

$$E_3 = \langle a, 9 \rangle : E_2 = \langle a, 9 \rangle : \langle b, 5 \rangle : \langle a, 12 \rangle : E_0.$$



On constate que l'environnement  $E_3$  contient deux variables de même nom **a** : l'une valant 12 qui dans le temps a été la première à être déclarée, l'autre valant 9 qui est la plus récente.

Quelle est alors la valeur de l'expression **a\*b** dans l'environnement  $E_3$  ? Dans l'expression, on remplace la variable **b** par sa valeur 5, mais qu'en est-il pour **a** ? 9 ou 12 ? La règle d'évaluation dit de choisir la valeur de la variable la plus récemment déclarée, autrement dit 9. Par conséquent, dans l'environnement  $E_3$  l'expression **a\*b** vaut 45.

$$a*b \Rightarrow 9*5 \Rightarrow 45.$$

Dans l'environnement  $E_3$ , toute référence à la variable **a** est une référence à la variable  $\langle a, 9 \rangle$ . L'autre variable  $\langle a, 12 \rangle$  est *masquée* par  $\langle a, 9 \rangle$ .

### 4.2.7 Déclaration simultanée de variables

Plusieurs variables peuvent être définies simultanément dans la même phrase **let** en utilisant le mot clé **and**.

**Syntaxe :** Déclaration simultanée de plusieurs variables

```
let < identificateur1 > = < expression1 >
and < identificateur2 > = < expression2 >
...
and < identificateurn > = < expressionn >
```

Les expressions  $\langle \text{expression}_i \rangle$  sont alors toutes évaluées dans l'environnement dans lequel la phrase de déclaration est exprimée.

**Exemple 6 :**

```
# let a = 1
  and b = 2;;
val a : int = 1
val b : int = 2
# a + b ;;
- : int = 3
```



La déclaration simultanée de plusieurs variables n'est pas équivalente à la déclaration séquentielle. Par exemple, on peut tout à fait effectuer les deux déclarations successives suivantes

```
# let c = 1. ;;
val c : float = 1.
# let d = c +. 2. ;;
val d : float = 3.
```

mais il n'est pas possible de demander la déclaration simultanée qui suit si aucune déclaration préalable d'une variable **e** de type **float** n'a été faite.

```
# let e = 1.
  and f = e +. 2. ;;
Unbound value e
```

En effet dans l'environnement d'évaluation de cette déclaration, la variable **e** n'existe pas, et il n'est donc pas possible d'évaluer l'expression **e +. 2.**

Mais si une déclaration préalable a été faite il n'y a aucun problème

```
# let e = -3.;;
val e : float = -3.
# let e = 1.
    and f = e +. 2. ;;
val e : float = 1.
val f : float = -1.
```

À noter que dans l'environnement obtenu après ces déclarations, la variable **e** vaut 1., et **f** vaut -1., autrement dit **f** a été calculée avec la valeur de **e** dans l'environnement précédent.

#### 4.2.8 Variables locales à une expression

Nous l'avons vu, l'environnement courant des variables déclarées ne peut que croître. Cela peut poser deux problèmes :

- un accroissement inutile de l'environnement avec des variables qui ne servent pas, et qui peut provoquer un encombrement, voire une saturation de la mémoire ;
- un masquage de certaines variables lorsqu'on utilise un même nom.

Bien souvent, on éprouve le besoin de déclarer une variable pour faciliter l'écriture d'une expression et/ou optimiser son évaluation.

Par exemple, si nous souhaitons calculer l'expression

$$\sqrt{a^2 + b^2} \times \frac{1 + \sqrt{a^2 + b^2}}{2 + \sqrt{a^2 + b^2}}$$

on pourrait écrire (en supposant que dans l'environnement courant il y ait deux variables **a** et **b** de valeurs flottantes)

```
sqrt(a**2. +. b**2.) *.
(1. +. sqrt(a**2. +. b**2.)) /.
(2. +. sqrt(a**2. +. b**2.))
```

mais cela n'est évidemment pas satisfaisant parce que d'une part l'expression n'est pas facilement lisible, et d'autre part dans son évaluation, la sous-expression  $\sqrt{a^2 + b^2}$  va être calculée trois fois.

Pour mener des calculs, il arrive souvent que, pour ne pas noyer le propos par de longues formules, on utilise des notations pour les simplifier. On « pose » que tel symbole vaut temporairement telle valeur, et ensuite on utilise ce symbole en lieu et place de celle-ci.

Sur notre exemple cela pourrait donner : soit  $\alpha = \sqrt{a^2 + b^2}$  dans l'expression

$$\alpha \times \frac{1 + \alpha}{2 + \alpha}$$

dont une traduction possible en OBJECTIVE CAML est

```
let alpha = sqrt(a**2. +. b**2.)

alpha *. (1. +. alpha) /. (2. +. alpha)
```

Cette façon de mener le calcul remédie aux deux défauts signalés ci-dessus :

- le code est beaucoup plus lisible ;
- la sous-expression  $\sqrt{a^2 + b^2}$  n'est calculée qu'une seule fois, et donc on gagne en efficacité.

Cependant, en procédant de la sorte on a introduit dans l'environnement courant une nouvelle variable dont l'intérêt est limité au seul calcul de notre expression. Par la suite la variable  $\alpha$  ne sera probablement plus utilisée mais persistera dans l'environnement : c'est une pollution.

Il existe une solution pour éviter la pollution de l'environnement par des variables d'intérêt temporaire : c'est la notion de *variable locale* à une expression.

Ce qui manque dans notre exemple c'est la traduction de posons  $\alpha = \dots$  **dans** l'expression  $\dots$ . Il ne faut pas déclarer une variable, puis calculer une expression, mais déclarer une variable à utiliser **dans** une expression. En OBJECTIVE CAML, cela se traduit avec la forme **let**  $\dots$  **in**  $\dots$ . Notre exemple devient alors

```
let alpha = sqrt(a**2. +. b**2.)
in
  alpha *. (1. +. alpha) /. (2. +. alpha)
```

Avec une telle déclaration, la variable **alpha** ne vient s'ajouter à l'environnement courant que pour la durée de l'évaluation de l'expression qui suit. Sitôt cette évaluation terminée, elle est supprimée, et n'existe donc pas dans l'environnement courant à l'issue du calcul. Il n'y a aucune différence entre l'environnement avant et après évaluation. Le problème de pollution d'environnement est réglé.

La syntaxe générale de déclaration de variables locales est donnée ci-dessous.

**Syntaxe :** Déclaration de variables locales à une expression

```
let <identificateur1> = <expression1>
and <identificateur2> = <expression2>
...
in
  <expression>
```

Les variables non locales déclarées par une simple forme **let** sont appelées variables *globales* par opposition à locales.

#### 4.2.9 Remarque sur la forme **let**

En OBJECTIVE CAML, la forme **let**  $\dots$  n'est pas une expression et a fortiori n'est pas une instruction.

Utilisée dans une séquence d'instructions elle ne donne pas l'effet auquel on pourrait penser. Dans la séquence qui suit

```
# let a = 1 ;
  a + 1;;
Warning S: this expression should have type unit.
Unbound value a
```


on constate un avertissement sur le fait que la première partie n'a pas le type **unit**, et un message d'erreur qui montre que dans l'environnement courant la variable **a** n'est liée à aucune valeur, et ceci malgré la forme la déclaration qui précède.

Utilisée dans une expression conditionnelle, elle provoque un message d'erreur de syntaxe.

```
# if true then begin
  let a = 1;
  print_string ("Fini")
end ;;
```

**Syntax error**

De même dans les instructions répétées (boucle **while** ou **for**).

 En conclusion, une déclaration globale de variable ne peut être encapsulée dans une séquence d'instructions, dans une expression conditionnelle ou dans une boucle.

En revanche, la forme **let ... in ...** est une expression. À ce titre elle peut être encapsulée.

```
# let a = 1 in print_int a ;
  print_newline () ;;
1
- : unit = ()
# if true then begin
    let a = 1 in
      print_int (a+1)
    end ;;
2- : unit = ()
```

## 4.3 Variables mutables

### 4.3.1 Motivation

#### Un problème

- État initial : Tas 1 un nombre quelconque de cartes, les autres tas vides.
- État final : peu importe
- **But** : afficher le nombre de cartes situées initialement sur le tas 1.

#### Algorithme

Une solution du problème consiste à déplacer (une à une) toutes les cartes du tas 1 vers un autre tas (le tas 2 par exemple), et à compter chaque déplacement effectué.

---

**Algorithme 4.1** Compter les cartes du tas 1

---

```
mise à zéro du compteur
tant que les tas 1 n'est pas vide faire
  déplacer une carte du tas 1 vers le tas 2
  ajouter 1 au compteur
fin tant que
```

---

Comme on peut le voir, dans cet algorithme la valeur du compteur évolue durant son exécution.

Une variable dont la valeur peut évoluer durant l'exécution d'un programme est appelée variable *mutable*. Par opposition, nous nommerons parfois les variables non mutables des *constantes*.

### 4.3.2 Déclaration d'une variable mutable

La déclaration d'une variable mutable est analogue à celle de toute variable, si ce n'est l'emploi du mot-clé **ref**.

**Syntaxe :** Déclaration d'une variable mutable

```
let < identificateur > = ref < expression >
```

**Exemple 7 :**

Déclaration d'une variable mutable **compteur** initialisée à 0.

```
# let compteur = ref 0 ;;
val compteur : int ref = {contents = 0}
```

Comme on peut le voir, cette variable n'est pas de type **int**, mais de type **int ref**. Et sa valeur est {contents = 0}.

```
# compteur ;;
- : int ref = {contents = 0}
```

Comme **compteur** n'est pas de type **int**, on ne peut pas l'utiliser tel quel dans le contexte d'une expression où on attend un **int**.

```
# compteur + 1;;
This expression has type int ref but is here used with type
```

**4.3.3 Référence à la valeur d'une variable mutable**

Pour faire référence à la valeur d'une variable mutable dans une expression, on utilise le préfixe **!** devant le nom de la variable.

**Exemple 8 :**

```
# !compteur ;;
- : int = 0
# !compteur + 1;;
- : int = 1
# !compteur;;
- : int = 0
```

Notez qu'après l'évaluation de la deuxième expression, la variable **compteur** a gardé la valeur 0. Pour modifier la valeur d'une variable mutable, il faut utiliser l'instruction d'*affectation*.

**4.3.4 Affectation**

L'*affectation* est une instruction qui permet de changer la valeur d'une variable mutable. La syntaxe de cette instruction est

**Syntaxe :** Affectation d'une valeur à une variable mutable

```
< identificateur > := < expression >
```

Le symbole **:=** signifie que la valeur *v* de l'expression < expression > est affectée à la variable désignée par l'identificateur < identificateur >. Après l'exécution de cette instruction la variable < identificateur > vaut *v*.

Comme toute instruction, le résultat d'une affectation est la valeur **()** de type **unit**.

**Exemples :**

1. Pour mettre à zéro la variable `compteur` précédemment déclarée

```
(* compteur = xxx *)
compteur := 0 ;
(* compteur = 0 *)
```

2. ajouter 1 au compteur

```
(* compteur = n *)
compteur := !compteur + 1 ;
(* compteur = n + 1 *)
```

**Remarques :**

1. Attention à ne pas confondre le symbole `:=` utilisé en CAML pour l'instruction d'affectation, avec le symbole `=` utilisé comme opérateur de comparaison, ainsi que dans la déclaration de variables.
2. En programmation, on est souvent amené à utiliser des instructions du type

```
a := !a + 1
```

qui augmentent de 1 la valeur d'une variable (mutable). Cette opération s'appelle *incrément* et on dit qu'on *incrément* une variable.

La *décrément* est la diminution de la valeur d'une variable.

```
a := !a - 1
```

3.  On peut penser que le code suivant est équivalent à une incrément.

```
let a = a+1
```

Il n'en est rien du tout ! Il s'agit d'une déclaration d'une **nouvelle** variable **a** qui vient masquer la précédente. De plus, on ne peut pas utiliser cette forme **let** dans une boucle. Par exemple, on peut écrire

```
# let a = ref 0 ;;
val a : int ref = {contents = 0}
# while !a < 5 do
  a := !a + 1;
  print_int !a;
  print_newline ()
done ;;
1
2
3
4
5
- : unit = ()
```

mais on ne peut pas écrire

```
# let a = 0 ;;
val a : int ref = {contents = 0}
# while a < 5 do
  let a = a + 1;
  print_int (a);
  print_newline () )
done ;;
Syntax error
```

4. L'affectation est une opération autorisée sur les variables mutables uniquement. Elle ne l'est pas sur les variables non mutables (ou constantes).

```
# let a = 1 ;;
val a : int = 1
# a := 2;;
This expression has type int but is here used with type 'a ref
```

## 4.4 Afficher des données

En CAML, lorsqu'on veut afficher des données, on utilise les instructions `print_int` pour les entiers, `print_float` pour les flottants.

Exemple 9 :

```
# print_int(1234);;
1234- : unit = ()
# let a = 1234;;
val a : int = 1234
# print_int(a*2);;
2468- : unit = ()
# print_int(1234.);;
This expression has type float but is here used with type int
# print_float(1234.);;
1234.- : unit = ()
```

Comme on peut le voir ces instructions impriment la valeur du paramètre sans passer à la ligne. Cela peut poser problème lorsqu'on veut afficher deux nombres successifs.

Exemple 10 :

```
# begin
  print_int(1);
  print_int(2)
end ;;
12- : unit = ()
```

L'instruction sans paramètre `print_newline` provoque un passage à la ligne.

Exemple 11 :

```
# begin
  print_int(1);
  print_newline();
  print_int(2);
  print_newline()
end ;;
1
2
- : unit = ()
```

L'instruction `print_string` imprime une chaîne de caractères.

Exemple 12 :

```
# let a = 12
  and b = 21
  in
  begin
    print_int(a);
    print_string("_+_");
    print_int(b);
    print_string("_=_");
    print_int(a+b);
    print_newline()
  end ;;
12 + 21 = 33
- : unit = ()
```

## 4.5 Le programme solution du problème 4.3.1

```
(* compter le nombre de cartes sur le tas 1 *)
let cpt = ref 0
in
begin
  init_tas(1, "[T]");
  init_tas(2, "");
  init_tas(3, "");
  init_tas(4, "");

  while tas_non_vide(1) do
    placersommet(1,2);
    cpt := !cpt + 1
  done;
  print_string("Nombre_de_cartes_initialement_sur_le_tas_1:_");
  print_int(!cpt);
  print_newline()
end
```



## 4.6 Exercices

### Exercice 4-1 Déclarations de variables

Décrivez les valeurs des variables déclarées à l'issue des deux sessions suivantes :

<pre>let a = 1 ;; let a = 2 ;; let b = 2*a + 1 ;;</pre>	<pre>let a = 1 ;; let a = 2 and b = 2*a + 1 ;;</pre>
---	--

### Exercice 4-2 Échange de variables

Donnez une séquence d'instructions qui échange les valeurs de deux variables mutables de type `int`.

### Exercice 4-3 Calcul de la valeur d'une fonction polynomiale

Soit  $f(x) = x^4 - 3x^3 - 2x^2 + x + 1$ .

Réalisez un programme qui affiche la valeur de  $f(x)$  lorsque  $x = 13$ . Concevez votre programme pour qu'il soit facile de le modifier pour le calcul de  $f(x)$  pour d'autres valeurs de  $x$ .

Cette dernière solution suit le schéma de Horner d'évaluation d'un polynôme.

### Exercice 4-4 Avec les cartes

Réalisez des programmes qui, à partir de la situation initiale

**Situation initiale :**

Tas 1 : "[K+T+P+C]"    Tas 2 : ""

Tas 3 : ""    Tas 4 : ""

1. calcule le nombre de trèfles,
2. calcule le nombre de cartes de couleur noire,
3. répartit équitablement les cartes rouges sur les tas 3 et 4, et les cartes noires sur les tas 1 et 2.



# Chapitre 5

## La boucle pour

### 5.1 La boucle Pour

On peut calculer la somme des entiers de 1 à  $n$  à l'aide d'une structure itérative **tant que** et de deux variables,  $i$  pour énumérer les entiers successifs de 1 à  $n$ , et  $S$  pour les cumuler.

---

**Algorithme 5.1** Calcul de la somme des  $n$  premiers nombres entiers.

---

**Entrée :**  $n$  un entier positif ou nul

**Sortie :**  $s = \sum_{k=1}^n k$

initialiser  $S$  à 0

initialiser  $i$  à 1

{on a bien  $S = \sum_{k=1}^{i-1} k$ }

**tant que**  $i \leq n$  **faire**

ajouter  $i$  à  $S$

incrémenter  $i$

{notons qu'on a toujours  $S = \sum_{k=1}^{i-1} k$ }

**fin tant que**

{la même égalité est satisfaite avec  $i = n + 1$ }

**renvoyer**  $s$

---

Compte tenu de la condition, il est possible de dire avant l'exécution de cet algorithme que la séquence d'instructions contenue dans la boucle **tant que** sera effectuée  $n$  fois.

De manière plus générale, si un algorithme s'écrit de cette façon

initialiser  $i$  à  $a$

**tant que**  $i \leq b$  **faire**

séquence d'instructions ne modifiant pas  $i$

incrémenter  $i$

**fin tant que**

et que la séquence d'instructions précédant l'incrément de la variable  $i$  ne modifie pas la valeur de cette variable, alors le nombre de fois que la séquence d'instructions sera exécutée est égal à  $b - a + 1$ .

De telles itérations dont le nombre d'étapes est déterminée par une variable qui parcourt l'ensemble des valeurs comprises entre deux bornes est appelée une *boucle pour*, et la variable  $i$  est appelée *indice* de la boucle.

On peut réécrire le calcul de la somme des entiers compris entre 1 et  $n$  en utilisant une boucle *pour*.

---

**Algorithme 5.2** Calcul de la somme des  $n$  premiers nombres entiers avec une boucle *pour*

---

**Entrée :**  $n$  un entier positif ou nul

**Sortie :**  $s = \sum_{k=1}^n k$

initialiser  $S$  à 0

**pour**  $i$  variant de 1 à  $n$  **faire**

ajouter  $i$  à  $S$

{on a  $S = \sum_{k=1}^i k$ }

**fin pour**

**renvoyer**  $s$

---

## 5.2 La boucle Pour en CAML

### 5.2.1 Syntaxe de la boucle pour

Toute séquence de deux instructions de la forme

```
let i = ref a
in
while !i <= b do
  (* sequence d'instructions *)
  i := !i + 1
done
```

peut être remplacée par une boucle *pour*

**Syntaxe :** Boucle pour


```
for i = a to b do
  (* sequence d'instructions *)
done
```

**Exemple 1 :**

Calcul de la somme des entiers de 1 à 100

```
# let s = ref 0
in
  for i = 1 to 100 do
    s := !s + i
  done;
  !s ;;
- : int = 5050
```

### 5.2.2 Remarque sur l'indice de boucle

 L'indice d'une boucle **for** en CAML est une variable locale à la boucle qui n'a pas besoin d'être déclarée<sup>1</sup>. Cette variable n'est définie que dans l'environnement de la boucle. Elle n'existe pas dans l'environnement englobant.

```
# for i = 1 to 5 do
  print_int(i);
  print_newline()
done;
print_int(i);;
Unbound value i
```

De plus, ce n'est pas une variable mutable, et on ne peut donc pas modifier la valeur de cet indice dans la séquence d'instructions du corps de la boucle.

### 5.2.3 Boucle pour à indice décroissant

Toute séquence de deux instructions de la forme

```
let i = ref a
in
while !i >= b do
  (* sequence d'instructions *)
  i := !i - 1
done
```

peut être remplacée par une boucle **pour**

**Syntaxe :** Boucle pour à indice décroissant

```
for i = a downto b do
  (* sequence d'instructions *)
done
```

**Exemple 2 :**

Calcul de la somme des entiers de 1 à 100

```
# let s = ref 0
in
  for i = 100 downto 1 do
    s := !s + i
  done;
  !s ;;
- : int = 5050
```

---

1. Ce n'est pas le cas dans tous les langages de programmation, comme par exemple C ou PASCAL

### 5.3 Suites récurrentes

Soit  $(u_n)_{n \in \mathbb{N}}$  une suite de nombres réels définie par son premier terme  $u_0 = a$  ( $a \in \mathbb{R}$ ) et pour tout  $n \geq 0$  une relation de récurrence de la forme

$$u_{n+1} = f(u_n)$$

où  $f$  est une fonction réelle d'une variable réelle.

Par exemple, on peut considérer la suite de réels définie par

$$\begin{aligned} u_0 &= 1 \\ u_{n+1} &= \frac{u_n}{2} + \frac{1}{u_n} \end{aligned}$$

on a ici  $a = 1$  et  $f(x) = \frac{x}{2} + \frac{1}{x}$

#### 5.3.1 Calcul du terme d'indice $n$

On veut calculer le terme  $u_n$  lorsqu'est donné l'indice  $n$ .

---

**Algorithme 5.3** Calcul du terme d'indice  $n$  d'une suite récurrente

---

**Entrée :**  $n$  un nombre réel,  $a$  un réel et  $f : \mathbb{R} \rightarrow \mathbb{R}$  une fonction.

**Sortie :** le terme  $u_n$  de la suite

```

 $u_0 \leftarrow a$ 
pour  $i$  variant de 1 à  $n$  faire
     $u_i \leftarrow f(u_{i-1})$ 
fin pour
renvoyer  $u_n$ 
```

---

Pour réaliser cet algorithme en CAML, il suffit d'utiliser une seule variable pour mémoriser les valeurs successives des termes de la suite.

```

(* on suppose declarees ici trois variables
- a de type float
- f de type float -> float
- n de type int
*)
let u = ref a
in
begin
  for i = 1 to n do
    u := f(!u)
  done;
  !u
end
```

#### 5.3.2 Calcul et affichage des termes d'indice 0 à $n$

On veut afficher la valeur de tous les termes de la suite  $(u_n)$  dont les indices sont compris entre 0 et  $n$ .

Il suffit de calculer successivement chaque terme de la suite et afficher immédiatement sa valeur.

```
(* on suppose declarees ici trois variables
- a de type float
- f de type float -> float
- n de type int
*)
let u = ref a
in
begin
  print_float(!u);
  print_newline();
  for i = 1 to n do
    u := f(!u);
    print_float(!u);
    print_newline()
  done
end
```

### 5.3.3 Calcul du premier terme satisfaisant une condition

On peut aussi vouloir chercher le premier terme de la suite qui satisfait une condition donnée. Cette condition peut porter sur le dernier terme calculé, ou bien sur plusieurs termes déjà calculés.

---

#### Algorithme 5.4 Calcul du premier terme satisfaisant une condition

---

```
u ← a
tant que u ne satisfait pas la condition voulue faire
  on calcule le terme suivant u ← f(u)
fin tant que
renvoyer u
```

---

**Remarque :** si aucun terme de la suite ne satisfait la condition donnée, la boucle est infinie (le programme ne s'arrête pas). Aussi avant de concevoir de tels programmes est-il important de s'assurer qu'au moins un terme de la suite vérifie la condition.

### 5.3.4 Autres suites récurrentes

#### Suites récurrentes d'ordre plus élevé

**ordre 2 :** la suite de Fibonacci définie par ses deux premiers termes et une relation de récurrence d'ordre 2

$$\begin{aligned} u_0 &= 0 \\ u_1 &= 1 \\ u_{n+2} &= u_{n+1} + u_n \quad \forall n \in \mathbb{N} \end{aligned}$$

Facile à programmer.

**ordre total :** la suite des nombres de Catalan définie par son premier terme et une relation de récurrence s'appuyant sur tous les termes qui précèdent

$$\begin{aligned}u_0 &= 1 \\u_{n+1} &= \sum_{k=0}^n u_k u_{n-k} \quad \forall n \in \mathbb{N}\end{aligned}$$

Difficile à programmer sans tableaux (cf cours d'APII).



## 5.4 Exercices

### Exercice 5-1 `downto` n'est pas indispensable

**Question 1** À l'aide d'une boucle à indices croissant, réécrivez l'instruction qui suit de sorte que l'affichage soit identique.

```
for i = n downto 1 do
  print_int(i);
  print_newline()
done
```

**Question 2** En supposant que l'instruction `action(i)` accomplisse une certaine tâche qui dépend de  $i$ , comment réécrire l'instruction

```
for i = b downto a do
  action(i)
done
```

avec une boucle à indices croissant ?

### Exercice 5-2 Tables de multiplication

Réalisez un programme qui affiche une table de multiplication par un nombre donné. Une trace d'exécution du programme demandé est donnée ci-dessous pour la table par 7 :

```
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
...
10 x 7 = 70
```

### Exercice 5-3 Calcul de puissances

Réalisez un programme qui calcule la valeur de l'entier  $a^n$  pour deux entiers  $a$  et  $n$ .

### Exercice 5-4 Calcul de factorielles

Réalisez un programme qui calcule l'entier  $n!$ ,  $n$  étant donné.

### Exercice 5-5

Boucles imbriquées pour stars académiques

#### Question 1

Réalisez un programme qui affiche à l'écran une ligne de  $n$  étoiles, l'entier  $n$  étant donné.

L'affichage du programme demandé est donné ci-dessous pour  $n = 5$  :

```
*****
```

**Question 2** Réalisez un programme qui affiche à l'écran un carré de  $n \times n$  étoiles, l'entier  $n$  étant donné.

L'affichage du programme demandé est donné ci-dessous pour  $n = 5$  :

```
*****
*****
*****
*****
*****
```

**Question 3** Réalisez un programme qui affiche à l'écran un triangle (isocèle) de hauteur  $n$ , l'entier  $n$  étant donné.

L'affichage du programme demandé est donné ci-dessous pour  $n = 5$  :

```
*
**
***
****
*****
```

**Question 4** Réalisez un programme qui affiche à l'écran un triangle (isocèle) de hauteur  $n$  pointé en bas, l'entier  $n$  étant donné.

Une trace d'exécution du programme demandé est donnée ci-dessous :

```
*****
****
***
**
*
```

### Exercice 5-6 Calcul de la suite de Heron

La suite de Heron est la suite récurrente de nombres réels définie par

$$\begin{aligned} u_0 &= a \\ u_{n+1} &= \frac{u_n}{2} + \frac{B}{2u_n} \end{aligned}$$

où  $a$  et  $B$  sont deux réels positifs.

**Question 1** Programmez le calcul du terme d'indice  $n$  de cette suite avec  $a = 1$  et  $B = 2$ .

**Question 2** Modifiez votre programme pour qu'à l'exécution on obtienne un affichage de tous les termes demandés, à raison d'un par ligne.

Par exemple, l'affichage produit aura la forme qui suit pour  $a = 1$ ,  $B = 2$  et  $n = 5$ .

```
u(1) = 1.5
u(2) = 1.41666666667
u(3) = 1.41421568627
u(4) = 1.41421356237
u(5) = 1.41421356237
```

**Question 3** Utilisez votre programme pour différentes valeurs de  $a$  et  $B$ .

**Exercice 5-7** *Suite de Fibonacci*

**Question 1** Programmez le calcul des premiers termes de la suite de Fibonacci. Faites le en utilisant le type `int`, puis le type `float`. Calculez les termes jusqu'à l'indice 50. Comparez les résultats selon le type utilisé.

**Question 2** Cette suite est croissante et tend vers  $+\infty$  lorsque  $n$  tend vers  $+\infty$ . Écrivez une fonction qui calcule l'indice du premier terme supérieur ou égal à un réel positif  $s$  passé en paramètre.



# Chapitre 6

## Les fonctions

### 6.1 Les fonctions

Nous avons déjà rencontré et utilisé plusieurs fonctions : `tas_non_vide`, `couleur_sommet`, `superieur` du module `Cartes`, ainsi que `not`, et bien d'autres encore.

Le but de ce chapitre est de voir comment créer nos propres fonctions.

Une *fonction* permet d'*abstraire* et de *nommer* un calcul ou une expression. Une fois définie, une fonction peut être utilisée pour produire de nouvelles *expressions* du langage.

Contrairement aux instructions, un appel à une fonction ne doit pas modifier l'état courant de l'environnement.

#### 6.1.1 Spécification d'une fonction

Spécifier une fonction, c'est

- choisir un identificateur pour la nommer ;
- préciser le nombre et le type de ses paramètres, et les nommer ;
- indiquer les conditions d'utilisation (CU) que doivent vérifier les paramètres lors d'un appel à la fonction ;
- indiquer le type de la valeur qu'elle renvoie ;
- et indiquer quelle est la relation entre la valeur renvoyée et les paramètres qui lui sont passés.

##### Exemple 1 :

La fonction prédéfinie donnant la négation d'un booléen en CAML est une fonction nommée `not`, qui opère sur un paramètre de type `bool`, dont la valeur renvoyée est aussi de type `bool` et pour laquelle il n'y a aucune contrainte d'utilisation. Nous résumons tout cela par la notation :

$$\begin{array}{lcl} \text{not} : \text{bool} & \longrightarrow & \text{bool} \\ a & \longmapsto & \neg a \end{array} .$$

##### Exemple 2 :

Un autre exemple est la fonction du module `Cartes` qui permet de comparer les hauteurs des cartes situées au sommet de deux tas. Cette fonction est nommée `superieur`, elle prend deux paramètres de type `Cartes.numero_tas`, et renvoie une valeur de type `bool`. Elle possède aussi

une contrainte d'utilisation qui exige qu'aucun des deux tas passés en paramètre ne soit vide. Tout cela est résumé par la notation :

$$\begin{array}{ccc} \text{superieur} : \text{numero\_tas} \times \text{numero\_tas} & \longrightarrow & \text{bool} \\ & & \left\{ \begin{array}{ll} \text{vrai} & \text{si la carte au sommet du tas } n \\ & \text{a une valeur strictement plus} \\ & \text{grande que celle du tas } p \\ \text{faux} & \text{sinon} \end{array} \right. \\ n, p & \longmapsto & \end{array}$$

CU : les tas  $n$  et  $p$  ne doivent pas être vides.

### 6.1.2 Déclaration d'une fonction en CAML

Contrairement à d'autres langages tels que C, ADA ou PASCAL, en CAML il n'est pas nécessaire d'apprendre de nouvelle forme syntaxique pour déclarer une nouvelle fonction. On utilise la même forme **let** que pour la déclaration des variables.

**Syntaxe :** Déclaration d'une fonction

```
let <nom> (<liste parametres>) =
  (* expression simple ou sequence d'instructions
    definissant la fonction *)
```

#### Exemple 3 :

Soit à écrire le prédicat **rouge** dont la spécification est

$$\begin{array}{ccc} \text{rouge} : \text{numero\_tas} & \longrightarrow & \text{bool} \\ & & \left\{ \begin{array}{ll} \text{vrai} & \text{si la carte au sommet du tas } n \\ & \text{est rouge} \\ \text{faux} & \text{sinon} \end{array} \right. \\ n & \longmapsto & \end{array}$$

CU : le tas  $n$  ne doit pas être vide.

Le code en CAML pour cette fonction s'écrit tout simplement

```
let rouge(n) =
  sommet_coeur(n) || sommet_carreau(n)
```

Comme on peut le constater, en CAML il n'est pas nécessaire au programmeur de préciser le type des paramètres et celui de la valeur renvoyée, contrairement au cas de langages comme C, ADA ou PASCAL. C'est le langage qui « calcule » ces types. On peut le voir dans la réponse fournie par l'interprète lorsque le code de la fonction **rouge** lui est fourni.

```
# let rouge(n) =
  sommet_coeur(n) || sommet_carreau(n) ;;
val rouge : Cartes.numero_tas -> bool = <fun>
```

### 6.1.3 Variables locales

Certaines fonctions nécessitent des variables qui leur sont propres pour écrire l'algorithme qui les réalise. Ces variables sont appelées des *variables locales*.

**Exemple 4 :**

La fonction *factorielle*

$$\begin{array}{lll} \text{factorielle} : & \text{int} & \longrightarrow \text{int} \\ & n & \longmapsto n! \\ \text{CU} : & n \geq 0 & \end{array}$$

peut s'écrire en CAML

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

On voit par l'utilisation de la forme **let ... in ...** l'utilisation d'une variable locale pour coder cette fonction.

Une fonction peut utiliser toute variable globale, à condition qu'elle soit déclarée avant la fonction.

**Exemple 5 :**

Le code

```
let un = 1
let plus_un(n) =
  n + un
```

définit une constante entière qui vaut 1 et la fonction

$$\begin{array}{lll} \text{plus\_un} : & \text{int} & \longrightarrow \text{int} \\ & n & \longmapsto n + 1 \end{array}$$

L'expression définissant cette fonction utilise la constante préalablement définie **un** et se comporte comme on peut s'y attendre.

```
# plus_un(0);;
- : int = 1
```

Mais si on redéfinit la constante sans redéfinir la fonction, c'est toujours l'ancienne valeur de la constante qui est prise en compte.

```
# let un = 2;;
val un : int = 2
# plus_un(0);;
- : int = 1
```

### 6.1.4 Fonctions à plusieurs paramètres

Comme la fonction **superieur**, une fonction peut avoir plus d'un paramètre. Ces paramètres peuvent être de types identiques ou non.

**Exemple 6 :**

La fonction **puissance** qui à partir de deux entiers  $a$  et  $n$  renvoie l'entier  $a^n$

$$\begin{array}{lll} \text{puissance1} : & \text{int} \times \text{int} & \longrightarrow \text{int} \\ & a, n & \longmapsto a^n \\ \text{CU} : & n \geq 0 & \end{array}$$

peut s'écrire en CAML avec le code

```
let puissance1(a,n) =
  let p=ref 1
  in
    for i=1 to n do
      p := !p*a
    done;
    !p
```

**Exemple 7 :**

Reprenons la fonction puissance pour des nombres réels.

$$\begin{array}{lll} \text{puissance2} : & \text{float} \times \text{int} & \longrightarrow \text{float} \\ & a, n & \longmapsto a^n \\ \text{CU} : & n \geq 0 & \end{array}$$

Elle peut s'écrire en CAML avec le code

```
let puissance2(a,n) =
  let p=ref 1.
  in
    for i=1 to n do
      p := !p*.a
    done;
    !p
```

Un appel à une fonction à plusieurs paramètres doit se faire avec le bon nombre et le bon type des paramètres. Ainsi, dans l'exemple

```
# puissance1(2,10);;
- : int = 1024
# puissance2(2.,10);;
- : float = 1024.
```

les deux appels de fonction sont corrects car effectués avec le bon nombre et les bons types de paramètres. Mais dans l'exemple

```
# puissance1(2);;
This expression has type int but is here used with type int * int
```



l'interprète signale une erreur dans l'appel à la fonction `puissance1` parce qu'un seul entier a été passé en paramètre au lieu des deux attendus, et dans l'exemple

```
# puissance2(2,10);;
This expression has type int * int but is here used with type float * int
```

il signale une erreur parce qu'au lieu du couple (réel, entier) attendu par la fonction `puissance2`, un couple (entier, entier) lui a été passé.

### 6.1.5 Fonctions sans paramètre

Il est possible de définir des fonctions sans paramètre. Le type de départ est alors le type `unit`.

#### Exemple 8 :

C'est le cas de la procédure prédéfinie `print_newline`.

#### Exemple 9 :

La fonction sans paramètre constante égale à 1

$$\begin{array}{ccc} \text{un} : \text{unit} & \longrightarrow & \text{int} \\ () & \longmapsto & 1 \end{array}$$

se code en CAML

```
let un() = 1
```

et tout appel à cette fonction produit l'entier 1.

```
# un();;
- : int = 1
```

On peut réaliser des fonctions sans paramètre non constantes.

#### Exemple 10 :

Comme son nom l'indique, la fonction

$$\begin{array}{ccc} \text{valeur\_de\_x} : \text{unit} & \longrightarrow & \text{type de x} \\ () & \longmapsto & \text{valeur de x} \end{array}$$

renvoie la valeur de la variable mutable `x`. En voici le code

```
let valeur_de_x() = !x
```

Il est bien entendu préalablement nécessaire d'avoir déclaré une variable mutable `x`.

```
# let valeur_de_x() = !x;;
Unbound value x
```

Si la variable `x` est déclarée alors le type de la valeur renvoyée par la fonction `valeur_de_x` est le même que celui de `x`.

```
# let x = ref 1;;
val x : int ref = {contents = 1}
# let valeurdex() = !x;;
val valeur_de_x : unit -> int = <fun>
```

```
# valeur_de_x();;
- : int = 1
# x := 2;;
- : unit = ()
# valeur_de_x();;
- : int = 2
```

On peut réaliser des fonctions sans paramètres un peu plus intéressantes.

#### Exemple 11 :

Supposons que nous voulions programmer une fonction qui simule le lancer d'une pièce de monnaie. Cette fonction doit donner le résultat "PILE" ou "FACE" de manière aléatoire.

$$\begin{array}{rcl} \text{pile\_ou\_face} : \text{unit} & \longrightarrow & \text{string} \\ () & \longmapsto & \text{PILE ou FACE} \end{array}$$

On peut la programmer en utilisant la fonction prédéfinie

$$\begin{array}{rcl} \text{Random.int} : \text{int} & \longrightarrow & \text{int} \\ n & \longmapsto & p \\ \text{CU} : n > 0 \end{array}$$

où  $p$  est un entier au hasard compris entre 0 inclus et  $n$  exclu.

La fonction `pile_ou_face` se code alors en faisant un appel à `Random.int(2)` qui produira soit l'entier 0, soit l'entier 1, et en prenant la convention que si c'est 0 alors elle renvoie la valeur "PILE" et si c'est 1 elle renvoie la valeur "FACE".

```
let pile_ou_face() =
  if Random.int(2) = 0 then
    "PILE"
  else
    "FACE"
```

Voici une série de quelques appels successifs à cette fonction.

```
# pile_ou_face();;
- : string = "PILE"
# pile_ou_face();;
- : string = "FACE"
# pile_ou_face();;
- : string = "FACE"
# pile_ou_face();;
- : string = "PILE"
```

#### 6.1.6 Intérêt des fonctions

- reflet de l'analyse du problème
- modularité
- lisibilité des programmes
- factorisation du code

## 6.2 Procédures

Nous avons déjà rencontré et utilisé plusieurs procédures : `init_tas` et `deplacer_sommet` du module `Cartes`.

Une *procédure* permet de créer une nouvelle action, ou encore d'*abstraire* et *nommer* une suite d'instructions. Une fois définie, une procédure peut être utilisée comme une nouvelle *instruction* du langage.

Comme toute instruction, un appel à une procédure modifie l'état courant de l'environnement : c'est un *effet de bord*.

### 6.2.1 Spécification d'une procédure

Spécifier une procédure, c'est

- choisir un identificateur pour la nommer ;
- préciser le nombre de paramètres, leur type, et les nommer ;
- indiquer les conditions d'utilisation (CU) que doivent vérifier les paramètres lors d'un appel à la procédure ;
- et indiquer l'action de la procédure sur l'environnement (effet de bord).

Par exemple, la procédure du module `Cartes` permettant de déplacer une carte d'un tas vers un autre se nomme `deplacer_sommet`, possède deux paramètres de type `numero_tas` désignant les tas de départ et d'arrivée, possède une contrainte d'utilisation qui est que le tas de départ ne doit pas être vide, et son action modifie les tas de cartes en déplaçant une carte du tas de départ vers le tas d'arrivée. Nous résumons tout cela par la notation :

$$\begin{array}{ll} \text{deplacer\_sommet} : \text{numero\_tas} \times \text{numero\_tas} & \longrightarrow \text{unit} \\ & \text{depart, arrivee} \longmapsto () \\ \text{Action} : & \text{déplace une carte du tas } \textit{depart} \text{ vers le tas } \textit{arrivee} \\ \text{CU} : & \text{le tas } \textit{depart} \text{ ne doit pas être vide} \end{array}$$

### 6.2.2 Déclaration d'une procédure en CAML

En CAML, une procédure est une fonction dont le résultat est de type `unit`. La seule valeur renvoyée par cette fonction est donc l'unique valeur du type `unit` à savoir `()`.

Les procédures se déclarent donc de la même façon que les fonctions.

**Exemple 12 :**

```
let tout_mettre_sur_tas1 () =
  (* vider le tas 2 sur le tas 1 *)
  while tas_non_vide(2) do
    deplacer_sommet(2,1)
  done;
  (* vider le tas 3 sur le tas 1 *)
  while tas_non_vide(3) do
    deplacer_sommet(3,1)
  done;
  (* vider le tas 4 sur le tas 1 *)
  while tas_non_vide(4) do
    deplacer_sommet(4,1)
  done
```

**Exemple 13 :**

```
let vider_tas(depart, arrivee) =  
  while tas_non_vide(depart) do  
    deplacer_sommet(depart, arrivee)  
  done
```

Cette procédure a pour nom **vider\_tas** et possède deux paramètres (**depart** et **arrivee**) qualifiés de *formels*, car lors de la conception (ou écriture) de cette procédure, ces paramètres n'ont aucune valeur. Ils servent à *nommer* les (futures) valeurs qui seront passées lors d'un appel à la procédure.

### 6.2.3 Appel à une procédure

Un appel à une procédure est une instruction. Le résultat de son exécution est une modification de l'état courant de l'environnement ou un affichage de données.

### 6.2.4 Intérêt des procédures

- reflet de l'analyse du problème ;
- modularité ;
- lisibilité des programmes ;
- factorisation du code.

### 6.2.5 Méthodologie

SPECIFIER, SPECIFIER, SPECIFIER

## 6.3 Exercices

### 6.3.1 Fonctions

#### Exercice 6-1

**Question 1** Donnez les spécifications des fonctions et procédures du module `Cartes`.

**Question 2** Écrivez la fonction `sommet_trefle` à l'aide de la fonction `couleur_sommet`.

#### Exercice 6-2

Réalisez et testez une fonction `tas_tous_vides` qui renvoie un booléen indiquant si tous les tas sont vides.

#### Exercice 6-3

**Question 1** Réalisez une fonction `tas_max` qui donne le numéro du tas dont le sommet a la carte de valeur la plus élevée, parmi les deux tas passés en paramètres. Ne pas oublier les CU.

**Question 2** Réalisez une fonction `tas_max3` analogue à la précédente pour trois tas passés en paramètres.

**Question 3** Même chose pour quatre tas! (paramètres ou non?)

#### Exercice 6-4 Ou exclusif

Réalisez une fonction qui renvoie le ou-exclusif des deux booléens passés en paramètres.

#### Exercice 6-5 Divisibilité

Réalisez le prédicat

$$\begin{array}{lll} \text{est\_divisible\_par} : \text{int}, \text{int} & \longrightarrow & \text{bool} \\ & n, p & \longmapsto \begin{cases} \text{vrai} & \text{si } n \text{ est divisible par } p \\ \text{faux} & \text{sinon} \end{cases} \\ \text{CU} : p \neq 0 \end{array}$$

#### Exercice 6-6 Années bissextiles

Une année *bissextile* est une année qui comprend 366 jours au lieu des 365 pour les années ordinaires.

Depuis l'instauration du calendrier grégorien, sont bissextiles, les années :

- divisibles par 4 mais non divisibles par 100
- ou bien divisibles par 400.

Réalisez la fonction spécifiée ci-dessous

$$\begin{array}{lll} \text{est\_bissextile} : \text{int} & \longrightarrow & \text{bool} \\ & \text{annee} & \longmapsto \begin{cases} \text{vrai} & \text{si } \text{annee} \text{ est bissextile} \\ \text{faux} & \text{sinon} \end{cases} \end{array}$$

#### Exercice 6-7 Le plus grand

**Question 1** Réalisez une fonction qui renvoie le plus grand des deux entiers passés en paramètres.

**Question 2** Réalisez de deux façons une fonction qui renvoie le plus grand des trois entiers passés en paramètres

1. en utilisant définie dans la question précédente;

2. puis sans utiliser cette fonction.

**Exercice 6-8** *Coefficients binomiaux*

Étant donnés deux entiers  $0 \leq p \leq n$ , on définit le coefficient binomial par

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

Réalisez de deux façons la fonction qui à deux entiers associe le coefficient binomial  $\binom{n}{p}$

1. une première façon utilisant la fonction **factorielle**;
2. une seconde façon exploitant la simplification

$$\binom{n}{p} = \frac{n \times (n-1) \times \dots \times (n-p+1)}{p \times (p-1) \times \dots \times 1}.$$

**Exercice 6-9** *Fonction polynomiale*

Réalisez une fonction qui permet de calculer les valeurs de la fonction réelle de variable réelle

$$f(x) = 3x^4 - x^2 + 2x + 1.$$

**Exercice 6-10** *Plancher et plafond d'un nombre réel*

On appelle *plancher* d'un nombre réel le plus grand nombre entier inférieur ou égal à  $x$ . On le note  $\lfloor x \rfloor$ . Ce nombre est aussi appelé partie entière de  $x$ .

On appelle *plafond* d'un nombre réel le plus petit nombre entier supérieur ou égal à  $x$ . On le note  $\lceil x \rceil$ .

Ces deux nombres coïncident lorsque  $x$  est un nombre entier.

$$\begin{aligned} \lfloor \sqrt{2} \rfloor &= 1 \\ \lceil \sqrt{2} \rceil &= 2 \\ \lfloor -\sqrt{2} \rfloor &= -2 \\ \lceil -\sqrt{2} \rceil &= -1 \end{aligned}$$

**Question 1** Réalisez une fonction nommée **plancher** qui calcule le plancher du nombre réel  $x$  passé en paramètre<sup>1</sup>. Il est conseillé de se concentrer d'abord sur le cas des réels positifs ou nuls, puis de généraliser.

**Question 2** De même, réalisez une fonction nommée **plafond** pour le plafond d'un réel.

**Exercice 6-11** *simulation d'un dé*

Réalisez une fonction qui simule un dé. Cette fonction renvoie au hasard un entier compris entre 1 et 6.

**Exercice 6-12** *Tas ordonné ?*

Réaliser et tester un prédicat **tas1\_ordonne** qui renvoie vrai si le tas numéro 1 est rangé dans l'ordre croissant. Le tas vide est ordonné, les tas d'une cartes le sont aussi. Un tas est ordonné lorsque toute carte du tas est supérieure au sens large à toutes celles qu'elle surmonte. (comme pour l'exercice précédent plusieurs versions sont possibles).

---

1. Cette fonction est prédéfinie dans l'unité **math** et se nomme **floor**.

### 6.3.2 Procédures

#### Exercice 6-13

Écrire l'entête de la procédure `deplacer_sommet` du module `Cartes`. (Le type qui définit les couleurs se nomme `couleurs`, et celui qui définit les tas se nomme `numero_tas`.)

#### Exercice 6-14

Expliquez la condition d'utilisation de la procédure `ViderTas`.

#### Exercice 6-15

**Question 1** Reprendre la procédure `tout_mettre_sur_tas1` en utilisant la procédure `vider_tas`. (attention à l'ordre des déclarations de ces procédures)

**Question 2** Faire une version paramétrée par le numéro du tas sur lequel on veut tout mettre.

#### Exercice 6-16

Les exercices de manipulation de cartes.

#### Exercice 6-17 *Tables de multiplication*

Spécifiez et réalisez une procédure qui affiche la table de multiplication par un entier donné en paramètre.





## Chapitre 7

# Caractères et chaînes de caractères

### 7.1 Les caractères

#### 7.1.1 Définition

Les *caractères* sont un ensemble de symboles comprenant les 26 lettres de l'alphabet latin en versions minuscules et majuscules, les 10 chiffres de 0 à 9, les différents symboles de ponctuation, l'espace, et bien d'autres symboles encore. Ils sont au nombre de 256 au total.

En CAML, ils forment le type de données **char**.

Il est possible de déclarer des constantes de type **char**.

**Exemple 1 :**

```
# let espace = ' ';;  
val espace : char = ' '
```

Les constantes littérales de type **char** sont désignées entre deux apostrophes.

On peut aussi déclarer des variables mutables de type **char**.

**Exemple 2 :**

```
# let c = ref 'a';;  
val c : char ref = {contents = 'a'}
```

Il est possible d'écrire une donnée de ce type avec la procédure **print\_char**.

**Exemple 3 :**

```
# print_char(!c);;  
a- : unit = ()
```

### 7.1.2 Fonctions `int_of_char` et `char_of_int`

Les caractères sont numérotés de 0 à 255.

**Exemple 4 :**

Par exemple, le caractère `ESPACE` porte le numéro 32, le caractère `A` porte le numéro 64 et le caractère `a` porte le numéro 97.

La figure 7.1 donne pour les 128 premiers caractères leur numéro associé (code ASCII<sup>1</sup>).

NUL	0	ESP	32	@	64	'	96
SOH	1	!	33	A	65	a	97
STX	2	"	34	B	66	b	98
ETX	3	#	35	C	67	c	99
EOT	4	\$	36	D	68	d	100
ENQ	5	%	37	E	69	e	101
ACK	6	&	38	F	70	f	102
BEL	7	'	39	G	71	g	103
BS	8	(	40	H	72	h	104
HT	9	)	41	I	73	i	105
LF	10	*	42	J	74	j	106
VT	11	+	43	K	75	k	107
FF	12	,	44	L	76	l	108
CR	13	-	45	M	77	m	109
SO	14	.	46	N	78	n	110
SI	15	/	47	O	79	o	111
DLE	16	0	48	P	80	p	112
DC1	17	1	49	Q	81	q	113
DC2	18	2	50	R	82	r	114
DC3	19	3	51	S	83	s	115
DC4	20	4	52	T	84	t	116
NAK	21	5	53	U	85	u	117
SYN	22	6	54	V	86	v	118
ETB	23	7	55	W	87	w	119
CAN	24	8	56	X	88	x	120
EM	25	9	57	Y	89	y	121
SUB	26	:	58	Z	90	z	122
ESC	27	;	59	[	91	{	123
FS	28	<	60	\	92		124
GS	29	=	61	]	93	}	125
RS	30	>	62	^	94	~	126
US	31	?	63	_	95	DEL	127

FIGURE 7.1 – Table des 128 premiers caractères (code ASCII)

La fonction `int_of_char` donne le code du caractère passé en paramètre.

**Exemple 5 :**

---

1. ASCII = American Standard Code for Information Interchange

```
# int_of_char(' ');
- : int = 32
# int_of_char('A');
- : int = 65
# int_of_char('a');
- : int = 97
```

Inversement, la fonction `char_of_int` donne le caractère associé à l'entier passé en paramètre qui doit être compris entre 0 et 255.

**Exemple 6 :**

```
# for i=65 to 70 do
  print_char (char_of_int(i))
done;;
ABCDEF- : unit = ()
```

### 7.1.3 Comparaison de caractères

Les opérateurs `=`, `<`, `<=`, `>` et `>=` permettent de comparer deux caractères.

Si `c1` et `c2` sont deux expressions de type `CHAR`, alors

1. `c1 = c2` est vrai si et seulement si les deux caractères `c1` et `c2` sont égaux ;
2. `c1 < c2` est vrai si et seulement si le numéro du caractère `c1` est strictement plus petit que celui de `c2` ;
3. `c1 <= c2` est vrai si et seulement si le numéro du caractère `c1` est inférieur ou égal à celui de `c2` ;
4. etc ...

## 7.2 Les chaînes de caractères

### 7.2.1 Définition

Les *chaînes de caractères* sont des séquences finies de caractères. Le nombre de caractères composant une chaîne de caractères est la *longueur* de cette chaîne. Par la suite nous noterons

$$|s| = \text{longueur de } s.$$

Une chaîne de longueur nulle ne comprend aucun caractère : c'est la *chaîne vide*.

En CAML, les chaînes de caractères forment le type `string` et leur longueur est limitée à  $2^{24} - 6 = 16\,777\,210$  caractères.

Il est possible de déclarer des variables de type `string`.

**Exemple 7 :**

```
# let chaine1 = "TIMO"
  and chaine2 = "LEON";;
val chaine1 : string = "TIMO"
val chaine2 : string = "LEON"
```

Les constantes littérales de type `string` sont désignées entre deux guillemets.

Il est possible d'écrire une donnée de ce type au moyen des procédures `print_string` et `print_endline`.

**Exemple 8 :**

```
# begin
  print_string("Bonjour,_je_m'appelle_");
  print_string(chaine1);
  print_endline(chaine2);
  print_endline("Et_vous,_comment_vous appelez-vous_?")
end;;
Bonjour, je m'appelle TIMOLEON
Et vous, comment vous appelez-vous ?
- : unit = ()
```



Une chaîne de 1 caractère n'est pas un caractère. Et en particulier on ne peut affecter une valeur de type `char` à une variable mutable de type `string`.

```
# let s = ref "" ;;
val s : string ref = {contents = ""}
# s := 'a';;
This expression has type char but is here used with type string
# s := "a" ;;
- : unit = ()
# s;;
- : string ref = {contents = "a"}
```

### 7.2.2 Notation indicielle

Une chaîne de caractères est, par définition, composée de caractères. Ces caractères sont numérotés dans l'ordre à partir de 0, et ces numéros sont appelés *indices*. Le premier caractère a pour indice 0, le second a pour indice 1, etc ... Le dernier caractère a pour indice  $|s| - 1$ .

On accède au caractère d'indice  $i$  d'une chaîne  $s$  avec la notation indicielle pointée :

**Syntaxe :** Accès au  $i$ -ème caractère d'une chaîne  $s$

`s.[i]`

Si  $s$  est une chaîne, alors  $s.[i]$  est un caractère.

**Exemple 9 :**

```
# chaine1.[0];;
- : char = 'T'
# chaine1.[1];;
- : char = 'I'
# chaine1.[2];;
- : char = 'M'
# chaine1.[3];;
- : char = 'O'
```


Les chaînes de caractères sont des structures de données mutables. On peut modifier un caractère d'une chaîne de caractères à l'aide de la notation indicielle. Pour modifier le caractère d'indice  $i$  d'une chaîne  $s$ , on utilise la flèche  $<-$ .

**Syntaxe :** Modifier la valeur du  $i$ -ème caractère d'une chaîne  $s$

```
s.[i] <- (* une expression de type char *)
```

**Exemple 10 :**

```
# let c = "TiMOLEON";;
val c : string = "TiMOLEON"
# c.[1] <- 'I';;
- : unit = ()
# c ;;
- : string = "TIMOLEON"
```

 L'accès à un caractère d'une chaîne  $s$  n'est possible que si l'indice  $i$  est compris entre 0 et  $|s| - 1$ . Toute tentative d'accès à un caractère d'indice situé en dehors de cet intervalle déclenche l'exception `Invalid_argument "index out of bounds"`.

```
# let chaine1 = "TIMO";;
val chaine1 : string = "TIMO"
# chaine1.[-1];;
Exception: Invalid_argument "index_out_of_bounds".
# chaine1.[4];;
Exception: Invalid_argument "index_out_of_bounds".
```

Et voici le code d'une procédure équivalente à la procédure prédéfinie `print_string`.

```
(* ecrire_chaine(s) écrit a l'ecran la chaine s
cette procedure est equivalente a print_string(s) *)
let ecrire_chaine(s) =
  for i=0 to String.length(s)-1 do
    print_char(s.[i])
  done
```

### 7.2.3 Concaténation

La *concaténation* de deux chaînes de caractères  $s_1$  et  $s_2$  est l'opération consistant à mettre bout à bout ces deux chaînes. La chaîne ainsi obtenue est appelée *concaténée* de  $s_1$  et  $s_2$ .

En CAML, c'est l'opérateur  $\wedge$  qui permet d'exprimer la concaténation de deux chaînes de caractères.

**Exemple 11 :**

```
# chaine1^chaine2;;
- : string = "TIMOLEON"
```

La longueur de la chaîne de caractères  $s_1 \wedge s_2$  est égale à la somme des longueurs de chacune des deux chaînes

$$|s_1 \wedge s_2| = |s_1| + |s_2|.$$

### 7.2.4 Comparaison de chaînes

Les opérateurs `=`, `<`, `<=`, `>` et `>=` permettent de comparer deux chaînes de caractères.

Si `s1` et `s2` sont deux expressions de type `STRING`, alors

1. `s1 = s2` est vrai si et seulement si les deux chaînes de caractères `s1` et `s2` sont égales ;
2. `s1 < s2` est vrai si et seulement si la chaîne de caractère `s1` vient strictement avant `s2` dans l'ordre lexicographique ;
3. `s1 <= s2` est vrai si et seulement si la chaîne de caractère `s1` vient avant `s2` dans l'ordre lexicographique ou est égale à `s2` ;
4. etc ...

## 7.3 Quelques fonctions prédéfinies sur les chaînes

Hormis la première, les fonctions présentées ici ne sont pas à connaître par cœur.

Les fonctions prédéfinies sur les chaînes de caractères font toutes partie du module `String`. Leur nom est donc préfixé par le mot `String` (avec un `S` majuscule).

### Longueur d'une chaîne

$$\begin{array}{lll} \text{String.length} : \text{string} & \longrightarrow & \text{int} \\ s & \longmapsto & \text{String.length}(s) = |s| \end{array}$$

Exemple 12 :

```
# String.length("TIMOLEON");;
- : int = 8
# String.length("");;
- : int = 0
```

### Création d'une chaîne d'une taille donnée

$$\begin{array}{lll} \text{String.create} : \text{int} & \longrightarrow & \text{string} \\ n & \longmapsto & \text{String.create}(n) , \\ \text{CU} : n \geq 0 \end{array}$$

où `String.create(n)` est une nouvelle chaîne de  $n$  caractères non spécifiés.

Exemple 13 :

```
# String.create(10);;
- : string = "<\135\236\183\024\017/\008\000\000"
```

### Copie d'une chaîne

$$\begin{array}{lll} \text{String.copy} : \text{string} & \longrightarrow & \text{string} \\ s & \longmapsto & \text{String.copy}(s) \end{array}$$

où `String.copy(s)` est une nouvelle chaîne copie de la chaîne `s`, c'est-à-dire une chaîne égale à `s`.

Exemple 14 :

```
# let s1 = "tIMOLEON" ;;
val s1 : string = "tIMOLEON"
# let s2 = String.copy (s1) ;;
val s2 : string = "tIMOLEON"
# s1 = s2 ;;
- : bool = true
# s1.[0] <- 'T' ;;
- : unit = ()
# s1 ;;
- : string = "TIMOLEON"
# s2 ;;
- : string = "tIMOLEON"
```

En OBJECTIVE CAML il n'y a pas de fonction `string_of_char` qui transforme un caractère en la chaîne constituée de ce seul caractère. C'est un excellent exercice que de réaliser cette fonction avec les fonctions présentées dans ce chapitre (cf exercice 7-3).

## 7.4 Exercices

### Exercice 7-1

Sur le modèle de la procédure `ecrire_chaine`, réalisez une procédure nommée `ecrire_vertical` qui écrit verticalement la chaîne passée en paramètre. Par exemple, pour la chaîne `vertical` la procédure écrira à l'écran

```
v
e
r
t
i
c
a
l
```

### Exercice 7-2

Sur le modèle de la procédure `ecrire_chaine`, réalisez une procédure nommée `ecrire_chaine_a_l_envers` qui écrit à l'envers la chaîne passée en paramètre. Par exemple, pour la chaîne `repus` la procédure écrira à l'écran `super`

### Exercice 7-3 `string_of_char`

Réalisez la fonction

$$\begin{array}{rcl} \text{string\_of\_char} : & \text{char} & \longrightarrow \text{string} \\ & c & \longmapsto s \end{array}$$

où  $s$  est la chaîne constituée du seul caractère passé en paramètre.

Par exemple on doit avoir

```
# string_of_char('a') ;;
- : string = "a"
# string_of_char('1') ;;
- : string = "1"
# string_of_char('+') ;;
- : string = "+"
```

### Exercice 7-4 `string_of_int`

Réalisez la fonction

$$\begin{array}{rcl} \text{string\_of\_int} : & \text{int} & \longrightarrow \text{string} \\ & n & \longmapsto s \end{array}$$

où  $s$  est la chaîne de caractère constituée des chiffres de l'écriture décimale de  $n$ .

Par exemple on doit avoir

```
# string_of_int(123) ;;
- : string = "123"
# string_of_int(-12) ;;
- : string = "-12"
```



**Exercice 7-5** *Sous-chaînes*

Une *sous-chaîne* d'une chaîne de caractères  $s$  est une chaîne composée de caractères consécutifs de  $s$ . Elles peuvent être caractérisées par l'indice de début et leur longueur.

Voici par exemple quelques sous-chaînes pour la chaîne  $s = \text{TIMOLEON}$  :

Sous-chaîne	début	longueur
T	0	1
IMO	1	3
LEON	4	4
TIMOLEON	0	8

Réalisez la fonction

$$\begin{aligned} \text{sous\_chaîne} : \text{string} \times \text{int} \times \text{int} &\longrightarrow \text{string} \\ s, \text{deb}, \text{long} &\longmapsto s[\text{deb}..\text{deb} + \text{long} - 1] \\ \text{CU} : 0 \leq \text{deb} < |s| \text{ et } 0 \leq \text{long} < |s| + 1 - \text{deb} \end{aligned}$$

où  $s[\text{deb}..\text{deb} + \text{long} - 1]$  désigne la sous-chaîne de  $s$  débutant à l'indice  $\text{deb}$  et de longueur  $\text{long}$ , donc terminant à l'indice  $\text{deb} + \text{long} - 1$ .

**Exemple 15 :**

```
# sous_chaine("TIMOLEON",0,1);;
- : string = "T"
# sous_chaine("TIMOLEON",1,3);;
- : string = "IMO"
# sous_chaine("TIMOLEON",4,4);;
- : string = "LEON"
# sous_chaine("TIMOLEON",0,8);;
- : string = "TIMOLEON"
```

**Exercice 7-6** *Miroir*

La chaîne *miroir* d'une chaîne  $s$  est la chaîne dont les caractères successifs sont ceux de la chaîne  $s$  dans l'ordre inverse. Par exemple, la chaîne miroir de **TIMOLEON** est **NOELOMIT**.

Réalisez la fonction

$$\begin{aligned} \text{miroir} : \text{string} &\longrightarrow \text{string} \\ s &\longmapsto \text{miroir}(s) \end{aligned}$$
**Exercice 7-7** *Palindromes*

Un *palindrome* est un mot qui se lit de la même façon de gauche à droite et de droite à gauche. **ICI**, **ETE**, **ELLE** et **RADAR** sont des palindromes.

Réalisez un prédicat qui est vrai si et seulement si la chaîne passée en paramètre est un palindrome.

**Exercice 7-8** *Initialisations de tas de cartes*

Les descriptions des tas de cartes sont des chaînes de caractères.

**Question 1** Initialisez le tas 1 avec une description de tas donnée par l'utilisateur.

**Question 2** Initialisez le tas 1 avec des trèfles dont le nombre est donné par l'utilisateur.

**Question 3** Même question mais chaque carte étant de couleur quelconque.

**Exercice 7-9** *Initiale et autres*

**Question 1** Réalisez une fonction `initiale` qui donne le caractère initial (c'est-à-dire le premier) d'une chaîne de caractères. Cette fonction a-t-elle des contraintes d'utilisation ?

**Question 2** Réalisez une fonction `finale` qui donne le caractère final (c'est-à-dire le dernier) d'une chaîne de caractères. Cette fonction a-t-elle des contraintes d'utilisation ?

**Question 3** Réalisez une fonction `sauf_initiale` qui donne la chaîne passée en paramètre sans son initiale. Cette fonction a-t-elle des contraintes d'utilisation ?

**Exercice 7-10** *Préfixe*

Une chaîne de caractères  $s_1$  est un *préfixe* d'une autre  $s_2$  si la chaîne  $s_1$  est le début de la chaîne  $s_2$ . À titre d'exemple, les préfixes de 'TIMOLEON' sont '', 'T', 'TI', 'TIM', 'TIMO', 'TIMOL', 'TIMOLE', 'TIMOLEO' et 'TIMOLEON'.

Réalisez de deux façons différentes une fonction qui teste si une chaîne (premier paramètre de la fonction) est un préfixe d'une autre (second paramètre de la fonction).

**Exercice 7-11**

Réalisez une fonction qui retourne le plus petit (aus sens de l'ordre défini dans la section 7.1.3) caractère de la chaîne de caractères passée en paramètre.

# Annexe A

## Les cartes

### A.1 Présentation

Le module **Cartes** offre une extension du langage **CamL** afin d'écrire des programmes destinés à un robot manipulateur de cartes.

Le domaine de travail du robot est constitué de tas, numérotés 1,2,3,4, sur lesquels sont empilées des cartes à jouer. Ces cartes ont les couleurs habituelles (♣, ♦, ♥ et ♠) et les valeurs habituelles (par ordre croissant : as, deux, trois, . . . , dix, valet, dame, roi). Les cartes proviennent de plusieurs jeux, il est donc possible de trouver plusieurs exemplaires d'une même carte.

Chacun des quatre tas peut contenir un nombre quelconque de cartes, y compris aucune.

### A.2 Utilisation du module Cartes

Le module **Cartes** est un module spécifiquement développé à l'université de Lille 1 pour l'enseignement d'InitProg. Ce module n'est donc pas livré avec les distributions d'OBJECTIVE CAML, et il est nécessaire de l'installer séparément. Nous supposons ici que cette installation est faite.

Pour utiliser ce module, plusieurs possibilités existent :

1. depuis un interprète, invoquer la directive **#load**

```
# #load "Cartes.cma" ;;  
Chargement des images. Patientez quelques instants.  
.....  
#
```

2. en lançant un interprète, ajouter le module sur la ligne de commande

```
$ ocaml Cartes.cma  
Chargement des images. Patientez quelques instants.  
.....  
Objective Caml version 3.10.0  
  
#
```

3. en lançant l'interprète dédié au module

```
$ ocamlcartes
Chargement des images. Patientez quelques instants.
.....
Objective Caml version 3.10.0

Camlcartes
FIL IEEA univ. Lille 1 (2009)
pour obtenir de l'aide : aide_moi () ;;
#
```

Dans tous les cas, une fenêtre graphique s'ouvre.

Les noms des différentes variables, fonctions et procédures déclarées dans ce module doivent normalement être préfixés par le nom du module. Par exemple, l'instruction `init_tas` pour initialiser un tas (cf ci-dessous) doit être invoquée par le nom `Cartes.init_tas`. Pour se dispenser de préfixer systématiquement tous les noms par celui du module, il suffit d'exécuter la commande

```
# open Cartes;;
#
```

Ceci accompli, on peut faire référence à tous les éléments déclarés dans le module en utilisant leur nom non préfixé par celui du module.

```
# init_tas (1,"T") ;;
- : unit = ()
```

Dans toute la suite, nous supposons que l'instruction `open Cartes` a été exécutée.

## A.3 Le langage du module Cartes

### A.3.1 Description de tas

Une fois le module `Cartes` chargé, la fenêtre graphique montre quatre tas de cartes. Ces tas, numérotés de 1 à 4 de gauche à droite, contiennent des cartes en nombre quelconques choisies au hasard. Les problèmes que nous traiterons nécessitent de pouvoir imposer certaines configurations de l'état de ces quatre tas.

Afin de décrire en début de programme, la configuration initiale des tas de cartes, le module `Cartes` offre une procédure `init_tas`<sup>1</sup>.

Syntaxe :

```
init_tas(n,s)
```

où *n* est le numéro du tas décrit, et *s* est une *chaîne de description* du tas.



En Caml, les chaînes sont délimitées par des guillemets " (cf chapitre 7).

La chaîne de description, lue de gauche à droite, indique les cartes d'un tas du bas vers le haut avec les conventions suivantes :

- Les lettres **T**, **K**, **C**, **P** représentent respectivement : ♣, ♦, ♥, ♠ ;
- Si *A* et *B* sont des chaînes de description, la chaîne *AB* est aussi une chaîne de description qui représente les cartes de *A* surmontées de celles de *B*.

1. `init_tas` est une fonction dont le résultat est l'unique valeur du type `unit`, c'est ce qu'on appellera une procédure. Le paramètre de la procédure `init_tas`, est un élément du produit cartésien de l'ensemble des entiers par l'ensemble des chaînes `int * string → unit`.

- Si  $A$  et  $B$  sont des chaînes de description, la chaîne  $A+B$  est aussi une chaîne de description qui représente un choix entre les cartes de  $A$  ou celles de  $B$  ;
- Si  $A$  est une chaîne de description,  $[A]$  représente la répétition des cartes de  $A$  un nombre quelconque (défini de manière aléatoire) de fois (y compris zéro).
- Les parenthèses sont autorisées pour éviter les ambiguïtés.

**Exemple 1 :**

1. `init_tas(1,"")` déclare le tas numéro 1 vide.
2. l'instruction `init_tas(2,"TCK")` décrit le tas numéro 2 contenant de bas en haut un ♣, un ♥ et un ♦.
3. l'instruction `init_tas(3,"T+P")` décrit le tas numéro 3 contenant une carte qui est soit un ♣ soit un ♠.
4. `init_tas(4,"[T+P]")` décrit le tas numéro 4 contenant un nombre quelconque, non déterminé, de cartes de couleur ♣ ou ♠.
5. l'instruction `init_tas(1,"T+[P]CC")` décrit le tas numéro 1 contenant soit une seule carte de couleur ♣, soit un nombre quelconque de ♠ surmonté de deux ♥.
6. l'instruction `init_tas(1,"(T+[P])CC")` décrit le tas numéro 1 contenant soit trois cartes un ♣ surmonté de deux ♥, soit un nombre quelconque de ♠ surmonté de deux ♥. À noter qu'une autre façon de décrire la même initialisation est d'utiliser la chaîne "TCC\_+[P]CC".

Une instance possible de l'état des quatre tas après exécutions des instructions numérotées 1, 2, 3 et 5 est donnée par la partie gauche de la figure A.1 page 86.

**Remarque :** Tout programme de résolution d'un problème sur les cartes doit débiter par une initialisation des quatre tas à l'aide de l'instruction `init_tas`. Une fois cette initialisation effectuée, cette instruction n'est plus utilisée. Il s'en suit qu'après l'étape d'initialisation, le nombre de cartes globalement présentes sur les quatre tas ne varie pas.

**A.3.2 Action**

La seule action permettant de modifier l'état des tas est le déplacement de la carte située au sommet d'un tas vers le sommet d'un autre tas. Cette action est donnée par un appel à la procédure `deplacer_sommet`<sup>2</sup> paramétrée par un couple :

**Syntaxe :**

`deplacer_sommet( $n, p$ )`

où  $n$  est le numéro du tas duquel on prend une carte, et  $p$  est celui du tas sur lequel on la pose.

**Exemple 2 :**

L'instruction `deplacer_sommet(2,4)` a pour effet de déplacer la carte au sommet du tas 2 pour la poser au sommet du tas 4. Appliquée à l'état des tas décrits par la partie gauche de la figure A.1, cette instruction transforme les tas 2 et 4 comme le montre la partie droite.

**CU :** Il n'est pas permis de déplacer une carte située sur un tas vide. Toute tentative d'action `deplacer_sommet( $n, p$ )` déclenche l'exception `Tas_Vide` si le tas  $n$  est vide.

---

2. dont le type est `int * int → unit`.

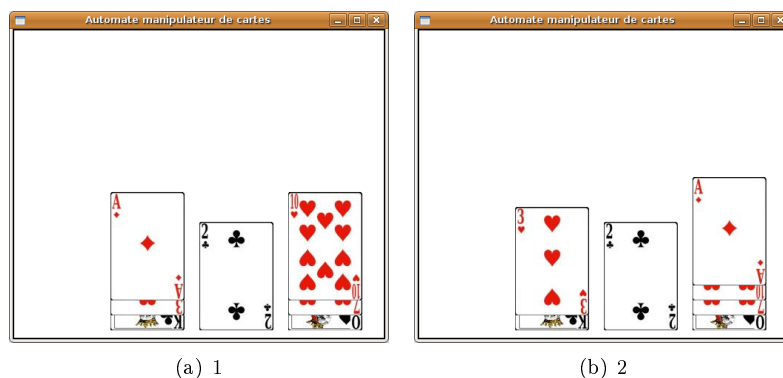


FIGURE A.1 – Transformation des tas par déplacement d'une carte

### A.3.3 Tests sur les tas

Certains traitements nécessitent des tests. Pour cela on dispose de fonctions.

#### Test de vacuité

Pour tester si un tas est vide, on fera appel à la fonction<sup>3</sup>

**Syntaxe :**

```
tas_vide(n)
```

qui donne la valeur **vrai** (**true** en anglais, et en Caml) si le tas numéro *n* est vide, **faux** (**false** en anglais et en Caml) dans le cas contraire.

On peut faire le test contraire en faisant appel à la fonction

**Syntaxe :**

```
tas_non_vide(n)
```

#### Test sur la couleur

Les quatre couleurs sont décrites dans le module **Cartes** par les quatre identificateurs (préfixés par le nom du module **Cartes**) :

**Cartes.TREFLE**, **Cartes.CARREAU**, **Cartes.COEUR**, **Cartes.PIQUE**.

Si la commande **open Cartes** a été exécutée, on peut se dispenser de préfixer les noms des couleurs dans l'écriture des programmes.



Il est important de désigner les couleurs par ces identificateurs en lettres MAJUSCULES.

Il est possible de connaître la couleur au sommet d'un tas en faisant appel à la fonction<sup>4</sup>

**Syntaxe :**

```
couleur_sommet(n)
```

3. dont le type est `int`  $\rightarrow$  `bool`.

4. dont le type est `int`  $\rightarrow$  `couleur`.

qui donne la couleur de la carte située au sommet du tas numéro  $n$ .

**Exemple 3 :**

```
if couleur_sommet(2)=PIQUE then ...
```

**CU :** Il est normalement dénué de sens de tester la couleur de la carte située au sommet d'un tas vide. Toute tentative d'appel à `couleur_sommet( $n$ )` déclenche l'exception **Tas\_Vide** si le tas  $n$  est vide.

Quatre autres fonctions<sup>5</sup> permettent aussi de tester la couleur du sommet d'un tas

**Syntaxe :**

```
sommet_trefle( $n$ )
sommet_carreau( $n$ )
sommet_coeur( $n$ )
sommet_pique( $n$ )
```

qui donnent la valeur **vrai** si le sommet du tas  $n$  est un ♣ (♦,♥,♠) et **faux** dans le cas contraire.

Elles sont soumises à la même contrainte d'utilisation que la fonction `couleur_sommet`, et déclenchent la même exception en cas de non respect de cette contrainte.

### A.3.4 Comparaison des valeurs

Une fonction permet de comparer les valeurs des cartes au sommet de deux tas :

**Syntaxe :**

```
superieur( $n,p$ )
```

qui donne la valeur **vrai** si la carte au sommet du tas numéro  $n$  a une valeur supérieure ou égale à celle du tas numéro  $p$ , et la valeur **faux** dans le cas contraire.

**CU :** On ne peut comparer les cartes situées au sommet de deux tas que s'ils ne sont pas vides. Tout appel à `superieur( $n,p$ )` déclenche l'exception **Tas\_Vide** si l'un des tas  $n$  ou  $p$  est vide.

### A.3.5 Contrôle de l'affichage

**Modes d'affichage**

À l'exécution, l'affichage peut se faire de trois façons :

1. **mode graphique** : Dans ce mode, on visualise l'évolution des tas de cartes dans une fenêtre graphique. C'est le mode par défaut.
2. **mode texte** : Dans ce mode, on visualise l'évolution des tas de cartes dans le terminal où s'exécute le programme. Voici un exemple d'affichage textuel produit par l'exécution d'une instruction :

---

5. dont le type est `int`  $\rightarrow$  `unit`.

```
# deplacer_sommet(4,1) ;;
deplacer_sommet(4,1);;
(*                                     .----- . *)
(*                                     | 3P  | *)
(*                                     .----- . *)
(*                                     | 10P | *)
(*                                     .----- . *)
(*                                     | vP  | *)
(*                                     .----- . *)
(*                                     | 5P  | *)
(*                                     .----- . *)
(*                                     | 4P  | *)
(*                                     .----- . *)
(*                                     | 5T  | *)
(*                                     .----- . *)
(* | 7P | | 6T | | 6K | | dP | *)
(*                                     .----- . *)
- : unit = ()
```

3. **mode texte et graphique** : Dans ce mode, on visualise l'évolution des tas de cartes à la fois dans le terminal et dans une fenêtre graphique.

Le mode d'affichage par défaut est le mode graphique. L'instruction

```
affichage_en_mode_texte ()
```

permet de passer en mode texte. L'instruction

```
affichage_en_mode_graphique ()
```

permet de passer en mode graphique. Et l'instruction

```
affichage_en_mode_texte_et_graphique ()
```

permet de combiner les deux modes texte et graphique<sup>6</sup>.

**Remarque :** Le mode texte permet de conserver une trace de l'exécution d'un programme dans un fichier texte. Par exemple, si un fichier nommé **exo.ml** contient un programme de résolution d'un problème sur les cartes, et s'il contient une instruction demandant un affichage textuel, alors dans un terminal la commande

```
$ ocamlcartes exo.ml > exo.result
```

produit un fichier texte nommé **exo.result** contenant toutes les instructions exécutées suivies de l'état des tas de cartes qu'elles transforment.

### Vitesse d'exécution

Un délai est imposé entre l'exécution des instructions de manipulation des tas. Le temps d'attente peut être ajusté en fonction des besoins. Le module fournit deux éléments permettant de consulter (**quel\_delai**)<sup>7</sup> et de modifier (**fixer\_delai**)<sup>8</sup> cette temporisation.

Pour augmenter la vitesse d'exécution, il suffit de diminuer la valeur réelle.

6. Ces trois procédures sont de type `unit → unit`

7. dont le type est `unit → float`

8. dont le type est `float → unit`



```
fixer_delai(0.1)
```

L'instruction précédente permet de fixer la durée de l'attente à un dixième de seconde.

### Pause

L'instruction **pause**<sup>9</sup> permet de faire une pause durant l'exécution d'un programme. Elle s'utilise en communiquant un message en paramètre, message qui sera imprimé dans le terminal lors de la pause. La pause s'arrête dès l'appui sur la touche ENTRÉE.

```
pause ("un_petit_repose_bien_merite")
```

### A.3.6 Réparer l'automate

Lorsqu'une contrainte d'utilisation n'est pas respectée, par exemple lorsqu'on veut déplacer une carte située sur un tas vide, une exception est déclenchée, et la fenêtre graphique apparaît barrée de deux lignes rouges qui indique que l'automate est « cassé ». À partir de ce moment plus aucune action sur l'automate n'est possible.

La session qui suit suppose que la situation actuelle est celle montrée à la figure A.1, et montre le déclenchement de l'exception **LesExceptions.Tas\_Vide 1** dû à la tentative de déplacer une carte depuis le tas 1 qui est vide, puis le déclenchement de l'exception **LesExceptions.AutomateCasse** qui montre l'impossibilité d'agir sur l'environnement des tas puisque l'automate a été cassé par la commande précédente.

```
# deplacer_sommet(1,2) ;;
Exception: LesExceptions.Tas_Vide 1.
# deplacer_sommet(2,3);;
Exception: LesExceptions.AutomateCasse.
```

Plus rien n'est possible avant l'exécution de l'instruction **Admin.repare ()**. Cette instruction permet de « réparer » l'automate (les lignes rouges de la fenêtre graphique disparaissent), et de poursuivre les actions sur les tas de cartes.

```
# Admin.repare ();;
- : unit = ()
# deplacer_sommet(2,3) ;;
- : unit = ()
```

### A.3.7 Obtenir de l'aide

Pour obtenir un aide-mémoire des principales fonctions et instructions du module **Cartes**, on fait appel à la fonction **aide\_moi**.

```
# aide_moi () ;;
Les principaux elements du Module Cartes
-----
* init_tas : numero_tas * string -> unit
  init_tas(num_tas,chaine) : initialise le tas numero num_tas
  avec la description donnee par chaine.
...
```

---

9. dont le type est `string → unit`

## A.4 Exercices

Ils sont classés en facile 😊, moyen 😊 et difficile 😊 (les premiers sont très faciles).

### A.4.1 Descriptions de tas

#### Exercice A-1

Pour chacune des descriptions qui suivent, donnez l'instruction d'initialisation du tas qui convient :

- le tas 1 contient une carte de couleur ♣ ;
- le tas 1 contient une carte de couleur ♣ ou ♠ ;
- le tas 1 contient une carte de couleur quelconque ;
- le tas 1 contient deux cartes de couleur ♥ ;
- le tas 1 contient une carte de couleur ♥ surmontée d'un ♦ ;
- le tas 1 contient un nombre quelconque de ♠ ;
- le tas 1 contient un nombre quelconque de ♠ ou bien un nombre quelconque de ♥ ;
- le tas 1 contient un nombre quelconque de cartes de couleur ♠ ou ♥ ;
- le tas 1 contient un nombre quelconque de cartes de couleur quelconque ;
- le tas 1 contient au moins un carreau ;
- le tas 1 contient un ♣ surmonté soit d'un nombre quelconque de ♥, soit d'un nombre quelconque non nul de ♠ ;
- le tas 1 contient un nombre pair de ♥ ;
- le tas 1 contient un nombre impair de ♥ ;
- le tas 1 contient un nombre pair de ♣ ou un nombre multiple de 3 de ♠ ;
- les deux cartes extrêmes du tas 1 (la plus basse et la plus haute) sont des ♣, entre les deux il y a un nombre quelconque de successions de deux cartes de couleur ♦♥.

### A.4.2 Séquence

Dans tous les exercices qui suivent, l'énoncé décrit une situation initiale des quatre tas de cartes (dans la syntaxe du module **Cartes**), et la situation finale à atteindre.

#### Exercice A-2 😊

**Situation initiale :**

Tas 1 : "TT"                      Tas 2 : ""

Tas 3 : ""                      Tas 4 : ""

**Situation finale :**

Tas 1 : ""                      Tas 2 : "TT"

Tas 3 : ""                      Tas 4 : ""

#### Exercice A-3 😊

**Situation initiale :**

Tas 1 : "TK"                      Tas 2 : ""

Tas 3 : ""                      Tas 4 : ""

**Situation finale :**

Tas 1 : "KT"                      Tas 2 : ""

Tas 3 : ""                      Tas 4 : ""

#### Exercice A-4 😊

**Situation initiale :**

Tas 1 : "TKTK"                      Tas 2 : ""

Tas 3 : ""                      Tas 4 : ""

**Situation finale :**

Tas 1 : "KKT"                      Tas 2 : ""

Tas 3 : ""                      Tas 4 : ""

#### Exercice A-5 😊

**Situation initiale :**

Tas 1 : "TKCP"                      Tas 2 : ""

Tas 3 : ""                      Tas 4 : ""

**Situation finale :**

Tas 1 : "PCKT"                      Tas 2 : ""

Tas 3 : ""                      Tas 4 : ""

### A.4.3 Conditionnelle

#### Exercice A-6 😊

**Situation initiale :**

Tas 1 : "T+P"                      Tas 2 : ""

Tas 3 : ""                      Tas 4 : ""

**Situation finale :**

Tas 1 : ""                      Tas 2 : "[T]"

Tas 3 : "[P]"                      Tas 4 : ""

#### Exercice A-7 😊

**Situation initiale :**

Tas 1 : "(T+K+C+P)(T+K+C+P)"      Tas 2 : ""

Tas 3 : ""      Tas 4 : ""

**Situation finale :**

Tas 1 : ""      Tas 2 : ""

Tas 3 : "(T+K+C+P)(T+K+C+P)"↑      Tas 4 : ""

le symbole ↑ signifiant que les cartes sont dans l'ordre croissant (i.e. la carte du dessous a une valeur inférieure ou égale à celle du dessus).

**Exercice A-8** 😊**Situation initiale :**

Tas 1 : "T+K+C+P"      Tas 2 : ""

Tas 3 : ""      Tas 4 : ""

**Situation finale :**

Tas 1 : "[T]"      Tas 2 : "[K]"

Tas 3 : "[C]"      Tas 4 : "[P]"

**Exercice A-9** 😊**Situation initiale :**

Tas 1 : "(T+K+C+P)(T+K+C+P)"      Tas 2 : ""

Tas 3 : ""      Tas 4 : ""

**Situation finale :**

Tas 1 : "[K+C]"      Tas 2 : "[T+P]"

Tas 3 : ""      Tas 4 : ""

## A.4.4 Itération

## Exercice A-10 😊

Situation initiale :

Tas 1 : "[T]"      Tas 2 : ""

Tas 3 : ""      Tas 4 : ""

Situation finale :

Tas 1 : ""      Tas 2 : "[T]"

Tas 3 : ""      Tas 4 : ""

## Exercice A-11 😊

Situation initiale :

Tas 1 : "[K+C][T+P]"      Tas 2 : ""

Tas 3 : ""      Tas 4 : ""

Situation finale :

Tas 1 : "[T+P][K+C]"      Tas 2 : ""

Tas 3 : ""      Tas 4 : ""

## Exercice A-12 😊

Situation initiale :

Tas 1 : "[K]"      Tas 2 : "[T]"

Tas 3 : ""      Tas 4 : ""

Situation finale :

Tas 1 : "[K]"      Tas 2 : ""

Tas 3 : "[KT]"      Tas 4 : ""

ou bien :

Tas 1 : ""      Tas 2 : "[T]"

Tas 3 : "[KT]"      Tas 4 : ""

## Exercice A-13 😊

Situation initiale :

Tas 1 : "[T]"      Tas 2 : ""

Tas 3 : ""      Tas 4 : ""

Situation finale :

Tas 1 : ""      Tas 2 : "[T]"

Tas 3 : "[T]"      Tas 4 : ""

le nombre de cartes des deux tas 2 et 3 différant d'au plus 1 dans la situation finale.

## Exercice A-14 😊

Situation initiale :

Tas 1 : "[T]"      Tas 2 : "[K]"

Tas 3 : "[P]"      Tas 4 : ""

Situation finale :

Tas 1 : ""      Tas 2 : "[T]"

Tas 3 : "[K]"      Tas 4 : "[P]"

En faire deux versions, la seconde utilisant une procédure `vider_tas(depart, arrivee)` qui vide le tas `depart` sur le tas `arrivee`.

**Exercice A-15** 😊**Situation initiale :**

Tas 1 : "[T]"	Tas 2 : "[K]"
Tas 3 : "[C]"	Tas 4 : "[P]"

**Situation finale :**

Tas 1 : "[P]"	Tas 2 : "[T]"
Tas 3 : "[K]"	Tas 4 : "[C]"

**Exercice A-16** 😊**Situation initiale :**

Tas 1 : "[T][K][C][P]"	Tas 2 : ""
Tas 3 : ""	Tas 4 : ""

**Situation finale :**

Tas 1 : "[P][C][K][T]"	Tas 2 : ""
Tas 3 : ""	Tas 4 : ""

**Exercice A-17** 😊**Situation initiale :**

Tas 1 : "[T]"	Tas 2 : "[K]"
Tas 3 : "[P]"	Tas 4 : ""

**Situation finale :**

Tas 1 : ""	Tas 2 : ""
Tas 3 : ""	Tas 4 : "[TKP][XY][Z]"

où  $X$  et  $Y$  désignent les deux couleurs restantes lorsque l'une des couleurs manque, et  $Z$  désigne la couleur restante lorsque  $X$  ou  $Y$  manque.

**Exercice A-18** 😊**Situation initiale :**

Tas 1 : "[T+K+C+P]"	Tas 2 : ""
Tas 3 : ""	Tas 4 : ""

**Situation finale :**

Tas 1 : "[T]"	Tas 2 : "[K]"
Tas 3 : "[C]"	Tas 4 : "[P]"

**Exercice A-19** 😊**Situation initiale :**

Tas 1 : "T[T]"	Tas 2 : ""
Tas 3 : ""	Tas 4 : ""

**Situation finale :**

Tas 1 : ""	Tas 2 : "T"—
Tas 3 : "[T]"	Tas 4 : ""

le symbole — indique que la carte est de valeur minimale.

**Exercice A-20** 😊**Situation initiale :**

Tas 1 : "[T]"	Tas 2 : ""
Tas 3 : ""	Tas 4 : ""

**Situation finale :**

Tas 1 : ""	Tas 2 : "[T]↑
Tas 3 : ""	Tas 4 : ""

le symbole ↑ signifiant que les cartes sont rangées par ordre croissant de valeurs de bas en haut.

**Exercice A-21** 😊

**Situation initiale :**

Tas 1 : "[T+K]"      Tas 2 : ""

Tas 3 : ""      Tas 4 : ""

**Situation finale :**

Tas 1 : "[X][Y]"      Tas 2 : ""

Tas 3 : ""      Tas 4 : ""

Le symbole  $X$  désigne la couleur ( $\clubsuit$  ou  $\diamondsuit$ ) la plus nombreuse, l'autre couleur étant désignée par  $Y$ .

#### Exercice A-22 🤖

**Situation initiale :**

Tas 1 : "K[T]"      Tas 2 : ""

Tas 3 : ""      Tas 4 : ""

**Situation finale :**

Tas 1 : "[T+K]"      Tas 2 : "[T+K]"

Tas 3 : "[T+K]"      Tas 4 : "[T+K]"

les trèfles étant équitablement répartis sur les quatre tas, l'unique carreau se trouvant n'importe où.

*Remarque :* ce problème est infaisable sans le carreau.

#### Exercice A-23 🤖

**Situation initiale :**

Tas 1 : "[T]"      Tas 2 : "[K]"

Tas 3 : "[C]"      Tas 4 : "[P]"

**Situation finale :**

Tas 1 : "[T]↑"      Tas 2 : "[K]↑"

Tas 3 : "[C]↑"      Tas 4 : "[P]↑"

le symbole  $\uparrow$  signifiant que les cartes sont rangées par ordre croissant de valeurs de bas en haut.

### A.4.5 Fonctions et procédures

#### Exercice A-24

Soit la fonction définie en CAML par

```
let meme_couleur (couleur, tas) =
  couleur = couleur_sommet(tas) ;;
```

**Question 1** Quel problème pose cette fonction si lors d'un appel le tas `tas` passé en paramètre est vide ?

**Question 2** Comment modifier la fonction `meme_couleur` pour qu'elle renvoie la valeur `faux` si le tas `tas` est vide.

#### Exercice A-25 Égalité de deux cartes

**Question 1** Écrivez une fonction qui teste l'égalité de la valeur de deux cartes situées au sommet de deux tas que l'on supposera non vides.

**Question 2** Puis écrivez une fonction qui teste l'égalité de deux cartes (valeur et couleur) situées au sommet de deux tas supposés non vides.

#### Exercice A-26 Inverser l'ordre des cartes d'un tas

Il s'agit d'écrire une procédure `inverser_tas1` pour inverser l'ordre des tas du tas 1. Par

exemple, si on a **Tas 1**: "TCCPK" alors après exécution de l'instruction `inverser_tas1 ()`, on doit avoir **Tas 1**: "KPCCT".

**Question 1** Faites-le avec l'hypothèse que les autres tas sont vides.

**Question 2** Réaliser la procédure sans supposer que les autres tas sont vides.



## Annexe B

# Le module Camlgraph

### B.1 Présentation

Le module **Camlgraph** est une adaptation du module **graphics** livré avec toutes les distributions de **OBJECTIVE CAML**. Il reprend les mêmes fonctions avec les mêmes noms. La seule différence réside dans le fait que toutes les fonctions y sont décurryfiées.

Ce module permet de dessiner des points, des segments, des polygones, des cercles, de remplir certaines zones dans une fenêtre graphique. Cette fenêtre graphique peut être considérée comme un tableau de points repérés par leurs coordonnées (cf figure B.1). Ces coordonnées sont des nombres entiers. Seuls les points dont les deux coordonnées sont positives ou nulles et inférieures à la largeur et à la hauteur de la fenêtre sont visibles.

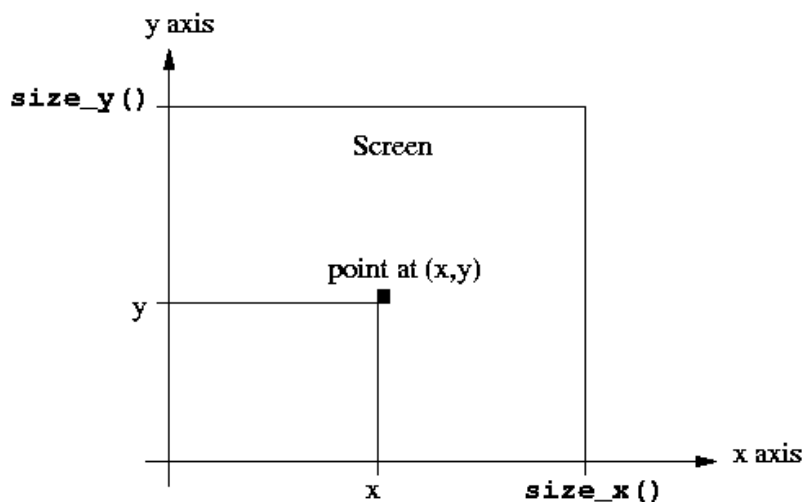


FIGURE B.1 – Système de coordonnées du module Camlgraph

Le programme ci-dessous illustre une utilisation de ce module. Son exécution produit l'image présentée sur la figure B.2.

```

(* auteur : EW *)
(* date : mai 2009 *)
(* objet : dessin de cercles *)
(* concentriques de *)
(* couleurs aleatoires *)
(* utilise le module Camlgraph *)

open Camlgraph ;;

let largeur = 600 (* largeur de la fenetre graphique *)
and hauteur = 400 (* hauteur de la fenetre graphique *) ;;

let n = (hauteur - 20) / 2 (* le nombre de cercles a dessiner *)
and cx = largeur / 2 (* abscisse du centre des cercles *)
and cy = hauteur / 2 (* ordonnee du centre des cercles *);;

begin
  Random.init (int_of_float (Unix.time()));
  open_graph ("_"^string_of_int(largeur)^"x"^string_of_int(hauteur)) ;
  set_window_title ("Cercles") ;
  for i = 1 to n do
    set_color (rgb(Random.int(256),Random.int(256),Random.int(256)));
    draw_circle (cx,cy,i)
  done;
  print_endline "appuyez_sur_une_touche_pour_terminer";
  ignore(read_key ()) ;
  close_graph ()
end

```

## B.2 Module Camlgraph

Author(s) : FIL

### B.2.1 Gestion des fenêtres graphiques

**val open\_graph : string -> unit**

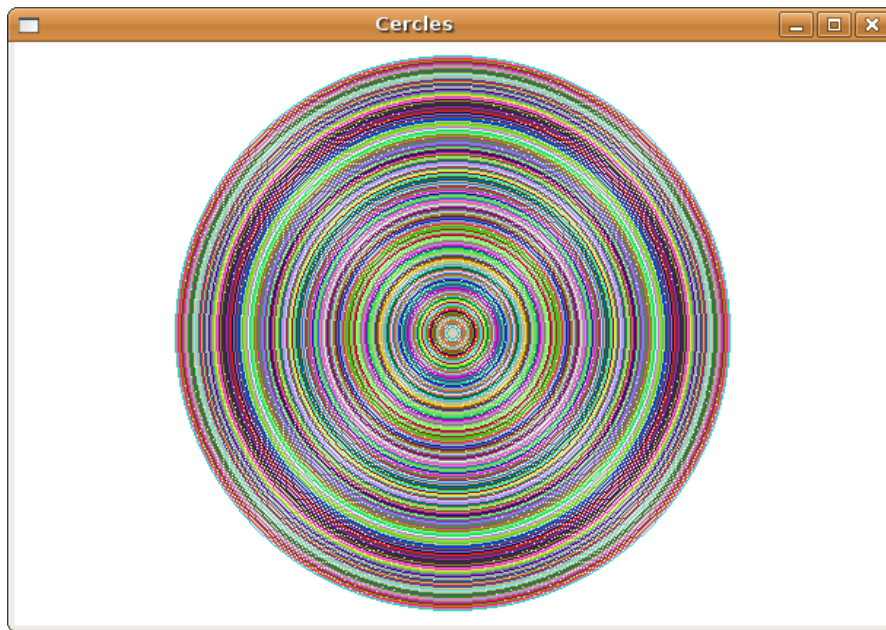
**open\_graph (s)** ouvre une fenêtre graphique dont les dimensions sont indiquées par la chaîne **s**. Pour une ouverture de la fenêtre sur l'écran par défaut, la chaîne doit commencer par un espace.

**val close\_graph : unit -> unit**

**close\_graph ()** ferme de la fenêtre graphique.

**val set\_window\_title : string -> unit**

**set\_window\_title (s)** donne un titre à la fenêtre graphique.

FIGURE B.2 – Figure produite en utilisant le module **Camlgraph**

```
val resize_window : int * int -> unit
    resize_window (w,h) redimensionne la fenêtre graphique w = largeur, h = hauteur.

val clear_graph : unit -> unit
    clear_graph () efface le contenu de la fenêtre graphique.

val size_x : unit -> int
    size_x () donne la largeur de la fenêtre graphique.

val size_y : unit -> int
    size_y () donne la hauteur de la fenêtre graphique.
```

### B.2.2 Gestion des couleurs

```
type color = Graphics.color
    représentation des couleurs

val rgb : int * int * int -> color
    rgb (r,g,b) donne une couleur à partir de ses trois composantes rouge, vert, bleu.

val set_color : color -> unit
    set_color (x) fixe la couleur courante.

val background : color
    couleur courante de l'arrière plan.
```

```
val foreground : color
    couleur courante de l'avant plan.
```

#### Couleurs prédéfinies

```
val black : color
val white : color
val red : color
val green : color
val blue : color
val yellow : color
val cyan : color
val magenta : color
```

### B.2.3 Point et dessin de lignes

```
val plot : int * int -> unit
    plot (x,y) dessine un point de coordonnées (x,y).

val point_color : int * int -> color
    point_color (x,y) donne la couleur du point de coordonnées (x,y).

val moveto : int * int -> unit
    moveto (x,y) positionne le crayon au point de coordonnées (x,y).

val rmoveto : int * int -> unit
    rmoveto (x,y) translate le crayon selon le vecteur de coordonnées (x,y).

val current_x : unit -> int
    current_x () donne l'abscisse de la position courante du crayon.

val current_y : unit -> int
    current_y () donne l'ordonnée de la position courante du crayon.

val current_point : unit -> int * int
    current_point () donne les coordonnées de la position courante du crayon.

val lineto : int * int -> unit
    lineto (x,y) trace un segment depuis la position courante jusqu'au point de coordonnées (x,y).

val rlineto : int * int -> unit
    rlineto (x,y) trace un segment depuis la position courante selon le vecteur de coordonnées (x,y).

val curveto : (int * int) * (int * int) * (int * int) -> unit
```

`curveto (b,c,d)` trace une courbe de Bezier du point courant jusqu'au point `d` avec les points `b` et `c` pour points de contrôle.

`val draw_rect : int * int * int * int -> unit`

`draw_rect (x1,y1,x2,y2)` dessine un rectangle dont le coin inférieur gauche a pour coordonnées `(x1,y1)` et le point supérieur droit `(x2,y2)`.

`val draw_arc : int * int * int * int * int * int -> unit`

`draw_arc (x,y,rx,ry,a1,a2)` dessine un arc d'ellipse de centre `(x,y)`, de rayon horizontal `rx`, de rayon vertical `ry`, entre les angles `a1` et `a2`.

`val draw_ellipse : int * int * int * int -> unit`

`draw_ellipse (x,y,rx,ry)` dessine une ellipse de centre `(x,y)`, de rayon horizontal `rx`, de rayon vertical `ry`.

`val draw_circle : int * int * int -> unit`

`draw_circle (x,y,r)` dessine un cercle de centre `(x,y)`, de rayon `r`.

`val set_line_width : int -> unit`

`set_line_width (n)` fixe l'épaisseur des traits

#### B.2.4 Texte

`val draw_char : char -> unit`

`draw_char (c)` dessine le caractère `c` à la position courante.

`val draw_string : string -> unit`

`draw_string (s)` dessine la chaîne de caractères `s` à la position courante.

`val set_font : string -> unit`

`set_font (s)` fixe la police pour dessiner les textes.

`val set_text_size : int -> unit`

`set_text_size (n)` fixe la taille des caractères pour dessiner les textes.

`val text_size : string -> int * int`

`text_size (s)` donne les dimensions du texte `s` s'il était dessiné avec les police et taille courantes.

#### B.2.5 Remplissage

`val fill_rect : int * int * int * int -> unit`

`fill_rect (x,y,w,h)` remplit le rectangle dont le coin inférieur gauche a pour coordonnées `(x,y)`, de largeur `w` et de hauteur `h`, avec la couleur courante

`val fill_arc : int * int * int * int * int * int -> unit`

`fill_arc (x,y,rx,ry,a1,a2)` remplit l'arc d'ellipse dont le centre a pour coordonnées `(x,y)`, de rayon horizontal `rx` et de rayon vertical `ry`, entre les angles `a1` et `a2`, avec la couleur courante

`val fill_ellipse : int * int * int * int -> unit`

`fill_ellipse (x,y,rx,ry)` remplit l'ellipse dont le centre a pour coordonnées `(x,y)`, de rayon horizontal `rx` et de rayon vertical `ry`, avec la couleur courante

`val fill_circle : int * int * int -> unit`

`fill_circle (x,y,r)` remplit le cercle dont le centre a pour coordonnées `(x,y)`, de rayon `r`, avec la couleur courante

## Annexe C

# Éléments du langage étudiés dans ce cours

### C.1 Syntaxe

#### C.1.1 Commentaires

Syntaxe : Commentaires en CAML

```
(* ceci est un commentaire *)
```

#### C.1.2 Expressions et instructions

Syntaxe : Séquence d'instructions

```
< instruction1 >;  
< instruction2 >;  
...  
< instructionn >
```

Syntaxe : Bloc d'instructions

```
begin  
  (* sequence d'instructions *)  
end
```

#### C.1.3 Expressions conditionnelles

Syntaxe : Expression conditionnelle

```
if < condition > then  
  < expression1 >  
else  
  < expression2 >
```

### C.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 *)
```

### C.1.5 Instructions conditionnellement répétées

**Syntaxe :** Boucle tant que

```
while < condition > do
  (* sequence d'instructions *)
done
```

### C.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 >
```



### C.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
```

### C.1.8 Les fonctions

**Syntaxe :** Déclaration d'une fonction

```
let <nom> (<liste parametres>) =
  (* expression simple ou sequence d'instructions
    definissant la fonction *)
```

### C.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 *)
```

## C.2 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, open, ref, then, to, true, while.**

**Rappel** Les mots-clés du langage ne peuvent pas servir d'identificateurs de variables.

## C.3 Fonctions prédéfinies du langage

### C.3.1 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 x` 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`.  
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.  
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.  
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.

### C.3.2 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.

### C.3.3 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`.

## C.4 Directives

`#quit` : pour quitter l'interprète.  
`#use <nom fichier>` : pour charger un fichier contenant des déclarations et expressions à évaluer.

## Annexe D

# Messages d'erreurs de l'interprète du langage

**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
```

# Bibliographie

- [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/>.



# Liste des algorithmes

0.1	Calcul de $n!$ . . . . .	6
1.1	Mettre les trois cartes du tas 1 vers les tas 2, 3 et 4 . . . . .	16
2.1	Solution du problème . . . . .	21
3.1	Solution du problème . . . . .	33
4.1	Compter les cartes du tas 1 . . . . .	44
5.1	Calcul de la somme des $n$ premiers nombres entiers. . . . .	51
5.2	Calcul de la somme des $n$ premiers nombres entiers avec une boucle pour . . . .	52
5.3	Calcul du terme d'indice $n$ d'une suite récurrente . . . . .	54
5.4	Calcul du premier terme satisfaisant une condition . . . . .	55





# Table des figures

1	QR-code du site du cours . . . . .	7
1.1	Le tas 1 après l'instruction <code>init_tas (1,"TCP")</code> . . . . .	16
1.2	Situation obtenue après l'instruction <code>deplacer_sommet (1,2)</code> . . . . .	17
1.3	Situation finale souhaitée . . . . .	18
2.1	Un état des tas de cartes . . . . .	24
7.1	Table des 128 premiers caractères (code ASCII) . . . . .	74
A.1	Transformation des tas par déplacement d'une carte . . . . .	86
B.1	Système de coordonnées du module <code>Camlgraph</code> . . . . .	97
B.2	Figure produite en utilisant le module <code>Camlgraph</code> . . . . .	99



# Table des matières

<b>1</b>	<b>Expressions et instructions</b>	<b>9</b>
1.1	Expressions . . . . .	9
1.2	Le logiciel OBJECTIVE CAML . . . . .	9
1.3	Premières expressions en CAML . . . . .	10
1.3.1	Expressions numériques . . . . .	10
1.3.2	Expressions booléennes . . . . .	12
1.3.3	Caractères et chaînes de caractères . . . . .	13
1.4	Instructions . . . . .	14
1.4.1	Instructions d'impression . . . . .	14
1.4.2	Instructions du module <b>Cartes</b> . . . . .	15
1.4.3	Séquence d'instructions . . . . .	15
1.5	Exercices . . . . .	19
1.5.1	Expressions numériques . . . . .	19
1.5.2	Instructions . . . . .	19
<b>2</b>	<b>Conditions, expressions et instructions conditionnelles</b>	<b>21</b>
2.1	Le problème . . . . .	21
2.1.1	Algorithme . . . . .	21
2.2	Expressions booléennes . . . . .	21
2.2.1	Booléens . . . . .	21
2.2.2	Expressions simples . . . . .	22
2.2.3	Expressions composées . . . . .	22
2.2.4	Propriétés des opérateurs . . . . .	23
2.3	En CAML . . . . .	24
2.3.1	Booléens . . . . .	24
2.3.2	Expressions booléennes . . . . .	24
2.3.3	Attention! . . . . .	25
2.3.4	Remarque sur les opérateurs <b>&amp;&amp;</b> et <b>  </b> . . . . .	26
2.4	Expressions conditionnelles . . . . .	26
2.5	Instructions conditionnelles . . . . .	27
2.5.1	Remarque sur le parenthésage . . . . .	28
2.6	Méthodologie . . . . .	29
2.7	Exercices . . . . .	30

<b>3</b>	<b>Instructions conditionnellement répétées</b>	<b>33</b>
3.1	Un problème . . . . .	33
3.2	Algorithme . . . . .	33
3.3	Sortie d'une boucle <b>tant que</b> . . . . .	33
3.4	La boucle <b>tant que</b> en CAML . . . . .	33
3.5	Méthodologie . . . . .	34
3.6	Exercices . . . . .	35
<b>4</b>	<b>Constantes, variables</b>	<b>37</b>
4.1	Types de données . . . . .	37
4.1.1	Les booléens . . . . .	37
4.1.2	Les nombres entiers . . . . .	37
4.1.3	Les nombres flottants . . . . .	38
4.1.4	Types définis par le module <b>Cartes</b> . . . . .	38
4.2	Variables . . . . .	39
4.2.1	Notion de variable . . . . .	39
4.2.2	Déclaration simple de variables . . . . .	39
4.2.3	Identificateurs de variables . . . . .	39
4.2.4	Environnement . . . . .	40
4.2.5	Évaluation d'une expression . . . . .	40
4.2.6	Masquage d'une variable . . . . .	40
4.2.7	Déclaration simultanée de variables . . . . .	41
4.2.8	Variables locales à une expression . . . . .	42
4.2.9	Remarque sur la forme <b>let</b> . . . . .	43
4.3	Variables mutables . . . . .	44
4.3.1	Motivation . . . . .	44
4.3.2	Déclaration d'une variable mutable . . . . .	44
4.3.3	Référence à la valeur d'une variable mutable . . . . .	45
4.3.4	Affectation . . . . .	45
4.4	Afficher des données . . . . .	47
4.5	Le programme solution du problème 4.3.1 . . . . .	48
4.6	Exercices . . . . .	49
<b>5</b>	<b>La boucle pour</b>	<b>51</b>
5.1	La boucle Pour . . . . .	51
5.2	La boucle Pour en CAML . . . . .	52
5.2.1	Syntaxe de la boucle pour . . . . .	52
5.2.2	Remarque sur l'indice de boucle . . . . .	53
5.2.3	Boucle pour à indice décroissant . . . . .	53
5.3	Suites récurrentes . . . . .	54
5.3.1	Calcul du terme d'indice $n$ . . . . .	54
5.3.2	Calcul et affichage des termes d'indice 0 à $n$ . . . . .	54
5.3.3	Calcul du premier terme satisfaisant une condition . . . . .	55
5.3.4	Autres suites récurrentes . . . . .	55
5.4	Exercices . . . . .	57

<b>6</b>	<b>Les fonctions</b>	<b>61</b>
6.1	Les fonctions	61
6.1.1	Spécification d'une fonction	61
6.1.2	Déclaration d'une fonction en CAML	62
6.1.3	Variables locales	63
6.1.4	Fonctions à plusieurs paramètres	64
6.1.5	Fonctions sans paramètre	65
6.1.6	Intérêt des fonctions	66
6.2	Procédures	67
6.2.1	Spécification d'une procédure	67
6.2.2	Déclaration d'une procédure en CAML	67
6.2.3	Appel à une procédure	68
6.2.4	Intérêt des procédures	68
6.2.5	Méthodologie	68
6.3	Exercices	69
6.3.1	Fonctions	69
6.3.2	Procédures	71
<b>7</b>	<b>Caractères et chaînes de caractères</b>	<b>73</b>
7.1	Les caractères	73
7.1.1	Définition	73
7.1.2	Fonctions <code>int_of_char</code> et <code>char_of_int</code>	74
7.1.3	Comparaison de caractères	75
7.2	Les chaînes de caractères	75
7.2.1	Définition	75
7.2.2	Notation indicielle	76
7.2.3	Concaténation	77
7.2.4	Comparaison de chaînes	78
7.3	Quelques fonctions prédéfinies sur les chaînes	78
7.4	Exercices	80
<b>A</b>	<b>Les cartes</b>	<b>83</b>
A.1	Présentation	83
A.2	Utilisation du module <code>Cartes</code>	83
A.3	Le langage du module <code>Cartes</code>	84
A.3.1	Description de tas	84
A.3.2	Action	85
A.3.3	Tests sur les tas	86
A.3.4	Comparaison des valeurs	87
A.3.5	Contrôle de l'affichage	87
A.3.6	Réparer l'automate	89
A.3.7	Obtenir de l'aide	89
A.4	Exercices	90
A.4.1	Descriptions de tas	90
A.4.2	Séquence	91
A.4.3	Conditionnelle	91
A.4.4	Itération	93
A.4.5	Fonctions et procédures	95

<b>B</b>	<b>Le module Camlgraph</b>	<b>97</b>
B.1	Présentation . . . . .	97
B.2	Module <b>Camlgraph</b> . . . . .	98
B.2.1	Gestion des fenêtres graphiques . . . . .	98
B.2.2	Gestion des couleurs . . . . .	99
B.2.3	Point et dessin de lignes . . . . .	100
B.2.4	Texte . . . . .	101
B.2.5	Remplissage . . . . .	101
<b>C</b>	<b>Éléments du langage étudiés dans ce cours</b>	<b>103</b>
C.1	Syntaxe . . . . .	103
C.1.1	Commentaires . . . . .	103
C.1.2	Expressions et instructions . . . . .	103
C.1.3	Expressions conditionnelles . . . . .	103
C.1.4	Instructions conditionnelles . . . . .	104
C.1.5	Instructions conditionnellement répétées . . . . .	104
C.1.6	Constantes, variables . . . . .	104
C.1.7	La boucle pour . . . . .	105
C.1.8	Les fonctions . . . . .	105
C.1.9	Caractères et chaînes de caractères . . . . .	105
C.2	Mots-clés du langage . . . . .	105
C.3	Fonctions prédéfinies du langage . . . . .	105
C.3.1	Fonctions de conversion . . . . .	105
C.3.2	Impressions . . . . .	106
C.3.3	Du module <b>String</b> . . . . .	106
C.4	Directives . . . . .	106
<b>D</b>	<b>Messages d’erreurs de l’interprète du langage</b>	<b>107</b>