I

Présentation du langage de programmation CAML

Ce premier chapitre est consacré à la présentation de Caml, le langage de programmation choisi dans cet ouvrage. Les sections 1 à 4 sont destinées au lecteur débutant en Caml, le lecteur averti complétera ses connaissances avec la section 5, enfin nous attendons remarques, suggestions, commentaires... du lecteur expert.

Cette introduction à Caml est indispensable au début de notre ouvrage. En effet, pour exposer de façon précise les méthodes de programmation et les algorithmes des chapitres à venir, il est préférable de commencer par s'accorder sur une « langue commune ». Nous aurions pu choisir de rester dans la généralité en écrivant les programmes en pseudo-langage comme :

début
  tant que l'étudiant ne comprend pas
  faire répéter l'explication
fin

Mais ceci est souvent, à notre avis, peu rigoureux, peu concis et écarte un attrait essentiel de l'informatique : la *confrontation à la réalité*. Afin que le lecteur puisse connaître la satisfaction de voir s'exécuter son programme sous ses yeux, il valait mieux choisir un langage de programmation existant sur ordinateur. Le langage Caml, plus précisément son implémentation pour micros Caml-Light, possède, outre sa gratuité, de nombreux attraits pédagogiques (interactivité, clarté, justesse) et ne déroutera pas le lecteur familier d'autres langages de programmation répandus[1].

Pour une étude exhaustive du langage Caml, nous renvoyons le lecteur aux ouvrages de Leroy et Weis [34] et de Cousineau et Mauny [12].

Signalons aussi que l'on peut se procurer *gracieusement* le logiciel à l'adresse internet suivante :

`http://www.inria.fr/pub/lang/caml-light`

1. Caml a été choisi comme langage de programmation de l'Option Informatique en Classes Préparatoires aux Grandes Écoles.

La version utilisée dans cet ouvrage est la 0.73 française[2]. Il va sans dire que nous ne présenterons pas toutes les possibilités de Caml. Les commandes de Caml sont très bien documentées dans l'aide fournie avec le logiciel[3], aide à laquelle il convient de se reporter.

Pour profiter pleinement de cette présentation, nous conseillons vivement au lecteur de tester au fur et à mesure les exemples proposés sur son ordinateur favori...

1 Caml est un langage interactif

Nous appellerons *session* Caml tout ce qui se passe entre le lancement de Caml et son arrêt par la commande `quit();;`. Une fois Caml lancé, l'utilisateur travaillant avec un système d'exploitation à fenêtres (comme Windows, Mac-OS, X11...) dispose d'une fenêtre d'`Entrée` (Caml Light Input en anglais) dans laquelle il saisit ses commandes et d'une fenêtre de `Résultats` (Caml Light Output en anglais) où le compilateur Caml affiche ses résultats ; les sorties graphiques sont représentées dans la fenêtre `Graphique` (voir figure I.1).

Figure I.1 : *Caml à l'écran.*

2. Caml-Light est destiné à l'enseignement de la programmation, Caml existe aussi en version plus « professionnelle » et objet : Objective Caml.

3. Le lecteur trouvera notamment sur le site Web les FAQs, questions fréquemment posées sur Caml, qui sont une source précieuse d'informations.

La convivialité de Caml vient tout d'abord de son caractère interactif. Ainsi lorsque l'utilisateur écrit une commande[4] à la suite du symbole d'invite `#` (le *prompt* en anglais) dans la fenêtre d'`Entrée` (terminée par '`;;`'[5]), Caml l'exécute (en fait l'*évalue*[6]), affiche le résultat dans la fenêtre `Résultats` et écrit une nouvelle invite (il *rend la main* à l'utilisateur). Ceci permet de vérifier rapidement la validité d'un code Caml (avant une inclusion éventuelle dans un programme plus important). L'édition séparée de programmes[7] à l'aide de votre éditeur préféré[8] est, bien sûr, également possible.

Commençons notre initiation à Caml par les opérations arithmétiques élémentaires (les textes en `caractères style machine à écrire` qui vont suivre sont des extraits de sessions Caml copiés depuis la fenêtre `Résultats`, ce qui suit le `#` est tapé par l'utilisateur, le reste est renvoyé par Caml) :

```
#1+1;;
- : int = 2
#2*3;;
- : int = 6
#4/2;;
- : int = 2
#5.5 *. 2.5;;
- : float = 13.75
```

On constate que Caml accompagne le résultat de ses calculs d'une information fondamentale : leur type. Ainsi il fait la distinction entre les entiers (`int`) et les réels informatiques ou *flottants* (`float`). Les opérations sur ces derniers sont caractérisées par l'usage du `.` après les opérateurs : `+.`, `*.` etc. Signalons que Caml connaît les fonctions usuelles comme `sin`, `cos`, `exp`, `log`, `mod`...

```
#exp(1.);;
- : float = 2.71828182846
#4. *. atan(1.);;
- : float = 3.14159265359
```

Ne nous attardons pas trop car Caml n'est pas une calculette (pour une utilisation exclusivement mathématique, on pourra préférer à Caml un outil de calcul formel comme Maple par exemple).

L'interactivité permet à Caml de détecter rapidement une erreur dans un programme. Il manifeste sa mauvaise humeur en indiquant par une succession d'accents circonflexes `^` l'endroit qu'il croit incorrect et en affichant un message d'erreur en général explicite.

4. Hormis un texte se trouvant entre les caractères '`(*`' et '`*)`' qui n'est pas lu par le compilateur car il s'agit là des délimiteurs de *commentaires* en Caml.
5. Le '`;`' simple est réservé à la séparation des instructions.
6. Signalons, pour le lecteur averti, que Caml n'est pas interprété. Caml compile en fait l'expression de l'utilisateur, renvoie son résultat, puis rend la main à l'utilisateur.
7. Un programme est chargé en mémoire par le menu `Inclure` (`Include` en anglais) et non par le menu `Charger` (`Load` en anglais) !
8. Signalons, pour les utilisateurs de Windows, l'existence de l'excellent éditeur *cmdcaml* interfacé avec Caml, disponible également sur le site Web.

```
#2 _ 3;;
Entrée interactive:
>2 _ 3;;
>  ^
Erreur de syntaxe.
#2 - 3;;
- : int = -1
```

Ici, Caml proteste car il ne connaît pas l'opérateur `_` (l'utilisateur l'a confondu avec le signe `-`).

2 Caml est un langage (fortement) typé

Rappelons que Caml ne se contente pas d'afficher le résultat d'une commande de l'utilisateur, il détermine également son *type* (on dit que Caml a « typé l'expression »). Nous avons vu les types `int` et `float` qui avertissent l'utilisateur que le résultat de son calcul est un entier ou un flottant. Cette indication de type est aussi donnée pour toutes les fonctions connues de Caml[9], par exemple :

```
#sin;;
- : float -> float = <fun>
```

signifie que `sin` est une application qui prend pour argument un flottant et renvoie comme résultat un flottant. C'est une indication qui peut se révéler précieuse pour la compréhension et le bon usage d'une application donnée.

Passons en revue d'autres types élémentaires[10] connus de Caml :

- les booléens `bool` avec comme opérateurs le « ou » (`or` ou `||`), le « et » (`&&`)...
- les caractères `char` encadrés par des accents graves,
- les chaînes de caractères `string` qui sont encadrées par des guillemets.

Pour comprendre la session Caml qui suit, il suffit de savoir que l'accent circonflexe désigne la concaténation (c'est-à-dire la mise bout à bout) de deux chaînes de caractères et que `nth_char` est la fonction Caml qui renvoie le n-ième caractère d'une chaîne de caractères donnée (les caractères sont numérotés à partir de 0).

```
#(1<2) && (4=3);;
- : bool = false
#(1<2) || (4=3);;
- : bool = true
#"Bonjour";;
- : string = "Bonjour"
#nth_char "Bonjour" 3;;
- : char = `j`
#"Ceci est"^" une chaîne de caractères";;
- : string = "Ceci est une chaîne de caractères"
```

9. ... et donc pour toutes celles que vous allez définir !
10. Le type `exn` et le type `unit` seront vus respectivement en 5.6 et 4.1.

Il existe en Caml des types plus élaborés comme :

- les tableaux qui sont délimités par `[|` et `|]` (on parle de *vecteurs* en Caml),
- les listes, délimitées par `[` et `]`,

les éléments étant séparés par des '`;`'.

À l'instar de beaucoup de langages de programmation, un tableau Caml, ou *vecteur*, est une suite d'objets munie d'une indexation qui assure un accès en temps constant à une composante (ou coordonnée pour conserver le vocabulaire vectoriel) quelconque donnée. Ainsi `v.(k)` retournera la k-ième composante du vecteur `v`. La longueur d'un vecteur est une constante fixée lors de sa création. Une liste Caml correspond à une liste chaînée en Pascal : il s'agit d'une suite d'objets dont la longueur est dynamique (c'est-à-dire variable au cours de la session) et qui, en contrepartie, ne permet un accès qu'à son premier élément à l'aide de la fonction `hd` (*head* en anglais, c'est-à-dire tête en français), l'accès à la k-ième composante nécessitant k opérations.

On peut illustrer ceci avec la session Caml suivante :

```
#[|"un" ; "vecteur"; "de"; "string" |];;
- : string vect = [|"un"; "vecteur"; "de"; "string"|]
#["une"; "liste"; "de"; "string"];;
- : string list = ["une"; "liste"; "de"; "string"]
#1::[2;3];;
- : int list = [1; 2; 3]
#hd [7;8;9];;
- : int = 7
#tl [1;2;3];;
- : int list = [2; 3]
#hd (tl [1;2;3]);;
- : int = 2
#[|7;8;9|];;
- : int vect = [|7; 8; 9|]
#[|7;8;9|].(2);;
- : int = 9
```

Les commandes Caml ci-dessus sont assez explicites : ainsi le '`::`' (prononcer[11] le « conse ») permet d'ajouter un élément en tête d'une liste ; la fonction `tl` (*tail* en anglais) renvoie la queue d'une liste, c'est-à-dire la liste privée de son premier élément (comme en C, les vecteurs de longueur n sont indexés de 0 à $n-1$). Signalons que la liste vide (resp. le vecteur vide) est notée `[ ]` (resp. `[| |]`).

On peut également définir des *produits cartésiens* de types, c'est-à-dire des paires (et plus généralement des n-uplets) d'objets de types identiques ou différents. La virgule '`,`' est le délimiteur des produits cartésiens et l'étoile '`*`' le symbole utilisé par Caml pour noter le produit cartésien :

```
#("une paire", "de string");;
- : string * string = "une paire", "de string"
#(1,"abc",true);;
- : int * string * bool = 1, "abc", true
```

11. « conse » pour constructeur.

Il existe des fonctions de conversion de type. Leur nom est toujours de la forme `type1_of_type2`[12] ce qui les rend assez explicites :

```
#string_of_int;;
- : int -> string = <fun>
#string_of_int 3;;
- : string = "3"
#float_of_int 4;;
- : float = 4.0
#list_of_vect [|1;2;3|];;
- : int list = [1; 2; 3]
```

Les types (et fonctions) automatiquement disponibles en début de session Caml font partie de la *bibliothèque du noyau* (*core library* en anglais) et sont documentés dans l'aide correspondante. Il existe d'autres types et fonctions prédéfinis regroupés en modules dans la *bibliothèque standard* (*standard library* en anglais) que l'on doit charger explicitement à l'aide d'une commande de la forme :

`#open "`*`nom-du-module`*`";;`

et qui sont documentés dans l'aide correspondante. Voici un exemple illustrant l'usage de la fonction `int n` (`n` désigne un entier) du module `random` qui renvoie un entier aléatoire entre `0` et `n-1` :

```
#int 10;;
Entrée interactive:
>int 10;;
>^^^
L'identificateur int n'est pas défini.
##open "random";;
#int 10;;
- : int = 4
#int 10;;
- : int = 2
#int 10;;
- : int = 3
```

(noter le double dièse `##` dû à l'invite et à la commande `#open`!).

L'utilisateur peut également enrichir les types connus par Caml et définir ses propres types. Voyons pour l'instant la construction la plus simple où l'on énumère[13] tous les objets appartenant au nouveau type :

```
#type animal_de_compagnie = Chien | Chat | Oiseau;;
Le type animal_de_compagnie est défini.
#Chien;;
- : animal_de_compagnie = chien
```

12. Enfin presque... la conversion d'un caractère en une chaîne de caractère (de longueur 1) se fait à l'aide de la fonction `char_for_read`.
13. Notez que l'usage veut que l'on commence le nom d'un constructeur de type par une majuscule et celui des variables par une minuscule.

Ceci présente un intérêt assez « limité »[14] ! (voir une utilisation en page 14). Les autres constructions de type seront exposées au 5.4 de ce chapitre.

Retenons en bref que *tous les objets, toutes les applications ont un type* en Caml, à la différence d'autres langages de programmation comme Pascal ou C qui ne sont pas aussi complètement typés. Il existe même des langages, comme LISP, qui sont totalement non typés : ceux-ci perdent alors la détection *a priori* d'erreurs de programmation, ce qui est un atout principal fourni par le typage à l'utilisateur.

Ajoutons que le compilateur Caml n'apprécie guère que l'on mélange les « torchons » et les « serviettes ». Les listes et les vecteurs sont ainsi *monotypes* c'est-à-dire constitués obligatoirement d'objets du même type :

```
#[1; "un" ; 2; 3];;
Entrée interactive:
>  [1;"un";2;3];;
> ^^^^^^
Cette expression est de type string list,
mais est utilisée avec le type int list.
```

De même, on ne peut pas mélanger pas abusivement flottants et entiers :

```
#1+2.3;;
Entrée interactive:
>1+2.3;;
>  ^^^
Cette expression est de type float,
mais est utilisée avec le type int.
#1.0 +. 2.3;;
- : float = 3.3
#float_of_int 1 +. 2.3;;
- : float = 3.3
```

Ainsi, on n'additionne pas un entier et un flottant (`1.0` est un flottant !). Il s'agit d'objets différents (ce qui est clair lorsque l'on songe à leur représentation machine[15]).

La rigueur de Caml confinerait-elle au rigorisme ? Nous allons expliquer pourquoi cette contrainte apparente qu'est la vérification systématique de type effectuée par Caml conduit à une programmation précise, souvent exhaustive et constitue ainsi *une aide* et non un obstacle pour le programmeur. Pour étayer ce propos, il nous faut à présent approfondir notre connaissance de Caml et introduire les « blocs de programme » que sont les *fonctions*... (enfin !).

14. On peut tout de même remarquer qu'il n'est pas nécessaire de disposer d'un type `bool` prédéfini : celui-ci peut très bien être déclaré par `type bool = true | false;;`.
15. ... et qui l'est plus encore lorsque l'on réfléchit à la construction mathématique des réels ! On verra en page 37 comment définir un type générique `nombre` qui permettra d'effectuer des opérations entre flottants et entiers.

3 Caml est un langage fonctionnel

3.1 Les fonctions Caml

Les fonctions constituent l'unité de base de la programmation Caml. La définition des fonctions en Caml suit la formulation mathématique usuelle. Ainsi :

```
#let f = function x -> x*x;;
f : int -> int = <fun>
```

signifie « soit `f` l'application qui à `x` associe son carré ». Le type de l'objet nouvellement créé est `<fun> int -> int` soit « une fonction qui à un entier associe un entier ». Son utilisation est des plus simples :

```
#f(3);;
- : int = 9
#f 2;;
- : int = 4
```

On note que les parenthèses sont optionnelles.

Le lecteur peut se demander comment Caml a trouvé que le type de la fonction est `int -> int`. Lorsqu'une nouvelle fonction est déclarée, Caml recherche le type qui convient à cette fonction[16] et pour cela utilise toutes les indications possibles ! Ici, le caractère `*` entre les deux `x` est une opération qui associe impérativement un entier[17] à deux entiers ; ceci indique alors à Caml que `x` ne peut être qu'entier et permet l'identification du type de `f`.

Signalons que la déclaration de fonctions peut également se faire selon les syntaxes :

```
#let f x = x*x;;
f : int -> int = <fun>
#let f = fun x -> x*x;;
f : int -> int = <fun>
```

(la subtile différence entre les mots-clés `fun` et `function` sera expliquée en 5.2, elle ne concerne pas les fonctions à un seul argument).

L'égalité entre fonctions peut être exprimée au niveau des valeurs :

```
#let g x = f x;;
g : int -> int = <fun>
#g(3);;
- : int = 9
```

ou bien, en poussant l'analogie avec les mathématiques, au niveau fonctionnel c'est-à-dire en ôtant les arguments[18] :

```
#let h = f;;
h : int -> int = <fun>
#h(3);;
- : int = 9
```

16. Caml recherche en fait le type « le plus général ». Nous reviendrons sur ce point en 5.1.
17. La commande `prefix *;;` vous permettra d'afficher le type de `*`. C'est la version préfixe de l'opérateur binaire.
18. Il s'agit d'un procédé assez fréquent en Caml issu de la η-abstraction du λ-calcul, utilisé notamment pour les fonctions à plusieurs arguments.

Pour terminer avec les fonctions à un argument, mentionnons qu'en Caml on n'est pas obligé de nommer les fonctions. On peut utiliser des fonctions anonymes comme dans l'exemple suivant :

```
#(function x -> x + 1) 2;;
- : int = 3
```

La définition des fonctions à plusieurs arguments[19] suit le même principe :

```
#let somme1 (x,y) = x+y;;
somme1 : int * int -> int = <fun>
#somme1 (1,4);;
- : int = 5
```

On peut alors instancier (c'est-à-dire donner une valeur à) certains arguments :

```
#let h z = somme1 (2,z);;
h : int -> int = <fun>
#h 3;;
- : int = 5
```

ce qui définit l'application partielle d'une variable entière `h(z) = somme1(2,z)`. On peut également définir l'addition de deux entiers selon la syntaxe :

```
#let somme2 x y  = x+y;;
somme2 : int -> int -> int = <fun>
#somme2 2 3;;
- : int = 5
```

ce qui fournit une fonction *somme* analogue, mais d'un type distinct. La seconde syntaxe permet l'*abstraction* :

```
#let h  = somme2 2;;
h : int -> int = <fun>
#h 3;;
- : int = 5
```

qui trouve son explication dans le type `int -> int -> int` proposé précédemment par Caml pour `somme2` : l'application qui à l'entier `x` associe l'application qui à l'entier `y` associe `x+y`.

Le mot-clé `let rec` est réservé à la programmation des *fonctions récursives*. Il s'agit d'applications dans la définition desquelles apparaît le nom de l'application elle-même. Considérons par exemple la définition d'une suite récurrente $(u_n)_{n\in\mathbb{N}}$ vérifiant la relation linéaire : $\forall\ n \geqslant 2$, $u_n = u_{n-1} + u_{n-2}$. Ceci se programme en Caml par :

```
#let rec u(n) = u(n-1) + u(n-2);;
u : int -> int = <fun>
```

Avec cette définition, Caml est incapable de calculer une quelconque valeur de u_n puisqu'il ne connaît ni u_0 ni u_1. Nous compléterons cet exemple en 3.3 et approfondirons la récursivité au chapitre II.

19. Citons aussi les fonctions *sans argument* qui emploient la double parenthèse `()` comme `quit();;`.

3.2 Les variables

Le mot « variable » est familier au lecteur. On parle en mathématiques de fonctions de plusieurs variables, de variable réelle etc. Cependant le terme variable évoque l'idée de « variation » qui est souvent éloignée de la pratique de Caml comme nous allons le voir par la suite. Aussi, nous préférerons le vocable « d'identificateur » à celui de « variable » à l'exception d'expressions consacrées comme « variable locale », « variable globale »...

Nous n'avons pour l'instant attribué un nom qu'à des fonctions, ceci à l'aide du mot-clé `let`. Cette construction est en fait générale et permet de nommer toute expression Caml. Nous allons le faire à présent pour des valeurs : c'est la définition des variables en Caml.

La commande :

`let x = a in` *une-expression-Caml* `;;`

définit la *variable locale* `x`. Elle signifie qu'au nom `x` on doit substituer la valeur `a` dans *toute* l'expression Caml qui suit le `in`. La valeur de `x` n'est connue que localement. En dehors de cette expression Caml la lettre `x` n'est plus définie. Voici un exemple :

```
#x;;
Entrée interactive:
>x;;
>^
L'identificateur x n'est pas défini.
#let x=2 in x+3;;
- : int = 5
#x;;
Entrée interactive:
>x;;
>^
L'identificateur x n'est pas défini.
```

Pour la définition des variables locales il existe également une syntaxe postposée (c'est-à-dire dans laquelle la définition suit, au lieu de précéder, l'expression principale) à l'aide du mot-clé `where` :

```
# x + 3 where x=2;;
- : int = 5
```

On peut faire des déclarations en cascade :

```
#let x = 3 in let y = 2 in x+y;;
- : int = 5
```

ou utiliser un couple (et plus généralement un n-uplet) d'identificateurs :

```
#let (x,y)=(3,2) in x+y;;
- : int = 5
```

ou encore le mot-clé `and`[20] :

```
#let x=3 and y=2 in x+y;;
- : int = 5
```

La définition des variables *globales* en Caml se fait comme suit :

`let x =` *`une-valeur`*`;;`

ce qui se traduit par « Soit x = une-valeur ». Par exemple :

```
#let pi=4. *. atan(1.);;
pi : float = 3.14159265359
```

Dans toute la suite de la session, l'identificateur choisi est relié à sa valeur, ainsi :

```
#let x=5;;
x : int = 5
#x+1;;
- : int = 6
#let f = fun y -> x+y;;
f : int -> int = <fun>
#f(2);;
- : int = 7
```

La valeur de `x` ne peut plus varier au cours de la session. On ne peut la modifier qu'en donnant une nouvelle définition de `x` : `let x=` *`une-autre-valeur`*`;;`. Attention celle dernière valeur n'est pas rétroactive ! Tout ce qui a précédé dans la session cette nouvelle définition et qui utilisait `x` continue d'employer son ancienne valeur[21], comme on le constate en poursuivant la session Caml précédente :

```
#let x=3;;
x : int = 3
#f(2);;
- : int = 7
```

Il s'agit bien en effet de *définitions* et non d'*affectations*, au sens où l'entendent la plupart des langages de programmation (en général avec le signe `:=`). L'égalité en Caml est bien plus proche de l'égalité mathématique[22] que ne l'est l'affectation traditionnelle dans un autre langage où l'on peut écrire par exemple : `x:= x+1` (ce qui peut paraître mathématiquement surprenant[23]).

Le comportement des variables globales en Caml s'apparente donc en fait à ce que l'on attend des constantes dans la plupart des autres langages de programmation ! Toutefois, comme nous le verrons, ces variables (locales et globales) suffisent amplement pour la programmation d'un grand nombre de problèmes.

20. Ces différentes syntaxes ne sont pas en fait complètement équivalentes. Si l'on désire impérativement que `x` soit évalué avant `y` alors il *faudra* utiliser les deux `let ... in` imbriqués, le `and` ne garantissant pas l'ordre d'évaluation.
21. Il en est de même (et c'est sécurisant) au niveau des définitions de fonctions : si l'utilisateur crée une fonction de même nom qu'une fonction prédéfinie en Caml, il ne modifie pas les autres fonctions du système qui font usage de cette dernière.
22. Il existe en Caml, un autre test d'égalité `==` dont nous parlerons en 5.3.
23. Le lecteur remarquera en effet que dans cette instruction le `x` du membre gauche désigne le nom de la variable, alors que le `x` du membre droit désigne sa valeur : quelle collusion de notations !

Heureusement, il existe aussi en Caml, et on peut soi-même définir, des objets « évolutifs » en cours de session. Nous aborderons ainsi la notion de référence au 4.3 et celle de *type mutable* au 5.4.

3.3 Le filtrage

Un motif pour utiliser Caml !

Voyons à présent un des aspects les plus appréciables de Caml : le *filtrage*. Il est fréquent d'avoir à programmer une fonction définie par cas, sous-cas etc. La syntaxe Caml se révèle alors à la fois concise et claire.

Par exemple, la définition récursive de $n!$ est :

$$0! = 1 \qquad \text{et} \qquad \forall\, n \geqslant 1 \quad n! = n \times (n-1)!$$

Le programme Caml correspondant suit directement cette description mathématique :

La première `|` est facultative, mais esthétique, non ?

```
#let rec factorielle = function
     | 0 -> 1
     | n -> n*factorielle (n-1);;
factorielle : int -> int = <fun>
#factorielle 5;;
- : int = 120
```

On lit le programme Caml ci-dessus de la façon suivante :

factorielle est la fonction qui à zéro associe 1
et qui à n associe $n \cdot factorielle(n-1)$.

Les différents cas sont séparés par des `|` et envisagés successivement (Caml choisit *le premier cas* qui est compatible avec son paramètre d'appel n).

Il faut faire très attention aux priorités implicites dans la syntaxe de Caml. Le compilateur comprend l'expression `n*factorielle n-1` comme `(n*factorielle n)-1` : les parenthèses[24] dans `factorielle (n-1)` sont indispensables. L'omission des parenthèses est une source d'erreurs fréquentes !

Excès de parenthèses ne nuit pas !

On peut rendre le paramètre n du filtrage explicite dans la définition de la fonction avec la syntaxe `match with` :

```
#let rec factorielle n = match n with
     | 0 -> 1
     | n -> n * factorielle (n-1);;
factorielle : int -> int = <fun>
#factorielle 5;;
- : int = 120
```

Complétons à présent notre exemple de suite récurrente linéaire d'ordre 2 avec cette fois-ci les valeurs initiales[25] $u_0 = 0$ et $u_1 = 1$. Ceci donne :

24. De façon générale, Caml associe à gauche. Ainsi `e1 e2 e3` sera interprété comme `(e1 e2) e3`.
25. Le lecteur reconnaît la suite de Fibonacci.

```
#let rec u = function
     | 0 -> 0
     | 1 -> 1
     | n -> u(n-1) + u(n-2);;
u : int -> int = <fun>
#u 8;;
- : int = 21
```

ou encore, en rendant le paramètre `n` explicite :

```
#let rec u n = match n with
     | 0 -> 0
     | 1 -> 1
     | n -> u(n-1) + u(n-2);;
u : int -> int = <fun>
#u 8;;
- : int = 21
```

Toutefois cette façon de programmer la suite de Fibonacci est loin d'être efficace. Nous l'expliquerons et proposerons d'autres méthodes pour son calcul au chapitre II.

La programmation du ou exclusif XOR (voir chapitre VI) fournit un troisième exemple de filtrage, non récursif cette fois. Le XOR est défini de la façon suivante en logique des propositions :

- true XOR true = false,
- true XOR false = true,
- false XOR true = true,
- false XOR false = false.

Le programme Caml correspondant s'écrit :

```
let XOR = function
     | (false, false) -> false
     | (false, true)  -> true
     | (true, false)  -> true
     | (true, true)   -> false;;
XOR : bool * bool -> bool = <fun>
#XOR (1=1, 2>0);;
- : bool = false
```

En fait, on peut abréger ce code[26] en remarquant qu'il suffit de définir les cas où XOR renvoie `true`. Pour cela, on peut utiliser le symbole réservé `_` qui traite *tous les autres cas.*

26. On pourrait faire encore plus bref en écrivant :
```
| (false,x) ->x
| (true,x) ->not x
```
mais pas en écrivant :
```
| (x,x) ->false
| _ ->true
```
comme l'explique la note suivante.

Cela donne :

```
let XOR = function
     | (false, false) -> false
     | (true, true)   -> false
     |     _          -> true;;
XOR : bool * bool -> bool = <fun>
#XOR (3=1, 5>0);;
- : bool = true
```

Plus généralement le _ sert à filtrer les parties indifférentes d'un motif. On peut ainsi définir le ou logique OR par :

```
let OR = function
     | (false, false) -> false
     | (true, _)      -> true
     | (_, true)      -> true;;
```

Le *filtrage* est l'opération qui consiste à faire correspondre, dans leur ordre d'écriture, l'argument de la fonction (dans l'exemple précédent un couple de booléens) avec les différents cas, ou *motifs*, intervenant dans sa définition.

Dès que l'on a défini un type, il est possible de filtrer sur celui-ci. En reprenant le type `animal_de_compagnie` de la page 6, on peut, par exemple, définir :

```
#let vole = function
     | Chien  -> false
     | Chat   -> false
     | Oiseau -> true;;
vole : animal_de_compagnie -> bool = <fun>
#vole Oiseau;;
- : bool = true
```

En fait, le filtrage est plus puissant que cela. L'intérêt d'un motif est, en effet, de pouvoir contenir des variables[27] ; un motif peut ainsi représenter un ensemble de cas à chaque fois. Pour illustrer ceci, construisons un exemple d'école : imaginons que l'on dispose de deux listes d'entiers, notées `liste1` et `liste2`, et que l'on désire les ajouter terme à terme (si l'on considère qu'il s'agit des listes des coefficients de deux polynômes, on est en train de calculer le polynôme somme). La programmation récursive de la fonction `add` en question distingue trois cas pour le couple `(liste1,liste2)` :

- soit l'une des deux listes est vide et l'on renvoie l'autre[28] (ceci regroupe deux cas),
- soit les deux listes sont au moins de longueur 1 chacune et l'on ajoute leurs deux premiers éléments avant de relancer `add` sur les restes.

27. Avec la restriction suivante toutefois : les variables d'un motif doivent toutes être distinctes. Nous reviendrons sur ce point au 5.3 lorsque nous évoquerons les *filtres gardés* et *l'unification*.
28. Ce cas se produit si les deux listes initiales ne sont pas de la même longueur.

L'implémentation Caml filtre les différents cas :

```
#let rec add (liste1,liste2) = match (liste1,liste2) with
     |  ([],liste)   -> liste
     |  (liste,[])   -> liste
     | (e1::r,e2::s) -> (e1+e2)::(add (r,s));;
add : int list * int list -> int list = <fun>
#add ([1;2;3],[0;4;5;6]);;
- : int list = [1; 6; 8; 6]
```

Il est fréquent, comme on l'a vu avec le mot-clé `function`, d'omettre le *dernier* argument d'une fonction définie par filtrage ; on peut ainsi proposer une nouvelle implémentation de la fonction `add` (qui n'envisage plus le cas où la première liste est vide) :

```
#let rec add2 liste = function
     | []   -> liste
     | e::r -> (hd(liste) + e)::(add2 (tl liste) r);;
add2 : int list -> int list -> int list = <fun>
#add2 [1;2;3] [0;1];;
- : int list = [1; 3; 3]
```

Que se passe-t-il lorsque l'on oublie un cas dans un filtrage ? Observons, par exemple, la réaction du compilateur si l'on oublie le cas de la liste vide dans la définition précédente :

```
#let rec add3 (liste1,liste2) = match (liste1,liste2) with
 (e1::r,e2::s) -> (e1+e2)::(add3 (r,s));;
Entrée interactive:
>.............................match (liste1,liste2) with
> (e1::r,e2::s) -> (e1+e2)::(add3 (r,s))..
Attention: ce filtrage n'est pas exhaustif.
add3 : int list * int list -> int list = <fun>
```

On constate que Caml lance un avertissement mais ne refuse pas de compiler `add3`. Il en sera ainsi chaque fois que le filtrage ne sera pas exhaustif (ici, l'exécution de `add3` provoquera une erreur).

Signalons un autre message d'erreur qui se manifeste lorsque certains motifs sont redondants :

```
#let rec add4 (liste1,liste2) = match (liste1,liste2) with
     |  ([],liste)   -> liste
     |  (liste,[])   -> liste
     | (e1::r,e2::s) -> (e1+e2)::(add4 (r,s))
     |  ([],[])      -> [];;
Entrée interactive:
>     |  ([],[])      -> [];;
>        ^^^^^
Attention: ce cas de filtrage est inutile.
add : int list * int list -> int list = <fun>
```

En effet, le cas `([],[])` est déjà filtré par le premier motif `([],liste)` (et par le deuxième d'ailleurs). Les motifs peuvent donc ne pas être disjoints. Dans ce cas, c'est le premier dans la liste de filtrage qui sera exécuté.

Pour conclure cette section, retenons que la programmation par filtrage est particulièrement *lisible*. Nous l'apprécierons notamment pour des motifs complexes (comme en III 4 au niveau des structures arborescentes). Il s'agit d'un style de programmation puissant en Caml, qu'il faudra privilégier.

4 C'est aussi un langage impératif

Cette section est destinée aux nostalgiques des `while`, `:=`, `begin...end` etc. Enfin pas seulement, car l'aspect impératif de Caml rend de nombreux services et est à connaître.

Un programme dans un langage impératif comme Pascal, C... est une suite d'instructions, généralement séparées par des ';', que l'ordinateur exécute séquentiellement. Dans un langage fonctionnel, ceci est remplacé par une expression évaluée par le compilateur. Caml permet ces deux styles de programmation (voir II 3).

Avant tout il nous faut introduire un nouveau type de base de Caml.

4.1 Le type unit

Il existe en Caml des fonctions qui ne renvoient aucun résultat[29] et qui agissent, selon l'expression consacrée, par *effets de bords*, comme par exemple les fonctions d'impression `print_string`, `print_int`... (voir la section 5.5 consacrée aux entrées-sorties). Tout étant typé en Caml, un type particulier leur est consacré : le type *unit*. Voyons cela :

```
#print_string "bonjour";;
bonjour- : unit = ()
#print_string;;
- : string -> unit = <fun>
```

Ainsi, `print_string` prend comme argument une chaîne de caractère, l'affiche sur la sortie standard (ici l'écran) : ceci est une action de type `unit`.

```
#let v=[|7;8;9|];;
v : int vect = [|7; 8; 9|]
#v.(2)<-10;;
- : unit = ()
#v;;
- : int vect = [|7; 8; 10|]
```

L'indexation va de 0 à n-1...

Vous aurez compris que la flèche `<-` est l'opérateur qui modifie une composante d'un vecteur. C'est à nouveau une opération de type `unit`.

Une instruction d'un langage de programmation impératif est « typiquement » une action de type `unit`. Le type `unit` est ainsi omniprésent dans ce qui va suivre.

29. Le lecteur reconnaîtra les procédures de Pascal.

4.2 La conditionnelle

Une construction de base dans un langage de programmation est le branchement conditionnel :

`if` *condition* `then` *action1* `else` *action2*

Si le booléen condition est vrai c'est l'action1 qui est effectuée sinon c'est l'action2. Il en va ainsi en Caml et l'on peut par exemple redéfinir la fonction valeur absolue `abs` de la façon suivante :

```
#let abs x =
    if x>0 then x else -x;;
abs : int -> int = <fun>
#abs (-5);;
- : int = 5
#abs 2;;
- : int = 2
```

La factorielle peut aussi s'écrire :

```
#let rec fact n=
   if n<=1 then 1 else n*fact (n-1);;
fact : int -> int = <fun>
#fact 12;;
- : int = 479001600
```

Attention, les deux actions du `if` doivent être du même type :

Encore une histoire de torchons et de serviettes !

```
#let mon_log x =
  if x>. 0. then log(x) else "ca ne marche pas";;
Entrée interactive:
>  if x>. 0. then log(x) else "ca ne marche pas";;
>                             ^^^^^^^^^^^^^^^
Cette expression est de type string,
mais est utilisée avec le type float.
```

(voir aussi la section 5.6 pour une définition du logarithme n'acceptant que des réels positifs).

On commence à le savoir...

Caml n'aimant pas l'imprécision, le `else` doit être impérativement présent. C'est même un peu plus compliqué que cela :

```
#let mon_log2 x =
  if x>. 0. then log(x);;
Entrée interactive:
>  if x>. 0. then log(x);;
>  ^^^^^^^^^^^^^^^^^^^^
Cette expression est de type unit,
mais est utilisée avec le type float.
```

Ce message d'erreur peut sembler étrange. Lorsque Caml ne rencontre pas de `else` à la suite du `if`, il complète automatiquement la commande par un `else ()` qui

signifie « sinon ne rien faire ». L'expression `()`, qui ne fait rien, est de type `unit`, donc d'un type différent de celui de `log(x)` ce qui explique le message d'erreur précédent.

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

En conséquence, si le type de l'*action1* est `unit` et si l'on désire ne rien faire dans le cas où la condition n'est pas vérifiée, le `else` peut être omis :

```
#let test x = if x<0 then print_string "Entier negatif";;
test : int -> unit = <fun>
#test (-3);;
Entier negatif- : unit = ()
#test 2;;
- : unit = ()
```

C'est en déclenchant des exceptions que l'on peut traiter les autres cas.

4.3 Les références

Les variables que nous avons jusqu'à présent rencontrées en Caml sont des objets informatiques qui n'évoluent pas au cours d'une session. Elles sont définies une fois pour toute à l'aide d'un `let`. Nous allons parler ici des variables évolutives de Caml.

Il s'agit des *références*. On les définit de la façon suivante :

`let` *nom-de-ref* `= ref` *une-valeur-initiale*`;;`

La définition d'un tel objet est donc forcément accompagnée de son initialisation.

```
#let x= ref 0;;
x : int ref = ref 0
```

Le type de l'identificateur `x` ci-dessus est donc `int ref`, une référence sur un entier.

C'est l'affectation qui permet de faire évoluer de tels objets. Le '`:=`' de Pascal, C... n'a donc pas disparu en Caml (il était en effet difficile d'occulter cette commande venue des tréfonds du langage machine et de la gestion des registres...). Cependant, comme nous l'avons déjà signalé, l'identificateur `x` dans `x:=x+1` (instruction venue d'un langage impératif classique) est ambigu. À gauche du signe `:=`, il désigne le nom de la variable informatique, à droite sa valeur. Cette ambiguïté est levée en Caml. L'identificateur `x` désigne le nom de l'objet informatique alors que `!x` désigne son contenu c'est-à-dire ici sa valeur[30] (voir figure I.2).

Ainsi en Caml, on écrira `x := !x + 1`.

```
#x;;
- : int ref = ref 0
#!x;;
- : int = 0
```

30. C'est du « hard » ! Une variable est une case mémoire qui a un contenu et une adresse... et l'on repense avec nostalgie aux pointeurs de Pascal.

Figure I.2 : *Une référence Caml.*

```
#x:=x+1;;
Entrée interactive:
>x:=x+1;;
>   ^
Cette expression est de type int ref,
mais est utilisée avec le type int.
#x:=!x+1;;
- : unit = ()
#x;;
- : int ref = ref 1
#!x;;
- : int = 1
```

Il faut faire attention à ne pas oublier le point d'exclamation `!` lors de l'appel au contenu d'une référence.

Nous reviendrons sur l'utilisation des références en page 28. Pour l'instant, illustrons leur utilisation dans les boucles.

4.4 Les boucles

Une boucle est une construction en programmation impérative qui permet de répéter plusieurs fois une instruction (ou un groupe d'instructions). Il y a deux types de boucles : les boucles où l'on connaît à l'avance le nombre d'itérations que l'on doit effectuer et les boucles pour lesquelles ce nombre d'itérations n'est pas connu mais est conditionné par un test booléen.

4.4.1 La boucle inconditionnelle

Il s'agit de la célèbre instruction[31] :

```
for i = départ to arrivée do
 corps-de-la-boucle
done;
```

31. Le `to` peut être remplacé par un `downto` qui a pour effet de décrémenter l'identificateur de boucle de `départ` à `arrivée`.

qui exécute le corps de la boucle depuis `départ` (qui s'évalue en un entier) jusqu'à `arrivée`[32] (qui s'évalue en un entier) en incrémentant de 1 la valeur de l'identificateur `i` à chaque itération. Le `for` définit l'identificateur de type entier `i` ; le corps de la boucle peut utiliser sa valeur mais non la modifier, ainsi on est certain de la terminaison de la boucle.

Encore elle ! La factorielle peut se définir d'une troisième façon en Caml :

```
#let fact n =
    let res = ref 1 in
      for i=1 to n do
          res:= !res*i
      done;
    !res;;
fact : int -> int = <fun>
#fact 3;;
- : int =6
```

Définissons, comme autre exemple, la fonction `sigma`[33] : qui renvoie la somme des entiers compris entre n et m (`sigma n m` $= \sum_{i=n}^{m} i$) :

```
#let sigma n m =
        let resultat = ref 0 in
              for i = n to m do
                 resultat := !resultat + i
              done;
        !resultat;;
sigma : int -> int -> int = <fun>
#sigma 1 10;;
- : int = 55
```

4.4.2 La boucle conditionnelle

Il s'agit de la non moins célèbre instruction[34] :

```
while condition do
 corps-de-la-boucle
done;
```

32. Si `départ>arrivée`, le corps de la boucle n'est pas exécuté.

33. Une version récursive de cette application peut être donnée par :

```
#let rec sigma_recursif n m =
    if n>m then 0
    else n + sigma_recursif (n+1) m;;
sigma_recursif : int -> int -> int = <fun>
#sigma_recursif 1 10;;
- : int = 55
```

Nous reparlerons de cela au chapitre II.

34. L'instruction `repeat` n'est pas prédéfinie en Caml.

qui exécute le corps de la boucle tant que `condition` s'évalue à `true`. Le danger d'une telle boucle est de pouvoir ne jamais terminer ! La vérification de l'arrêt sera l'une de nos préoccupations du chapitre II.

Illustrons l'utilisation de `while` par la convergence d'une suite récurrente simple. La méthode de Newton pour la recherche des racines de $X^2 - 2$ conduit à la suite récurrente $u_{n+1} = g(u_n)$ avec $g(x) = \frac{x}{2} + \frac{1}{x}$. Une valeur de $u_0 > \sqrt{2}$ entraînera la convergence de la suite vers $\sqrt{2}$. On peut en Caml définir la fonction auxiliaire g puis donner comme condition d'arrêt $|{u_n}^2 - 2| < \varepsilon$, avec ε une précision donnée[35].

```
#let newton epsilon u0 =
           (* epsilon est la précision, u0 la valeur initiale *)
   let g x = 1./.2. *. x +. 1./. x in
           (* on travaille avec des flottants ! *)
   let rac2 = ref u0 in
           while (abs_float(2. -. !rac2*.(!rac2)) > epsilon) do
              rac2:= g !rac2
           done;
   !rac2;;
newton : float -> float -> float = <fun>
#newton 1e-6  5.;;
- : float = 1.4142135858
```

On peut avoir envie de savoir en combien d'itérations une précision donnée a été obtenue. On utilise pour cela un compteur :

```
#let newton_bis epsilon u0 =
   let g x = 1./.2. *. x +. 1./. x in
   let compteur = ref 0 in
(* on initialise le compteur à zéro *)
   let rac2 = ref u0 in
       while (abs_float(2. -. !rac2*.(!rac2)) > epsilon) do
              rac2:= g !rac2;
              compteur:= !compteur + 1
       done;
   print_string "nombre d'iterations : ";
   print_int !compteur;
   print_newline ();;
(* on retourne la valeur du compteur cette fois-ci
   plutôt que la racine *)
newton_bis : float -> float -> unit = <fun>
#newton_bis 1e-6 5.;;
nombre d'iterations : 5
- : unit = ()
```

35. Le lecteur vérifiera facilement que pour $u_0 \in \mathbb{R}_+^*$, la suite décroît à partir du rang 1 vers $\sqrt{2}$ ce qui justifie la condition d'arrêt.

Signalons l'existence des fonctions `succ` (qui rajoute 1 à un entier) et `incr` (qui fait de même sur une référence d'entier) qui permettent de remplacer la ligne :

```
compteur:=!compteur + 1
```

en :

```
compteur:= succ !compteur
```

ou plus simplement encore en :

```
incr compteur
```

Pour conclure sur `while`, gageons que si le lecteur produit une boucle infinie, il découvrira vite l'usage de la commande '`Interrompre Caml-Light`' du menu !

4.5 Délimiteurs

Et il y a même... des `begin` et des `end` ! Ceux-ci groupent des instructions (séparées par des '`;`') et jouent le rôle en quelque sorte de parenthèses. Voyons un exemple (assez explicite) :

```
#let test n =
let a=ref 0 and b=ref 0 in
if n=1 then begin a:=1; b:=11 end else begin a:=2;  b:=22 end;
(!a,!b);;
test : int -> int * int = <fun>

#test 1;;
- : int * int = 1, 11
#test 2;;
- : int * int = 2, 22

#let test2 n =
let a=ref 0 and b=ref 0 in
if n=1 then begin a:=1; b:=11 end else a:=2;  b:=22;
(!a,!b);;
test2 : int -> int * int = <fun>

#test2 1;;
- : int * int = 1, 22
#test2 2;;
- : int * int = 2, 22
```

L'instruction `b:=22` ne fait plus en effet partie de la commande `if... then... else...;` et est toujours exécutée. Voyons ce qui se passe lorsque l'on ôte les premiers `begin...end` :

```
#let test3 n =
let a=ref 0 and b=ref 0 in
if n=1 then  a:=1; b:=11 else a:=2;  b:=22;
(!a,!b);;
```

```
Entrée interactive:
>if n=1 then  a:=1; b:=11 else a:=2;  b:=22;
>                          ^^^^
Erreur de syntaxe.
```

Le dernier test n'a même pas voulu se compiler car Caml a jugé l'instruction `if` terminée après le premier '`;`' qui suit `a:=1` (il l'a complétée avec un `else ()`[36]) et donc n'accepte pas l'intrusion du nouveau `else`. Rappelons encore ici que le '`;`' est le séparateur d'instructions et qu'il n'a donc rien à faire devant un `else` de façon générale.

On peut également employer les `begin... end` pour limiter la portée d'une définition de variable locale. Observons le programme suivant :

```
let Juillet97 () =
  let s = "britannique" in
  let s = "chinoise" in
        print_string "Hong Kong est une ville ";
        print_string s;
  print_string " sous souveraineté ";
  print_string s;;
```

La deuxième définition de `s` remplace la première. À l'exécution, on obtient :

```
#Juillet97 ();;
Hong Kong est une ville chinoise sous souveraineté chinoise
- : unit = ()
```

Alors que le deuxième programme :

```
let Juin97 () =
  let s = "britannique" in
  begin
  let s = "chinoise" in
        print_string "Hong Kong est une ville ";
        print_string s;
  end;
  print_string " sous souveraineté ";
  print_string s;;
```

produit :

```
#Juin97 ();;
Hong Kong est une ville chinoise sous souveraineté britannique
- : unit = ()
```

ce qui est tout de même différent, non ?

36. Un autre type que `unit()` pour l'action suivant le `then` aurait également provoqué une erreur de syntaxe.

4.6 La programmation impérative est-elle indispensable ?

Pour conclure cette section sur la programmation impérative en Caml, il est intéressant de montrer qu'il existe des équivalents fonctionnels aux instructions impératives. Le `if... then... else...;` peut se programmer ainsi :

```
let conditionnelle condition action1 action2 =
  match condition with
    | true  -> action1
    | false -> action2;;
```

Les boucles, quant à elles, ne sont que des itérations d'une fonction f donnée :

```
let rec boucle_inconditionnelle n f x =
           if n=0 then x else f(boucle_inconditionnelle (n-1) f x)

let rec boucle_conditionnelle test_arret f x =
           if (test_arret x) then x
                   else boucle_conditionnelle test_arret f (f x);;
```

Ainsi, il ne coûte rien alors de programmer fonctionnellement une boucle `repeat... until` (la boucle impérative `repeat... until` est classiquement une boucle conditionnelle où le test d'arrêt est évalué *à la suite* du corps de la boucle et qui à la différence de `while` exécute celui-ci au moins une fois). On peut écrire :

```
let rec repeat_until test_arret f x =
   boucle_conditionnelle test_arret f (f x);;
```

On pourra aussi se reporter à l'exercice I.12 pour constater la simplicité de la récursivité sur un problème traditionnellement traité avec moult pointeurs et force boucles `while`...

Le choix d'une programmation fonctionnelle ou impérative d'un même problème informatique (en l'absence d'un argument d'efficacité en faveur d'une méthode ou de l'autre), doit, à notre avis, être guidé par un souci de simplicité, de clarté... d'esthétique ! C'est une affaire de goût moins personnelle qu'il n'y paraît puisque le programme est amené à être utilisé par un tiers[37]...

5 Compléments

Nous allons dans cette section approfondir les notions vues précédemment et donner quelques compléments. Nous conseillons au lecteur débutant en Caml d'aborder cette section après avoir traité les exercices I.1 à I.10.

5.1 Variables de type

Le système de typage de Caml est d'une grande expressivité. Nous allons voir ici ce que sont les variables de type et les variables de type faibles.

37. Un tiers... qui est souvent soi-même quelque temps plus tard !

5.1.1 Polymorphisme

Le lecteur curieux s'est peut-être déjà demandé quel était le type de la liste vide. S'il a posé la question à Caml, il a eu la surprise de voir s'afficher :

```
#[];;
- : 'a list = []
```

Nous avons déjà dit que Caml lors de la détermination du type d'un objet recherchait le type le plus général. Ici, la lettre `'a` est une *variable de type* : elle représente, comme ce nom l'indique, un type d'objets quelconque. La liste vide est, en effet, une liste de n'importe quoi. Caml emploie ainsi ces variables de type pour « typer » les objets sans contrainte de type. Examinons, par exemple, le type de la fonction `hd` :

```
#hd;;
- : 'a list -> 'a = <fun>
```

La fonction `hd` agit sur une liste de type quelconque dont elle renvoie le premier élément. Il est à noter que c'est le même `'a` qui apparaît des deux côtés de la flèche `->`, car le type de l'élément de tête est le type de tous les éléments de la liste. Caml a donc bien raison ici d'employer une seule variable de type.

Plusieurs variables de type distinctes peuvent être produites par Caml. Prenons l'exemple de la fonction `map`, prédéfinie en Caml, qui permet l'application d'une fonction donnée à tous les éléments d'une liste :

```
#let FoisDeux x = 2*x;;
FoisDeux : int -> int = <fun>
#map FoisDeux [1;2;3];;
- : int list = [2; 4; 6]
```

Son type est :

```
#map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

L'analogie entre type et ensemble de définition d'une application est ici éclairante. En effet, la fonction `map` a deux arguments : d'une part une fonction f très générale qui, a priori, prend des objets de type `'a` et renvoie des objets de type `'b`, d'autre part une liste d'objets sur lesquels f doit pouvoir s'appliquer et qui sont donc nécessairement de type `'a`, le résultat étant alors une liste d'objets de type `'b`. On pourra aussi regarder le type des fonctions `fst` et `snd` qui renvoient les premières et secondes composantes d'un couple.

Les fonctions typées par des variables de types s'appellent fonctions *polymorphes*. Le polymorphisme permet une *grande expressivité*.

Reprenons par exemple le cas de la fonction `sigma` du 4.4.1 qui calcule $\sum_{i=n}^{m} i$. La programmation de la fonction `prod` définie par `prod n m` $= \prod_{i=n}^{m} i$ est identique à substitution près du caractère `+` par le caractère `*`. On peut donc souhaiter définir une fonction plus abstraite `itere_loi` comprenant deux arguments supplémentaires : la loi de composition interne considérée, puis une valeur d'initialisation du calcul (en

général l'élément neutre de la loi). Rappelons, pour la bonne compréhension de la session Caml qui suit, que la version préfixée (c'est-à-dire celle où l'opérateur figure avant ses opérandes[38]) de l'addition (resp. la multiplication) est `prefix +` (resp. `prefix *`). Cela donne :

```
#let itere_loi loi  initialisation n m =
    let resultat = ref initialisation in
            for i = n to m do
                resultat := loi !resultat i
            done;
    !resultat;;
itere_loi : ('a -> int -> 'a) -> 'a -> int -> int -> 'a = <fun>
#let sigma = itere_loi (prefix +) 0;;
sigma : int -> int -> int = <fun>
#sigma 1 10;;
- : int = 55
#let prod = itere_loi (prefix *) 1;;
prod : int -> int -> int = <fun>
#prod 1 10;;
- : int = 3628800
```

On a les jeux qu'on peut !

Le code de la fonction `itere_loi` n'oblige pas à travailler forcément avec des lois de composition de $\mathbb{N} \times \mathbb{N}$ dans $\mathbb{N}$, comme nous l'indique Caml avec la variable de type `'a`. Pour s'amuser on peut par exemple définir la fonction x^n ou x est un réel et n un entier[39] d'un `float` par un `int` :

```
#let rec puissance_entiere x = function
    | 0 -> 1.0
    | 1 -> x
    | n -> x *. puissance_entiere x (n-1);;
puissance_entiere : float -> int -> float = <fun>
#puissance_entiere 2.5 3;;
- : float = 15.625
```

et la fonction `itere_loi puissance_entiere 2.0` se met à calculer des puissances itérées de 2.

```
#let puissance_iteree_de_deux = itere_loi puissance_entiere 2.0;;
puissance_iteree_de_deux : int -> int -> float = <fun>
#puissance_iteree_de_deux 1 5;;
- : float = 1.32922799578e+36
```

On a : `puissance_iteree_de_deux n m` $= \left(\left((2^n)^{n+1}\right)^{\cdots}\right)^m$.

Si d'aventure on désire que `itere_loi` ne concerne que $\mathbb{N}$, il est possible de *forcer son type*. On peut en effet imposer le type d'une expression polymorphe en Caml en remplaçant :

expression par (*expression*:*type-voulu*)

dans du code Caml.

38. La notation `+ 1 2` est préfixe alors que `1 + 2` est une notation infixe.
39. Nous verrons diverses façons plus ou moins efficaces de programmer ce calcul.

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

Ainsi, la version modifiée de `itere_loi` s'écrit :

```
#let itere_loi loi  initialisation n m =
           let resultat = ref initialisation in
                   for i = n to m do
                             resultat := loi (!resultat:int) i
                   done;
           resultat;;
itere_loi : (int -> int -> int) -> int -> int -> int -> int = <fun>
```

(dans cet exemple, il a suffi d'imposer un type à `!resultat` pour imposer le type de la fonction).

L'usage de `itere_loi` n'est alors plus possible avec `puissance_entiere` dont le type ne convient plus :

```
#let puissance_iteree_de_deux = itere_loi puissance_entiere 2.0;;
Entrée interactive:
>let puissance_iteree_de_deux = itere_loi puissance_entiere 2.0;;
>                                         ^^^^^^^^^^^^^^^^^^
Cette expression est de type float -> int -> float,
mais est utilisée avec le type int -> int -> int.
```

5.1.2 Variables de type faibles

En plus des variables de type, Caml introduit également les *variables de type faibles* qu'il note `'_a`, `'_b`... Il s'agit de types que l'on pourrait qualifier de « types inconnus en attente d'être reconnus ». Ces variables de type faibles ont ainsi une durée de vie brève ; elles apparaissent notamment lors de l'appel d'une fonction polymorphe par une fonction dont tous les paramètres ne sont pas précisés.

C'est un peu plus subtil que les variables de type toutes simples. À titre d'exemple, imaginons que l'on veuille programmer la fonction `miroir` qui symétrise une liste (`miroir [1;2;3] = [3;2;1]`) (voir à ce propos l'exercice I.4) et que l'on utilise pour cela une fonction auxiliaire définie comme suit :

```
#let rec miroir_aux accu = fun
        | [] -> accu
        | (a::suite) -> miroir_aux (a::accu) suite;;
miroir_aux : 'a list -> 'a list -> 'a list = <fun>
#miroir_aux [3;2;1] [6;7;8];;
- : int list = [8; 7; 6; 3; 2; 1]
```

Cette fonction est donc polymorphe. Elle est appelée par `miroir` de la manière suivante :

```
#let miroir = miroir_aux [];;
miroir : '_a list -> '_a list = <fun>
```

et voilà `miroir` de type inconnu ! Le type de `miroir_aux` étant polymorphe, la définition de `miroir` ne permet pas pour l'instant à Caml de déterminer son type. Il lui faut attendre une utilisation de `miroir`. Ainsi la première application effective de `miroir` va lui affecter un type :

```
#miroir [1;2;3];;
- : int list = [3; 2; 1]
#miroir;;
- : int list -> int list = <fun>
```

Ceci peut être ennuyeux si l'on désire que `miroir` puisse travailler sur des listes d'objets de type quelconque, c'est-à-dire si l'on veut que `miroir` demeure polymorphe. Une légère modification de la définition de `miroir` le permet :

```
#let miroir_polymorphe l = miroir_aux [] l;;
miroir_polymorphe : 'a list -> 'a list = <fun>
#miroir_polymorphe [1;2;3];;
- : int list = [3; 2; 1]
#miroir_polymorphe;;
- : 'a list -> 'a list = <fun>
```

On a *explicité tous les arguments* et la nouvelle fonction est bien dotée d'un type polymorphe pour toute la durée de la session.

Signalons que l'on peut faire apparaître un type inconnu bien plus vite :

```
#let e = ref [];;
e : '_a list ref = ref []
#e:=[1];;
- : unit = ()
#e;;
- : int list ref = ref [1]
```

On constate ainsi que les variables de type faibles sont courantes dans l'univers des références.

5.2 Fonctionnelles, opérateurs, curryfication

L'examen attentif de l'exemple précédent montre que nous avons utilisé une référence, non sur un entier, mais sur une liste. La portée syntaxique du mot-clé `ref` est en fait générale. C'est l'*expression*, c'est-à-dire tout code Caml syntaxiquement correct, qui constitue l'objet de base de Caml. Les mots-clés que nous avons déjà vus comme `ref`, `let`, `where`... s'appliquent sur toute expression. Voici par exemple une référence sur un objet structuré :

```
#let x= ref (true,[1;2;3]);;
x : (bool * int list) ref = ref (true, [1; 2; 3])
#x:=(false,[4;5]);;
- : unit = ()
```

Il est usuel en Caml de définir une fonction auxiliaire à l'intérieur d'une fonction principale à l'aide d'un `where`. Par exemple[40] :

```
#let binomial n p = (fact n)/((fact p) * (fact (n-p)))
    where rec fact  = function
          | 0 -> 1
          | n -> n* fact (n-1);;
binomial : int -> int -> int = <fun>
#binomial 4 2;;
- : int = 6
```

Retenons que, de façon générale, tout est emboîtable en Caml c'est-à-dire toute expression peut être utilisée comme une sous-expression d'une autre expression.

Nous avons ainsi déjà constaté que Caml se place indifféremment au niveau des valeurs ou des fonctions (voir pages 8 à 9). Il est alors facile de définir des fonctionnelles (des fonctions qui agissent sur des fonctions) en Caml. La composition des applications en est l'exemple le plus célèbre :

```
#let compose g f x = g (f x);;
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
#let f x = x+1;;
f : int -> int = <fun>
#let g x = x*x;;
g : int -> int = <fun>
#let h = compose g f;;
h : int -> int = <fun>
#h 3;;
- : int = 16
```

On a défini h comme la composée $g \circ f$ soit l'application qui à n associe $(n+1)^2$. Vous avez remarqué que Caml a convenablement typé la composition des fonctions et impose des ensembles de départ et d'arrivée cohérents.

Si l'on veut retrouver les notations mathématiques usuelles, on peut noter `o` l'opérateur binaire et le déclarer infixe (c'est-à-dire placé entre ses arguments) à l'aide de la directive[41] Caml `#infix` :

```
#let o = compose;;
o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
##infix "o";;
#let h = g o f;;
h : int -> int = <fun>
#h 3;;
- : int = 16
```

Esthétique, non ?

40. Attention, cet exemple n'est là que pour illustrer l'assemblage des expression Caml, ce n'est certainement pas une programmation efficace du calcul des coefficients binomiaux (on fait beaucoup trop de multiplications !). Étudiez l'exercice I.8 pour des implémentations plus raisonnables de ce calcul.

41. Nous avons déjà vu la directive `#open` de Caml.

Les formes infixées et préfixées des opérateurs ne sont pas toujours rigoureusement équivalentes. Caml implémente en effet les opérateurs booléens *infixes* de façon *paresseuse*. Plus précisément, si Caml doit évaluer le booléen `b = e1 or e2` et si l'évaluation de `e1` retourne `true`, alors il conclut, fort efficacement, à la vérité de `b` sans évaluer `e2` (le comportement est analogue pour `&&`). Par contre, si l'on déclare l'opérateur booléen préfixe à l'aide de la commande `prefix`[42] alors, comme c'est le cas lors d'un appel fonctionnel, les arguments sont tous les deux évalués et le caractère paresseux disparaît. Nous avons illustré ce comportement dans la session qui suit à l'aide d'une fonction `test` qui retourne toujours `true` sauf pour une valeur exceptionnelle, `n=0`, auquel cas un message d'erreur est affiché :

```
#let test n  = (n/n = 1);;
test : int -> bool = <fun>
#test 1;;
- : bool = true
#test 0;;
Exception non rattrapée: Division_by_zero
#(2=2) or (test 0);;
- : bool = true
#let ou = prefix or;;
ou : bool -> bool -> bool = <fun>
#ou (2=2)  (test 0);;
Exception non rattrapée: Division_by_zero
```

Retenons que, hormis le cas des booléens infixes, les arguments d'une fonction sont toujours évalués : c'est l'*appel par valeurs*.

Revenons aux fonctionnelles en considérant la forme des fonctions à plusieurs arguments. Nous avons vu en 3.1 deux versions de la fonction `somme` de deux entiers, une version dite *curryfiée*[43] :

```
let somme_curry x y = x+y;;
```

et une version *non-curryfiée* dont le paramètre est un couple d'entiers :

```
let somme_uncurry (x,y) = x+y;;
```

qui correspond à l'usage mathématique.

Nous avons également déjà mentionné la facilité d'abstraction de paramètre sur des fonctions curryfiées[44] pour la définition d'autres fonctions (voir l'exercice I.10). Clairement les définitions de ces fonctions sont mathématiquement isomorphes et il est facile d'écrire des fonctionnelles permettant de passer d'une forme à l'autre.

```
#let curry = fun
     f x y -> f (x,y);;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

#let somme = curry somme_uncurry;;
somme : int -> int -> int = <fun>
```

42. Nous avons déjà regardé ainsi le type de `+`.
43. Du nom du logicien Haskell Curry (1900-1982).
44. Celles-ci sont aussi implémentées de manière plus efficace.

On retrouve la fonction somme curryfiée. Inversement, on définit :

```
#let uncurry = fun
    f (x,y) -> f x y;;
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

Remarquons que la définition :

```
#let curry = function
 f x y -> f (x,y);;
```

conduit à un message d'erreur :

```
Entrée interactive:
> f x y -> f (x,y);;
> ^^^
Le constructeur f n'est pas défini.
```

Nous voici arrivés à la subtile différence entre les mots-clés `fun` et `function`. La construction `fun` est curryfiée à la différence de la construction `function` qui elle n'abstrait qu'un seul argument. Concrètement, on réservera l'usage de `function` aux définitions des fonctions à un argument (un argument pouvant être un uplet !) afin d'éviter des erreurs déroutantes comme :

```
#let XOR = function
    | true x -> not x
    | false x -> x;;

Entrée interactive:
>    | true x -> not x
>      ^^^^^^
Le constructeur true est constant:
   il ne peut recevoir un argument.
```

Les formes suivantes sont cependant correctes :

```
let XOR = fun
    | true x -> not x
    | false x -> x;;

let XOR = function
    | (true,x) -> not x
    | (false,x) -> x;;
```

Enfin, seule l'antériorité de ce chapitre par rapport au chapitre III consacré aux structures de données, nous retient d'évoquer ici la fonctionnelle `list_it`. Nous vous donnons donc rendez-vous en page 150 pour de plus amples informations.

5.3 Filtrage et égalité

Nous avons déjà insisté sur la puissance et la lisibilité du filtrage en Caml. Nous avons également annoncé (se reporter aux différentes implémentations de XOR) que

la répétition de variables identiques était proscrite dans un motif Caml. Détaillons ce point avec un exemple de filtrage sur des motifs plutôt complexes. La fonction suivante est correcte :

```
let test1 = function
    | ... -> ...
    | ((1, x),(2,(1,y))) -> ...
    | ... -> ...;;
```

alors que celle-ci ne l'est pas :

```
let test2 = function
    | ... -> ...
    | ((1, x),(2,(1,x))) -> ...
    | ... -> ...;;
```

car `x` ne peut être utilisé à nouveau.

Cette restriction provient d'une raison informatique majeure. Considérons ce qui se passe lorsque Caml doit évaluer

`test1 ((1, (0,0) ),(2,(1, (0,(0,0)) ))).`

La comparaison entre le motif et la donnée se fait en un temps proportionnel à la taille du motif. Caml parcourt en effet la donnée jusqu'à retrouver la structure arborescente du motif. Ici, l'essai étant concluant, il n'y a aucun échec qui fasse directement passer au motif suivant et le nombre de tests est maximal. À la fin de la comparaison, `x` vaut `(0,0)` et `y` vaut `(0,(0,0))`. Si l'on considère le motif de la fonction `test2`, la rencontre du deuxième `x` impose à Caml de tester l'égalité entre `(0,(0,0))` et `(0,0)` et donc de parcourir toute la donnée (on pourrait imaginer un exemple de taille plus conséquente encore[45]). Afin de garantir un filtrage en temps linéaire en fonction de la complexité des motifs, le choix a été fait en Caml d'imposer des variables toutes distinctes dans les motifs.

C'est en fait toute la difficulté de l'implémentation pratique de l'égalité qui se cache derrière ce problème. Dans l'exemple précédent, la conclusion était manifeste (`(0,(0,0))` $\neq$ `(0,0)` !) mais l'on peut fort bien songer à une comparaison entre deux expressions syntaxiquement différentes mais sémantiquement identiques pour le programmeur (comme par exemple `2/2` et `1`) et ceci avec une complexité quelconque. Comme Caml n'est pas devin et qu'en général c'est le programmeur qui connaît le mieux son contexte de travail, il a été décidé de laisser au programmeur le choix du test d'égalité qu'il juge approprié. Ainsi, il existe en Caml ce que l'on appelle des *motifs gardés*, qui sont des motifs suivis d'une condition au libre choix du programmeur. La syntaxe en est :

`|` *motif* `when` *condition* `->...`

On peut ainsi écrire :

```
let test3 = function
    | ... -> ...
    | ((1, x),(2,(1,y))) when x=y -> ...
    | ... -> ...;;
```

45. Le nombre de parenthèses de la donnée est déjà suffisant...

mais aussi plus généralement :

```
let test4 = function
    | ... -> ...
    | ((1, x),(2,(1,y))) when x=y+1 -> ...
    | ... -> ...;;
```

Signalons également que rien n'interdit au programmeur d'écrire quelque chose comme :

```
let test5 = function
    | ... -> ...
    | ((1, x),(2,(1,y))) -> if x=y then ... else ...
    | ... -> ...;;
```

Notez que `test5` n'est pas sémantiquement identique à `test3`, car la donnée `((1, (0,0) ),(2,(1, (0,(0,0)) )))` est filtrée par le motif de `test5` mais pas par celui de `test3` pour laquelle Caml envisage les motifs qui suivent.

On peut également définir des abréviations au sein d'un motif avec le mot-clé `as`. Ainsi on pourra écrire :

```
let f = function
    | [] -> ...
    | a::r as L -> if list_length L = ...
```

au lieu de :

```
let f = function
    | [] -> ...
    | a::r -> if list_length (a::r) = ...
```

Revenons sur les tests d'égalité en Caml. Il en existe deux `=` et `==` qui sont tous deux polymorphes de type: `'a -> 'a -> bool`. Il s'agit de deux tests complètement différents.

Le `==` est le plus « informatique » des deux : il teste l'adresse mémoire de ses arguments. Il n'échoue donc jamais mais peut renvoyer des résultats sémantiquement faux :

```
#"test" == "test";;
- : bool = false
#"test" = "test";;
- : bool = true
```

(les deux chaînes de caractères `"test"` sont stockées dans deux mémoires différentes).

Le `=` est plus « mathématique » : il teste l'égalité de ses arguments en effectuant un parcours de ceux-ci. Il peut ainsi lui arriver de boucler. Essayez donc :

```
#1/2=3/2-1;;
```

alors que le test suivant termine :

```
#1/2==3/2-1;;
- : bool = true
```

En conclusion, *il faudra toujours privilégier le test spécifique d'égalité sur la structure de données que l'on manipule* (et s'il n'existe pas, en écrire un est la solution la plus sûre !). Il faut aussi retenir que `==` est plus rapide mais, testant l'adresse mémoire des objets, est moins « intelligent », il peut même se révéler faux (deux chaînes de caractères comportant les mêmes caractères ou deux listes comportant les mêmes éléments ne sont pas `==` mais `=`) ; alors que `=` est sémantiquement plus juste, mais peut ne pas terminer ou ne pas conclure sur des valeurs fonctionnelles, ou affirmer que deux valeurs mutables sont égales, ce qui peut être juste à un instant donné mais faux par la suite.

```
#let x = ref 1 and y = ref 1;;
x : int ref = ref 1
y : int ref = ref 1
#x = y;;
- : bool = true
#x == y;;
- : bool = false
#x := 2;;
- : unit = ()
#x = y;;
- : bool = false
```

(pratiquement, utilisez `==` pour des références).

Enfin, pour conclure sur le filtrage et l'égalité, mentionnons que Caml réalise en permanence une égalité bien plus générale entre des termes arborescents comportant *chacun* des variables *avec possibilité de répétitions*, égalité appelée *unification*, ceci pour la *synthèse des types* des expressions. Mais ceci est une autre histoire...

5.4 Type défini par l'utilisateur

Nous connaissons déjà en Caml les types constants prédéfinis comme `bool`, `int`, `float`... ainsi que les opérateurs de types que sont le produit cartésien `*` et la flèche fonctionnelle `->`.

L'utilisateur peut créer ses propres types, comme nous allons le voir à présent.

5.4.1 Type somme et type produit (enregistrement)

On peut définir un type de deux façons en Caml :

- soit en faisant une union disjointe de types déjà connus, c'est ce que l'on appelle un *type somme* ;
- soit en faisant un produit cartésien de types existants, ce que l'on appelle un *type produit*.

On peut bien sûr mélanger ces deux constructions.

Les définitions de type se font à l'aide du mot-clé `type`. L'identificateur de l'union disjointe dans les types somme est la barre verticale '`|`'. Nous l'avons déjà rencon-

trée lors de la définition du type somme `animal_de_compagnie`. Nous avions alors simplement énuméré tous les objets du nouveau type :

```
type animal_de_compagnie = Chien | Chat | Oiseau;;
Le type animal_de_compagnie est défini.
```

Les types produits sont analogues aux *records* de Pascal. Il s'agit d'enregistrements dont chaque composante est nommée (on parle d'*étiquette*). La syntaxe en est la suivante :

```
type  nom-du-type = {Etiquette1 :  type-de-l-etiquette1;
                     Etiquette2 :  type-de-l-etiquette2;
                           ...};;
```

Définissons par exemple les nombres complexes sous la forme de couples de réels dont les composante sont `Partie_reelle` et `Partie_imaginaire` :

```
#type complexe =
      {Partie_reelle : float; Partie_imaginaire : float};;
Le type complexe est défini.
#let i={Partie_reelle = 0.0 ; Partie_imaginaire = 1.0};;
i : complexe = {Partie_reelle = 0.0; Partie_imaginaire = 1.0}
```

Comme on peut le constater la définition d'objets du nouveau type se fait à l'aide d'accolades et du signe '='. À l'instar d'un vecteur, l'accès aux composantes d'un objet du type enregistrement se fait à l'aide du signe '·'. La conjugaison dans $\mathbb{C}$ peut être définie par :

```
#let conjugaison {Partie_reelle = a ; Partie_imaginaire = b} =
                    {Partie_reelle = a ; Partie_imaginaire = -.b};;
conjugaison : complexe -> complexe = <fun>
#let i_barre = conjugaison i;;
i_barre : complexe = {Partie_reelle = 0.0; Partie_imaginaire = -1.0}
#i_barre.Partie_imaginaire;;
- : float = -1.0
```

Les composantes d'un enregistrement peuvent contenir tout type connu (en particulier des types somme) :

```
#type pere_de_famille =
{Nom : string ; Age : int ; Enfants : string list ;
                Animaux : animal_de_compagnie list};;
Le type pere_de_famille est défini.
#let papa =
{Nom = "jean" ; Age = 60 ; Enfants = ["philippe"; "luc"] ;
                            Animaux = [chien]};;
papa : pere_de_famille =
 {Nom = "jean"; Age = 60; Enfants = ["philippe"; "luc"];
                           Animaux = [chien]}
```

Signalons qu'il existe aussi des *abréviations de type* (qu'il ne faut pas confondre avec une définition de type) sous la syntaxe :

```
type nom-de-l-abréviation == type-à-abréger
```

Par exemple :

```
#type point == float * float;;
Le type point est défini.
```

et par la suite l'abréviation `point` pourra être utilisée en lieu et place de `float*float` (notamment lorsque l'on désire forcer un type).

```
#let rec première_projection = function
   | [] -> []
   | (x,y)::q -> x::(première_projection q);;
première_projection : ('a * 'b) list -> 'a list = <fun>
#let rec première_projection = function
   | [] -> []
   | ((x,y):point)::q -> x::(première_projection q);;
première_projection : point list -> float list = <fun>
#let (translation: point -> point) = function
    (x,y) -> x +. 1., y +. 1.;;
translation : point -> point = <fun>
```

5.4.2 Type mutable

Il est vraisemblable que l'on souhaite pouvoir modifier tous les ans l'âge du père de famille défini dans l'exemple précédent. Ceci n'est pas possible avec la construction choisie. Si l'on désire qu'une étiquette d'un type enregistrement puisse évoluer en cours de session, on doit la déclarer, lors de la définition du type, comme `mutable`. Cela donne :

```
#type pere_de_famille = {Nom : string ; mutable Age : int ;
                         Enfants : string list ;
                         Animaux : animal_de_compagnie list};;
Le type pere_de_famille est défini.
```

La modification d'une étiquette mutable se fait à l'aide du signe `<-` :

```
#let papa =
{Nom = "jean" ; Age = 60 ; Enfants = ["philippe"; "luc"] ;
                Animaux = [chien]};;
papa : pere_de_famille =
 {Nom = "jean"; Age = 60; Enfants = ["philippe"; "luc"];
                Animaux = [chien]}
#let rajeunir x = x.age <- 20;;
- : unit = ()
#rajeunir papa;;
- : unit = ()
#papa;;
- : pere_de_famille =
 {Nom = "jean"; Age = 20; Enfants = ["philippe"; "luc"];
                          Animaux = [chien]}
```

5.4.3 Type avec argument

Caml n'en reste pas là au niveau des définitions de type. Il offre aussi la possibilité de paramétrer les constructeurs de types de la façon suivante :

type *Identificateur* of *type-déjà-connu*,

un élément du nouveau type étant désigné par :

Identificateur(*valeur-du-type-déjà-connu*).

Ceci peut nous permettre de créer un nouveau type, que nous appellerons `nombre` destiné à contourner les distinctions syntaxiques entre flottants et entiers :

```
#type nombre = Entier of int | Reel of float;;
Le type nombre est défini.
#let x = Entier(1);;
x : nombre = Entier 1
#let y=Reel(2.5);;
y : nombre = Reel 2.5
#let add = fun
    | (Entier n) (Entier m) -> Entier (n+m)
    | (Entier n) (Reel x)   -> Reel ((float_of_int n) +. x)
    | (Reel x)   (Entier n) -> Reel (x +. (float_of_int n))
    | (Reel x)   (Reel y)   -> Reel (x +. y);;
add : nombre -> nombre -> nombre = <fun>
##infix "add";;
#x add y;;
- : nombre = Reel 3.5
```

(remarquer ici l'usage de `fun`).

5.4.4 Type polymorphe

On peut très bien employer des variables de type dans les constructions précédentes et définir ainsi des types polymorphes. En voici un exemple avec des enregistrements :

```
#type 'a boite = {Provenance : string ; Contenu : 'a list};;
Le type boite est défini.
#let boulier = {Provenance = "chine" ;
                Contenu = [0;1;2;3;4;5;6;7;8;9]};;
boulier : int boite =
 {Provenance = "chine";
  Contenu = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]}
```

Voici un autre exemple :

```
#type 'a paire = Paire of 'a  * 'a ;;
Le type paire est défini.
#let anglais_francais = Paire (["un";"deux";"trois"],
                               ["one";"two"; "three"]);;
anglais_francais : string list paire =
 Paire (["un"; "deux"; "trois"], ["one"; "two"; "three"])
```

Remarquons que l'objet `Paire(1,"un")` n'est pas reconnu comme étant du type `'a paire` puisque les types des deux composantes sont différents :

```
#Paire(1,"un");;
Entrée interactive:
>Paire(1,"un");;
>        ^^^^
Cette expression est de type string,
mais est utilisée avec le type int.
```

5.4.5 Type récursif

On peut très bien définir en Caml des objets dont la structure est récursive. Prenons l'exemple de la structure de donnée arbre binaire. Il s'agit d'objets informatiques définis ainsi : un arbre binaire est soit un nœud avec un fils gauche et un fils droit qui sont eux-mêmes des arbres binaires[46], soit tout simplement une feuille (on décide d'étiqueter les feuilles de l'arbre avec des entiers). Cette structure de données récursive est illustrée en figure I.3 :

Figure I.3 : *La structure d'arbre binaire accompagnée d'un exemple.*

Elle est amplement étudiée au chapitre III.

Cette définition récursive se traduit immédiatement en Caml par :

```
type arbre_binaire = Feuille of int
                   | Noeud of  arbre_binaire * arbre_binaire;;
```

Voici le codage de l'exemple d'arbre de la figure I.3 :

```
#let mon_arbre =
           Noeud ( Feuille 1, Noeud (Feuille 2, Feuille 3));;
mon_arbre : arbre_binaire =
           Noeud (Feuille 1, Noeud (Feuille 2, Feuille 3))
```

et voici, par exemple, une fonction comptant le nombre de feuilles d'un arbre binaire donné :

46. Si vous entendez parler d'arbres en informatique pour la première fois, retenez qu'à la différence des arbres « biologiques » ceux-ci ont la racine en haut et les feuilles en bas (pensez à l'arborescence des fichiers sur votre disque dur).

```
#let rec nombre_de_feuilles  = function
    | (Feuille n) -> 1
    | Noeud (sous_arbre_droit, sous_arbre_gauche) ->
                   (nombre_de_feuilles sous_arbre_droit) +
                   (nombre_de_feuilles sous_arbre_gauche);;

nombre_de_feuilles : arbre_binaire -> int = <fun>
#nombre_de_feuilles mon_arbre;;
- : int = 3
```

Étudions à présent un autre exemple récursif. Nous disposons de tous les outils nécessaires pour définir un type `expression_arithmetique` comportant des constantes entières, les opérateurs binaires d'addition et de multiplication et les variables formelles.

```
#type expression = Constante of int
                | Variable of string
                | Addition of expression * expression
                | Multiplication of expression * expression
                | Exponentielle of expression;;

Le type expression est défini.
#let expr1 =
     Exponentielle(Addition (Variable "x", Constante 1));;
expr1 : expression =
     Exponentielle (Addition (Variable "x", Constante 1))
#let expr2 = Addition (Multiplication (Variable "x",
                        Addition (Variable "y", Constante 1)),
                       Constante 2);;
expr2 : expression =
 Addition
  (Multiplication (Variable "x",
         Addition (Variable "y", Constante 1)),
   Constante 2)
```

L'expression `expr1` (resp. `expr2`) représente $\exp(x+1)$ (resp. $(x*(y+1))+2$). Nous verrons en page 43 comment coder de façon un peu moins fastidieuse ces expressions.

On peut alors définir une dérivation formelle :

```
#let rec derive variable_de_derivation = function
    | (Constante n)  -> (Constante 0)
    | (Variable x)   -> if (x=variable_de_derivation)
                        then (Constante 1)
                        else (Constante 0)
    | (Addition (expr1, expr2)) ->
              Addition (derive variable_de_derivation expr1,
                        derive variable_de_derivation expr2)
```

```
      | (Multiplication (expr1, expr2)) ->
         Addition
          (Multiplication (derive variable_de_derivation expr1,
                           expr2),
           Multiplication (expr1,
                           derive variable_de_derivation expr2))
      | (Exponentielle expr) ->
         Multiplication (derive variable_de_derivation expr,
                         Exponentielle (expr));;
derive : string -> expression -> expression = <fun>
```

Voici un exemple d'utilisation de `derive` :

```
#derive "x" expr1;;
- : expression =
 Multiplication
  (Addition (Constante 1, Constante 0),
   Exponentielle (Addition (Variable "x", Constante 1)))
#derive "x" expr2;;
- : expression =
 Addition
  (Addition
    (Multiplication
          (Constante 1, Addition (Variable "y", Constante 1)),
     Multiplication (Variable "x",
           Addition (Constante 0, Constante 0))),
   Constante 0)
```

On est certes loin encore d'une présentation agréable[47]...

Pour conclure sur les types récursifs, signalons que le type `'a list` prédéfini en Caml pourrait être redéfini simplement de la façon suivante :

```
#type 'a liste = Liste_vide | Conse of 'a*'a liste;;
Le type liste est défini.
#Conse (1, Liste_vide);;
- : int liste = Conse (1, Liste_vide)
```

5.4.6 Types mutuellement récursifs

À l'instar des fonctions mutuellement récursives (voir chapitre II 2.5), on peut définir en Caml plusieurs types récursifs dépendant simultanément les uns des autres. En voici un *exemple* d'école :

```
#type alpha = a | Mot   of alpha * beta
   and beta = b | Suite of beta  * alpha;;
Le type alpha est défini.
Le type beta est défini.
```

47. La simplification est un problème crucial en calcul formel !

```
#let essai1 = Mot ( Mot (a, Suite (b,a)), b);;
essai1 : alpha = Mot (Mot (a, Suite (b,a)), b)
#let essai2 = Suite (b,essai1);;
essai2 : beta = Suite (b, Mot (Mot (a, Suite (b,a)), b))
```

5.5 Entrées-sorties, flots, compilation

Hormis le graphisme qui fait l'objet d'une section indépendante, nous avons regroupé dans cette sous-section ce qui concerne la communication de votre programme Caml avec « l'extérieur » c'est-à-dire les entrées-sorties (banales), les flots (que nous n'aborderons que pour simplifier la saisie d'expressions) et la réalisation d'exécutables indépendants.

5.5.1 Entrées-sorties

Il est bien utile de connaître les fonctions d'entrées-sorties d'un langage de programmation. Nous avons pour l'instant évoqué des fonctions d'impression dont le nom est `print_`*nom-du-type-de-l'-expression-à-imprimer*, comme `print_int` pour imprimer un entier ou `print_string` pour imprimer une chaîne de caractères. La commande `print_newline ();;` (qui est une fonction sans argument) permet l'affichage d'une ligne vide. Les fonctions analogues de lecture ont pour nom : `read_int`, `read_float`... Tous les détails sont donnés dans l'aide du module[48] `io`. Citons aussi dans le module `graphics`, les fonctions `read_key` et `key_pressed` (qui, comme c'est l'usage, gèrent les frappes de touches du clavier).

Signalons également qu'il existe une impression formatée, directement héritée de C, accessible avec la primitive `printf` du module `printf`. La fonction `printf` prend en premier argument une chaîne de caractères (le « format »), dans laquelle on indique le type des arguments à imprimer. Conventionnellement chaque argument à imprimer est représenté dans le format par un symbole `%` suivi de son type symbolique. Ainsi '`%s`' désignera un argument chaîne de caractères et '`%d`' un argument entier. Le tout est suivi des arguments à imprimer :

```
#printf__printf "Le nombre %s est %d" "un" 1;;
Le nombre un est 1- : unit = ()
```

(Noter que l'on n'a pas chargé le module entier à l'aide d'un `#open`, mais simplement la fonction `printf` en la préfixant du nom du module et d'un double souligné : `printf__`.)

Toutes ces entrées-sorties se font pour l'instant sur la sortie et l'entrée standard (clavier et écran). On peut décider de les rediriger vers des fichiers. Pour cela, on doit ouvrir des *canaux*[49] (qu'il faut refermer après utilisation). Les commandes correspondantes sont les suivantes :

- `open_in` *`nom-de-fichier`*`;;` qui ouvre le canal d'entrée,
- `close_in` *`nom-de-fichier`*`;;` qui ferme le canal d'entrée,

48. `io` pour *input-output*.
49. Signalons que l'on peut également rediriger séparément les messages d'erreurs.

- `open_out` *`nom-de-fichier`*`;;` qui ouvre le canal de sortie,
- `close_out` *`nom-de-fichier`*`;;` qui ferme le canal de sortie,

et on y écrit avec `output_string`, `output_char` ou `output`. Les primitives les plus commodes sont `output_value` et `input_value` qui écrivent ou lisent nimporte quelle valeur.

Considérons l'exemple suivant :

```
#let rec fact = fun
    | 0 -> 1
    | n -> n* fact (n-1);;
fact : int -> int = <fun>
#let sortie = open_out "mon_fichier";;
sortie : out_channel = <abstr>
```

On écrit à présent dans le fichier `mon_fichier` :

```
#output_value sortie (fact 10);;
- : unit = ()
#output_string sortie "C'est fini\n";;
- : unit = ()
```

On ferme la sortie :

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

Attention, les écritures dans un canal sont stockées dans un tampon (et non pas écrites immédiatement) : il convient donc régulièrement de le vider explicitement à l'aide de la fonction `flush` (ici `flush sortie;;`). La fonction `close_out` vide automatiquement ce tampon.

À présent on peut lire les résultats depuis `mon_fichier` :

```
#let entree = open_in "mon_fichier";;
entree : in_channel = <abstr>
#let fact10 = input_value entree;;
fact10 : '_a = <poly>
#print_int fact10;;
3628800- : unit = ()
#(fact10:int);;
(* remarquer le forçage de type pour récuperer la valeur *)
- : int = 3628800
#input_line entree;;
- : string = "C'est fini"
#input_line entree;;
Exception non rattrapée: End_of_file
```

Le dernier message d'erreur était prévisible, nous avions déjà atteint la fin du fichier. Enfin, on n'oublie pas de fermer l'entrée :

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

5.5.2 Flots

Il existe une structure de données prédéfinie en Caml très utile pour gérer des entrées : le *flot*[50]. À la page 39, nous avons vu la lourdeur de la saisie d'une expression arithmétique. Nous allons utiliser les flots pour simplifier cela à l'aide de la conversion d'expressions sous forme de syntaxe concrète (c'est-à-dire « naturelle » pour l'utilisateur) en syntaxe abstraite (c'est-à-dire la syntaxe effectivement manipulée par le compilateur). On peut se reporter à [12] pour avoir une description complète du problème de l'analyse lexicale. Ici, nous allons simplement donner un exemple de conversion « syntaxe concrète → syntaxe abstraite » dans le cas du type `expression` étudié précédemment.

Voici le code source que l'on peut adapter à ses besoins[51] :

```
(* Définition des Lexèmes *)

type Lexeme = PARO
            | PARF
            | PLUS
            | MULT
            | EXP
            | CST of int
            | VAR of string;;

(* ANALYSEUR LEXICAL *)

let Identificateur_Long = make_string 32 ' ';;

let rec Ident i = function
    | [< ' 'a'..'z'|'A'..'Z'|'0'..'9'|'_'  as c ;
(* on a droit à des identificateurs qui ne commencent pas *)
(* par un chiffre, de longueur < 33 et indexés au besoin ! *)
     (if i >= 32
      then Ident i
       (* on tronque au dessus de 32 caractères *)
      else begin
              set_nth_char Identificateur_Long i c;
              Ident (succ i)
           end) s >]    -> s
    | [<>]              ->
        (match sub_string Identificateur_Long 0 i with
           | "+"   -> PLUS
           | "*"   -> MULT
           | "exp" -> EXP
           | s     -> VAR s);;
```

50. ou flux (*stream* en anglais).
51. *Token* est la traduction anglaise de *Lexème*.

```
let rec Nombre i = function
    | [< ''0'..'9'  as d ;
(* on a droit à des nombres de longueur < 33 *)
     (if i >= 32 then Nombre i
      else begin
               set_nth_char Identificateur_Long i d;
               Nombre (succ i)
           end) n >]    -> n
    | [<>]              ->
        (match sub_string Identificateur_Long 0 i with
           | "+"   -> PLUS
           | "*"   -> MULT
           | "exp" -> EXP
           | n     -> CST (int_of_string n));;

let rec Blancs = function
  | [< '' '|'\t'|'\n'; Blancs _ >] -> ()
  | [<>] -> ();;

let rec LexemesFlot_of_flot flot = Blancs flot ; match flot with
  | [< ''(' ; Blancs _ >] -> [< 'PARO ; LexemesFlot_of_flot flot >]
  | [< '')' ; Blancs _ >] -> [< 'PARF ; LexemesFlot_of_flot flot >]
  | [< ''e';''x';''p'; Blancs _ >]
                          -> [< 'EXP;   LexemesFlot_of_flot flot >]
  | [< ''+' ; Blancs _ >] -> [< 'PLUS ; LexemesFlot_of_flot flot >]
  | [< ''*' ; Blancs _ >] -> [< 'MULT ; LexemesFlot_of_flot flot >]
  | [< ''0'..'9' as d ;
       (set_nth_char Identificateur_Long 0 d ; Nombre 1 ) lex >]
                          -> [< 'lex ; LexemesFlot_of_flot flot >]
  | [< ''a'..'z'|'A'..'Z' as c ;
      (set_nth_char Identificateur_Long 0 c ; Ident 1 ) lex >]
                          -> [< 'lex ; LexemesFlot_of_flot flot >]
(* dans les 2 derniers cas on accumule les caractères pour *)
(* constituer un identificateur de plus d'une lettre ou un *)
(* nombre de plus d'un chiffre                             *)
  | [<>]                  -> [<>];;

let LexemesFlot_of_string string =
          LexemesFlot_of_flot (stream_of_string string);;

(* ANALYSEUR SYNTAXIQUE *)

let rec Gram1 lexFlot =
  let e1=Gram2 lexFlot in
  match lexFlot with
  | [< 'PLUS ; Gram1 e2 >] -> Addition (e1,e2)
  | [<>] -> e1
```

```
and Gram2 lexFlot =
  let e1=Gram3 lexFlot in
  match lexFlot with
  | [< 'MULT ; Gram2 e2 >] -> Multiplication (e1,e2)
  | [<>] -> e1
and Gram3 = function
  | [< 'EXP ; Gram4 e >] -> Exponentielle (e)
  | [< Gram4 e >] -> e
and Gram4 = function
  | [< 'PARO ; Gram1 e ; 'PARF >] -> e
  | [< 'CST n >] -> Constante n
  | [< 'VAR s >] -> Variable s;;

let Analyse string = Gram1 (LexemesFlot_of_string string);;
```

Et voici un exemple de son utilisation :

```
#Analyse  "exp(x+1)";;
- : expression = Exponentielle
                   (Addition (Variable "x", Constante 1))
#Analyse "x*(y+1)+2";;
- : expression =
 Addition
  (Multiplication
       (Variable "x", Addition (Variable "y", Constante 1)),
        Constante 2)
#Analyse "x_1+y*567*z";;
- : expression =
 Addition
  (Variable "x_1",
   Multiplication
    (Variable "y", Multiplication (Constante 567, Variable "z")))
```

Brièvement, le principe de cet analyseur est le suivant : la chaîne de caractères d'entrée est transformée en un flot de caractères. Celui-ci est ensuite transformé en flot de lexèmes à l'aide de `LexemesFlot_of_Flot` (c'est là que l'on reconnaît les différents constituants de l'expression : constantes, variables, opérateurs...), cette phase s'appelle l'*analyse lexicale*. Chaque lexème a sa signification propre. Ensuite, on passe le tout dans la grammaire des expressions arithmétiques[52] pour reconstituer l'expression arborescente de la syntaxe abstraite (fonction `Gram1`), c'est la phase dite d'*analyse syntaxique*. On a choisi les conventions d'écriture mathématique usuelles.

La conversion inverse se fait à l'aide d'un simple parcours d'arbre (voir le chapitre III).

52. Voyez les `Grami` comme des non-terminaux.

5.5.3 Compilation

La gestion des entrées-sorties trouve toute son utilité dans la réalisation de programmes exécutables indépendants du système Caml interactif (rappelons que si vous avez créé un fichier toto.ml son exécution dans le système interactif se fait à l'aide de la commande `Inclure`). La création d'un programme exécutable autonome est possible si vous travaillez dans un système d'exploitation disposant d'une ligne de commande[53] (comme DOS, Windows ou Unix).

Notre utilisation de la compilation va rester très rudimentaire. Nous n'évoquerons pas la subdivision d'un programme en modules séparés qui coopèrent à l'aide de fichiers d'interface (tout ceci est très bien expliqué dans [26]).

Supposons que l'on dispose du fichier source `test.ml` qui contient :

```
let rec fact = function
    | 0 -> 1
    | n -> n*fact (n-1);;

print_newline ();
print_string "Entrez un entier naturel :";
print_int (fact (int_of_string (read_line ())));
print_newline ();;
```

La création d'un exécutable indépendant se fait depuis le système d'exploitation avec la commande :

`camlc -o` *`nom-de-l'exécutable`*`.exe` *`fichier-source`*`.ml`

(`camlc` fait partie de la distribution[54] de Caml-Light). Sur notre exemple, ceci produit un fichier exécutable (`.exe`) et deux fichiers compilés suffixés `.zo` (le code compilé) et `.zi` (l'interface compilée). L'exécutable est alors directement utilisable depuis la ligne de commande. La figure I.4 montre la compilation et l'exécution de notre fichier exemple `test.ml` sous DOS.

En fait, le fichier exécutable produit n'est que l'appel de la commande `camlrun` (qui est également un fichier de la distribution de Caml Light) sur le fichier objet `.zo`. La présence de ces deux fichiers est donc indispensable à l'exécutable.

Signalons que la compilation peut aussi se faire depuis le système interactif à l'aide de la commande `compile "nom-du-fichier";;`. Comme `camlc`, elle produit les fichiers `.zo` et `.zi`. Le fichier `.zo` peut être chargé depuis le système interactif avec la commande `load_object` (ou `Charger un objet` du menu) et produit alors une exécution similaire au `.exe`.

```
#compile "d:/test.ml";;
- : unit = ()
#load_object "d:/test.zo";;
```

53. Sous Mac OS, il faut utiliser l'application MPW ou se contenter de la compilation depuis le système interactif à l'aide de `compile`.
54. L'appel direct à `camlc` suppose que la variable d'accès aux exécutables (en général PATH) contient le chemin menant aux binaires de Caml.

Figure I.4 : *Compilation et exécution d'un programme Caml*

```
Entrez un entier naturel :10
3628800
- : unit = ()
```

5.6 Valeurs exceptionnelles

Quittons les entrées-sorties et observons ce qui se produit lorsque l'on demande :

```
#hd [];;
Exception non rattrapée: Failure "hd"
```

Caml renvoie un message d'erreur : il n'y a pas de premier élément dans la liste vide ! En fait c'est un peu plus précis que cela :

Caml *déclenche une exception*.

Une exception, c'est le vilain petit canard, c'est LE cas (ou les cas) où la fonction ne marche pas. Enfin, pas seulement... Dans cette section, nous allons détailler la gestion des exceptions, exposer la définition par l'utilisateur de nouvelles exceptions avant d'aborder un point de vue plus général concernant l'utilisation des exceptions comme style de programmation.

5.6.1 Gestion des exceptions

En Caml, nous pouvons :

- utiliser les exceptions prédéfinies
- ou bien en créer de nouvelles.

Pour l'instant, passons en revue quelques exceptions prédéfinies en Caml. On distingue deux types d'exceptions en Caml : les exceptions constantes et les exceptions paramétrées (cf. les types).

Parmi les exceptions constantes, citons l'exception `Not_found` qui apparaît lorsqu'une fonction de recherche n'aboutit pas, `Division_by_zero`, qui se passe de commentaire, et enfin, celle que l'on ne souhaite pas : `Out_of_memory`.

```
#1/(2-2);;
Exception non rattrapée: Division_by_zero
#index "luc" ["pierre";"paul";"luc";"jean"];;
- : int = 2
#index "luc" ["pierre";"paul";"jacques";"jean"];;
Exception non rattrapée: Not_found
```

(le lecteur a compris que `index` recherche la présence d'un élément dans une liste).

La plus célèbre exception paramétrée en Caml est sans doute l'exception d'échec nommée `Failure` qui est paramétrée par le nom de la fonction déclenchant l'échec. Nous l'avons rencontrée avec `hd []`. Il y a également l'exception `Invalid_argument`, elle aussi accompagnée d'une chaîne de caractère : elle signifie bien sûr que l'argument appelé n'a pas de sens avec la fonction considérée. En voici une illustration :

```
#let v = [|"pierre";"paul";"luc";"jean"|];;
v : string vect = [|"pierre"; "paul"; "luc"; "jean"|]
#v.(1);;
- : string = "paul"
#v.(5);;
Exception non rattrapée: Invalid_argument "vect_item"
```

Le déclenchement d'une exception se fait avec le mot-clé `raise` (qui se traduit par *lever* en français). La fonction `hd` peut ainsi être définie par :

```
let hd = function
    | []  -> raise (Failure "hd")
    | a::l -> a;;
```

Avez-vous remarqué que la fonction factorielle, que nous avons plusieurs fois définie, boucle sur un entier négatif ? Il vaut mieux dans ce cas déclencher l'exception `Invalid_argument` :

```
#let rec fact  n =
    if n >= 0
    then if n=0 then 1 else n* fact(n-1)
    else raise (Invalid_argument "n positif, SVP !");;
fact : int -> int = <fun>
#fact 5;;
- : int = 120
#fact (-3);;
Exception non rattrapée: Invalid_argument "n positif, SVP !"
```

Les commandes `raise Invalid_argument` et `raise Failure` sont si courantes qu'il en existe en Caml des abréviations : `invalid_arg` et `failwith`. Par exemple, le code de `hd` ci-dessous est équivalent au précédent :

```
let hd = function
    | []  -> failwith "hd"
    | a::l -> a;;
```

L'utilisateur peut facilement définir ses propres exceptions. Signalons au passage que les exceptions ont leur propre type de base dont nous n'avons pas encore parlé : le type `exn`.

```
#Division_by_zero;;
- : exn = Division_by_zero
```

Pour créer une nouvelle exception, on utilise le mot-clé `exception`. On peut par exemple faire plaisir à l'académie française :

```
#exception Division_par_zero;;
L'exception Division_par_zero est définie.
#let ma_division n = fun
    | 0 -> raise Division_par_zero
    | m -> n/m;;
ma_division : int -> int -> int = <fun>
#ma_division 8 4;;
- : int = 2
#ma_division 5 0;;
Exception non rattrapée: Division_par_zero
```

On procède de manière analogue pour définir de nouvelles exceptions paramétrées :

```
#exception Argument_non_valide of float;;
L'exception Argument_non_valide est définie.
#let mon_log x =
   if x <. 0.
   then raise (Argument_non_valide x)
   else log x;;

mon_log : float -> float = <fun>
#mon_log 2.;;
- : float = 0.69314718056
#mon_log (-2.);;
Exception non rattrapée: Argument_non_valide -2.0
```

Jusqu'à présent, nous n'avons pas fait grand chose des exceptions (mis à part produire des messages d'erreurs). Il faut remarquer la formule employée par Caml : `Exception non rattrapée`. En effet, on peut *rattraper une exception*, principalement lorsqu'une fonction appelante reçoit un déclenchement d'exception de la part d'une fonction appelée et qu'elle sait comment « récupérer la situation ». La syntaxe Caml est alors la suivante :

```
try expression-qui-déclenche-l'-exception
with attitude-à-adopter;;
```

« L'attitude à adopter » est un filtrage. Elle est de la forme :

```
exception1 -> action1
| exception2 -> action2
| ...
```

Par exemple :

Pourquoi un nom si long ?

```
#let division_qui_retourne_toujours_un_resultat n  m =
try ma_division n m
with Division_par_zero -> 0;;
division_qui_retourne_toujours_un_resultat :
                              int -> int -> int = <fun>
#division_qui_retourne_toujours_un_resultat 8 4;;
- : int = 2
#division_qui_retourne_toujours_un_resultat 8 0;;
- : int = 0
```

Pour réfléchir sur un exemple plus élaboré, reprenons encore une fois la factorielle. Nous avons déjà remarqué que l'argument ne doit pas être négatif, mais il ne doit pas non plus être trop grand sous peine de dépasser la taille maximale allouée aux entiers [55]. Les calculs en Caml sur les entiers sont effectués modulo $2^{31} = 2147483648$, un dépassement de cette borne *ne provoquant aucun message d'erreur*. Le calcul est erroné dès que l'on dépasse $2^{30} = 1073741824$:

```
#1073741823;;
- : int = 1073741823
#1073741823+1;;
- : int = -1073741824
```

Si l'on ne fait pas attention, on aboutit rapidement à des surprises :

```
#fact 12;;
- : int = 479001600
#fact 13;;
- : int = -215430144
```

On peut donc définir une exception `Depassement_sur_les_entiers` et l'utiliser avec profit :

```
#exception Depassement_sur_les_entiers;;
L'exception Depassement_sur_les_entiers est définie.
#let fact n =
   if n <0
   then raise (Invalid_argument "n positif, SVP !")
   else let res = ref 1 in
        for i=1 to n do
          begin
            res:= !res*i;
            if !res < 0 then raise Depassement_sur_les_entiers
          end
        done;
        !res;;
fact : int -> int = <fun>
```

55. Il ne s'agit pas d'entiers, mais... regardez quand même le module `num`, un module de calcul rationnel en précision arbitraire.

```
#fact 12;;
- : int = 479001600
#fact 13;;
Exception non rattrapée: Depassement_sur_les_entiers
#fact (-2);;
Exception non rattrapée: Invalid_argument "n positif, SVP !"
```

on a choisi une version impérative de factorielle pour déterminer précisément quand le calcul devient erroné.

On peut alors utiliser ce code pour calculer des coefficients binomiaux :

```
#let binomial n p =
     try
 string_of_int ((fact n)/( (fact p) * (fact (n-p)) ))
     with
  Invalid_argument s ->  "Calcul de factoriel impossible"
| Depassement_sur_les_entiers -> "Dépassement dans factoriel";;
binomial : int -> int -> string = <fun>
#binomial 4 2;;
- : string = "6"
#binomial 2 4;;
- : string = "Calcul de factoriel impossible"
#binomial 15 14;;
- : string = "Dépassement dans factoriel"
```

(ceci est particulièrement inefficace, reportez vous à l'exercice I.8).

L'exemple de factorielle nous montre que le déclenchement d'une erreur arrête l'évaluation de la fonction en cours et passe l'erreur à la fonction appelante ou au *top-level* (la ligne de commande Caml) le cas échéant. Cette considération peut permettre d'optimiser nos programmes Caml. Le lecteur trouvera une utilisation d'exception pour la résolution du problème des n reines sur l'échiquier (voir page 182).

5.6.2 Utilisation des exceptions en programmation

L'un des apports essentiels des exceptions réside dans la gestion de valeurs exceptionnelles qui deviennent récupérables par la fonction appelante. Nous allons voir maintenant une autre façon d'utiliser les exceptions : la possibilité d'interrompre toute exécution dès que la valeur cherchée est trouvée. Pour illustrer ces deux aspects, étudions un exemple issu du monde impitoyable de la finance. Imaginons qu'un banquier tienne à jour pour chacun de ses clients une liste des découverts de compte courant. Chaque client a une autorisation de découvert (disons -3000 pour fixer les idées) et en fin d'année, le banquier regarde cette liste et compare le minimum de cette liste avec la valeur -3000. Trois cas se présentent :

Cette obsession de l'argent, ça cache quelque chose non ?

- le plus grand découvert n'a pas franchi le cap des -3000 et le client est jugé « passable »,
- le plus grand découvert est inférieur à -3000 et là, malheur au client fautif !
- il n'y a pas eu de découvert dans l'année (et le client est un oiseau rare à soigner tout particulièrement).

Le banquier ne sait pas le faire tout seul...

Le banquier a demandé à un informaticien de lui programmer la fonction `minlist` qui détermine le minimum d'une liste d'entiers. L'informaticien, étant consciencieux, prévoit le cas de la liste vide en déclenchant une erreur. Le banquier a sa fonction `bilan_annuel` qui fonctionne ainsi :

Le terme «valeur exceptionnelle» est adapté pour un découvert bancaire, non ?

```
#let bilan_annuel liste_des_decouverts =
try
  if (minlist liste_des_decouverts) < -3000
  then "interdit_bancaire"
  else "ca_passe"
with liste_vide -> "Bon client";;
bilan_annuel : int list -> string = <fun>

#let liste_des_decouverts =
[-10;-512;-300;-1000;-2000;-3010;-700];;

#bilan_annuel liste_des_decouverts;;
- : string = "interdit_bancaire"

#let liste_des_decouverts =
[-10;-50;-300;-1000;-2000;-500;-700];;

#bilan_annuel liste_des_decouverts;;
- : string = "ca_passe"

#let liste_des_decouverts = [];;

#bilan_annuel liste_des_decouverts;;
- : string = "Bon client"
```

La fonction `minlist` fournie par l'informaticien est :

```
let rec minlist = function
    | [] -> failwith "liste_vide"
    | [a] -> a
    | a::r -> min a (minlist r);;
```

Cet exemple est l'illustration de la gestion du cas exceptionnel (pour l'informaticien) récupérable (pour le banquier).

L'informaticien, grand pédagogue, l'a convaincu des vertus de Caml

À présent le banquier (qui s'y connaît quand même un peu en informatique) réalise qu'il peut améliorer lui-même la fonction `minlist`. Il est en effet inutile de continuer à parcourir la liste des découverts si l'on a déjà trouvé une valeur inférieure à -3000. La modification est alors la suivante :

```
let rec detecte = function
    | [] -> failwith "liste_vide"
    | a::r -> if a < (-3000)
              then failwith "interdit_bancaire"
              else r<> [] && detecte r;;
```

Notez également l'utilisation du connecteur paresseux `&&`.

Le parcours de liste est effectué tant que les valeurs sont supérieures à -3000. La fonction appelante filtre alors les différentes valeurs exceptionnelles :

```
let bilan_annuel liste_des_decouverts =
  try
    detecte  liste_des_decouverts ;
    "ça passe"
(* on est dans ce cas si minlist n'a pas déclenché d'exceptions *)
  with Failure "liste_vide" -> "bon_client"
     | Failure s -> s;;
```

Signalons pour conclure que la gestion des valeurs exceptionnelles peut également se faire directement par le typage. On utilise un type particulier :

```
type 'a option = None | Some of 'a;;
```

prédéfini en Caml 0.73.

Les fonctions sont modifiées pour retourner un type `option`. Définissons par exemple une fonction `minlist` qui retourne non pas un `int` mais un `int option` :

```
#let rec minlist = function
    | []   -> None
    | [a]  -> Some a
    | a::r -> min (Some a) (minlist r);;

minlist : 'a list -> 'a option = <fun>
#minlist [1;2;6;-3;10];;
- : int option = Some -3
#minlist [];;
- : '_a option = None
```

La valeur exceptionnelle est ainsi repérée par `None`. L'exception est ainsi prise en charge par le typage et non plus à l'aide de messages d'erreur.

5.7 Débogage

Cette section aurait dû arriver plus tôt !

Il est beaucoup plus facile de réaliser un débogueur en programmation impérative, où un programme est une suite d'instructions que l'on peut exécuter pas à pas, qu'en programmation fonctionnelle, où il s'agit dévaluer une expression. La recherche des erreurs est donc malaisée en Caml. Néanmoins, face aux bogues, vous avez des alliés. Nous allons énumérer les principaux et illustrer leur action sur quelques exemples (trop simples sans doute...).

Tout d'abord et tout simplement *le typage fort* de Caml détecte les erreurs les plus grossières.

```
#let rec fact  = fun
     | 0 -> 1.
     | n -> n* fact (n-1);;
```

```
Entrée interactive:
>     | n -> n* fact (n-1);;
>             ^^^^^^^^^^^^
Cette expression est de type int,
mais est utilisée avec le type float.
```

C'était évident, on a laissé traîner un « point » après le 1 ce qui le rend réel !

Le *filtrage* fournit lui aussi des indications :

```
#let rec fact = fun
    | 0 -> 1
    | n -> n * fact (n-1)
    | 1 -> 1;;
Entrée interactive:
>    | 1 -> 1;;
>      ^
Attention: ce cas de filtrage est inutile.
fact : int -> int = <fun>
```

L'*ajout de contraintes de type* est plutôt efficace. Cela permet à Caml d'améliorer sa détection d'erreur. Considérons un exemple :

```
#let rec f =  fun x ->
    if x = 0 then f true else f (x-1);;
Entrée interactive:
>    if x = 0 then f true else f (x-1);;
>                                ^^^
Cette expression est de type int,
mais est utilisée avec le type bool.
```

L'utilisateur désire en fait programmer une fonction de type `int -> int`. En le précisant, Caml va détecter que c'est le booléen `true` qui est fautif.

```
#let rec (f:int->int) = fun x->
      if x =0 then f true else f (x-1);;
Entrée interactive:
>      if x =0 then f true else f (x-1);;
>                     ^^^^
Cette expression est de type bool,
mais est utilisée avec le type int.
```

Retenons qu'il ne faut pas hésiter à parsemer son programme de contraintes de type avisées.

Si on ne peut faire d'exécution pas à pas, il existe quand même *la fonction* `trace` qui permet d'afficher tous les appels à une fonction donnée. L'argument de `trace` est le nom de la fonction en question (la commande `untrace` permet d'annuler l'appel à `trace`). Considérons (encore une fois) la factorielle :

```
let rec fact = fun
    | 0 -> 1
    | n -> n * fact(n-1);;
```

Nous avons déjà remarqué qu'elle ne terminait pas avec un argument négatif. On peut visualiser cela en « traçant » fact :

```
#trace "fact";;
La fonction fact est dorénavant tracée.
- : unit = ()
#fact (-2);;
fact <-- -2
fact <-- -3
fact <-- -4
fact <-- -5
fact <-- -6
fact <-- -7
fact <-- -8
fact <-- -9
fact <-- -10
fact <-- -11
fact <-- -12
...
Interruption.
```

L'observation d'une fonction à l'aide de `trace` permet souvent de déterminer l'erreur (qui est sémantique cette fois, puisque l'utilisation de `trace` n'est possible que sur des fonctions acceptées par le compilateur donc syntaxiquement correctes). Toutefois, `trace` ne sait pas afficher les appels de fonctions polymorphes. L'affichage est alors sans intérêt. Reprenons, par exemple, la fonction `minlist` précédente :

```
#let rec minlist = function
    | [] -> failwith "liste_vide"
    | [a] -> a
    | a::r -> min a (minlist r);;

minlist : 'a list -> 'a = <fun>
#trace "minlist";;
La fonction minlist est dorénavant tracée.
- : unit = ()
#minlist [-2;0;-4;1];;
minlist <-- [<poly>; <poly>; <poly>; <poly>]
minlist <-- [<poly>; <poly>; <poly>]
minlist <-- [<poly>; <poly>]
minlist <-- [<poly>]
minlist --> <poly>
minlist --> <poly>
minlist --> <poly>
minlist --> <poly>
- : int = -4
```

On peut facilement contourner le problème en forçant le type de `minlist` :

```
#let rec (minlist:int list->int) = function
    | [] -> failwith "liste_vide"
    | [a] -> a
    | a::r -> min a (minlist r);;
minlist : int list -> int = <fun>
#trace "minlist";;
La fonction minlist est dorénavant tracée.
- : unit = ()
#minlist [-2;0;-4;1];;
minlist <-- [-2; 0; -4; 1]
minlist <-- [0; -4; 1]
minlist <-- [-4; 1]
minlist <-- [1]
minlist --> 1
minlist --> -4
minlist --> -4
minlist --> -4
- : int = -4
```

Le dernier recours du programmeur devant une fonction « récalcitrante » est de provoquer lui-même l'affichage de variables pertinentes à l'aide d'insertions dans le code d'instructions `print_int`, `print_string`... supplémentaires. Cette méthode rudimentaire a fait ses preuves...

Si malgré tout cela, le programme persiste dans ses dysfonctionnements... il faut consulter les « gourous » de la caml-list sur internet. Voici par exemple une erreur classique sur les matrices.

Un étudiant doit faire un programme de transposition de matrice d'entiers 3×3. Il propose la solution suivante :

```
#let transpose m =
    let q = make_vect 3 (make_vect  3 0) in
            for i = 0 to 2 do
                for j = 0 to 2 do
                    q.(j).(i) <- m.(i).(j)
                done
            done;
    q;;
transpose : int vect vect -> int vect vect = <fun>
```

(`make_vect n x` est l'instruction qui crée un vecteur de longueur `n` rempli de `x`).

Prudent, il fait quand même un essai :

```
#transpose
[|  [| 1; 2; 3 |];
    [| 0; 1; 2 |];
    [| 0; 0; 1 |] |];;
```

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

Manifestement, le programme ne fonctionne pas convenablement ! Les trois lignes sont identiques. Il ne s'agit pas d'un problème de type. L'étudiant modifie alors le code de `transpose` :

```
#let transpose m =
  let q = make_vect 3 (make_vect  3 0) in
(* on initialise par la matrice nulle *)
    for i = 0 to 2 do
      for j = 0 to 2 do
        q.(j).(i) <- m.(i).(j);
        print_newline ();
        printf__printf "la valeur de q(%d,%d) est %d" j i q.(j).(i);
        print_newline ();
        printf__printf "la valeur de m(%d,%d) est %d" i j m.(i).(j)
      done
     done;;

transpose : int vect vect -> unit = <fun>
#transpose
[|  [| 1; 2; 3 |];
    [| 0; 1; 2 |];
    [| 0; 0; 1 |] |];;

la valeur de q(0,0) est 1
la valeur de m(0,0) est 1
la valeur de q(1,0) est 2
la valeur de m(0,1) est 2
la valeur de q(2,0) est 3
la valeur de m(0,2) est 3
la valeur de q(0,1) est 0
la valeur de m(1,0) est 0
la valeur de q(1,1) est 1
la valeur de m(1,1) est 1
la valeur de q(2,1) est 2
la valeur de m(1,2) est 2
la valeur de q(0,2) est 0
la valeur de m(2,0) est 0
la valeur de q(1,2) est 0
la valeur de m(2,1) est 0
la valeur de q(2,2) est 1
la valeur de m(2,2) est 1- : unit = ()
```

et constate que les affectations sont bonnes ! Seul un gourou[56] peut tirer l'étudiant

56. L. Cheno en l'occurrence.

de son désarroi : l'instruction `make_vect 3 (make_vect 3 0)` crée un vecteur bidimensionnel où toutes les lignes *partagent* (en mémoire) la même colonne. Il fallait taper :

```
...
let une_col = make_vect 3 0 in
    let q = make_vect 3 une_col in
        for i = 0 to ...
```

ou encore utiliser tout simplement le type prédéfini `matrix` :

```
...
let q = make_matrix 3 3 0 in
    for i= 0 to ...
```

5.8 Graphisme

Nous allons terminer par le complément le plus esthétique : le graphisme en Caml. Le module `graphics` contient les primitives du langage destinées au graphisme. Une session graphique commence donc par la commande `#open "graphics";;`.

Figure I.5 : *La fenêtre graphique de Caml.*

L'instruction de base est `open_graph s;;` où `s` est une chaîne de caractères. Elle commande l'ouverture de la fenêtre graphique suivant les paramètres précisés par `s`. Typiquement `open_graph "nxm";;` ouvre une fenêtre graphique de `n` pixels de largeur et de `m` pixels de hauteur[57]. Si `s=""`, la taille par défaut est appliquée, taille que l'on peut connaître à l'aide des commandes `size_x ();;` et `size_y ();;`. L'origine des coordonnées dans la fenêtre graphique, `(0,0)`, est le coin en bas à gauche de la fenêtre, les abscisses (resp. les ordonnées) varient ainsi entre `0` et `size_x () -1` (resp. entre `0` et `size_y () -1`).

57. Certains systèmes d'exploitation ne permettent pas cette option et l'argument `s` est ignoré. C'est alors la taille par défaut qui est utilisée.

Tout dessin en dehors de la fenêtre est coupé et ne provoque pas d'erreur. La fermeture de la fenêtre graphique se fait par `close_graph ();;`. Les fonctions graphiques sont documentées dans l'aide du module `graphics`, citons ici les principales :

- `clear_graph ()` efface la fenêtre graphique ;
- `set_color` *`couleur`* fixe la couleur courante à *couleur* (les couleurs de base sont prédéfinies, on peut également utiliser le système RGB) ;
- `plot n m` dessine un point de la couleur courante au point de coordonnées (n, m) ;
- `moveto n m` positionne le point courant en (n, m) ;
- `lineto n m` trace une ligne de la couleur courante (dans l'épaisseur courante) du point courant au point de coordonnées (n, m) ;
- `draw_circle n m r` dessine un cercle de la couleur courante de rayon `r` dont le centre a pour coordonnées (n, m).

Figure I.6 : *Débuts en graphisme...*

Par exemple, on peut observer le résultat de la session Caml suivante sur la figure I.6 :

```
##open "graphics";;
#open_graph "";;
- : unit = ()
#let xmax = (size_x ())-1 and ymax = size_y () -1;;
xmax : int = 479
ymax : int = 279
#draw_circle (xmax/4) (ymax/4) (min (xmax/4) (ymax/4));;
- : unit = ()
#set_color red;;
- : unit = ()
#fill_rect (xmax/2) (ymax/2) xmax ymax;;
- : unit = ()
#set_color black;;
- : unit = ()
#moveto (xmax/2) 0;lineto (3*xmax/4) (ymax/4); lineto xmax 0;;
- : unit = ()
```

Il existe également des fonctions pour placer du texte sur la fenêtre graphique, des fonctions de contrôle du clavier (dont `key_pressed`, `read_key`) et de la souris, un type `image` (représentation sous la forme de matrices de couleurs) et les fonctions correspondantes pour sauvegarder et travailler toutes vos créations...

Écrivons, par exemple, un petit programme de tracé de fonctions :

```
#open "graphics";;
open_graph "";;

(* la taille de la fenêtre par défaut *)
let w=float_of_int (size_x ());;
let h=float_of_int (size_y ());;

(* la fonction *)
let f x = (sin x) /. x;;

(* l'intervalle d'étude *)
let a=(-15.);;
let b=15.;;

(* le nombre de points d'interpolation *)
let n=100;;

(* le pas suivant x *)
let pas = (b-.a)/.(float_of_int (n-1));;

(* les valeurs de f *)
let v = make_vect n 0.0;;

(* init_valeurs calcule de plus les max et min de f *)
let init_valeurs () =
  v.(0) <- (f a);
  let fmax = ref (v.(0)) and fmin = ref (v.(0)) in
  let x= ref a in
    for i=1 to (n-1) do
      x:= !x +. pas;
      v.(i) <- (f !x);
      if v.(i) > !fmax then fmax:=v.(i);
      if v.(i) < !fmin then fmin:=v.(i)
    done;
    if (!fmin = !fmax)
    then (!fmin -. 1., !fmax +. 1.)
(* cas d'une fonction constante *)
    else (!fmin,!fmax);;

(* on place le graphe de f dans la fenêtre en *)
(* laissant une marge de 10% *)
```

```
let dilatation () =
     let (ymin,ymax) = init_valeurs () in
     let coefx = (b -. a) /. (0.9 *. w) and
         coefy = (ymax -. ymin) /. (0.9 *. h) in
     let decal_x= 0.05 *. w  and
         decal_y= 0.05 *. h -. ymin /. coefy in
     (coefx,coefy,decal_x,decal_y);;

let trace_fonction () =
     clear_graph ();
     let (coefx,coefy,decal_x,decal_y) = dilatation () in
     moveto (int_of_float decal_x)
            (int_of_float (decal_y +. v.(0) /. coefy));
     for i=1 to (n-1) do
     lineto (int_of_float (decal_x +. (float_of_int i)
                                  *. pas /. coefx))
            (int_of_float (decal_y +. v.(i) /. coefy ))
     done;;
```

Afin de tracer toute la courbe dans la fenêtre, on a calculé les valeurs extrémales de f et ajusté l'échelle en conséquence (le dessin n'est donc pas *a priori* isométrique). L'appel de `trace_fonction ();;` produit alors la figure I.7.

Figure I.7 : *Sinus cardinal.*

La récursivité permet de produire des dessins intéressants. Faites donc l'exercice I.15 pour produire la figure I.8.

Figure I.8 : *Carrés emboîtés.*

6 Conclusion

L'étude exhaustive de Caml ne constituant pas notre objectif, nous allons arrêter là cette brève présentation. Pour tout approfondissement sur le langage ou la programmation fonctionnelle, nous renvoyons le lecteur aux ouvrages [34] et [12] déjà cités. Signalons que, entre autres richesses, le site Web permet l'accès à la liste de discussion sur Caml où l'échange d'informations est très fructueux.

En guise de conclusion, nous allons satisfaire la curiosité du lecteur qui a eu la patience (le courage ?) de lire cette présentation jusqu'au bout et lui donner (pour l'anecdote) la signification et l'origine du mot Caml[58]. Caml est la contraction des mots CAM et ML. CAM pour « Categorical Abstract Machine » et ML pour « Meta-Language ». L'explication de ces sigles nécessite un tout petit peu d'histoire. À l'origine (dans les années 1930), il y eut le λ-calcul, théorie sur le fondement des mathématiques (aspect fonctionnel et calculabilité) où excella A. Church. Évidemment, le λ-calcul n'avait pas à l'époque, vocation à développer un langage de programmation. Puis, il y eut les recherches sur les méthodes de preuve qui conduisirent R. Milner en 1978 à la mise au point de ML, un méta-langage dédié aux démonstrations de programmes. Enfin, également dérivés du λ-calcul, il y eut des travaux sur les catégories... et les techniques de compilation de ces combinateurs catégoriques donnèrent naissance à la CAM. Finalement, l'implémentation de ML bâtie sur la CAM[59] s'est naturellement appelée Caml...

58. cf. le Caml Tutorial de M. Mauny. Une référence précieuse, disponible en anglais sur le site Web.
59. L'implémentation sur micros de Caml, Caml-Light, n'a aujourd'hui plus rien à voir avec la CAM, seul le mot est resté...

Exercices

Exercice I.1.

Définir la fonction `th` (tangente hyperbolique) en ne faisant qu'un calcul d'exponentielle.

Exercice I.2.

Lorsque l'on introduit l'ensemble $\mathbb{N}$ à l'aide des axiomes de Peano, la première opération que l'on présente est l'addition. La multiplication dans $\mathbb{N}$ est ensuite définie récursivement de la façon suivante :

- $\forall\ n \in \mathbb{N},\ n * 0 = 0$,
- $\forall\ (n, p) \in \mathbb{N} \times \mathbb{N}^*,\ n * p = n * (p - 1) + n$.

En suivant cette définition, programmer la fonction `Mult n p`.

Exercice I.3. Ordre lexicographique.

Il existe plusieurs ordres qui généralisent sur $\mathbb{Z}^2$ l'ordre naturel $\leqslant$ sur les entiers. L'un d'entre eux, très célèbre, est l'ordre lexicographique. Il est défini par :

$$(a, b) \preccurlyeq (c, d) \qquad \text{si} \qquad (a \leqslant c) \quad \text{ou} \quad \text{si} \quad (a = c \quad \text{et} \quad b \leqslant d)$$

Il s'agit d'un ordre total qui peut être généralisé à des triplets, des quadruplets etc. À partir de l'ordre usuel des lettres de l'alphabet, on retrouve en fait, sur les mots, l'ordre du dictionnaire d'où le nom d'ordre « lexicographique ».

1° Implémenter le test `inferieur_lexico` sur $\mathbb{Z}^2$.

2° Proposer une généralisation pour des uplets d'entiers de longueur quelconque.

Exercice I.4.

Nous avons en page 27 écrit en Caml une fonction `miroir` de symétrisation de listes (`miroir [1;2;3] = [3;2;1]`) utilisant une fonction auxiliaire récursive `miroir_aux`.

```
let rec miroir_aux accu = fun
        | [] -> accu
        | (a::suite) -> miroir_aux (a::accu) suite;;
let miroir l = miroir_aux [] l;;
```

et dont l'exécution donne :

```
#miroir_aux [3;2;1] [6;7;8];;
- : int list = [8; 7; 6; 3; 2; 1]
#miroir [1;2;3];;
- : int list = [3; 2; 1]
```

1° Proposer une version impérative de `miroir` à l'aide d'une boucle.

2° Quel désavantage présente la version suivante naïve de `miroir` :

```
let rec miroir_naif = function
    | [] -> []
    | a::r -> (miroir_naif r) @ [a];;
```

Exercice I.5. Palindrome.

Un palindrome est un mot qui est identique lorsqu'on le lit de droite à gauche ou de gauche à droite, comme par exemple : « Bob » ou le célèbre « Tu l'as trop écrasé César ce Port Salut »[60]. Nous allons nous intéresser aux entiers dont l'écriture décimale est un palindrome.

1° Définir la fonction `renverse` qui renverse l'écriture décimale d'un entier (`renverse 123 = 321`). Pour cela, convertir le nombre en la liste de ses chiffres considérés comme des caractères...

2° En déduire le test `palindrome` sur les entiers.

Exercice I.6. Rendez la monnaie !

Le but de l'exercice est d'écrire une fonction `change` qui détermine une façon de payer une somme d'argent s à l'aide d'une liste de valeurs de billets disponibles. Nous supposerons que cette liste est donnée suivant un ordre décroissant. Par exemple :

```
#change 1790 [500;200;100;50;10];;
- : int list = [500; 500; 500; 200; 50; 10; 10; 10; 10]
```

signifie que l'on peut payer 1790 Fr avec 3 billets de 500 Fr, un de 200, un de 50 et 4 de 10.

Ça existe encore les billets de 10 ??

Écrire la fonction change récursivement en prévoyant le cas d'impossibilité de paiement.

Exercice I.7.

Définir la fonction `dirac` qui à `x` réel associe sa fonction de dirac δ_x, qui est l'application qui vaut 1 en `x` et 0 ailleurs.

Exercice I.8.

Nous avons dit que le calcul des coefficients binomiaux effectué en page 29 directement à l'aide de la factorielle était particulièrement inefficace. Nous allons en proposer d'autres implémentations :

1° Définir une fonction `produit n m` qui calcule le produit des entiers compris (au sens large) entre `n` et `m` et en déduire une autre implémentation de `binomial`.

2° À l'aide de formules usuelles, proposer d'autres implémentations.

Exercice I.9. Marche aléatoire.

On suppose disponible une fonction `hasard n : int -> int` qui fournit aléatoirement un entier compris entre `0` et `n-1`. On utilise `hasard 2` pour simuler une marche aléatoire sur un axe : selon le résultat, on fait un pas en avant ou un pas en arrière ; de même, on utilise `hasard 4` pour simuler une marche aléatoire sur un quadrillage plan : selon le résultat, on fait un pas vers le Nord, ou vers l'Est, ou vers le Sud, ou vers l'Ouest. Dans tous les cas, on part de l'origine, et on appelle retour à l'origine tout passage par le point de départ après k pas ($k > 0$).

60. Oubliez pour cet exemple saugrenu l'apostrophe et les blancs.

1° Écrire, en Caml, une fonction qui simule une marche aléatoire de n pas sur un axe ; cette fonction devra retourner l'abscisse du point d'arrivée, l'abscisse du point le plus à droite atteint au cours de la marche, et le nombre de retours à l'origine.

2° Écrire, en Caml, une fonction qui simule une marche aléatoire d'au plus n pas, sur un quadrillage plan, jusqu'au premier retour à l'origine, et calcule le nombre de pas effectués.

Exercice I.10.

À l'aide de la composition des applications définie en page 29, programmer récursivement `itere f n` qui à la fonction f et l'entier n associe l'application $f \circ f \circ \ldots \circ f$ (f itérée n fois).

Exercice I.11. Jouons aux cartes.

On considère un jeu de cartes à jouer traditionnel de 32 cartes c'est-à-dire composé de figures (Roi, Dame, Valet), de cartes numérotées de 7 à 10 et des quatre As. La valeur des cartes numérotées est 0, la valeur des figures est 1 pour le Valet, 2 pour la Dame et 3 pour le Roi.

1° Définir le type `cartes`.

2° Définir une application `valeur` de type `cartes list -> int` qui à une liste de cartes associe la somme des valeurs de celles-ci.

Exercice I.12. Le problème du drapeau hollandais.

Le problème connu sous ce nom consiste à trier une liste de « couleurs » (Rouge, Blanc et Bleu) de telle sorte que les Rouges se retrouvent d'abord, puis les Blancs et enfin les Bleus (on peut voir ce problème comme le tri d'une liste d'entiers ne contenant que des 1, des 2 et des 3). On procède par échange d'éléments.

Avec de telles couleurs, on aurait pu faire le « problème du drapeau français » mais on est à l'heure de l'europe, non ?

On définit le type `couleur` comme suit :

```
type couleur = Rouge | Blanc | Bleu;;
```

1° Programmez une fonction `une_passe` de type `couleur list -> couleur list * bool` qui parcourt une liste de couleurs et permute deux couleurs consécutives lorsqu'elles ne sont pas dans l'ordre final voulu. Le booléen retourné signale s'il y a eu un échange effectué.

```
#une_passe
[Blanc ; Bleu ; Rouge ; Bleu; Bleu ; Blanc; Rouge; Bleu; Rouge];;
- : couleur list * bool =
[Blanc; Rouge; Bleu; Bleu; Blanc; Bleu; Rouge; Rouge; Bleu], true
```

2° En déduire la fonction `drapeau_hollandais` qui effectue le tri complet.

Exercice I.13. Un jeu de « société » : le fizzbuzz.

Des invités (moins de 7) sont assis autour d'une table ronde et comptent « à la fizzbuzz » : ils comptent à tour de rôle à haute voix les entiers à partir de 1 en suivant les règles suivantes (données par ordre de priorité) :

- si n est un multiple de 35, le joueur ne dit pas n mais « fizzbuzz »,

- si n est un multiple de 5, le joueur ne dit par n mais « buzz »,
- si n est un multiple de 7 ou n contient un 7 dans son écriture décimale, le joueur ne dit pas n mais « fizz »,
- dans les autres cas, le joueur dit effectivement n.

Les nombres annoncés par les joueurs sont ainsi, si personne ne se trompe : 1, 2, 3, 4, *buzz*, 6, *fizz*, 8, 9, *buzz*, 11, 12, 13, *fizz*, *buzz*, 16, *fizz*, 18... Le but de la tablée est d'obtenir le plus haut score[61]. Si un joueur se trompe, on recommence à partir de lui à 1. Le but de l'exercice est de programmer la suite des mots prononcés par les joueurs.

Évidemment, il convient de traiter ce problème simple en créant des fonctions auxiliaires.

1° Définir les fonctions suivantes :

a) la fonction `charlist_of_string`, qui convertit une chaîne de caractères en la liste de ses caractères,

b) la fonction `contient c s` qui teste si le caractère `c` apparaît dans la chaîne de caractères `s`.

2° Définir la fonction `fizzbuzz` n qui écrit à l'écran la valeur qui doit être prononcée en fonction de `n`.

3° En déduire, la fonction `list_fizzbuzz n` qui affiche la séquence des `fizzbuzz k` pour $k \in [\![1, n]\!]$.

Exercice I.14. Crible d'Ératosthène.

Le crible d'Ératosthène permet d'obtenir rapidement une (petite) liste de nombres premiers ; par exemple, pour calculer la liste des nombres premiers inférieurs à 100, on part d'une liste de tous les entiers inférieurs ou égaux à 100, puis on barre tous les multiples de 2 (sauf 2), tous les multiples de 3 (sauf 3), tous les multiples de 5 (sauf 5), et tous les multiples de 7 (sauf 7) ; les « survivants » forment la liste souhaitée.

1° En voici une traduction[62] en Caml :

```
let n = 100;;
let grille =  make_vect (n+1) true;;
let crible n v p =
...;;
let eratosthene () =
  let res = ref [] in
   crible n grille 2;
   crible n grille 3;
   crible n grille 5;
   crible n grille 7;
```

61. On peut augmenter la difficulté du jeu en imposant un changement de sens à chaque multiple de 5 c'est-à-dire à chaque buzz ou fizzbuzz.
62. Pour simplifier les indices on décide de ne pas se servir des cases 0 et 1 du vecteur de booléens `grille`. Ainsi `grille.(i)` teste effectivement la primalité de l'entier i.

```
for i = 2 to n do
    if grille.(i) then res:= i::!res
done;
!res;;
```

a) Compléter la fonction `crible`. Quel est le nombre d'entiers premiers inférieurs à 100 ?

b) Si on remplace 100 par 200, dans la définition de la variable globale `n`, comment faut-il modifier la fonction `eratosthene ()` ?

2° Évidemment, ça n'est pas très pratique et l'on désire disposer d'une fonction `eratosthene n` qui retourne la liste des entiers premiers entre 2 et `n` pour tout `n`.

a) Modifier `eratosthene` en faisant de `n` et du vecteur `grille` des variables locales à cette fonction.

b) Proposer une nouvelle version d'`eratosthene`, récursive cette fois. On pourra écrire une fonction `intervalle n m` qui retourne la liste des entiers successifs de `n` à `m`.

Exercice I.15. Les carrés emboîtés.

Retrouver le programme récursif Caml qui a créé la figure I.8.

Éléments de correction des exercices

Exercice I.1. La fonction `th`.

Il suffit de définir une variable locale contenant la valeur de `exp 2x`. Le programme se présente ainsi :

```
let th x =
  let exp2 = exp (2. *. x) in
    (exp2 -. 1.)/. (exp2 +. 1.);;
```

Ce n'est pas un calcul très efficace par rapport à un calcul de série, mais il faut bien commencer ! Attention à ne pas oublier les `.` après les opérateurs arithmétiques sur les flottants !

Exercice I.2. Multiplication dans $\mathbb{N}$.

La programmation récursive suit la définition :

```
let rec Mult = fun
   | n 0 -> 0
   | n p -> n + Mult n (p-1);;
```

On verra dans le chapitre II comment prouver cette fonction sur $\mathbb{N}^2$. Signalons que l'on aurait pu définir une fonction sur des couples (voir la section sur la curryfication) mais que cela nous aurait empêché de rendre l'opération infixe :

```
#Mult 3 4;;
- : int = 12
##infix "Mult";;
#3 Mult 4;;
- : int = 12
```

Exercice I.3. Ordre lexicographique.

1° On propose simplement comme test strict :

```
let inferieur_lexico (x1,y1) (x2,y2) =
          (x1<x2) || ((x1=x2) && (y1<=y2));;
```

2° C'est la structure de liste que nous allons considérer pour représenter des uplets. On va tester les deux éléments en tête de liste et éventuellement les queues récursivement. Si les listes ne sont pas de même longueur (où si l'une des deux est vide) on retourne un message d'erreur.

```
let rec inferieur_lexico l1 l2 = match (l1,l2) with
   | [], _ -> failwith "Comparaison impossible"
   | _, [] -> failwith "Comparaison impossible"
   | (a::r,b::s) -> (a<b) || ((a=b) && (inferieur_lexico r s));;
```

Le connecteur paresseux `&&` permet d'éviter un parcours complet des listes.

Exercice I.4. La fonction `miroir`.

1° On utilise une boucle `while` qui accumule jusqu'à épuisement de la liste initiale :

```
let miroir l =
    let accumulateur = ref [] in
    let lref = ref l in
    while !lref <> [] do
       accumulateur := (hd !lref)::(!accumulateur);
       lref:= tl (!lref)
    done;
    !accumulateur;;
```

2° Cette version utilise une concaténation d'une liste réduite à un seul élément en fin de la liste renvoyée ce qui nécessite un parcours complet de cette dernière. Si les deux algorithmes précédents étaient linéaires (en nombre de `::`), celui-ci devient inutilement quadratique (cf. page 150).

Exercice I.5. Palindrome.

1° On convertit le nombre en une chaîne de caractères que l'on lit caractères après caractères. Le nombre renversé est formé au fur et à mesure :

```
let renverse n =
   if n<10 then n
   else let s = string_of_int n and res= ref "" in
        for i=(string_length s) -1 downto 0 do
           res:=(!res)^(char_for_read s.[i])
        done;
        int_of_string(!res);;
```

2° La fonction palindrome est alors immédiate :

```
let palindrome n = (n = (renverse n));;
```

Exercice I.6. Rendez la monnaie.

À chaque étape de la récursion, on compare la somme restant à payer avec la liste des billets. Cela donne :

```
let rec change s liste_billets = match (s,liste_billets) with
    | (0, _) -> []
    | (_,[]) -> failwith "Je ne peux pas faire de change"
    | (s, n::r) -> if s >= n
                   then n::(change (s-n) (n::liste_billets))
                   else change s r;;
```

Exercice I.7. Fonction de Dirac.

On peut proposer :

```
let dirac x = diracx where diracx y = if (x=y) then 1. else 0.;;
```

On constate la facilité avec laquelle on écrit des fonctionnelles en Caml. Voyons-la à l'œuvre :

```
#let dirac_3 = dirac 3.;;
dirac_3 : float -> float = <fun>
#dirac_3 (0.);;
- : float = 0.0
#dirac_3 (5.);;
- : float = 0.0
#dirac_3 (3.);;
- : float = 1.0
```

Exercice I.8. Binomiaux.

1° On peut proposer une version itérative de `produit` :

```
let produit n m =
    let res = ref n in
       for i=n+1 to m do
         res := !res*i
       done;
    !res;;
```

en notant que celle-ci suppose $n \geqslant m$. Pour diminuer le nombre de multiplications on peut selon le cas calculer $\binom{n}{p}$ ou $\binom{n}{n-p}$, ce qui donne :

```
let binomial n p =
    if (n-p>p)
    then  (produit (n-p+1) n)/ (fact p)
    else  (produit (p+1) n)  / (fact (n-p));;
```

2° On peut utiliser $\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$:

```
let rec binomial n = function
  | 0 -> 1
  | p -> n*(binomial (n-1) (p-1))/p;;
```

et éventuellement là aussi ajouter le test $n - p > p$:

```
let binomial n p =
   if p>n then 0
   else binomial_aux n (min p (n-p))
          where rec binomial_aux n = fun
         | 0  -> 1
         | p -> n*(binomial_aux (n-1) (p-1))/p;;
```

On pourrait envisager d'utiliser la formule du triangle de Pascal mais il faut faire attention à ne pas faire de calculs superflus (cf. l'implémentation de la suite de Fibonacci). Attention également aux débordements arithmétiques dès que l'on manipule des factorielles ou des binomiaux.

Le procédé serait complètement différent si l'on cherchait à calculer une ligne entière du (voire tout le) triangle de Pascal.

Exercice I.9. Marche aléatoire.

Le choix peut se faire entre une programmation récursive (en prenant soin de vérifier la terminaison) et une programmation impérative. Dans ce cas, pour la première question où le nombre d'itérations est connu, il faut utiliser une boucle `for` alors que pour la deuxième, où l'arrêt est conditionné par un test, c'est une boucle `while` qui convient. Cela donne :

```
(* marche aléatoire linéaire *)
let hasard = random__int;;

let deplace position = match (hasard 2) with
   | 0 -> decr position; !position
   | 1 -> incr position; !position;;

let marche_aléatoire_axe n =
   let position = ref 0 in
   let retour_origine = ref 0 in
   let position_droite = ref 0 in
     for i=1 to n do
       position := deplace position;
       if !position > !position_droite
       then position_droite := !position;
       if !position = 0
       then incr retour_origine
     done;
(!position, !position_droite, !retour_origine);;

(* marche aléatoire planaire *)
let deplace (x,y) = match (hasard 4) with
   | 0 -> decr x; (x,y)
   | 1 -> incr x; (x,y)
   | 2 -> decr y; (x,y)
   | 3 -> incr y; (x,y);;

let non_origine (position_x,position_y) =
   (!position_x <> 0) || (!position_y <> 0);;

let marche_aléatoire_plane n =
(* n doit être supérieur ou égal à 1 *)
   let position = ref (ref 0, ref 0) in
   let nombre_pas = ref 0 in
(* il faut exécuter la boucle qui suit une fois *)
   position := deplace !position;
   incr nombre_pas;
   while (non_origine !position) && (!nombre_pas <= n)  do
    position := deplace !position;
    incr nombre_pas
   done;
```

```
    if (not (non_origine !position))
    then begin
         print_string "Retour à l'origine au bout de ";
         print_int !nombre_pas; print_string " pas."
         end
    else begin
         print_string "Pas de retour à l'origine au bout de ";
         print_int n; print_string " pas."
         end;;
```

Exercice I.10. Composition itérée.

On peut proposer :

```
let rec itere f = fun
   | 0 -> (function x -> x)
   | 1 -> f
   | n -> f  (itere f (n-1));;
```

Exercice I.11. Jeu de cartes.

Il suffit d'écrire :

```
type cartes = Sept | Huit | Neuf | Dix | Valet | Dame | Roi | As;;

let rec valeur_main = function
   | [] -> 0
   | Valet::r -> 1 + valeur_main r
   | Dame::r  -> 2 + valeur_main r
   | Roi::r   -> 3 + valeur_main r
   | f::r     -> valeur_main r;;
```

Exercice I.12. Drapeau Hollandais.

On peut proposer :

```
let rec une_passe = function
    | []   -> [],false
    | (Blanc::Rouge::r)
          -> let l,b = une_passe r in Rouge::Blanc::l,true
    | (Bleu::Rouge::r)
          -> let l,b = une_passe r in Rouge::Bleu::l,true
    | (Bleu::Blanc::r)
          -> let l,b = une_passe r in Blanc::Bleu::l,true
    | x::r -> let l,b = une_passe r in x::l,b;;

let rec drapeau_hollandais l =
    let l',modifiée = une_passe l in
    if modifiée then drapeau_hollandais l' else l;;
```

Exercice I.13. Le Fizzbuzz.

On suit les étapes données par l'énoncé :

```
let rec charlist_of_string s =
         let result= ref [] in
                for i=((string_length s)-1) downto 0 do
                      result := (nth_char s i)::(!result)
                done;
         !result;;

let contient  c s = mem c (charlist_of_string s);;

let fizzbuzz_aux n =
      if (n mod 35) = 0 then 35
      else if (n mod 5)=0 then 5
              else if (n mod 7)=0
                      or (contient `7` (string_of_int n))
                   then 7 else 0;;

let fizzbuzz n = match (fizzbuzz_aux n) with
        | 35 -> print_string "fizzbuzz"
        | 5  -> print_string "buzz"
        | 7  -> print_string "fizz"
        | _  -> print_int n;;

let list_fizzbuzz n =
        for i = 1 to n do print_newline(); fizzbuzz i done;;
```

Exercice I.14. Crible d'Ératosthène.

1° a) On barre les multiples de p dans le vecteur `v` :

```
let crible n v p =
  for i= p to (n/p) do
    v.(p*i) <- false
  done;;
```

On fait commencer le crible avec `i=p*p` car les multiples de `p` précédents ont déjà été barrés. Ceci évite de nombreux calculs inutiles. L'exécution fournit alors :

```
#list_length (eratosthene ());;
- : int = 25
```

Il y a 25 nombres premiers inférieurs à 100.

b) Si on remplace 100 par 200, dans la définition de `n`, il faut, bien sûr, ajouter :

```
crible n grille 11;
crible n grille 13;
```

dans la fonction `eratosthene ()`.

2° a) Il faut appeler la fonction `crible` pour tout nombre premier inférieur à la racine carrée de `n` ; soit :

```
let eratosthene  n =
  let grille =  make_vect (n+1) true in
     crible n grille 2;
     let i = ref 3 in
     while (!i*(!i) <= n) do
       if grille.(!i) then crible n grille !i; i:= !i+2
     done;
   let res = ref [] in
     for i = 2 to n do
       if grille.(i) then res := i::!res
     done;
   !res;;
```

On enlève les multiples de 2 puis de 2 en 2 on récupère les premiers qui induisent un `crible`.

b) On peut proposer différentes implémentations pour `intervalle`. Voici une première version récursive :

```
 let rec intervalle n m =
   if m < n then []
   else n :: (intervalle (n+1) m);;
```

Voici une version impérative :

```
let intervalle n m =
    let accu= ref [] in
        for i =m downto n do
            accu := i :: !accu
        done;
    !accu ;;
```

Et voici enfin une version plus savante, à l'aide du récurseur du système T de Gödel :

```
let rec recurseur z f = function
 | 0 -> z
 | n -> f (n-1) (recurseur z f (n-1));;
let intervalle n m =
    recurseur [] (fun k reste -> (m-k)::reste) (m-n+1);;
```

Il suffit alors d'appliquer la fonction `eratosthene` à la liste des entiers compris entre 2 et `n`. On utilise pour cela une fonction auxiliaire `eratosthene_aux` dont les arguments sont la liste des entiers premiers déjà déterminés et le reste de la grille à tester.

```
let eratosthene n =
    eratosthene_aux [] (intervalle 2 n)
    where rec eratosthene_aux liste_premiers = function
      | [] -> liste_premiers
      | p::reste -> eratosthene_aux (p::liste_premiers)
                                    (crible p reste);;
```

La fonction crible filtrant à présent une liste a été modifiée :

```
let rec crible p  = function
   | [] -> []
   | n::r -> let reste = crible p r in
             if n mod p =0 then reste else n::reste;;
```

Exercice I.15. Carrés emboîtés.

Pour fixer les idées, numérotons les carrés du plus grand au plus petit (il y en a 12). Le but est de créer les deux premiers carrés emboîtés et d'itérer l'opération avec une dimension divisée par 4. Pour cela, on commence par se placer au coin en bas à gauche du carré numéro 3 (ce sont les deux premiers `lineto`). Là, la récursivité dessine pour nous les carrés 3 à 12 et revient en ce point. On termine alors les carrés 1 et 2 en revenant au point initial (ce qui assure une récursion correcte) : ce sont les 8 autres `lineto`.

```
#open "graphics";;

open_graph "260x260";;

(*  niveau d'imbrication *)
let n=6;;

let rec dessin x y l n =
if n>0 then  begin
lineto (int_of_float (x +. l /. 2.)) (int_of_float y);
lineto (int_of_float (x +. l /. 4.)) (int_of_float (y +. l /. 4.));
dessin (x +. l /. 4.)  (y +. l /. 4.) (l /. 2.) (n-1);
lineto (int_of_float x) (int_of_float (y +. l /. 2.));
lineto (int_of_float (x +. l /. 2.)) (int_of_float (y +. l));
lineto (int_of_float (x +. l)) (int_of_float (y +. l /. 2.));
lineto (int_of_float (x +. l /. 2.)) (int_of_float y);
lineto (int_of_float (x +. l)) (int_of_float y);
lineto (int_of_float (x +. l)) (int_of_float (y +. l));
lineto (int_of_float x) (int_of_float (y +. l));
lineto (int_of_float x) (int_of_float y)
end;;

let carres_imbriques () =
   moveto 10 10;
   dessin 10. 10. 240. n;;
```

II

Éléments de programmation

La résolution d'un problème est souvent décrite en combinant les notions de répétition, choix, calculs etc. La programmation *itérative* (encore appelée *impérative*) consiste à implémenter une telle résolution sous la forme d'une suite d'instructions, comprises par l'ordinateur comme autant d'ordres à exécuter séquentiellement. La *récursion*, présente dans un langage comme Caml, est une alternative permettant de programmer une solution en l'autorisant à se citer elle-même (dans un cadre bien précis). C'est une idée incontournable dans les langages fonctionnels et en informatique (moderne) en général. Nous allons nous efforcer, dans ce chapitre, de familiariser le lecteur avec cette notion qui de magique deviendra, nous l'espérons, naturelle...

1 Quelques conseils pour une « bonne » programmation...

> [le précepte est] de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre.
> Descartes — *Discours de la méthode*

Avant d'entrer dans le vif du sujet, il convient de rappeler au programmeur (ou aux programmeurs dans le cadre d'un travail collectif) quelques recommandations de bon sens. Proposons une liste d'étapes essentielles au bon développement d'un travail informatique, ce que nous nommerons, par commodité (et bien que cela ne soit pas complètement fidèle) : la « programmation d'une fonction » donnée. Plus les intervenants sont nombreux, plus les étapes que nous allons brièvement décrire se révèlent indispensables.

1° **Spécification.** C'est l'étape préliminaire. Elle consiste à définir exactement ce que l'on *veut* faire c'est-à-dire à préciser totalement la fonction à programmer

a priori. Ainsi, on doit détailler son domaine d'application (ses arguments) et faire de même avec le ou les résultats calculés. Il convient de formaliser tout ceci au mieux avec au besoin des équations, des quantificateurs, des intervalles etc. Ces informations constituent *la spécification* de la fonction recherchée ; c'est en quelque sorte un *cahier des charges* exhaustif[1]. Notons que la description de ce que la fonction *ne fait pas* peut être aussi importante que la description de ce qu'elle doit *savoir faire*.

2° Validation. C'est une étape difficile[2] ! Elle consiste à s'assurer que la spécification obtenue au 1° correspond bien à l'objectif recherché. Plus précisément, cela se traduit principalement par la vérification de la *complétude*[3] et de la *cohérence* (on dit aussi *consistance*) de la spécification *i.e.* on s'assure que toutes les situations ont été envisagées et qu'il ne subsiste pas de contradiction.

3° Structuration des données. Il faut en effet décider de la représentation informatique des objets manipulés par la fonction (arguments, résultats, objets intermédiaires...) : ce point fait l'objet du chapitre III.

4° Programmation modulaire. Il y a deux approches : celle dite *top-down* (de « haut en bas ») où l'on préfère commencer par programmer la fonction principale en y incluant au fur et à mesure des fonctions auxiliaires qui sont détaillées par la suite (et dont on suppose simplement qu'elles vérifient une spécification précise) ; et celle dite *bottom-up* (de « bas en haut ») où l'on agrège des fonctions élémentaires pour arriver à ses fins[4]. Il faut avoir une ligne de conduite très précise pour réussir de cette seconde façon ; c'est pourquoi nous conseillerons au débutant la première approche.

De façon générale, le principe consistant à partager un problème en plusieurs sous-problèmes que l'on traite successivement ne date pas d'hier... Il ne s'applique d'ailleurs pas qu'à la programmation !

5° Preuve de correction. Le programme étant écrit, à l'instar d'une démonstration mathématique, on le *prouve*. Quand le programme est simple, on peut le faire à l'aide des méthodes que nous allons évoquer. Lorsque le programme découle directement des spécifications, cette étape devient redondante (et ce sera assez souvent le cas en Caml, langage issu des travaux sur la preuve automatique de programmes).

6° Tests. Si tout ce qui précède a été fait correctement, cette étape est superflue... Néanmoins il est en général prudent de s'assurer sur des exemples que l'on a atteint son but, la difficulté étant de n'oublier aucun cas particulier. En général, la phase de tests s'effectue de « bas en haut » : on teste d'abord les fonctions élémentaires pour finir avec la fonction principale.

1. La spécification d'un problème informatique est souvent traduite logiquement et fonctionnellement. C'est un avantage dans le cadre d'une approche fonctionnelle de la programmation.
2. Tester la complétude d'une spécification est en fait indécidable.
3. On rappellera le sens logique de l'adjectif *complet* au chapitre VI.
4. C'est le cas de certains langages de programmation (Maple par exemple) où seules quelques fonctions de base sont programmées dans un langage extérieur, le reste étant défini dans le nouveau langage lui-même et s'enrichissant de versions en versions du logiciel.

7° **Présentation.** Il convient de bien présenter (indentation etc.) et de commenter sans retenue le programme. Ce n'est pas seulement une préoccupation esthétique ! Ce souci d'explication du programme permet sa compréhension par un public non restreint à son seul auteur... et assure sa diffusion et son évolution éventuelle en le rendant plus facile à maintenir.

En espérant que tout ceci ne reste pas lettre morte, nous allons à présent passer à ce qui constitue l'essentiel de ce chapitre : la programmation récursive et la programmation impérative.

2 Récursivité

Naturam expellas furca,
tamen usque recurret.
Horace — *Ep. 1,10,24*

2.1 Introduction

Le mot « récursion » vient du latin *recursare* qui signifie *courir en arrière*, *revenir*. L'idée de réapparition, de retour est donc étymologiquement liée à la récursivité. Ainsi, nous appellerons *fonction récursive* toute fonction qui, dans sa définition, fait appel à son propre identificateur. C'est le cas, par exemple, de la fonction élémentaire `somme`[5] vue dans le chapitre I, qui calcule la somme des entiers compris entre `1` et `n` (son argument) :

```
let rec somme n = if n = 0 then 0 else n + somme (n-1);;
```

La récursivité n'a pas attendu la programmation fonctionnelle pour se manifester... Le monde des arts nous le montre à travers les variantes de la *mise en abîme* : cela va de la pièce de théâtre dans la pièce de théâtre (comme *L'illusion comique* de Corneille), du film dans le film (*La nuit américaine* de Truffaut) au dessin dans le dessin (comme les graphismes d'Escher, la sixième case de la page 35 de *Tintin au Congo*... ou plus communément les couvercles des boîtes de fromage *Vache qui rit*), etc. Bref s'il ne s'agit pas d'un phénomène banal, il est tout du moins commun.

Revenons à Caml et, à l'aide de la fonction `trace` que nous avons déjà évoquée dans la section du chapitre I consacrée au débogage (page 53), observons précisément le fonctionnement de la récursion.

Voyons, par exemple, comment est calculé `somme 3`. Si l'on s'en tient à la définition de `somme`, Caml est conduit à effectuer la suite des opérations suivantes :

```
somme 3
3 + somme 2
3 + (2 + somme 1)
3 + (2 + (1 + somme 0))
3 + (2 + (1 + 0))
3 + (2 + 1)
3 + 3
6
```

5. L'utilisation d'un filtrage ne modifierait en rien `somme` ni ce que nous allons en dire.

L'appel de `somme 3` nécessite celui de `somme 2`, qui lui-même nécessite `somme 1`, qui appelle à son tour jusqu'à aboutir à `somme 0`. Ce dernier appel est le premier qui retourne un résultat (en l'occurrence 0). Nous appellerons cette première phase une *descente* dans la récursion. Cela permet alors de calculer `somme 1`, puis `somme 2` et enfin `somme 3` qui vaut 6 : nous appellerons cette deuxième phase une *remontée* dans la récursion.

C'est exactement ce que `trace` nous montre :

```
#trace "somme";;
La fonction somme est dorénavant tracée.
- : unit = ()
#somme 3;;
somme <-- 3
somme <-- 2
somme <-- 1
somme <-- 0
somme --> 0
somme --> 1
somme --> 3
somme --> 6
- : int = 6
```

Les appels à `somme` sont repérés par le signe `<--` et les valeurs retournées sont repérées par le signe `-->`.

Cet exemple est assez simple car, dans ce cas d'école, les phases de descente et de remontée sont uniques. Nous allons à présent envisager un problème un peu moins élémentaire (et fondamental en informatique) celui du *tri* : on considère une liste d'entiers quelconque qu'il faut réordonner suivant l'ordre naturel de N noté $\leqslant$.

Plus précisément, on va s'intéresser[6] à la version basique de l'algorithme de « tri rapide » (ou *quicksort* en anglais) dont le principe est le suivant :

- si la liste d'entiers est vide ou réduite à un élément, l'algorithme laisse la liste inchangée,
- si la liste, notée `l`, est de longueur supérieure ou égale à deux, on considère son élément de tête, appelons-le `e`, on sépare la queue de `l` en deux sous-listes : celle des éléments plus grands que `e` et celle des éléments plus petits. Puis on trie, suivant ce même principe, chacune de ces deux sous-listes et enfin on assemble le tout pour former la liste finale triée.

C'est une illustration de la stratégie, dite *diviser pour régner*, dont nous vanterons les mérites au chapitre IV.

Par exemple, si la liste initiale est la suivante :

```
[18;3;10;25;9;3;11;13;23;8]
```

la liste des éléments inférieurs à l'entier en tête `18` est `[3;10;9;3;11;13;8]`; celle des entiers supérieurs est `[25;23]`. Le tri de ces deux listes fournit `[3;3;8;9;10;11;13]`

6. Nous présenterons d'autres algorithmes de tri dans cet ouvrage, notamment dans les exercices II.2 à II.5.

et `[23;25]`. Le résultat est alors obtenu en mettant bout à bout la première liste triée suivie de `18` suivi de la deuxième liste triée soit `[3;3;8;9;10;11;13;18;23;25]`.

Voici une implémentation en Caml de cet algorithme[7] :

```
let rec tri_rapide  = function
  |  []    -> []
  |  [e]   -> [e]
  |  e::r  -> let l1,l2 = partition  e r in
              (tri_rapide l1)@(e::(tri_rapide l2));;
```

où `partition` est la fonction qui réalise la séparation en deux sous-listes en fonction de l'élément de tête (nous suivons une mise au point « de haut en bas ») :

```
#let rec partition (e:int)  = function
  | [] -> ([], [])
  | d::r -> let l1,l2 = partition e r in
            if (d < e) then (d::l1,l2) else (l1,d::l2);;
partition : int -> int list -> int list * int list = <fun>
#partition 18 [3;10;25;9;3;11;13;23;8];;
- : int list * int list = [3; 10; 9; 3; 11; 13; 8], [25; 23]
```

Observons le comportement de `tri_rapide`[8] sur la liste `[3;4;2;1;5]` :

```
#trace "tri_rapide";;
La fonction tri_rapide est dorénavant tracée.
- : unit = ()
#tri_rapide [3;4;2;1;5];;
tri_rapide <-- [3; 4; 2; 1; 5]
tri_rapide <-- [4; 5]
tri_rapide <-- [5]
tri_rapide --> [5]
tri_rapide <-- []
tri_rapide --> []
tri_rapide --> [4; 5]
tri_rapide <-- [2; 1]
tri_rapide <-- []
tri_rapide --> []
tri_rapide <-- [1]
tri_rapide --> [1]
tri_rapide --> [1; 2]
tri_rapide --> [1; 2; 3; 4; 5]
- : int list = [1; 2; 3; 4; 5]
```

Cette fois-ci (et c'est ce que nous voulions illustrer) il se produit plusieurs descentes et plusieurs remontées dans la récursion. Le tri de `[3;4;2;1;5]` conduit à la

7. La traitement à part des listes de longueur 1 (`[e] ->[e]`) est simplement là pour accélérer le tri et aider à la compréhension de l'exécution pas à pas. Ce cas peut très bien être supprimé.
8. On a imposé le type entier à l'aide de `(e:int)` dans `partition` donc dans `tri_rapide` afin de pouvoir « tracer » la fonction (voir chapitre I, page 55).

partition par rapport à `3` suivante : `[2;1]` et `[4;5]`. La fonction `tri_rapide` s'engage alors dans la branche `tri_rapide l2` (qui est d'abord [9] traitée par Caml) soit `tri_rapide [4;5]` qui lui-même conduit à `tri_rapide [5]` qui retourne un résultat et ainsi de suite. On peut visualiser cette suite de montées et descentes dans la récursion à l'aide du tableau suivant :

Listes à trier	Construction du résultat		
[3;4;2;1;5]	tri [3;4;2;1;5]	=	(tri [2;1])-3-(tri [4;5])
[4;5] [2;1]	tri [4;5]	=	(tri [ ])-4-(tri [5])
[5] [ ] [2;1]	tri [5]	=	[5]
[ ] [2;1]	tri [ ]	=	[ ]
[2;1]	tri [4;5]	=	[4;5]
[2;1]	tri [2;1]	=	tri [1]-2-(tri [ ])
[ ] [1]	tri [ ]	=	[ ]
[1]	tri [1]	=	[1]
-	tri [2;1]	=	[1;2]
-	tri [3;4;2;1;5]	=	[1;2;3;4;5]

(Le traitement des listes à trier se fait de gauche à droite.)

On constate que, même sur une liste courte, il est malaisé de suivre les appels récursifs de la fonction `tri_rapide` (essayez donc de suivre le traitement de la liste `[18;3;10;25;9;3;11;13;23;8]`). Comme, en outre, il est très facile d'écrire une fonction récursive qui boucle indéfiniment, nous allons à présent exposer un cadre théorique permettant, dans des cas simples, de prouver la *terminaison* d'une fonction récursive et sa *correction* (c'est-à-dire son adéquation avec sa spécification). Nous appliquerons bien sûr tout ceci, entre autres, à nos deux exemples `somme` et `tri_rapide`.

2.2 Problèmes de terminaison

Entendons-nous bien : nous dirons pour une fonction f et un argument x donné que $f(x)$ *termine* si ce calcul nécessite un *nombre fini* d'opérations élémentaires (quelles qu'elles soient). Pour l'instant, peu nous importe la grandeur de ce nombre et ses conséquences : les aspects temporels seront considérés au chapitre IV.

Revenons à la récursivité. Le corps d'une fonction récursive comporte en général deux parties :

- le traitement des *cas de base* ($n = 0$ pour `somme`, ou le cas des listes de longueur inférieure ou égale à un pour `tri_rapide`),
- la partie récursive (qui comprend un ou plusieurs appels récursifs).

Prouver la terminaison d'une fonction récursive consiste à montrer que la fonction parvient toujours au traitement des cas de base en un nombre fini d'appels. Ce nombre fini n'étant pas en général facile à préciser, nous allons préférer des arguments mathématiques *ad hoc* (relations d'ordre, éléments minimaux...) pour parvenir à nos fins.

9. C'est l'instruction la plus profonde de l'expression `(tri_rapide l1)@(e::(tri_rapide l2))`.

2.2.1 Si peu de mathématiques...

Nous allons formaliser la notion intuitive suivante. Si l'on dispose d'une quantité liée à la fonction récursive et à ses arguments telle que :

- cette quantité évolue dans un ensemble ordonné qui ne comporte pas de suite infinie strictement décroissante,
- à chaque appel de la fonction, la quantité en question décroît strictement,
- la fonction récursive achève son calcul pour les cas de base,

alors la fonction termine bien dans tous les cas.

En reprenant l'exemple de la fonction `somme`, on remarque que l'argument `n` de la fonction qui est un élément de $\mathbb{N}$ décroît strictement à chaque appel récursif et finit donc par atteindre 0, qui est un cas de base de la fonction.

Pour notre second exemple, on choisit comme quantité liée à `(tri_rapide l)` la longueur de la liste `l`. La définition de `tri_rapide` ne faisant appel qu'à des listes de longueur strictement inférieure, cette quantité décroît strictement pour atteindre 1 ou 0 qui correspondent aux longueurs des deux cas de base prévus.

Donnons quelques définitions et propriétés afin de préciser les choses sur le plan mathématique. Dans la suite, E va désigner un ensemble non vide, $\preccurlyeq$ une relation d'ordre (partiel ou total) définie sur E et $\prec$ la relation d'ordre stricte associée ($\leqslant$ et $<$ continuent de désigner l'ordre usuel (et sa version stricte) sur les entiers, les réels...).

Définition II.1. $(E, \preccurlyeq)$ *est dit* bien fondé[10] *s'il n'existe pas de suite infinie d'éléments de* E *strictement décroissante.*

Par exemple, $(\mathbb{N}, \leqslant)$ est bien fondé alors que $(\mathbb{Z}, \leqslant)$ ne l'est pas.

On a alors immédiatement la propriété (que nous utiliserons souvent par la suite sans le dire...) :

Proposition II.2. *Si* $(E, \preccurlyeq)$ *est bien fondé et si* F *est une partie non vide de* E *alors* $(F, \preccurlyeq)$ *est bien fondé.*

Comme exemple moins élémentaire, on peut considérer un ordre très utile pour la suite : l'ordre *lexicographique*. Sur $\mathbb{N}^2$, il est défini par :

$$(a, b) \preccurlyeq (c, d) \qquad \text{si} \qquad (a \leqslant c) \quad \text{ou} \quad \text{si} \quad (a = c \quad \text{et} \quad b \leqslant d)$$

Il s'agit d'un ordre total qui peut être facilement généralisé[11] à $\mathbb{N}^p$. On a :

Proposition II.3. $\mathbb{N}^2$ *muni de l'ordre lexicographique est bien fondé.*

10. On trouve aussi la terminologie de *bien ordonné* pour un ensemble muni d'un ordre *total* bien fondé. Les preuves qui vont suivre ne dépendant pas du caractère partiel ou total de l'ordre, nous ne ferons pas de distinction et utiliserons le terme « bien fondé » dans les deux cas.
11. À l'aide de l'ordre usuel sur les lettres de l'alphabet, cette construction fournit, sur les mots, l'ordre du dictionnaire.

Preuve. Supposons l'existence d'une suite infinie $(a_n, b_n)_{n\in\mathbb{N}}$ de couples d'entiers, strictement décroissante. Tous les couples étant strictement inférieurs à (a_0, b_0), il existe un rang n_0 à partir duquel $a_n < a_0$ (en effet, l'ensemble des couples de $\mathbb{N}^2$ inférieurs strictement à (a_0, b_0) ayant la même première coordonnée que (a_0, b_0) est de cardinal fini b_0). De même, il existe un rang $n_1 \geqslant n_0$ à partir duquel $a_n < a_{n_0} < a_0$. En itérant cette construction, on obtient une suite infinie d'entiers, la suite des premières coordonnées a_{n_i}, qui est strictement décroissante dans $\mathbb{N}$ ce qui constitue une contradiction. □

On dispose dans un ensemble bien fondé de la propriété caractéristique suivante :

Proposition II.4. *Un ensemble $(E, \preccurlyeq)$ est bien fondé si et seulement si toute partie non vide de E admet un élément minimal.*

Rappelons, avant de prouver cette proposition, que, si l'ordre est total, la notion d'élément minimal coïncide avec celle de minimum. Dans le cas d'un ordre partiel, un élément m d'une partie non vide $A \subset E$ est dit élément minimal de A si

$$\forall\, a \in A,\quad a \preccurlyeq m \implies m = a$$

Par exemple, dans $\mathbb{N} \setminus \{0, 1\}$ muni de la relation d'ordre partiel $\mid$ (« divise »), les éléments minimaux sont les nombres premiers.

Le lecteur vérifiera de même que, si $F = \{a, b, c\}$ est un ensemble contenant trois éléments, les éléments minimaux de $E = \mathcal{P}(F) \setminus \{\emptyset\}$, l'ensemble des parties non vides de F, muni de la relation d'ordre partiel $\subset$ (« est inclus dans »), sont les trois singletons $\{a\}$, $\{b\}$ et $\{c\}$.

Revenons à la preuve de la proposition :

Preuve. Soit E un ensemble non vide dans lequel toute partie non vide admet un élément minimal. Supposons l'existence de $(u_n)_{n\in\mathbb{N}}$ une suite infinie d'éléments de E telle que $\forall\ n \in \mathbb{N}$, $u_{n+1} \prec u_n$. Considérons alors le support de la suite $S = \{u_n, n \in \mathbb{N}\}$ et m un élément minimal de S. On dispose donc de $n_0 \in \mathbb{N}$ tel que $m = u_{n_0}$. L'élément u_{n_0+1} apporte la contradiction : c'est un élément de S qui est, par hypothèse, strictement plus petit que m. Une telle suite $(u_n)_{n\in\mathbb{N}}$ n'existe donc pas et E est bien fondé.

Réciproquement, soit $(E, \preccurlyeq)$ un ensemble bien fondé et soit $A \subset E$ une partie non vide de E sans éléments minimaux. On dispose donc de la négation de la définition donnée précédemment, soit :

$$\forall\, m \in A,\quad \exists\, a \in A \quad \text{tel que } a \prec m$$

La partie A est nécessairement de cardinal infini et on construit alors aisément (par récurrence) une suite d'éléments de A strictement décroissante infinie, ce qui est contraire à l'hypothèse. □

On peut illustrer notre proposition dans $\mathbb{N}^2$ muni de l'ordre lexicographique. Le minimum d'une partie A non vide est le couple (n_0, m_0) défini par :

$$n_0 = \min\{n \in \mathbb{N} \mid \exists\, m \in \mathbb{N}, (n, m) \in A\} \text{ et } m_0 = \min\{m \in \mathbb{N} \mid (n_0, m) \in A\}.$$

Disposant à présent de la notion d'ordre bien fondé et de quelques exemples, nous allons pouvoir démontrer le théorème de terminaison :

Théorème II.5 (Terminaison). *Soit f une fonction récursive, $\mathcal{A}$ l'ensemble de ses arguments, φ une application de $\mathcal{A}$ dans un ensemble bien fondé $(E, \preccurlyeq)$ et $\mathcal{B}$ la partie de $\mathcal{A}$ constituée des éléments dont l'image par φ est minimale dans $\varphi < \mathcal{A} >$ (les cas de base).*
Si $f(b)$ termine pour tout $b \in \mathcal{B}$ et si dans la définition de $f(x)$ n'apparaissent, en nombre fini, que des appels à $f(y)$ tels que $\varphi(y) \prec \varphi(x)$ alors $f(x)$ termine pour tout x dans $\mathcal{A}$.

Preuve. Faisons une démonstration par l'absurde.

Supposons non vide la partie $\mathcal{X}$ constituée des éléments de $\mathcal{A}$ sur lesquels f ne termine pas. Soit $F = \varphi < \mathcal{X} >$. En tant que partie non vide de E, on dispose de m_0 un élément minimal de F. Notons x_0 un antécédent par φ de m_0 dans $\mathcal{X}$.

Comme f termine sur tous les arguments dont l'image par φ est un élément minimal de $\varphi < \mathcal{A} >$, m_0 n'est pas minimal dans $\varphi < \mathcal{A} >$. Ainsi, l'ensemble $M = \{m \in \varphi < \mathcal{A} > \mid m \prec m_0\}$ est non vide. D'autre part, puisque m_0 est minimal dans F, on a $M \subset \varphi < \mathcal{A} > \setminus F$ et f termine pour tous les arguments dont l'image par φ appartient à M.

Finalement, le calcul de $f(x_0)$, faisant directement appel à un nombre fini de $f(y)$ avec $\varphi(y) \prec \varphi(x_0) = m_0$, termine : ce qui est en contradiction avec $x_0 \in \mathcal{X}$. □

2.2.2 Quelques exemples

L'exemple de `somme` est un exemple particulièrement simple puisque $\mathcal{A} = E = \mathbb{N}$ et $\varphi = id_{\mathbb{N}}$. Le théorème précédent nous permet donc de rédiger :

- `somme 0` termine,
- le seul appel à `somme` intervenant dans sa définition a pour argument `n-1` $<$ `n`, donc, $\forall\ n \in \mathbb{N}$, `somme n` termine.

Pour `tri_rapide`, on peut écrire : l'ensemble $\mathcal{A}$ des arguments de la fonction est l'ensemble des listes d'entiers. On prend comme application φ l'application *longueur-de-la-liste* qui est à valeurs dans $\mathbb{N}$. Le cas de la liste vide étant envisagé et les deux appels de `tri_rapide` dans la définition de la fonction n'étant faits que sur des listes de longueur strictement inférieure à celle de la liste initiale, la fonction `tri_rapide` termine bien sur toute liste d'entiers.

Voyons à présent d'autres exemples.

Le calcul des coefficients binomiaux peut se faire à l'aide de la formule du triangle de Pascal :

$$\forall\ (n,p) \in \mathbb{N}^2, \quad 1 \leqslant p < n, \qquad \binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$$

Ce qui donne la fonction récursive suivante définie sur $\mathbb{N}^2$:

```
let rec binome = function
    | (n,0) -> 1
    | (n,p) -> if (p>n)
               then 0
               else binome ((n-1),p) + binome ((n-1),(p-1));;
```

Cette méthode est assez inefficace car des calculs de binomiaux identiques sont effectués de multiples fois (se reporter à l'exercice I.8). Cependant cet algorithme, appelé avec des entiers *naturels*, termine bien. En effet :

- les arguments de `binome` varient dans $\mathcal{A} = \mathbb{N}^2$. On peut choisir à nouveau $E = \mathcal{A}$ et $\preccurlyeq$ l'ordre lexicographique,
- les deux appels à `binome` intervenant dans sa définition le sont avec les couples $(n-1, p) \prec (n, p)$ et $(n-1, p-1) \prec (n, p)$,
- l'élément minimum $(0, 0)$ de $(\mathbb{N}^2, \preccurlyeq)$ est traité dans le cas `(n,0)`.

Attention, l'instruction `(n,0) -> 1` est indispensable. Elle garantit que p (et donc n) reste bien dans $\mathbb{N}$. Si l'on remplaçait ce motif par le simple `(0,0)` alors le calcul de `binome (0,1)` ne terminerait pas. En effet, on parviendrait à des entiers négatifs et $(\mathbb{Z}^2, \preccurlyeq)$, on le sait, n'est pas bien fondé.

Sur cet exemple, on peut également constater que les choix de E et φ ne sont pas nécessairement uniques. En effet, on pourrait aussi conclure en considérant $E = \mathbb{N}$ et $\varphi(n, p) = n + p$ qui décroît elle aussi strictement à chaque appel.

Voyons à présent le cas d'une fonction chère aux informaticiens : la fonction d'Ackermann [12]. Elle est définie comme suit :

```
let rec ackermann  = fun
  | 0 p -> p+1
  | n 0 -> ackermann (n-1)  1
  | n p -> ackermann (n-1) (ackermann n (p-1));;
```

C'est une fonction à croissance très rapide comme on le montrera dans l'exercice II.7. On peut par exemple calculer :

```
#ackermann 1 1;;
- : int = 3
#ackermann 2 3;;
- : int = 9
#ackermann 3 7;;
- : int = 1021
```

mais le lecteur peut attendre longtemps pour le calcul de `ackermann 4 4` !

Cette fois-ci, nous ne programmons pas une fonction banale et la preuve de terminaison prend tout son intérêt. Nous allons encore utiliser l'ordre lexicographique sur $\mathbb{N}^2$. On sait que le calcul de `ackermann n p` termine si `n=0` (à nouveau les motifs `0 p` et `n 0` garantissent le maintien dans $\mathbb{N}^2$). Sinon il fait directement appel à

12. La fonction d'Ackermann constitue un exemple de fonction récursive « non primitive », cette notion étant fondamentale en théorie de la calculabilité.

(`ackermann (n-1) 1`) et $(n-1,1) \prec (n,p)$ ou à (`ackermann n (p-1)`) et $(n,p-1) \prec (n,p)$ ou à (`ackermann (n-1) X`) et $(n-1,X) \prec (n,p)$ (où `X = ackermann n (p-1)`). D'après le théorème de terminaison, la fonction termine donc bien pour tout couple d'arguments entiers naturels.

En outre, on remarque que, pour cet exemple, les applications à valeurs dans $(\mathbb{N}, \leqslant)$ que sont $\varphi(n,p) = n+p$ ou $\varphi(n,p) = \max(n,p)$... ne permettent pas de conclure.

Le lecteur trouvera une autre occasion de s'exercer en étudiant la terminaison de la fonction de McCarthy proposée dans l'exercice II.8. Nous allons quant à nous terminer ces quelques exemples par un dernier cas d'école. Considérons la fonction dite « fonction de Morris » définie par :

```
let rec morris = fun
    | 0 _ -> 1
    |   _ -> morris (m-1) (morris m n);;
```

Une preuve trop rapide de terminaison de cette fonction conduirait à écrire :

- on considère l'ordre lexicographique sur $\mathbb{N}^2$,
- le calcul de `morris m n` fait appel au calcul de `morris (m-1) X` et $(m-1,X) \prec (m,n)$ pour tout `X`,
- enfin, la fonction termine lorsque son premier argument est nul.

Mais `X = morris m n`! Alors l'exécution en Caml de `morris 1 0` ne termine pas, car ce calcul conduit à l'exécution de `morris 0 (morris 1 0)`, donc à nouveau au calcul de `morris 1 0` et ainsi de suite. En effet, comme dans la plupart des langages de programmation, en Caml, les arguments sont évalués avant d'être passés à la fonction (on parle « d'appel par valeurs »). Nous l'avions déjà signalé au début de ce chapitre lors de l'exécution pas à pas des fonctions : Caml évalue en priorité les expressions les plus profondes d'une expression donnée.

Bref il faut être attentif !

2.2.3 Indécidabilité de la terminaison

Hélas, trois fois hélas, on ne sait pas toujours conclure quant à la terminaison d'une fonction récursive. Le lecteur en quête de célébrité peut se lancer dans la preuve de terminaison de la fonction :

```
let rec collatz n =
   if  n<= 0 then invalid_arg "n>0 SVP"
   else  (n>1) && if (impair n)
                  then collatz (3*n + 1)
                  else collatz (n quo 2);;
```

où `impair n` teste la parité de son argument entier :

```
impair n = ((n mod 2) = 1).
```

Il s'agit de la célèbre conjecture (c'en est une) de Collatz (encore appelée « de Syracuse ») qui énonce que, lorsque l'on part d'un entier u_0 supérieur à 1, la suite $(u_k)_{k\in\mathbb{N}}$

définie par :

$$u_{k+1} = \begin{cases} u_k/2, & \text{si } u_k \text{ est un entier pair,} \\ 3u_k + 1 & \text{sinon,} \end{cases}$$

finit toujours par atteindre la valeur 1.

Considérons par exemple les termes issus de $u_0 = 13$, soit :

$$u_1 = 40,\ u_2 = 20,\ u_3 = 10,\ u_4 = 5,\ u_5 = 16,\ u_6 = 8,\ u_7 = 4,\ u_8 = 2,\ u_9 = 1.$$

(testez vous-même d'autres valeurs de départ).

Le théorème II.5 ne nous est ici d'aucune utilité (puisque $3n+1 > n$). Pire encore, on pourrait croire que, tel le grand théorème de Fermat, toutes ces « fonctions récursives - conjectures » attendent leurs A. Wiles[13] : il n'en est rien. Il a été démontré au début du XX$^{\text{e}}$ siècle[14] qu'il était impossible d'espérer trouver un programme universel sachant tester si une fonction récursive quelconque termine son calcul.

Imaginons en effet un monde merveilleux dans lequel existerait une fonction `termine` de type `'a -> 'b -> bool` qui répondrait `true` si la fonction récursive passée en argument (de type général `'a -> 'b`) termine et `false` dans le cas contraire. Par exemple, `termine(somme)` renverrait `true` et `termine(sans_fin)` renverrait `false` avec `sans_fin` la fonction définie par :

```
let rec sans_fin x = if x then sans_fin x else false;;
```

(considérer l'exécution de `sans_fin true`).

Hélas l'essai de `termine` sur la fonction `test` suivante montre que nous ne vivons pas dans ce monde :

```
let rec test () = if (termine test) then (test ()) else true;;
```

Si `test ()` ne termine pas alors `termine test` retourne `false` et la valeur de `test ()` est `true` et donc `test ()` termine. Et si `test()` termine alors `termine test` retourne `true` et `test ()` est à nouveau évalué et la fonction boucle. Il y a donc une contradiction et une telle fonction `termine` ne peut, et c'est bien dommage, exister.

2.3 Preuves sur les programmes récursifs

Nous allons à présent généraliser ce qui précède avec pour objectif la preuve de *correction* d'une fonction récursive donnée. On complétera ces considérations au chapitre III par les preuves sur les fonctions définies sur des ensembles inductifs.

Soit f une fonction récursive et $\mathfrak{p}_f$ un prédicat[15] sur les valeurs calculées par f. On va énoncer un théorème permettant de prouver :

$$\forall\ x \in \mathcal{A}, \quad \mathfrak{p}_f(x)$$

($\mathcal{A}$ désigne à nouveau l'ensemble des arguments de f).

13. qui a finalement démontré en 1994 le grand théorème que Fermat avait conjecturé au XVII$^{\text{e}}$ siècle.
14. C'est l'œuvre des logiciens Gödel et Turing dans les années 1930.
15. C'est-à-dire une application à valeurs dans {vrai, faux}.

Si le prédicat $\mathfrak{p}_f$ considéré est « le calcul de $f(x)$ termine», on retrouve le problème de la preuve de terminaison. En ce qui concerne les preuves de correction, le prédicat considéré sera en général[16] : « $f(x)$ calcule bien ce que je veux ». Par exemple, « `tri_rapide l` effectue bien le tri de la liste d'entiers `l` » ou « `somme n` calcule bien la somme des entiers de `0` à `n` » ...

Succinctement, prouver un prédicat $\mathfrak{p}_f$ sur une fonction récursive f va consister à démontrer que celui-ci est vérifié à chaque étape de la récursion *i.e.* qu'il est vrai sur les cas de base, et que s'il l'est sur les appels récursifs, il l'est aussi sur l'argument principal.

C'est une démarche tout à fait analogue à celle du théorème II.5 de terminaison. Montrons tout d'abord une proposition préliminaire concernant les ensembles bien fondés :

Proposition II.6. *Soit $(E, \preccurlyeq)$ un ensemble bien fondé et $\mathfrak{p}$ un prédicat sur les éléments de E. Si la propriété suivante*[17] *est vérifiée :*

$$(1) \qquad \forall\, x \in E, \quad \Big([\forall\, y \prec x,\ \mathfrak{p}(y)] \Longrightarrow \mathfrak{p}(x)\Big)$$

alors $\forall\, x \in E,\ \mathfrak{p}(x)$.

Comme la propriété $\forall\, z \in \emptyset, P(z)$ est toujours vraie quel que soit le prédicat P, la propriété (1) contraint $\mathfrak{p}$ à être vérifiée sur tous les éléments minimaux de E (dont l'ensemble des minorants est vide). La proposition précédente est donc équivalente à la propriété redondante mais « plus pratique » suivante :

Ah ! La logique et l'ensemble vide...

Proposition II.7 (Induction sur un ensemble bien fondé).

Soit $(E, \preccurlyeq)$ un ensemble bien fondé et $\mathfrak{p}$ un prédicat sur les éléments de E. Si $\mathfrak{p}$ est vérifiée sur tous les éléments minimaux de E et si

$$\forall\, x \in E, \quad \Big([\forall\, y \prec x,\ \mathfrak{p}(y)] \Longrightarrow \mathfrak{p}(x)\Big)$$

alors $\forall\, x \in E,\ \mathfrak{p}(x)$.

Preuve. Effectuons une démonstration par l'absurde.

Supposons non vide la partie F constituée des éléments de E sur lesquels $\mathfrak{p}$ est faux. On dispose alors d'un élément minimal m_0 dans F. Donc $\forall\, y \prec m_0$, $y \notin F$ et donc $\mathfrak{p}(y)$ est vrai. De (1) on déduit $\mathfrak{p}(m_0)$: ce qui constitue la contradiction. □

Si l'on applique la proposition précédente avec l'ensemble bien fondé $(\mathbb{N}, \leqslant)$ on retrouve que si $\mathfrak{p}(0)$ est vrai et si $\forall\, n \in \mathbb{N},\ [\forall k \leqslant n,\ \mathfrak{p}(k)] \Longrightarrow \mathfrak{p}(n+1)$ alors $\forall\, n \in \mathbb{N},\ \mathfrak{p}(n)$: soit la preuve par récurrence (généralisée[18]) classique sur les entiers. Ainsi, dans la rédaction des preuves qui vont suivre et qui utilisent l'induction sur

16. mais ce prédicat pourrait très bien être tout autre.
17. Classiquement, nous entendons « $\mathfrak{p}(z)$ » comme « $\mathfrak{p}(z)$ est vrai ».
18. À la différence de la preuve par récurrence « élémentaire » où l'on suppose seulement $\mathfrak{p}(n)$ pour montrer $\mathfrak{p}(n+1)$.

un ensemble bien fondé, le lecteur trouvera l'expression « par hypothèse d'induction » à la place de l'habituelle « par hypothèse de récurrence ». Nous retrouverons encore la récurrence au chapitre III (page 148) lorsque nous parlerons de l'induction structurelle.

Nous pouvons à présent énoncer le théorème de preuve d'un prédicat sur une fonction récursive :

Théorème II.8 (Correction).

Soit f une fonction récursive, $\mathcal{A}$ l'ensemble de ses arguments, φ une application de $\mathcal{A}$ dans $(E, \preccurlyeq)$ un ensemble bien fondé, $\mathcal{B}$ la partie de $\mathcal{A}$ constituée des éléments dont l'image par φ est minimale dans $\varphi < \mathcal{A} >$ (les cas de base) et $\mathfrak{p}_f$ un prédicat sur les valeurs calculées par f.
Si $\mathfrak{p}_f(b)$ est vérifié pour tout b de $\mathcal{B}$ et si, en notant de nouveau (y) les arguments d'appel de f intervenant dans la définition de la fonction et x l'argument d'appel initial, on a :

- $\forall\ y,\ \varphi(y) \prec \varphi(x)$,
- $[\forall\ y,\ \mathfrak{p}_f(y)] \implies \mathfrak{p}_f(x)$,

alors $\mathfrak{p}_f$ est vérifié sur toutes les valeurs calculées par f.

Preuve. Il suffit d'appliquer la proposition précédente au prédicat `q` défini sur l'ensemble bien fondé $\varphi < \mathcal{A} >$ par : $\mathfrak{q}(e)$ est vrai si $\forall\ x \in \mathcal{A}$ tel que $\varphi(x) = e$ on a $\mathfrak{p}_f(x)$ vrai. □

On notera la similitude des théorèmes de terminaison et de correction (substituez « $f(x)$ termine » par « $\mathfrak{p}_f(x)$ »). De plus, *comme le test de $\mathfrak{p}_f(x)$ n'a de sens que si le calcul de $f(x)$ termine*, on mène en général de front les preuves de terminaison et de correction[19].

Considérons, une fois encore, nos exemples favoris :

- Définissons, pour `n`$\in \mathbb{N}$ le prédicat $\mathfrak{p}_s$(`n`) = « la fonction `somme n` termine et sa valeur est $\sum_{i=0}^{n} i$ ». On a $\mathfrak{p}_s$(`0`) car `somme 0` termine et vaut 0. De plus, si l'on suppose que `somme (n-1)` termine et vaut $\sum_{i=0}^{n-1} i$ alors `somme n` = `n + somme (n-1)` termine car `n-1` < `n` et vaut $\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$, on a donc bien $\mathfrak{p}_s$(`n`). Finalement, on a : $\forall\ n \in \mathbb{N}$, $\mathfrak{p}_s$(`n`).

  Sur cet exemple, on pourrait très bien considérer d'autres prédicats plus ou moins simples à montrer comme par exemple « le calcul `somme n` termine et sa valeur est $\frac{n(n+1)}{2}$ » ou « le calcul `somme n` termine et sa valeur n'est jamais un nombre premier sauf pour `n=2` »...

  (considérer les propriétés démontrées sur `ackermann` dans l'exercice II.7).

19. Souvent même, les prédicats considérés ne mentionnent pas explicitement la terminaison du calcul et ne font que la sous-entendre (car elle reste, bien sûr, indispensable).

– Définissons pour `l` une liste d'entiers, le prédicat $\mathfrak{p}_t(\texttt{l})$ = « le calcul `tri_rapide l` termine et vaut la liste triée des éléments de `l` ». On a $\mathfrak{p}_t(\texttt{[]})$ car `tri_rapide []` termine et vaut la liste vide. Par définition, `tri_rapide l` vaut `(tri_rapide l1) @(e::(tri_rapide l2))`. Les listes `l1` et `l2` sont de longueurs strictement inférieures à la longueur de la liste initiale `l` et si, par hypothèse d'induction, `tri_rapide l1` vaut la liste triée des éléments de `l1` et `tri_rapide l2` vaut la liste triée des éléments de `l2` alors `(tri_rapide l1)@(e::(tri_rapide l2))` est bien la liste triée des éléments de la liste `l` puisque l'élément `e` se trouve justement à sa place. On a donc bien montré par induction la propriété pour toute liste d'entiers.

  Les listes étant des objets de structure inductive (voir chapitre III), nous commenterons à nouveau cette preuve en 2.2 page 148.

– Nous laissons au lecteur le soin de montrer de façon analogue le prédicat défini sur $\mathbb{N}^2$ muni de l'ordre lexicographique par : $\mathfrak{p}_b(\texttt{n,p})$ = « `binome n p` termine et calcule $\binom{n}{p}$ ».

  Se reporter à l'exercice II.9 pour un autre exemple d'application du théorème de correction[20]. Est-ce la peine d'ajouter que prouver un prédicat quelconque sur une fonction récursive n'est pas toujours facile voire faisable ? À nouveau considérez la fonction `collatz` et montrez qu'elle vaut toujours `false`...

2.4 Programmer c'est prouver

Dans la section précédente nous avons écrit `somme` et `tri_rapide` puis nous avons démontré que ces fonctions étaient correctes c'est-à-dire en adéquation avec leur spécification. Est-il légitime de séparer ainsi les choses ? La réponse n'est pas si simple. Nous allons considérer à ce propos l'exemple (célébrissime) du jeu des tours de Hanoï. Rappelons brièvement le principe du jeu.

On dispose de n rondelles de tailles différentes et de trois piquets A, B et C sur lesquels les rondelles peuvent être enfilées. Au départ les n rondelles sont sur le piquet A, dans l'ordre des diamètres décroissants, le plus large en bas (voir figure II.1).

Figure II.1 : *Position initiale.*

20. où la fonction récursive f n'est pas définie partout, où $\mathcal{A}$ et $f < \mathcal{A} >$ ne sont pas du même type et où φ n'est pas surjective...

Figure II.2 : *Position finale.*

Le jeu consiste à déplacer les rondelles sur le piquet C, rangées comme au départ (voir figure II.2) en déplaçant à chaque coup une rondelle au sommet d'une pile sur un autre piquet et en ne posant jamais une rondelle sur une rondelle plus petite.

Par exemple dans le cas $n = 3$, on peut procéder comme suit :

- la rondelle au sommet de A est placée sur C,
- la rondelle au sommet de A est placée sur B,
- la rondelle au sommet de C est placée sur B,
- la rondelle au sommet de A est placée sur C,
- la rondelle au sommet de B est placée sur A,
- la rondelle au sommet de B est placée sur C,
- la rondelle au sommet de A est placée sur C.

Comment résoudre le problème avec n rondelles?

Nous proposons la solution (récursive) suivante :

- Pour déplacer une pile de 0 rondelle, il n'y a rien à faire,
- Pour déplacer une pile de $n > 0$ rondelles de A vers C :

il suffit de déplacer $n - 1$ rondelles de A vers B :

puis de déplacer la rondelle (la plus grande du jeu) de A vers C :

et enfin de déplacer la pile de $n-1$ rondelles de B vers C :

La règle de décroissance du diamètre des rondelles enfilées sur un piquet n'est pas violée lorsque la pile est déplacée puisque les rondelles de la pile de $n-1$ éléments peuvent toujours être enfilées sur un piquet en bas duquel se trouve une rondelle plus grande.

Ainsi, si l'on sait résoudre le problème avec $n-1$ rondelles, on sait le résoudre avec n ; comme le cas $n=0$ est trivial, le problème est résolu !

Le programme Caml traduit directement cette méthode :

```
let rec hanoi n départ intermédiaire arrivée =
    if n>0 then begin
       hanoi (n-1) départ arrivée intermédiaire;
       deplace départ arrivée;
       hanoi (n-1) intermédiaire départ arrivée
    end;;
```

avec `deplace` la fonction :

```
let deplace piquet1 piquet2 =
    print_newline ();
    print_string "la rondelle au sommet de ";
    print_string piquet1;
    print_string " est placée sur ";
    print_string piquet2;;
```

La spécification de `hanoi p x y z` est la suivante : cette fonction déplace conformément aux règles du jeu `p` rondelles du piquet `x` au piquet `z` en se servant du piquet intermédiaire, le piquet `y`.

Et c'est la récursivité qui fait tout le travail !

```
#hanoi 3 "A" "B" "C";;

la rondelle au sommet de A est placée sur C
la rondelle au sommet de A est placée sur B
la rondelle au sommet de C est placée sur B
la rondelle au sommet de A est placée sur C
la rondelle au sommet de B est placée sur A
la rondelle au sommet de B est placée sur C
la rondelle au sommet de A est placée sur C
- : unit = ()
```

Évidemment on pourrait à présent prouver la correction de la fonction `hanoi`. Mais cela serait vraiment redondant avec le programme puisque celui-ci est exactement la programmation de notre solution. Nous tenions à présenter ce problème car, d'une part, il constitue un « incontournable » de l'étudiant en informatique et, d'autre part, il est l'illustration que, dans ce langage issu des travaux sur les preuves de programme qu'est Caml, « preuve de programme » et « programme » peuvent parfois ne faire qu'un, pour la joie du programmeur !

En guise de complément, s'il est vrai qu'ici c'est le compilateur qui fait tout le travail, nous vous renvoyons au chapitre IV pages 213 et 218 pour étudier la complexité (et l'optimalité) de l'algorithme.

Signalons que nous devons ce célèbre problème à l'imagination du mathématicien français E. Lucas. Ce dernier a de plus assorti en 1883 son exercice de la légende suivante : selon des bonzes de Hanoï qui passent leur vie à résoudre le problème avec $n = 64$ la fin du monde surviendra à la fin du jeu. En considérant qu'ils ont commencé le 1er Janvier 1883 et qu'ils déplacent une rondelle par seconde (!), l'exercice IV.5 vous donnera le temps qu'il nous reste d'ici à la fin du monde...

2.5 Récursivité croisée

Avant d'en terminer (provisoirement !) avec la récursivité, nous voulions en présenter un complément. La récursivité en Caml ne s'arrête pas, en effet, au traitement des fonctions récursives f dont la définition est du type $f(x) = \Phi(f, x)$ comme nous l'avons vu jusqu'à présent. Elle permet une plus grande expressivité encore au travers de la *récursivité croisée* ou *récursivité mutuelle*.

Les fonctions mutuellement récursives sont définies simultanément en une seule instruction à l'aide des mots-clés :

`let rec` *définition de* f_1 `and` *définition de* f_2 `and` *etc...* `;;`

où chacun des identificateurs f_j peut apparaître dans toutes les définitions des autres fonctions.

Nous allons illustrer cela par la programmation d'un couple de suites récurrentes croisées. Il s'agit de la moyenne arithmético-géométrique qui est sans doute déjà

familière au lecteur :

$$\begin{cases} a_0 = a \\ a_n = \dfrac{a_{n-1} + g_{n-1}}{2} \end{cases} \qquad \text{et} \qquad \begin{cases} g_0 = g \\ g_n = \sqrt{a_{n-1} \cdot g_{n-1}} \end{cases}$$

avec $a > g > 0$.

La programmation en Caml suit exactement cette définition :

```
let rec a = function
    | 0 -> a0 (* valeur initiale *)
    | n -> (a(n-1) +. g(n-1))/. 2.
    and g = function
    | 0 -> g0 (* valeur initale *)
    | n -> sqrt( a(n-1) *. g(n-1));;
```

On sait que ces deux suites forment un couple de suites adjacentes. Elles interviennent d'ailleurs dans le calcul de π à l'aide de la formule de Brent-Salamin :

$$\pi \approx \pi_n = \frac{4{a_{n+1}}^2}{1 - \displaystyle\sum_{j=1}^{n} 2^{j+1}({a_j}^2 - {g_j}^2)} \qquad \text{avec} \quad a_0 = 1 \text{ et } g_0 = \frac{1}{\sqrt{2}}$$

La suite $(\pi_n)_{n\in\mathbb{N}}$ des approximations de π converge quadratiquement ; π_4 réalise déjà une approximation de π à $5 \cdot 10^{-41}$ près !

Les fonctions mutuellement récursives sont plus que la traduction exacte de leur équivalent mathématique ; elles sont essentielles en analyse syntaxique, problème dont on reparlera au chapitre V (regarder attentivement le code de la fonction `Gram1` du convertisseur *syntaxe concrète* $\longmapsto$ *syntaxe abstraite* de la page 43). Nous proposons dans l'exercice II.16 une première solution au problème du batelier à l'aide de fonctions mutuellement récursives.

Les preuves (en particulier la terminaison) sur ces fonctions mutuellement récursives sont évidemment plus complexes ; l'exemple qui suit est « réjouissant » de non terminaison :

```
#let rec tic() = print_string "tic "; tac()
     and tac() = print_string "tac "; tic();;
#tic ();;
```

3 Programmes impératifs, éléments de preuve

Changeons à présent de style et passons à la programmation impérative.

Nous avons présenté dans le chapitre I page 16 différents aspects de la programmation impérative comme l'usage des références, des boucles conditionnelles (`while`) et inconditionnelles (`for`) etc. Nous allons ici aborder la démonstration de programmes écrits en style impératif. Nous ne ferons que « survoler » ce sujet car la preuve de programme est, comme nous l'avons déjà dit, une question difficile[21].

21. Le lecteur intéressé par la question trouvera son bonheur dans les bonnes bibliothèques d'informatique au rayon *génie logiciel*.

Nous allons restreindre notre propos aux *preuves de boucles*, les boucles jouant en quelque sorte pour la programmation impérative le rôle des appels récursifs d'une fonction en programmation récursive. La méthode consiste à déterminer une propriété, dite *invariant de boucle*, liant les éléments variables d'une boucle, et qui reste vraie quel que soit le nombre de passages dans celle-ci. Détaillons cela sur des exemples.

Reprenons tout d'abord le cas élémentaire de la fonction `somme`, cette fois dans sa version impérative :

```
#let somme n =
    let resultat = ref 0 in
       for i = 1 to n do
           resultat := !resultat + i
       done;
       !resultat;;
```

et considérons l'invariant de boucle noté $\mathfrak{p}(i)$: « après i itérations de la boucle la valeur de résultat, notée $resultat_i$, est $\sum_{j=0}^{i} j$ ».

On vérifie sans peine que $\mathfrak{p}(i)$ est vrai pour tout $i \geqslant 0$. On a en effet $\mathfrak{p}(0)$ car la valeur d'initialisation de `resultat` est bien 0. Si l'on suppose qu'après k itérations de la boucle $resultat_k = \sum_{j=0}^{k} j$ la $(k+1)$-ième itération de la boucle effectue $resulat_{k+1} := resultat_k + k + 1 = \sum_{j=0}^{k} j + k + 1 = \sum_{j=0}^{k+1} j$. □

Ceci nous fournit en fait la preuve de correction de la fonction `somme`.

Prenons un exemple un (tout petit) peu plus conséquent [22], avec cette fois une boucle `while`. Que calcule la fonction `f` suivante ?

```
let f n =
  let x = ref n and y = ref n in
     while not(!y=0) do
       x:= !x + 2;
       y:= !y -1
     done;
  !x;;
```

À chaque itération de la boucle, `y` décroît strictement. On peut donc penser que la boucle va s'exécuter `n` fois si `n` est positif ou bien ne pas terminer dans le cas contraire. La valeur initiale de `x` est `n` ; comme on ajoute 2 à `x` à chaque passage dans la boucle, on pressent que la valeur finale de `f` sur $\mathbb{N}$ est `3n`.

Pour montrer ce résultat, on va noter x_i et y_i les contenus des références `x` et `y` après i itérations de la boucle et définir l'invariant de boucle $\mathfrak{p}(i)$ suivant : « $x_i + 2y_i = 3n$ », n étant le paramètre d'appel entier naturel de `f`.

22. Exemple tiré de *Mathématiques discrètes*. M. Marchand, De Boeck ed.

Initialement, $x_0 = n$ et $y_0 = n$, on a donc bien $x_0 + 2y_0 = 3n$. Si l'on suppose, $\mathfrak{p}(i)$ alors après i+1 itérations, on a $x_{i+1} = x_i + 2$ et $y_{i+1} = y_i - 1$ d'où $x_{i+1} + 2y_{i+1} = x_i + 2 + 2(y_i - 1) = x_i + y_i = 3n$, soit $\mathfrak{p}(i+1)$. □

À la sortie de la boucle `!y = 0` donc `!x = 3n` et `f` renvoie bien le triple de son argument entier naturel.

On trouve d'autres exemples dans les exercices II.12 et II.13.

Certes, avec cette méthode nous sommes loin d'être capables de prouver un programme quelconque écrit en style impératif. Il suffit de songer à un programme utilisant des branchements conditionnels (en cascade), des boucles avec un niveau d'imbrication conséquent... pour comprendre que l'invariant de boucle a ses limites. Toutefois cette méthode a le mérite d'être simple et d'exister ! Les méthodes plus réalistes imposent de définir des spécifications substantielles des programmes considérés et de s'assurer que l'on couvre tous les cas envisagés dans le programme. Comme nous l'avons déjà dit : nous renvoyons le lecteur à un traité de *génie logiciel*.

4 Itération contre récursivité ?

l'itération est humaine,
la récursion divine...
Peter L. Deutsch —

Nous avons déjà vu, notamment avec `fact` et `somme`, des versions récursives et impératives d'une même fonction :

```
let rec somme n =
    if n = 0
    then 0
    else n + somme (n-1);;

let somme n =
    let r = ref 0 in
      for i = 1 to n do
        r := !r + i
      done;
    !r;;
```

```
let rec fact n =
    if n = 1
    then 0
    else n * fact (n-1);;

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

Ces programmes se comprennent aisément et sont d'efficacité comparable (en nombre d'opérations arithmétiques effectuées).

Nous verrons dans le chapitre III comment programmer de manière impérative une procédure récursive à l'aide d'une pile. Traiter la théorie générale de la *dérécursification* (c'est-à-dire une conversion *programme récursif* $\longmapsto$ *programme impératif*) nous amènerait beaucoup trop loin et sortirait du cadre de cet ouvrage. Nous allons nous contenter ici de proposer une dérécursification de certaines fonctions simples, puis de faire quelques comparaisons entre les différentes versions obtenues.

Les fonctions récursives simples que nous allons tout d'abord considérer sont les fonctions *récursives terminales* qui sont des fonctions récursives dans la définition

desquelles n'apparaît qu'une seule fois l'identificateur de la fonction et ce en dernière instruction. Par exemple, les fonctions `somme`, `fact` sont récursives terminales.

Donnons un autre exemple, la fonction qui à n associe a^n :

```
let rec puissance a  = function
    | 0 -> 1
    | n -> a * puissance a (n-1);;
```

En reprenant les notations des théorèmes de terminaison et de correction, on peut mettre les fonctions récursives terminales sous la forme générale :

$$\begin{cases} \forall\, x \in \mathcal{B}, & f(x) = \textit{une valeur}, & \text{(les cas de base)} \\ \text{sinon} & f(x) = \Phi(x, f(\sigma(x))) & \text{avec } \varphi(\sigma(x)) \prec \varphi(x). \end{cases}$$

où Φ et σ sont des fonctions quelconques.

Par exemple

- pour `somme`, $\mathcal{B} = \{0\}$, $\sigma(x) = x - 1$ et $\Phi(X, Y) = X + Y$,
- pour `fact`, $\mathcal{B} = \{0\}$, $\sigma(x) = x - 1$ et $\Phi(X, Y) = X * Y$,
- pour `puissance a`, $\mathcal{B} = \{0\}$, $\sigma(x) = x - 1$ et $\Phi(X, Y) = a * Y$.

Le lecteur aura sans doute remarqué que pour les fonctions récursives terminales les phases de descente et de remontée dans la récursion sont uniques. Leur dérécursification peut se faire simplement sous la forme :

```
let f x =
    if x∈ B
    then valeur
    else let y=ref x and resultat= ref init in
       while (!y∉ B) do
         resultat := Φ(!y, !resultat);
         y:=σ(y)
       done;
    !resultat;;
```

où `init` désigne une valeur initiale dépendant du contexte.

On montrerait facilement l'équivalence entre les deux versions d'une même fonction avec les méthodes de preuve vues précédemment. Voici ce que l'on obtient pour `puissance` :

```
let puissance a n =
    if n =0 then 1
    else let y=ref n and resultat = ref a in
           while (!y > 1) do
             resultat := a * !resultat;
             y:= !y-1
           done;
           !resultat;;
```

Comme nous l'avons déjà fait pour `somme` et `fact`, lorsque l'on constate, comme c'est le cas ici pour `puissance`, que le nombre d'itérations de la boucle `while` est connu, on peut employer une boucle `for`, ce qui donne finalement :

```
let puissance a n =
    let r = ref 1 in
       for i=1 to n do
         r:=a*!r
       done;
    !r;;
```

Les boucles se substituent donc à la récursivité terminale. Signalons que nous avions vu la transformation inverse dans le chapitre I page 24.

Dans les cas des fonctions `fact`, `sommme` et `puissance`, en matière d'efficacité, nous n'avons aucune raison de préférer l'un ou l'autre des styles de programmation, car il est facile de vérifier que le nombre d'opérations arithmétiques élémentaires effectuées par les deux implémentations est proportionnel à l'argument n. On dit que la complexité de ces algorithmes est linéaire en n (voir chapitre IV).

Nous ne pouvons pas ne pas signaler, en ce qui concerne `puissance`, qu'il existe une méthode plus efficace basée sur le paradigme *diviser pour régner*. L'algorithme d'*exponentiation binaire*, puisque c'est de lui qu'il s'agit, part du constat simple suivant : si, par exemple, on doit calculer a^{11} alors, plutôt que de faire 10 multiplications $a \times a \times \ldots \times a$, on peut avantageusement calculer $b = a \times a$, puis $c = b \times b$ puis $d = c \times c$ et finalement faire le produit $a \times b \times d$.

Ceci est bien plus « économique » en nombre de multiplications. Qu'avons-nous fait ? Si on écrit 11 en base 2, soit $\overline{b_3 b_2 b_1 b_0} = \overline{1011}$, on a calculé les trois premiers termes de la suite $(a^{2^k})_{k \geqslant 1}$, puis multiplié entre eux les carrés correspondants aux b_i égaux à 1.

Dans le cas général, on effectue donc $\lfloor \log_2 n \rfloor$ multiplications pour le calcul des termes de la suite des carrés et « le nombre de 1 intervenant dans l'écriture binaire de n » pour les produits finaux. Ce deuxième entier est majoré par $\lceil \log_2 n \rceil$ et donc, de linéaire en n, le nombre total de multiplications de notre calcul de puissance est devenu logarithmique.

Le programme Caml, version itérative, correspondant est le suivant :

```
let puissance a n =
    let r= ref 1 and base2 = ref n and carré = ref a in
    while (!base2 >0) do
          if (!base2 mod 2 = 1)
          then r := !r * !carré;
          base2 := !base2 / 2;
          carré := !carré * !carré
    done;
    !r;;
```

La référence `carré` représente successivement a, a^2, a^4... et les chiffres de l'écriture binaire de n sont testés avec `base2`.

Le lecteur ne manquera pas de prouver[23] ce programme.

Le programme récursif se comprend sans commentaire :

```
let rec puissance a  = function
    | 0 -> 1
    | n -> let r= puissance a (n / 2) in
                  if (n mod 2)=0 then r*r else a*r*r;;
```

et la complexité est inchangée.

Les fonctions qui ne sont pas récursives terminales ne se dérécursifient pas toujours facilement et le résultat peut souvent paraître abscons de prime abord. Considérons l'exemple classique de la suite de Fibonacci. Nous avons déjà vu son implémentation récursive :

```
let rec fib = function
     | 0 -> 0
     | 1 -> 1
     | n ->  fib (n-2) +  fib (n-1);;
```

Mais celle-ci présente un grave défaut : celui de faire énormément de calculs redondants. Cet algorithme naïf n'a en effet « pas de mémoire » ; il recommence le calcul de `fib (n-2)` sans se souvenir qu'il a déjà obtenu ce résultat dans `fib (n-1)`. Et tout cela à chaque appel récursif...

```
#fib 5;;
fib <-- 5
fib <-- 4
fib <-- 3
fib <-- 2
fib <-- 1
fib --> 1
fib <-- 0
fib --> 0
fib --> 1
fib <-- 1
fib --> 1
fib --> 2
fib <-- 2
fib <-- 1
fib --> 1
fib <-- 0
fib --> 0
fib --> 1
fib --> 3
fib <-- 3
fib <-- 2
fib <-- 1
```

23. Considérez l'invariant de boucle $r_i \cdot (carré_i)^{base2_i} = a^n$.

```
fib --> 1
fib <-- 0
fib --> 0
fib --> 1
fib <-- 1
fib --> 1
fib --> 2
fib --> 5
- : int = 5
```

Estimons le nombre, noté $recfib$, d'appels à la fonction `fib` nécessaires au calcul de `fib` n. On a $recfib(0) = recfib(1) = 1$ et $recfib(n) = 1+recfib(n-1)+recfib(n-2)$. Donc $recfib(n) >$ `fib` n qui lui-même vaut :

$$\frac{\beta^n - \alpha^n}{\sqrt{5}} \qquad \text{avec} \quad \alpha = \frac{1-\sqrt{5}}{2} \quad \text{et} \quad \beta = \frac{1+\sqrt{5}}{2}$$

Le nombre d'appels croît donc au moins[24] exponentiellement en fonction de n.

Alors que, bien évidemment, la méthode naturelle pour calculer `fib n` consiste à calculer `fib 2`, puis `fib 3`... jusqu'à aboutir à `fib n`, ce qui nécessite un nombre d'appels linéaire. On peut donc proposer une implémentation impérative de cette méthode plus efficace mais sans doute moins lisible :

```
let fib n =
 let pred = ref 0 and succ = ref 1 and aux =ref 0 in
    for i = 2 to n do
      aux  := !pred;
      pred := !succ;
      succ := !succ + !aux
    done;
  if n<= 1 then n else !succ;;
```

Les références `pred` et `succ` représentent deux termes consécutifs de la suite de Fibonacci. On notera l'usage d'une variable supplémentaire `aux` pour permettre le calcul des termes suivants.

Le lecteur ne résistera pas à la tentation de prouver ce programme. Hum, hum !

Doit-on déduire de ce qui précède que :

– récursivité = clarté = inefficacité

alors que

– itération = obscur = efficacité ?

Pas du tout.

24. Bien sûr, on pourrait donner une forme close de $recfib(n)$.

Tout d'abord, il est facile à l'aide d'une fonction auxiliaire (voir l'exercice I.4), de proposer une version linéaire récursive de la suite de Fibonacci :

```
let rec fib_aux = fun
     | 0 -> (1,0)
     | n ->  f (fib_aux (n-1))
where f(x,y) = (x+y,x);;

let fib n =  snd (fib_aux n);;
```

En outre, on peut sans trop de peine rendre plus lisible l'algorithme impératif. Pour cela, il suffit de s'inspirer de ce qui précède et de se souvenir qu'une suite récurrente linéaire d'ordre 2 numérique n'est rien d'autre qu'une suite récurrente linéaire d'ordre 1 vectorielle. Il suffit donc de passer en dimension 2 ! On a en effet :

$$v_{n+1} = A\ v_n \quad \text{avec} \quad v_n = \begin{pmatrix} fib_{n+1} \\ fib_n \end{pmatrix}, \quad A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix},$$

et le vecteur initial $v_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

On en déduit le programme impératif plus lisible :

```
let fib n =
    let f(x,y) = (x+y,x) and  v=ref (1,0) in
       for k=1 to n do
          v:=f(!v)
       done;
    snd !v;;
```

qui peut être transformé à l'aide de `boucle_inconditionnelle` vue en page 24 en :

```
let fib n =
snd(boucle_inconditionnelle n (fun (x,y) -> (x+y,x)) (1,0));;
```

et ces deux programmes sont de complexité linéaire (voir l'exercice II.14).

Mais on peut faire mieux ! Le lecteur un peu futé aura trouvé lui-même que si l'on est capable de calculer une puissance numérique avec une complexité logarithmique, on est capable de faire de même avec une matrice ! On peut donc proposer un programme « logarithmique » pour la suite de Fibonacci utilisant l'exponentiation rapide de la matrice[25] A indifféremment sous forme impérative ou récursive. Par exemple :

```
let mult a b =
    [|
      a.(0)*b.(0)+a.(1)*b.(2);
      a.(0)*b.(1)+a.(1)*b.(3);
      a.(2)*b.(0)+a.(3)*b.(2);
      a.(2)*b.(1)+a.(3)*b.(3);
    |];;
```

25. Pour rester lisible, on a représenté une matrice 2x2 par un vecteur de 4 coefficients.

```
let rec puissance_matrice2x2 m  = function
 | 0 ->  [|1;0;0;1|] (* l'identité *)
 | n ->  let moitié = puissance_matrice2x2 m (n / 2) in
              if (n mod 2)=0
              then (mult moitié moitié)
              else (mult m (mult moitié moitié));;

let puissance_matrice2x2 m n =
    let résultat = ref [|1;0;0;1|] (* l'identité *)
        and base2 = ref n and carré = ref m in
    while (!base2 >0) do
          if (!base2 mod 2 = 1)
          then résultat := mult !résultat  !carré;
          base2 := !base2 / 2;
          carré := mult !carré !carré
    done;
    !résultat;;

let fib n =
   if n<2 then n else
   (puissance_matrice2x2 [|1;1;1;0|] (n-1)).(0);;
(* on prend le coefficient A11 de la matrice *)
```

Nous aurions pu unifier ces différentes fonctions[26] pour un résultat certainement moins clair et moins pédagogique.

En conclusion, c'est la facilité d'écriture qui doit trancher le débat récursion/itération. Signalons que la suite de M. Fibonacci[27] n'a pas régalé que les informaticiens, elle est aussi intervenue dans la démonstration du dixième problème de Hilbert.

26. Ou ôter le caractère vectoriel pour éviter une perte de temps due aux accès, ou déterminer, en la réduisant, la puissance n-ième de A etc.
27. Ou Léonard de Pise (1175?-1240?).

Exercices

Exercice II.1. Le schéma de Horner.

Étant donné un polynôme $P = a_0 + a_1X + \ldots + a_nX^n$ à coefficients réels et un nombre réel x, on veut calculer la valeur $P(x)$. La méthode naïve conduit à calculer séparément les puissances de x : $1, x, \ldots, x^n$ puis à les multiplier par les coefficients a_i correspondants, et à additionner le tout.

Le schéma de Horner consiste à utiliser l'égalité mathématique suivante :

$$a_nx^n + a_{n-1}x^{n-1} + \cdots + a_0 = (((a_nx + a_{n-1})x + a_{n-2})x \cdots)x + a_0.$$

1° On représente un polynôme par la liste de ses coefficients ; proposez un algorithme d'évaluation d'un polynôme P en un réel x en tirant profit du membre droit de l'égalité ci-dessus.

2° Quels sont les avantages de la méthode de Horner par rapport à la méthode naïve ?

Exercice II.2. Le tri par sélection.

C'est sans doute la méthode de tri la plus élémentaire. Elle consiste à trouver le minimum de la liste (qui peut ne pas être unique), à le placer en tête de la liste résultat et à recommencer cette opération sur les éléments restants.

La fonction principale est donc la suivante :

```
let rec tri_selection = function
    | [] -> []
    | l  -> let (m,r) = (minimum_et_reste l) in
                         m::(tri_selection r);;
```

1° Proposer une implémentation de la fonction `minimum_et_reste` qui isole le minimum d'une liste des autres éléments :

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

2° On mesure l'efficacité d'un algorithme de tri en estimant le nombre de comparaisons (tests `a>b`) nécessaires pour produire la liste triée (voir chapitre IV). Donnez la complexité de l'algorithme tri par sélection.

3° On considère à présent que les données initiales ne sont plus rangées sous la forme d'une liste mais dans un vecteur. Programmez à nouveau le tri par sélection sur ce type de données. On pourra écrire les fonctions auxiliaires suivantes :

- `echange i j v` qui échange le contenu des cases `i` et `j` d'un `vecteur v`.
- `minimum a b v` qui détermine le minimum du vecteur `v` entre les indices `a` et `b`.

Exercice II.3. Le tri par insertion.

C'est le tri du joueur de cartes. On considère en quelque sorte que les éléments de la liste à trier sont donnés un par un. Le premier élément constitue à lui tout seul

une liste de longueur 1 triée. On range alors à sa bonne place le deuxième élément pour former une liste de longueur 2 triée et on continue. On insère ainsi les éléments successivement dans des sous-listes triées.

Par exemple, sur la liste initiale `[5;4;8;2;6;7;7;2;9]`, les étapes de tri successives sont :

```
J'insère 9 dans  []
         ce qui donne [9]
J'insère 2 dans  [9]
         ce qui donne [2; 9]
J'insère 7 dans  [2; 9]
         ce qui donne [2; 7; 9]
J'insère 7 dans  [2; 7; 9]
         ce qui donne [2; 7; 7; 9]
J'insère 6 dans  [2; 7; 7; 9]
         ce qui donne [2; 6; 7; 7; 9]
J'insère 2 dans  [2; 6; 7; 7; 9]
         ce qui donne [2; 2; 6; 7; 7; 9]
J'insère 8 dans  [2; 2; 6; 7; 7; 9]
         ce qui donne [2; 2; 6; 7; 7; 8; 9]
J'insère 4 dans  [2; 2; 6; 7; 7; 8; 9]
         ce qui donne [2; 2; 4; 6; 7; 7; 8; 9]
J'insère 5 dans  [2; 2; 4; 6; 7; 7; 8; 9]
         ce qui donne  [2; 2; 4; 5; 6; 7; 7; 8; 9]
```

(la récursion fait que l'on considère la liste par la fin).

1° a) Commencez par écrire la fonction `insere e l` qui insère un élément dans une liste *déjà triée* `l`.

b) En déduire une implémentation récursive du tri par insertion.

2° Pourquoi ne pas abstraire l'ordre sur les objets de la liste à trier ? Modifiez le code précédent pour trier, par exemple, des listes de couples d'entiers suivant l'ordre lexicographique.

3° Comme dans l'exercice précédent, on désire déterminer la complexité de l'algorithme de tri par insertion. Nous allons mener un étude en moyenne c'est-à-dire faire la moyenne des coûts du tri sur toutes les listes possibles (voir chapitre IV) en utilisant le modèle des permutations[28]. On va supposer dans ce modèle que les éléments qui composent la liste de longueur n à trier sont les entiers de 1 à n. En effet, le nombre de comparaisons à effectuer pour un tri ne dépend pas des éléments à trier mais de l'ordre dans lequel ils apparaissent. L'hypothèse précédente n'est donc pas restrictive[29]. À une telle liste l on peut associer bijectivement une permutation σ de $\mathfrak{S}_n$ par $\sigma(i) = j$ si l'entier j se trouve en j-ème position dans l (en considérant cette fois-ci que l'élément de tête est en position 1).

a) Pour une permutation $\sigma \in \mathfrak{S}_n$ (et donc sa liste associée), une inversion est

28. Cette étude s'inspire de l'ouvrage *Algorithmes et Programmation* de R. Cori et J.J. Lévy aux éditions de l'École Polytechnique.
29. À la précision suivante près : les éléments de la liste doivent être tous distincts.

un couple (i, j) tel que $i < j$ et $\sigma(i) > \sigma(j)$. Combien d'inversions comporte la liste `[8;1;5;10;4;2;6;7;9;3]` ?

b) À une permutation $[\sigma(1); \sigma(2); \ldots; \sigma(n)]$ on peut associer la permutation « miroir » $[\sigma(n); \ldots; \sigma(2); \sigma(1)]$. En remarquant que le couple (i, j) est une inversion de la première permutation si et seulement s'il n'est pas une inversion de la seconde permutation, déterminez le nombre moyen d'inversions d'une permutation de $\mathfrak{S}_n$ c'est-à-dire :

$$\frac{\sum_{\sigma \in \mathfrak{S}_n} inv(\sigma)}{\#\mathfrak{S}_n} \qquad \text{où} \quad inv(\sigma) \quad \text{est le nombre d'inversions de } \sigma.$$

c) En déduire la complexité moyenne, en nombre de comparaisons, du tri par insertion.

4° Programmez le tri par insertion sur des vecteurs.

Exercice II.4. Le tri bulle.

Une variante du tri par sélection est le *tri bulle*. Son principe est de parcourir la liste à trier et de transposer toute inversion rencontrée (voir l'exercice précédent). À la fin de cette passe, le minimum de la liste se retrouve en tête (si la suite de transpositions s'est faite, en raison de la récursion, en commençant par la fin). On recommence alors l'opération sur la queue de la liste.

Si l'on se représente une liste comme une « colonne verticale de liquide » (la tête de la liste en haut), les nombres les plus légers (les plus petits) remontent tels des bulles vers le haut en poussant des bulles successives du bas vers le haut : c'est l'origine du nom « tri bulle ».

Dans le détail, les passes successives du tri bulle sont, par exemple sur la liste initiale `l = [5;4;8;2;6;7;7;2;9]` :

```
On effectue une passe sur la liste : [5; 4; 8; 2; 6; 7; 7; 2; 9]
                ce qui donne pour l : [2; 5; 4; 8; 2; 6; 7; 7; 9]
On effectue une passe sur la liste : [5; 4; 8; 2; 6; 7; 7; 9]
                ce qui donne pour l : [2; 2; 5; 4; 8; 6; 7; 7; 9]
On effectue une passe sur la liste : [5; 4; 8; 6; 7; 7; 9]
                ce qui donne pour l : [2; 2; 4; 5; 6; 8; 7; 7; 9]
On effectue une passe sur la liste : [5; 6; 8; 7; 7; 9]
                ce qui donne pour l : [2; 2; 4; 5; 6; 7; 8; 7; 9]
On effectue une passe sur la liste : [6; 7; 8; 7; 9]
                ce qui donne pour l : [2; 2; 4; 5; 6; 7; 7; 8; 9]
On effectue une passe sur la liste : [7; 7; 8; 9]
Ce qui donne  [2; 2; 4; 5; 6; 7; 7; 8; 9]
```

Vous remarquez que la dernière passe est inutile car il n'y a plus d'inversion, mais le programme doit s'en assurer. Une façon simple de s'arrêter à temps (pour éviter de boucler sur des passes inutiles) consiste en effet à adjoindre au résultat de la fonction `une_passe` un booléen qui signale si une transposition a été effectuée.

1° a) Programmez récursivement la fonction `une_passe`.

```
#une_passe [5; 4; 8; 2; 6; 7; 7; 2; 9];;
- : bool * int list = true, [2; 5; 4; 8; 2; 6; 7; 7; 9]
#une_passe [7; 7; 8; 9];;
- : bool * int list = false, [7; 7; 8; 9]
```

b) En déduire une implémentation récursive du tri bulle.

2° a) Montrer que le nombre de comparaisons effectuées par tri bulle pour trier une liste de longueur n est majoré par $\frac{n(n-1)}{2}$.

b) À l'aide du modèle des permutations (voir l'exercice précédent), estimez le nombre moyen d'*échanges* effectués par le tri bulle.

3° Programmez le tri bulle sur des vecteurs.

Exercice II.5. Le tri fusion.

Il s'agit à nouveau d'un tri suivant le paradigme diviser pour régner. Le principe en est le suivant :

- on divise en deux moitiés la liste à trier,
- on trie chacune d'entre elles,
- on fusionne les deux moitiés obtenues pour reconstituer la liste complète triée.

1° a) Écrivez une fonction `divise` récursive qui produit à partir d'une liste donnée deux moitiés de longueur égale (ou presque dans le cas d'une liste initiale de longueur impaire). Pour cela, inspirez-vous d'un croupier qui distribue un paquet de cartes à deux joueurs en remettant alternativement une carte à l'un puis à l'autre des deux adversaires.

une obsession de l'argent, du jeu...

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

b) Écrivez une fonction `fusion` qui fusionne deux listes triées en une seule.

```
#fusion [5; 6; 7; 8;9] [2; 2; 4; 7];;
- : int list = [2; 2; 4; 5; 6; 7; 7; 8; 9]
```

c) En déduire finalement la fonction `tri_fusion`.

2° On note $c(n)$ le nombre de comparaisons nécessaires à `tri_fusion` pour trier une liste de longueur n. Montrez que $c(n)$ vérifie une récurrence du type diviser pour régner :

$$c(n) = 2c\left(\frac{n}{2}\right) + dn$$

(traitez le cas n pair).

En déduire la complexité du tri fusion.

3° La longueur d'un vecteur étant fixée lors de sa création, l'accès à une coordonnée s'effectuant en temps constant, il est rapide de diviser en deux un vecteur en Caml. La structure de données vecteur semble donc toute désignée pour implémenter l'algorithme de tri fusion. Comme il s'agit d'économiser la place mémoire, nous allons

implémenter l'algorithme à l'aide d'une seule mémoire auxiliaire : un autre vecteur de même longueur que le vecteur à trier initial. La fonction principale se présente ainsi :

```
let tri_fusion v =
       let n = (vect_length v) in
       let aux =  make_vect n 0 in
       tri v 0 (n-1) aux
       where rec tri v i j aux =
         if j>i then
             begin
               let m=(i+j)/2 in
                  tri v i m aux;
                  tri v (m+1) j aux;
                  fusion v i j aux
             end;;
```

La fonction ne renvoie aucun résultat, elle modifie physiquement le vecteur initial : on dit qu'elle procède par effets de bord. Vous constatez que `tri_fusion` requiert une fonction auxiliaire `tri v i j aux` qui réalise le tri de la partie du vecteur `v` située entre les indices `i` et `j` à l'aide du vecteur auxiliaire `aux`. Il ne manque plus qu'à écrire la fonction `fusion` de type

```
#fusion;;
- : 'a vect -> int -> int -> 'a vect -> unit = <fun>
```

À vous de jouer !

Exercice II.6. Tests de terminaison.

On considère les fonctions suivantes :

```
let rec u n =
   if n=0 then 0 else n - u (n-1);;

let rec v n =
   if n=0 then 1 else n-v(v(n-1));;
```

1° La fonction `u` termine-t-elle ? Si tel est le cas, quelle est sa valeur ?

2° a) La fonction `v` termine-t-elle ?

b) On modifie le code précédent en imposant $v(0) = 0$. Donner une version itérative de la fonction v.

Exercice II.7. Quelques compléments sur la fonction d'Ackermann (voir page 86).

Par commodité d'écriture nous noterons $A(n, p)$ pour `ackermann n p`.

1° Donner les expressions explicites de $A(1, p)$, $A(2, p)$ et $A(3, p)$.

2° Que vaut $A(4, 4)$? Comparer cet entier avec 10^{80} (qui est une estimation du nombre d'atomes de l'univers !).

3° Montrer que $\forall\ (n, p) \in \mathbb{N}^2$, $A(n, p) > p$.

4° Montrer que si $q > p$, alors $\forall\ n \in \mathbb{N}$, $A(n,q) > A(n,p)$.

5° Montrer que $A(n+1,p) > A(n,p)$ pour tout couple $(n,p) \in \mathbb{N}^2$.

6° On définit la fonction α de $\mathbb{N}$ dans $\mathbb{N}$ comme l'*inverse fonctionnel* de A : $\alpha(n)$ est le plus petit entier k tel que $A(k,k) \geqslant n$. Montrer que la fonction α est bien définie sur $\mathbb{N}$.

Que peut-on dire de la croissance de α ?

Exercice II.8. McCarthy.

La fonction de McCarthy est définie par :

```
let rec f n =
    if n>100
    then n-10
    else f (f (n+11));;
```

La fonction `f` termine-t-elle sur $\mathbb{Z}$? Si tel est le cas que vaut-elle ?

Exercice II.9. Inversions paires.

On appelle *inversion paire* d'une liste d'entiers *non vide de longueur paire* deux éléments successifs tels que :

- le premier d'entre eux est situé en *position paire* (on considère ici que l'élément en tête de la liste est en position 0, que son suivant est en position 1 etc.),
- le premier est strictement supérieur au second.

Par exemple, `[2;5;-3;8;11;10;0;8]` n'admet que `(11,10)` comme inversion paire.

1° Programmer récursivement une fonction `inversion` qui teste la présence d'une inversion paire dans une liste *de longueur paire non vide*.

Voici comment doit se comporter cette fonction :

```
#inversion [2;5;3;8;11;10;0;8];;
- : bool = true
#inversion [2;5;3;8;11;12;0;8];;
- : bool = false
```

2° Prouver la fonction.

Exercice II.10. Parité.

Voici une définition mutuellement récursive des fonctions `pair` et `impair` qui testent la parité des entiers :

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

Proposer une version de ces deux fonctions sans récursivité croisée (ni appel aux fonctions modulo ou division, bien sûr...).

Exercice II.11. Un peu de trigonométrie.

d'après [13]. Le but de cet exercice n'est pas de réaliser une implémentation efficace du calcul des fonctions sin et cos mais d'utiliser la récursivité mutuelle.

On sait qu'*au voisinage de* 0 on a $\sin x \sim x$ et $\cos x \sim 1 - \frac{x^2}{2}$. On connaît également les formules de trigonométrie usuelles :

$$\begin{aligned}
\cos(x+2\pi) &= \cos x \\
\sin(x+2\pi) &= \sin x \\
\cos(x+\pi) &= -\cos x \\
\sin(x+\pi) &= -\sin x \\
\cos x &= \sin(x+\frac{\pi}{2}) = \sin(\frac{\pi}{2}-x) \\
\sin(2x) &= 2\sin x \cos x \\
\cos(2x) &= 1-2(\sin x)^2
\end{aligned}$$

À l'aide de ces formules, proposer un programme Caml de calcul des fonctions sin et cos n'utilisant que les formules ci-dessus et les approximations de Taylor au voisinage de 0.

Exercice II.12. Un invariant de boucle.

Déterminer et prouver le résultat calculé par la fonction suivante :

```
let f x y =
    let r = ref 0 and s = ref y in
        while (!s >0) do
          r:= !r + x;
          s:= !s -1
        done;
    s:= y -1;
    let t = ref !r in
        while (!s>0) do
          r:= !r + !t;
          s:= !s -1
        done;
    !r;;
```

Exercice II.13. Suite et vecteur.

On considère la suite u_n définie par la récurrence suivante :

$$u_0 = 1 \qquad u_{n+1} = \sum_{k=0}^{n} u_k u_{n-k},\ n \geqslant 0$$

1° Écrire en Caml, de manière impérative, la fonction `suite` qui calcule u_n.

2° Prouver le programme obtenu.

Exercice II.14. Fonction de deux variables.

Nous avons vu en page 102 une implémentation de la suite de Fibonacci de complexité linéaire qui utilise la fonctionnelle `boucle_inconditionnelle` (l'équivalent récursif d'une boucle `for`) et une fonction de deux variables *ad hoc*.

1° Sur ce modèle, proposer une nouvelle implémentation de la factorielle.

2° Peut-on calculer de même la somme de la série harmonique ?

(indication : on peut voir une des variables de la fonction à itérer comme le résultat intermédiaire et l'autre comme l'incrément)

Exercice II.15. Le flocon de Von Koch.

Figure II.3 : *Flocons de Von Koch.*

La figure II.3 montre les flocons de Von Koch pour $n = 0$, $n = 1$, $n = 2$ et $n = 4$. En regardant attentivement vous vous rendrez compte que ces figures géométriques sont obtenues à l'aide de la transformation illustrée en figure II.4 : pour obtenir B, on a remplacé le tiers médian du segment droit initial A de longueur L par deux segments de longueur $L/3$ chacun (la longueur de B est ainsi $4/3L$).

Figure II.4 : *Le segment A devient le segment B.*

1° On désire programmer récursivement le dessin des flocons de Von Koch. On se propose d'utiliser pour cela la fonction principale :

```
let flocon n =
(* n représente la profondeur de la récursion *)
   clear_graph ();
(* on se place en l'origine du dessin *)
   moveto 100 100;
```

```
(* puis on trace les trois côtés du flocon  *)
   côté 0.0 100. n;
   côté ((2. *. pi) /. 3.) 100. n;
   côté (-.((2. *. pi) /. 3.)) 100. n;;
```

La fonction `flocon` appelle la fonction `côté angle longueur n` qui trace récursivement un des trois côtés du flocon à partir du point courant de longueur `longueur` et faisant un angle `angle` avec l'horizontale. Proposer une implémentation de `côté`.

2° La fonction `côté` est-elle insensible aux erreurs d'arrondis (dues aux calculs de cosinus et sinus)? Comment améliorer ce fait?

Exercice II.16. Le loup, la chèvre et le chou.

Un loup, une chèvre et un chou doivent traverser une rivière. Un batelier propose sa barque, mais il ne peut pas transporter plus d'un passager à la fois. Or, si le loup et la chèvre restent seuls sur l'une des deux rives, le loup mangera la chèvre et, de même, si la chèvre et le chou restent seuls, la chèvre mangera le chou. Il faut écrire un programme qui détermine quelles sont les traversées à effectuer pour que les trois se retrouvent sains et saufs sur l'autre rive.

Précisons qu'il est possible de traverser à vide, et qu'on peut ramener quelqu'un qui a déjà traversé.

Nous présentons dans cet exercice, une méthode pour la résolution du problème utilisant un couple de fonctions mutuellement récursives, l'exercice III.2 proposera une autre approche.

Nous allons représenter ici la solution au problème à l'aide de la liste des états successifs de la rivière c'est-à-dire sous la forme d'une liste de triplets dont la première coordonnée est la position de la barque (1 pour rive n° 1 et 2 pour rive n° 2), dont la deuxième coordonnée est l'état de la rive n° 1 et dont la troisième coordonnée est l'état de la rive n° 2.

Par exemple, la configuration `(1,["chevre"],["loup";"chou"])` signifie que la barque est à la rive n° 1, que seule la chèvre s'y trouve alors que le chou et le loup sont sur la rive n° 2.

1° Quelques fonctions utiles pour la suite :

a) Écrire la fonction `isole i l` qui renvoie le couple formé du i-ème élément de la liste `l` et de celle-ci privée de cet élément :

```
#isole 1 ["chevre"; "chou"; "loup"];;
- : string * string list = "chou", ["chevre"; "loup"]
```

b) Écrire la fonction `insère s l` qui insère à sa place la chaîne de caractères `s` dans la liste triée `l` (suivant l'ordre total de Caml `<`).

```
#insère "chou" ["chevre"; "loup"];;
- : string list = ["chevre"; "chou"; "loup"]
```

c) Écrire la fonction `correcte` qui vérifie que l'état d'une rive n'est pas interdit.

2° Écrire deux fonctions mutuellement récursives `rive1_à_2` et `rive2_à_1` qui à partir de la liste de configurations en cours essaient toutes les différentes traversées possibles.

Le programme principal sera lancé par :

```
rive1_à_2 [(1,["chevre";"chou";"loup"],[])];;
```

Attention à ne pas boucler en testant indéfiniment la même solution (du genre « j'emmène la chèvre d'une rive à l'autre puis je recommence »).

3° Conclure en proposant un affichage des solutions trouvées.

Éléments de correction des exercices

Exercice II.1. Horner.

1° On peut proposer :

```
let rec horner P x = match P with
  | [] -> 0.
  | a::r -> a +. x *. (horner r x);;
```

2° Si le polynôme est de degré n, la méthode naïve nécessite n additions et $2n-1$ multiplications ($n-1$ pour le calcul des x^k et n pour les produits par les coefficients) pour réaliser l'évaluation alors que la méthode de Horner n'effectue que n multiplications et n additions.

Le gain de temps est d'autant plus conséquent que le coût d'une addition est négligeable devant celui d'une multiplication. Il convient d'évaluer un polynôme suivant cette méthode, surtout si la variable est complexe (matrice...).

Exercice II.2. Tri par sélection.

1° On peut proposer :

```
let rec minimum_et_reste = function
    | [x] ->(x,[])
    | x1::r1 -> let (m2,l2) = (minimum_et_reste r1) in
                if x1<m2 then (x1,m2::l2) else (m2,x1::l2);;
```

2° Si la liste est de longueur n, l'algorithme applique successivement la fonction `minimum_et_reste` à la liste initiale, à la liste privée de son minimum, à celle-ci privée de son minimum... jusqu'à la liste vide, donc à des listes de longueur n, $n-1$, $n-2$,..., 0. Pour déterminer le minimum d'une liste `l`, la fonction `minimum_et_reste` parcourt entièrement `l` et effectue (longueur `l`) -1 comparaisons. L'algorithme de tri par sélection effectue donc en tout $\dfrac{n(n-1)}{2}$ comparaisons. On pourra comparer ce résultat à ceux obtenus par le tri rapide dans le chapitre IV.

3° Classiquement, sur des vecteurs, on peut échanger deux éléments à l'aide d'une mémoire supplémentaire[30] :

```
let echange i j v =
    let t = v.(i) in
      begin
        v.(i) <- v.(j);
        v.(j) <- t
      end;;
```

30. L'échange entre valeurs numériques peut se faire sans variable auxiliaire :

```
let echange_bis i j v =
    v.(j) <- v.(i) + v.(j);
    v.(i) <- v.(j) - v.(i);
    v.(j) <- v.(j) - v.(i);;
```

mais au prix d'opérations arithmétiques supplémentaires.

Le tri par sélection devient alors :

```
let minimum a b v =
       let mini = ref a in
          for i = a to b  do
             if v.(i) < v.(!mini)
             then mini:=i
          done;
       !mini;;

let tri_selection_vect v =
       let N = vect_length v and minimum_temporaire = ref 0 in
         for i=0 to N-1 do
            minimum_temporaire := minimum i (N-1) v;
            echange !minimum_temporaire i v
         done;
       v;;
```

Exercice II.3. Tri par insertion.

1° a) On propose :

```
let rec insere element = function
    | [] -> [element]
    | x::reste -> if element <= x
                  then element::x::reste
                  else x::(insere element reste);;
```

b) On a alors :

```
let rec tri_insertion = function
    | [] -> []
    | x::reste -> insere x (tri_insertion reste);;
```

2° On commence par implémenter l'ordre lexicographique :

```
let lexico (x1,y1) (x2,y2) = (x1<x2) or ((x1=x2) & (y1<=y2));;
```

Les fonctions modifiées deviennent alors :

```
let rec insere_generalise ordre element = function
  | [] -> [element]
  | x::reste -> if (ordre element x)
                then element::x::reste
                else x::(insere_generalise ordre element reste);;

let rec tri_insertion_generalise ordre = function
  | [] -> []
  | x::reste -> insere_generalise ordre x
                   (tri_insertion_generalise ordre reste);;
```

Ce qui donne par exemple :

```
#let lc=[(12,10);(3,2);(3,1);(0,0);(10,5);(11,1);(10,6);(1,2)];;
#tri_insertion_generalise lexico lc;;
- : (int * int) list =
         [0, 0; 1, 2; 3, 1; 3, 2; 10, 5; 10, 6; 11, 1; 12, 10]
```

3° L'insertion dans une liste déjà triée n'impose pas son parcours complet contrairement à une recherche de minimum. Le tri par insertion devrait donc mieux se comporter que le tri par sélection.

a) Sauf erreur de notre part, il y en a 21.

b) La somme du nombre d'inversions d'une permutation et de sa permutation miroir est $\frac{n(n-1)}{2}$, le nombre de couples possibles. En regroupant deux par deux les permutations, le nombre moyen d'inversions d'une permutation de $\mathfrak{S}_n$ est donc $\frac{n(n-1)}{4}$.

c) Le nombre de comparaisons effectuées par `insere` pour insérer l'élément l_i dans le bout de liste déjà trié correspond au nombre d'inversions présentes dans la suite des éléments d'indice i à n de la liste initiale, plus 1 (le premier test). Pour une permutation σ donnée, le nombre total de comparaisons effectuées est donc :

$$c_\sigma = n - 1 + inv(\sigma)$$

Le nombre moyen de comparaisons effectuées par le tri par insertion est donc :

$$\frac{1}{n!}\sum_{\sigma\in\mathfrak{S}_n} c_\sigma = n - 1 + \frac{n(n-1)}{4} = \frac{n(n+3)}{4} - 1$$

soit encore un nouvel algorithme quadratique, plus efficace en pratique que le tri par sélection. Dans le cas où la liste initiale est « presque en ordre », le nombre de comparaisons effectuées est faible, ce qui n'est pas le cas du tri par sélection.

4° Notons que l'on ne peut pas profiter de la structure vectorielle pour faire de l'insertion dichotomique en raison du décalage inévitable.

```
let insere_element v i =
       let j=ref i and e=v.(i) in
          while (!j>0) & (v.(!j-1) > e) do
                v.(!j) <- v.(!j-1) ; j:=!j-1
          done;
       v.(!j) <-e;;

let tri_insertion_vect v =
       for i=1 to (vect_length v)-1 do
           insere_element v i
       done;;
```

Exercice II.4. Tri bulle.

1° On peut proposer :

```
let rec une_passe  = function
  | [(x:int)] -> false,[x]
  | x::reste  -> let bool,res = (une_passe reste) in
                   if x<=(hd res)
                   then bool,x::res
                   else true,(hd res)::x::(tl res);;

let rec tri_bulle  = function
  | [] -> []
  | l -> let (modifiee, liste) = une_passe l in
           if modifiee
           then (hd liste)::(tri_bulle (tl liste))
           else liste;;
```

2° a) Comme le tri par sélection, cet algorithme effectue successivement `une_passe` sur la liste initiale, sur la liste privée de son minimum, sur celle-ci privée de son minimum... jusqu'à tomber sur une liste triée (qui au pire est de longueur 1). Dans ce cas, le parcours complet des listes successives impose d'effectuer $(n-1)+(n-2)$ $+(n-3)+\cdots+1$ comparaisons. Le nombre total de comparaisons est donc bien majoré par $\dfrac{n(n-1)}{2}$.

b) Chaque échange d'éléments de tri bulle supprime une et une seule inversion et, une fois le tri terminé, il n'y a plus aucune inversion. Ainsi le nombre total d'échanges effectués est égal au nombre d'inversions dans la liste initiale. Le nombre moyen d'échanges effectués par tri bulle est donc le nombre moyen d'inversions dans $\mathfrak{S}_n$ que nous avons déjà calculé à la question **3**° b) de l'exercice précédent, soit $\dfrac{n(n-1)}{4}$.

3° On utilise la fonction `echange` vue plus haut :

```
let une_passe_vect fin v =
  let modifié = ref false in
      for j=1 to fin do
        if v.(j-1) > v.(j)
        then begin
             echange (j-1) j v;
             modifié := true
             end
      done;
  !modifié;;

let tri_bulle_vect v =
  let i = ref ((vect_length v)-1) in
     while (!i>=0)  && (une_passe_vect !i v) do
       decr i
     done;;
```

Exercice II.5. Tri fusion.

1° On peut proposer :

```
let rec divise = function
    | [] -> ([],[])
    | [e] -> ([e],[])
    | a::b::r -> let (m1,m2) = divise r in
                 (a::m1,b::m2);;

let rec fusion = fun
    | l [] -> l
    | [] l -> l
    | (a::r as l1) (b::s as l2) -> if a<b
                                   then a::(fusion r l2)
                                   else b::(fusion l1 s);;

let rec tri_fusion = fun
    | [] -> []
    | [e] -> [e]
    | l -> let (m1,m2) = divise l in
           fusion (tri_fusion m1) (tri_fusion m2);;
```

2° La division d'une liste de longueur n en deux moitiés ne nécessite aucune comparaison, leur fusion en requiert $2\lfloor n/2 \rfloor$. La définition de `tri_fusion` montre que :

$$c(n) = c(n) = c\big(\lfloor n/2 \rfloor\big) + c\big(\lceil n/2 \rceil\big) + 2\lfloor n/2 \rfloor$$

et cette récurrence entraîne (voir chapitre IV) que $c(n) = \Theta(n \lg n)$.

On voit donc que dans tous les cas (le pire, le meilleur et en moyenne), la complexité de l'algorithme est en $n \lg n$. Ceci rend le `tri_fusion` plus intéressant que le tri rapide (voir l'analyse du chapitre IV) qui reste quadratique dans le pire des cas.

3° Amis des indices dans les vecteurs, bonjour ! Voici notre implémentation de la fusion sur des vecteurs (un conseil : faites un dessin pour vous représenter les mouvements de `i` et `j`) :

```
let fusion v debut fin aux =
       let m = (debut+fin)/2 and i=ref debut in let j=ref (m+1) in
            for k=0 to (fin-debut) do
              if (!i<=m)
              then
                begin
                  if (!j<=fin)
                  then
                    begin
                      if v.(!i)<=v.(!j)
                      then begin aux.(k)<- v.(!i); incr i end
                      else begin aux.(k)<- v.(!j); incr j end
                    end
```

```
      else
        begin aux.(k)<- v.(!i); incr i end
    end
  else begin aux.(k)<- v.(!j); incr j end
done;
for k=0 to (fin-debut) do
  v.(debut+k)<-aux.(k)
done;;
```

C'est beau les listes et la récursion, non ?

Exercice II.6. Tests de terminaison.

1° On conclut immédiatement à la terminaison de `u`. Une récurrence (puisqu'il s'agit ici d'une fonction récursive à un argument entier) montre alors que $u(2n) = n$ et $u(2n+1) = n+1$.

2° Il suffit de calculer $v(1)$ pour s'apercevoir que v ne termine pas (il n'y a pas de solution entière). Par contre, en considérant $v(0) = 0$ on peut vérifier que $v(k) < k+1$ et prouver la terminaison du programme.

Comme le calcul de $v(n)$ impose de connaître tous les termes $v(i)$ avec $i < n$; il est commode alors, pour dérécursifier `v`, d'utiliser un vecteur.

Exercice II.7. La fonction d'Ackermann.

1° Calcul de $A(1,p)$.

On a $A(1,0) = A(0,1) = 2$ et $A(1,p+1) = A(0,A(1,p)) = A(1,p)+1$ donc par récurrence $A(1,p) = p+2$.

Calcul de $A(2,p)$.

On a $A(2,0) = A(1,1) = 3$ et $A(2,p+1) = A(1,A(2,p)) = A(2,p)+2$ donc par récurrence $A(2,p) = 2p+3$.

Calcul de $A(3,p)$.

On a $A(3,0) = A(2,1) = 5$ et $A(3,p+1) = A(2,A(3,p)) = 2A(3,p)+3$ donc par récurrence $A(3,p) = 2^{p+3} - 3$.

2° On a $A(4,4) = 2^{A(4,3)+3} - 3 > 2^{A(4,3)}$ d'où, comme $A(4,0) = 13 > 2$, on a :

$$A(4,4) \gg 2^{\left(2^{\left(2^{\left(2^2\right)}\right)}\right)} = 2^{65536} \gg 10^{80} \ !$$

Exactement, $A(4,4) = 2^{\left(2^{\left(2^{\left(2^{13}\right)}\right)}\right)} - 3$.

3° On va montrer le résultat par induction dans $(\mathbb{N}^2, \preccurlyeq)$ suivant le théorème de correction. Soit $\mathfrak{p}_A$ le prédicat défini pour tout $(n,p) \in \mathbb{N}^2$ par :

$$\mathfrak{p}_A(n,p) = \text{« } A(n,p) > p \text{ »}.$$

– Si $n = 0$, alors $A(0,p) = p+1 > p$, soit $\mathfrak{p}_A(0,p)$.

– Par définition, $A(n,0) = A(n-1,1)$. Par hypothèse d'induction, $A(n-1,1) > 1$ donc on a bien $A(n,0) = A(n-1,1) > 1 > 0$.
– Par définition, $A(n,p) = A(n-1, A(n,p))$. Par hypothèse d'induction, $A(n,p-1) > p-1$ et $A(n-1, A(n,p-1)) > A(n,p-1)$ donc $A(n,p) = A(n-1, A(n,p-1)) > A(n,p-1) > p-1$ d'où $A(n,p) > p$ (on travaille dans $\mathbb{N}$!).

On a donc bien montré $\mathfrak{p}_A$ pour tout $(n,p) \in \mathbb{N}^2$.

4° Il suffit de montrer que $\forall\ (n,p) \in \mathbb{N}^2$, $A(n,p+1) > A(n,p)$. C'est clair si $n = 0$. Soit $n \in \mathbb{N}^*$, on a, pour tout p, d'après la question précédente, $A(n,p+1) = A(n-1, A(n,p)) > A(n,p)$ soit le résultat.

5° On va montrer le résultat à nouveau par induction. Soit $\mathfrak{p}_A$ le prédicat défini pour tout $(n,p) \in \mathbb{N}^2$ par :

$\mathfrak{p}_A(n,p) =$ « $A(n+1,p) > A(n,p)$ ».

– Si $n = 0$, alors $A(1,p) = p+2 > A(0,p) = p+1$.
– Par définition, $A(n+1,0) = A(n,1)$. Par hypothèse d'induction, on a $A(n,1) > A(n-1,1)$ d'où $A(n+1,0) = A(n,1) > A(n,0) = A(n-1,1)$.
– Par définition, $A(n,p) = A(n-1, A(n,p))$. Par hypothèse d'induction, on a $A(n+1,p-1) > A(n,p-1)$ d'où, d'après le **4°**, $A(n-1, A(n+1,p-1)) > A(n-1, A(n,p-1))$. Et, par induction on peut supposer $A(n, A(n+1,p-1)) > A(n-1, A(n+1,p-1))$ et on en déduit bien

$$A(n+1,p) = A(n, A(n+1,p-1)) > A(n-1, A(n,p-1)) = A(n,p)$$

On a donc bien montré $\mathfrak{p}_A(n,p)$ pour tout $(n,p) \in \mathbb{N}^2$.

6° Pour vérifier la définition de α, il suffit de montrer que la suite $(A(n,n))_{n\in\mathbb{N}}$ tend vers $+\infty$. On a $A(0,0) = 1$, $A(1,1) = 3$, $A(2,2) = 7$, $A(3,3) = 61$ et $A(4,4) = ...$ Pour $n \geqslant 3$, à l'aide des questions précédentes, $A(n+1,n+1) = A(n, A(n+1,n)) > A(n+1,n) > A(n,n)$, d'où la stricte croissance de la suite et le résultat s'en déduit.

La fonction α est évidemment une fonction à la croissance très lente. Elle intervient notamment dans le calcul de la complexité du problème *Set-Union-Find* (union et recherche dans les ensembles). On connaît en effet un algorithme d'union d'ensembles[31] « quasi-linéaire » (de complexité $\mathcal{O}(n\,\alpha(n))$).

On pourra se reporter au problème de Mathématiques appliquées et Algorithmique posé en 1995 à l'ENS de Lyon (attention, pour simplifier les calculs, l'auteur a choisi une définition d'Ackermann légèrement différente de la définition usuelle).

Exercice II.8. Fonction de McCarthy.

Nous allons montrer que le calcul de `f (x)` termine toujours sur $\mathbb{Z}$ et vaut : $x - 10$ si $x > 100$ et 91 si $x \leqslant 100$.

L'intervalle d'entiers $[\![100, +\infty[\![$ va constituer notre ensemble $\mathcal{B}$ de cas de base ; nous allons en effet, pour $x \leqslant 100$, procéder par induction sur $([\![-\infty, 100]\!], \geqslant)$ qui est bien fondé (car majoré).

Mais si, mais si !

31. On emploie aussi le vocabulaire de « fusion de classes d'équivalence ».

Nous allons définir le prédicat suivant :

$\mathfrak{p}_f(x) =$ « $f(x)$ termine et

- si $100 < x$, $f(x) = x - 10$,
- si $x \leqslant 100$, $f(x) = 91$ »

et le montrer en suivant le théorème II.8 (attention, l'ordre considéré est l'opposé de l'ordre naturel sur les entiers ; notre preuve inductive va donc aller *a contrario* de nos habitudes de preuve par récurrence). Trois cas se présentent :

- soit $x > 100$, $f(x)$ termine et vaut bien $x - 10$, c'est un cas de base.
- soit $90 \leqslant x \leqslant 100$ et par définition $f(x) = f(f(x + 11))$. Comme $101 \leqslant x + 11$, $f(x + 11) = x + 11 - 10 = x + 1$ et donc dans cet intervalle $f(x) = f(x + 1)$. On peut recommencer ce raisonnement sur $x + 1$, puis $x + 2$... tant que l'on reste dans l'intervalle $[\![90, 100]\!]$. On obtient alors $f(x) = f(x + 1) = ... = f(100) = f(101)$. Car $f(101)$ constitue le cas d'arrêt rencontré. La valeur calculée vaut donc $f(101) = 91$. Le prédicat est donc à nouveau vérifié dans ce cas.
- soit $x < 90$ et $f(x)$ conduit encore au calcul de $f(f(x + 11))$. Comme $101 \geqslant x + 11 > x$, par hypothèse d'induction $f(x + 11)$ termine et vaut $91 > x$. Donc $f(x)$ termine et vaut $f(f(x + 11)) = f(91) = 91$. Le prédicat est donc à nouveau vérifié dans ce cas.

Finalement, on a bien montré, en suivant le théorème de correction, le résultat annoncé.

Exercice II.9. Inversion paire.

1° On peut proposer pour `inversion` le code :

```
let rec inversion = function
  | [a;b]   -> (a>b)
  | a::b::r -> (a>b) || (inversion r);;
```

remarquez la vertu de la paresse : l'usage du connecteur `||` permet un arrêt dès qu'une inversion paire est trouvée.

2° L'ensemble $\mathcal{A}$ des arguments de `inversion` est l'ensemble des listes d'entiers de longueur paire non vides. On peut considérer l'application φ « longueur de la liste » à valeur dans l'ensemble ordonné bien fondé $(\mathbb{N}, \leqslant)$. Notez que, pour une fois, φ n'est pas surjective car $\varphi < \mathcal{A} >= 2\mathbb{N} \setminus \{0\}$. Le minimum de $\varphi < \mathcal{A} >$ est 2 et les cas de base sont les listes de longueur 2.

Soit $\mathfrak{p}_i$(`l`) le prédicat : « `inversion l` termine et retourne `true` si et seulement s'il y a une inversion paire dans la liste `l` ».

Le prédicat est vérifié si la liste est de longueur 2. Si l'on suppose $\mathfrak{p}_i$(`r`), alors le calcul de $\mathfrak{p}_i$(`a::b::r`) termine et, testant si les deux premiers éléments `a` et `b` forment une inversion paire, renvoie le booléen recherché.

On a donc montré la correction de `inversion`.

Exercice II.10. Pair et impair.

On commence par créer une fonction qui alterne le couple `(true, false)`.

```
let rec parité = function
          | 0 -> (true, false)
          | n -> let (p,i) = parité (n-1) in (i,p);;
```

et on n'a plus qu'à considérer la bonne coordonnée :

```
let impair n = snd (parité n);;
let pair n = fst (parité n);;
```

Exercice II.11. Un peu de trigonométrie.

Dans [13], P Cousot propose :

```
let pi=4. *. atan(1.);;
let rec cos x =
    if x>= 2. *. pi then cos (x -. 2.*. pi)
    else if x >=  pi then -. cos (x -.  pi)
    else if x >= (pi /. 2.) then sin ((pi /. 2.)-. x)
    else if x <= -2.*.pi then cos (x +. 2. *. pi)
    else if x <= -.pi then -. cos (x +. pi)
    else if x <= -.(pi /. 2.) then sin ( x +. (pi /. 2.))
    else if abs_float (x) < 0.003 then 1. -. x *. x /. 2.
    else let  s= sin (x /. 2.) in 1. -. 2. *. s *. s
and sin x =
    if x>= 2. *. pi then sin (x -. 2.*. pi)
    else if x >=  pi then -. sin (x -.  pi)
    else if x >= (pi /. 2.) then cos ((pi /. 2.)-. x)
    else if x <= -2.*.pi then sin (x +. 2. *. pi)
    else if x <= -.pi then -. sin (x +. pi)
    else if x <= -.(pi /. 2.) then -. cos ( x +. (pi /. 2.))
    else if abs_float (x) < 0.00001 then x
    else 2. *. sin (x /. 2.) *. cos (x /. 2.);;
```

L'appel aux fonctions sin et cos termine puisque l'argument décroît strictement en valeur absolue. Cette méthode est cependant très coûteuse : pour $x = 10\pi$ il y a 861 appels de cos et 944 appels de sin ! Numériquement, elle est également contestable car fortement sujette aux erreurs d'arrondi.

Exercice II.12. Invariant de boucle.

Il y a deux boucles à considérer. Définissons (avec les conventions usuelles de notation) comme premier invariant de boucle :

$$r_i + xs_i = xy$$

On a bien $r_0 = 0$ et $s_0 = y$ soit l'invariant pour $i = 0$. Ensuite si l'on suppose la relation après i itérations de la première boucle, comme $r_{i+1} = r_i + x$ et $s_{i+1} = s_i - 1$, on a $r_{i+1} + xs_{i+1} = r_i + x + x(s_i - 1) = r_i + xs_i = xy$. □

À la fin de la première boucle, `!s = 0` donc `!t = !r = x y`.

On définit alors un deuxième invariant de boucle : « $r_i + xys_i = xy^2$ ».

Il est à nouveau vérifié à chaque étape car avant la seconde boucle $r_0 = xy$ et $s_0 = y - 1$ puis, en supposant la relation après i itérations de boucle, comme $r_{i+1} = r_i + !t = r_i + xy$ et $s_{i+1} = s_i - 1$, on a $r_{i+1} + xys_{i+1} = r_i + xy + xy(s_i - 1) = r_i + xys_i = xy^2$. □

Finalement, comme la seconde boucle termine lorsque `!s=0`, la valeur retournée par `f` est `!r` $= xy^2$.

Exercice II.13. Suite et vecteur.

Le calcul d'un terme nécessite la mémoire de tous les précédents (surtout éviter l'algorithme naïf exponentiel qui consisterait à faire des appels récursifs répétés de `u`!). Il est donc souhaitable de stocker ceux-ci sous la forme d'un vecteur (accès en temps constant). Cette structure de données conduit à la programmation impérative qui suit.

1° Les premiers termes de la suite (1,1,2,5,14,42,...) sont obtenus avec :

```
let suite n =
(* calcule TOUS les termes de la suite entre 1 et n *)
  let u = make_vect (n+1) 0 in
   u.(0) <- 1;
   for i=1 to n do
   for j=0 to i-1 do
     u.(i) <- u.(i) + u.(j) * u.(i-1-j)
   done done;
 u.(n);;
```

(évidemment si l'on désire faire plusieurs appels à `suite`, on a intérêt à passer en variable globale, avec une taille maximale choisie, le vecteur `u`...)

On peut raffiner en remarquant la symétrie de la somme :

$$\sum_{k=0}^{2p+1} u_k u_{2p+1-k} = 2\sum_{k=0}^{p} u_k u_{2p+1-k} \quad \text{et} \quad \sum_{k=0}^{2p} u_k u_{2p-k} = {u_p}^2 + 2\sum_{k=0}^{p-1} u_k u_{2p-k}$$

2° Qui dit boucles dit invariants de boucle... On peut par exemple poser (avec les notations usuelles : $\mathtt{u_k(l)}$ représente u_l après k itérations de la boucle) pour la boucle la plus interne l'invariant de boucle :

« après j itérations de la boucle, l'entier $\mathtt{u_j(i)}$ vaut $\displaystyle\sum_{k=0}^{j-1} \mathtt{u(k)u(i-1-k)}$ ».

La valeur de $\mathtt{u_0(i)}$ est bien nulle et si l'on suppose qu'après j itérations de la boucle $\mathtt{u_j(i)} = \displaystyle\sum_{k=0}^{j-1} \mathtt{u(k)u(i-1-k)}$ alors la $(j+1)$ième itération effectue

$$\mathtt{u_{j+1}(i)} = \mathtt{u_j(i)} + \mathtt{u(j)u(i-1-j)} = \sum_{k=0}^{j} \mathtt{u(k)u(i-1-k)}.$$

La boucle externe quant à elle ne fait qu'itérer ce procédé sur les $n+1$ (indices 0 à n) éléments du tableau u.

Ainsi `suite n` calcule bien la somme donc les termes de la suite u_n.

Exercice II.14. Fonction de deux variables.

1° Il suffit d'utiliser la fonction qui au couple (x, y) associe $(x+1, (x+1)\cdot y)$. Cela donne :

```
let fact n =
snd(boucle_inconditionnelle n (fun (x,y) -> (x+1,(x+1)*y)) (0,1));;
```

2° Pour calculer $\sum_{k=1}^{n} \frac{1}{k}$, on emploie cette fois la fonction qui au couple (x, y) associe $(x+1, y+\frac{1}{x+1})$. Attention à ne pas oublier que le résultat est réel :

```
let harmonique n =
    snd(boucle_inconditionnelle (n-1)
       (fun (x,y) -> (x+1,y+.1./.((float_of_int x)+.1.)))
       (1,1.));;
```

Exercice II.15. Von Koch.

1° L'implémentation directe de la fonction `côté` est la suivante :

```
let rec côté angle longueur n  =
    let (x,y) = current_point () in
    if (n=0)
(* arrêt de la récursion et tracé du segment *)
    then lineto
         (x + int_of_float (longueur *. cos (angle)))
         (y + int_of_float (longueur *. sin (angle)))
(* le déplacement est relatif *)
    else begin
(* si n<>0 on subdivise le côté en quatre segments
   de longueur "longueur/3" et on trace récursivement *)
         côté angle (longueur /. 3.) (n-1);
         côté (angle -. (pi/.3.)) (longueur /. 3.) (n-1);
         côté (angle +. (pi/.3.)) (longueur /. 3.) (n-1);
         côté angle (longueur /. 3.) (n-1)
         end;;
```

« On pallie généralement au manque de matériel par des hommes» (sic !) Camus, La peste.

2° Mais celle-ci souffre des arrondis dus aux calculs de cosinus et sinus. Vous avez sans doute fait mieux ! Nous proposons pour pallier ces défaillances de forcer le tracé correct des extrémités des segments ; ce qui donne :

```
let flocongood n =
    clear_graph ();
    let (ox,oy) = (ref 100., ref 100.) in
    côté 0.0 200. n ;
```

```
    côté ((2. *. pi) /. 3.) 200. n;
    côté (-.((2. *. pi) /. 3.)) 200. n
    where rec côté  angle longueur n  =
      if (n=0)
      then begin
           moveto (int_of_float !ox) (int_of_float !oy);
(* voilà la correction *)
           ox := !ox +. longueur *. cos (angle);
           oy := !oy +. longueur *. sin (angle);
           lineto (int_of_float !ox) (int_of_float !oy)
           end
      else begin
(* si n<>0  on subdivise le côté en quatre segments
   de longueur "longueur/3" et on trace récursivement *)
           côté angle (longueur /. 3.) (n-1);
           côté (angle -. (pi/.3.)) (longueur /. 3.) (n-1);
           côté (angle +. (pi/.3.)) (longueur /. 3.) (n-1);
           côté angle (longueur /. 3.) (n-1)
           end;;
```

Exercice II.16. Le batelier etc.

Voici une solution possible complète commentée :

```
(*  FONCTIONS AUXILIAIRES *)
let rec isole = fun
       | i []     ->  "",[]
       | 0 (a::r) -> (a,r)
       | i (a::r) -> if (i < 0)
                     then failwith "erreur sur i"
                     else let e,reste = isole (i-1) r in
                          e, a:: reste;;

let rec insère  = fun
       | "" l -> l
       | e [] -> [e]
       | e (a::r)  -> if e<a then e::a::r else a::(insère e r);;
(* on insère une chaîne dans une liste maintenue ordonnée afin de
   pouvoir par la suite tester l'identité de deux configurations *)

let correcte = function
    | ["chevre";_] -> false
    | _ -> true;;

(* L'AFFICHAGE DES SOLUTIONS *)

let rec print_list = function
       | [] -> ()
       | a::r -> print_string a; print_string " "; print_list r;;
```

```
let rec print_rive i = function
       | [] -> print_string "la rive "; print_int i;
               print_string " est vide"
       | r  -> print_string "sur la rive "; print_int i;
print_string " on a : "; print_list r;;

let rec afficher (i,rive1,rive2) =
           print_string "La barque est sur la rive ";
           print_int i; print_newline ();
           print_rive 1 rive1; print_newline ();
           print_rive 2 rive2;print_newline ();;

let rec afficher_solution  = function
    | [] -> ()
    | [etat] -> print_newline ();
                print_string "j'ai trouvé une solution :";
                print_newline ();
                afficher etat
    | etat::reste -> afficher_solution reste;
                     print_string "------- une traversée -------";
                     print_newline ();
                     afficher etat;;
    (* on affiche dans l'ordre chronologique *)

(* LE CORPS DU PROGRAMME *)

let  rec rive1_à_2   = function
  | (1,rive1,rive2)::reste_lc as liste_configuration ->
      for i=0 to list_length rive1  do
       let element,nouvelle_rive1 = isole i rive1 in
        if correcte nouvelle_rive1
        then let nouvelle_rive2=insère element rive2 in
         if not ( mem (2,nouvelle_rive1,nouvelle_rive2) reste_lc )
              (* on teste si on ne boucle pas *)
         then rive2_à_1
           ((2,nouvelle_rive1,nouvelle_rive2)::liste_configuration)
      done
  |   _ -> failwith "erreur dans rive1_à_2"
and rive2_à_1  = function
  | (2,[],rive2)::reste_lc    as liste_configuration ->
      afficher_solution liste_configuration (* on a trouvé *)
  | (2,rive1,rive2)::reste_lc as liste_configuration ->
      for i=0 to list_length rive2  do
        let element,nouvelle_rive2 = isole i rive2 in
          if correcte nouvelle_rive2
          then let nouvelle_rive1=insère element rive1 in
            if not ( mem (1,nouvelle_rive1,nouvelle_rive2) reste_lc )
                  (* on teste si on ne boucle pas *)
```

```
              then rive1_à_2
               ((1,nouvelle_rive1,nouvelle_rive2)::liste_configuration)
        done
    |   _ -> failwith "erreur dans rive2_à_1";;
  (* on emploie des boucles for pour trouver TOUTES les solutions *)
```

L'exécution de

```
rive1_à_2 [1,["chevre";"chou";"loup"],[]];;
```

produit alors :

```
j'ai trouvé une solution :
La barque est sur la rive 1
sur la rive 1 on a : chevre chou loup
la rive 2 est vide
------- une traversée -------
La barque est sur la rive 2
sur la rive 1 on a : chou loup
sur la rive 2 on a : chevre
------- une traversée -------
La barque est sur la rive 1
sur la rive 1 on a : chou loup
sur la rive 2 on a : chevre
------- une traversée -------
La barque est sur la rive 2
sur la rive 1 on a : loup
sur la rive 2 on a : chevre chou
------- une traversée -------
La barque est sur la rive 1
sur la rive 1 on a : chevre loup
sur la rive 2 on a : chou
------- une traversée -------
La barque est sur la rive 2
sur la rive 1 on a : chevre
sur la rive 2 on a : chou loup
------- une traversée -------
La barque est sur la rive 1
sur la rive 1 on a : chevre
sur la rive 2 on a : chou loup
------- une traversée -------
La barque est sur la rive 2
la rive 1 est vide
sur la rive 2 on a : chevre chou loup

j'ai trouvé une solution :
La barque est sur la rive 1
sur la rive 1 on a : chevre chou loup
la rive 2 est vide
```

```
------- une traversée -------
La barque est sur la rive 2
sur la rive 1 on a : chou loup
sur la rive 2 on a : chevre
------- une traversée -------
La barque est sur la rive 1
sur la rive 1 on a : chou loup
sur la rive 2 on a : chevre
------- une traversée -------
La barque est sur la rive 2
sur la rive 1 on a : chou
sur la rive 2 on a : chevre loup
------- une traversée -------
La barque est sur la rive 1
sur la rive 1 on a : chevre chou
sur la rive 2 on a : loup
------- une traversée -------
La barque est sur la rive 2
sur la rive 1 on a : chevre
sur la rive 2 on a : chou loup
------- une traversée -------
La barque est sur la rive 1
sur la rive 1 on a : chevre
sur la rive 2 on a : chou loup
------- une traversée -------
La barque est sur la rive 2
la rive 1 est vide
sur la rive 2 on a : chevre chou loup
- : unit = ()
```

Il y a en effet deux solutions : on part avec la chèvre, on revient à vide, on repart avec le loup (ou le chou), on ramène la chèvre, on repart avec le chou (ou le loup), on revient à vide et on fait une dernière traversée avec la chèvre.

Li était de ceux qui ne pouvaient passer pour un humain
de la Terre sans une grande altération physique.
Il était velu et possédait la mauvaise silhouette :
imaginez un croisement entre Quasimodo et un singe.
Mais franchement, je crois que vous auriez pu le prendre
pour un vendeur de chez IBM tout à fait ordinaire.
Iain M. Banks — *L'état des arts*