Le langage fonctionnel Caml

Jacques Le Maitre

UNIVERSITÉ du SUD Toulon-Var

Table des matières

1
Introduction

Le langage Caml (il faut prononcer « Camel ») est un langage de programmation fonctionnel de la famille ML. ML a été inventé par l'équipe de Robert Milner à Edimbourg. C'était à l'origine un métalangage (d'où son nom) pour un système de vérification de preuves. C'est devenu ensuite un langage de programmation complet.

Plusieurs versions de ML sont maintenant disponibles, dont les plus connues sont Standard ML et Caml. Caml a été conçu à l'INRIA (Institut National de Recherche en Informatique et Automatique). Deux versions de Caml existent actuellement : Caml Light et Objective Caml. Caml Light est bien adapté à l'apprentissage de la programmation. Objective Caml, qui intègre Caml Light, permet la manipulation d'objets et dispose d'un compilateur très performant. Ces deux langages tournent sous UNIX, Windows et MacOS. C'est Caml Light qui est utilisé comme support de ce cours. C'est un logiciel libre qui peut être téléchargé à partir du site Web de de l'INRIA (`http://www.inria.fr`).

Les principales caractéristiques de Caml, sont les suivantes :

- Caml est un langage fonctionnel. Les fonctions sont des valeurs à part entière qui peuvent être argument ou valeur d'une fonction.
- Caml possède une syntaxe conviviale.
- Caml est un langage typé. Les types des variables ne sont pas déclarés par le programmeur mais calculés automatiquement par l'interpréte Caml.
- Caml supporte le filtrage, le polymorphisme et le traitement des exceptions.
- Caml permet la programmation impérative.

Un programme Caml manipule des valeurs. Une valeur peut être une *valeur de base* (un nombre, un booléen, un caractère, une chaîne de caractères), une *valeur structurée* (un *n-uplet*, une *liste* ou un *enregistrement*) ou bien une *fonction*. Toute valeur a un type. Un programme Caml se présente comme une suite de phrases qui sont soit des *expressions* à *évaluer*, soit des *définitions*. Une expression est construite à partir : de constantes littérales représentant des valeurs de base, de *noms de valeurs*, de *constructeurs de valeurs structurées* et de l'opération d'*application* d'une fonction. La valeur d'une expression est calculée en effectuant toutes les applications qu'elle contient. On dit que l'expression a été évaluée, réduite ou simplifiée. Une définition consiste à donner un nom à la valeur d'une expression.

Voici, par exemple, un programme Caml qui définit la longueur et la largeur d'un rectangle puis en calcule le périmètre.

```
#let lon = 15;;
#let lar = 5;;
#2 * (lon + lar);;
```

La valeur de ce périmètre est obtenue par les applications successives des fonctions prédéfinies `+` et `*`.

Avant de commencer l'étude de Caml voici quelques informations sur son utilisation en mode interactif. Le caractère d'invite (« prompt » en anglais) est le caractère `#`. A son invite, le programmeur peut entrer une phrase qui peut tenir sur plusieurs lignes et doit se terminer par `;;`. La phrase est validée par un retour chariot. Caml affichera alors :

– soit un message d'erreur,

– soit une réponse constituée en général d'une valeur et de son type.

Par exemple :

```
#(3 + 7) * 5;;
- : int = 50
```

L'expression `(3 + 7) * 5` est anonyme (-) de type `int` (entier) et a pour valeur `50`.

Trois types d'unités lexicales apparaissent dans un programme Caml : des *identificateurs*, des *mots-clés* réservés et des *symboles spéciaux*. Un identificateur est un mot composé d'une lettre suivie ou non d'une suite de caractères dont chacun peut être une lettre, un chiffre, le caractère _ ou le caractère `'`. Par exemple :

```
l1 les_villes x'1
```

Les symboles spéciaux sont utilisés comme séparateurs ou bien pour désigner les opérateurs les plus courants (`+` ou `<=`, par exemple). Les mots-clés réservés sont des identificateurs qui jouent un rôle particulier dans le langage (`if`, `then` ou `else`, par exemple).

Plusieurs livres sur Caml ont été écrits. Ce cours est fortement inspiré de trois d'entre eux :

– Pierre Weis, Xavier Leroy, *Le langage Caml*, InterEditions.

– Xavier Leroy, Pierre Weis, *Manuel de référence du langage Caml*, InterEditions.

– Thérèse Accart Hardin, Véronique Donzeau-Gouge Viguié, *Concepts et outils de programmation. Le style fonctionnel, le style impératif avec Caml et Ada*, InterEditions.

Les deux premiers livres ont été écrits par les concepteurs de Caml et de Caml Light. Le premier, remarquablement écrit, est plus qu'un cours sur Caml : c'est un véritable cours de programmation. Dans sa deuxième partie il présente un choix d'exemples très complets qui démontrent l'expressivité et la puissance de Caml. Le troisième livre est issu d'un cours de premier cycle du CNAM. L'idée était d'introduire la programmation fonctionnelle puis la programmation impérative à partir des langages Caml et Ada. La première partie du livre est consacrée à Caml dont la sémantique est présentée de façon très claire. On trouvera de plus, dans ce livre, un grand nombre d'exercices corrigés.

2
Valeurs, environnements et expressions

Ce chapitre introduit les concepts de base du langage Caml : les valeurs manipulées ; les expressions, qui sont des calculs soumis à l'interprète Caml ; la définition de valeurs, qui consiste à leur donner un nom ; l'*environnement* qui mémorise les *liaisons* entre noms et valeurs et auquel Caml se réfère lors d'un calcul et enfin le fonctionnement de l'interprète lui-même.

2.1 Valeurs

Caml manipule des valeurs qui sont classées par types. Un type représente un ensemble de valeurs. Toute valeur appartient à un et un seul type. Il faut faire une distinction entre l'expression d'un type et son extension. L'*expression d'un type* exprime la façon dont il est construit à partir d'autres types. L'*extension d'un type* est l'ensemble des valeurs de ce type. Si *t* est une expression de type nous noterons *ext*(*t*) l'extension de *t*.

Les types constructibles en Caml seront étudiés pas à pas tout au long de ce cours. Pour débuter nous considérerons un sous-ensemble des types de Caml, que nous appellerons *types de base*. Ce sont les suivants :

- Le type `bool` dont l'extension est constitué des valeurs `true` et `false` (*ext*(`bool`) = {`true`, `false`}).
- Le type `int` dont l'extension est l'ensemble des entiers appartenant à un intervalle dépendant de l'implémentation ([-2^{30}, $2^{30}-1$] en général).
- Le type `float` dont l'extension est l'ensemble des flottants appartenant à un intervalle dépendant de l'implémentation (si possible des flottants double précision à la norme IEEE).
- Le type `char` dont l'extension est l'ensemble des caractères dont le code numérique est compris entre 0 et 255. Les caractères dont le code est compris entre 0 et 127 sont du standard ASCII et ceux dont le code est compris entre 128 et 255 dépendent de l'implémentation.
- Le type `string` dont l'extension est l'ensemble des chaînes de 0 à *n* caractères où *n* dépend de l'implémentation (65535 au minimum).
- Le type `unit` dont l'extension ne contient qu'une seule valeur notée `()` et qui signifie « rien ».

2.2 Environnements

Une valeur peut être nommée. Un nom de valeur est :

- soit un identificateur différent des mots-clés réservés. Par exemple :

Soit *Env* un environnement.

VALEUR. Si *v* est une valeur de base de type *t* alors *v* est une expression telle que *type*(*v*, *Env*) = *t* et *val*(*v*, *Env*) = *v*.

NOM DE VALEUR. Si *n* est un nom de *Env* lié à une valeur *v* de type *t*, alors *n* est une expression telle que *type*(*n*, *Env*) = *t* et *val*(*n*, *Env*) = *v*.

EGALITE ET INEGALITE. Si e_1 et e_2 sont des expressions telles que *type*(e_1, *Env*) = *type*(e_2, *Env*) et *op* ∈ {=, <>} alors e_1 *op* e_2 est une expression telle que *type*(e_1 *op* e_2, *Env*) = `bool` et *val*(e_1 *op* e_2, *Env*) = *val*(e_1, *Env*) *op* *val*(e_2, *Env*).

OPPOSE D'UN ENTIER. Si *e* est une expression telle que *type*(*e*, *Env*) = `int` alors -*e* est une expression telle que *type*(-*e*, *Env*) = `int` et *val*(-*e*, *Env*) = -*val*(*e*, *Env*).

OPERATIONS ARITHMETIQUES SUR LES ENTIERS. Si e_1 et e_2 sont des expressions telles que *type*(e_1, *Env*) = `int` et *type*(e_2, *Env*) = `int` et *op* ∈ {+, -, *, /, mod} alors e_1 *op* e_2 est une expression telle que *type*(e_1 *op* e_2, *Env*) = `int` et *val*(e_1 *op* e_2, *Env*) = *val*(e_1, *Env*) *op* *val*(e_2, *Env*).

COMPARAISONS D'ENTIERS. Si e_1 et e_2 sont des expressions telles que *type*(e_1, *Env*) = `int` et *type*(e_2, *Env*) = `int` et si *op* ∈ {<, <=, >=, >} alors e_1 *op* e_2 est une expression telle que *type*(e_1 *op* e_2, *Env*) = `bool` et *val*(e_1 *op* e_2, *Env*) = *val*(e_1, *Env*) *op* *val*(e_2, *Env*).

OPERATIONS SUR LES FLOTTANTS. Les opérateurs `-. +. -. *. /. <. <=. >=. >.` sont définis sur les flottants de façon analogue aux opérateurs `- + - * / < <= >= >` sur les entiers.

CONCATENATION DE CHAINES (^). Si e_1 et e_2 sont des expressions telles que *type*(e_1, *Env*) = `string` et *type*(e_2, *Env*) = `string` alors e_1 ^ e_2 est une expression telle que *type*(e_1 ^ e_2, *Env*) = `string` et *val*(e_1 ^ e_2, *Env*) = *val*(e_1, *Env*) *concaténé avec* *val*(e_2, *Env*).

EXPRESSION PARENTHESEE. Si *e* est une expression alors `(`*e*`)` une expression telle que *type*(`(`*e*`)`, *Env*) = *type*(*e*, *Env*) et *val*(`(`*e*`)`, *Env*) = *val*(*e*, *Env*).

Figure 1.1. Les expressions de base

```
lon  lar  prénom1
```

– soit un symbole d'opérateur précédé du mot-clé `prefix`. Par exemple :

```
prefix +   prefix >=
```

Les noms de valeurs sont aussi appelés *variables*. Ils doivent être différent des mots-clés réservés de Caml comme cela est classique dans la plupart des langages de programmation.

Une liaison est une association entre un nom et une valeur. Une liaison entre un nom *n* et une valeur *v* de type *t* sera notée :

n : *t* = *v*

Par exemple :

```
lon : int = 15
```

Un environnement est un ensemble de liaisons. Par exemple :

{`lon : int = 15, lar : int = 5`}

Au début d'une session Caml il existe un environnement initial qui contient des liaisons prédéfinies, nous l'appellerons *Env_init*.

Un environnement peut être étendu par un autre environnement. L'extension d'un environnement Env_1 par un environnement Env_2, que nous noterons $Env_1 \oplus Env_2$, est obtenue en ajoutant les liaisons de Env_2 aux liaisons de Env_1 dont le nom n'est pas présent dans Env_2. On a :

$$Env_1 \oplus Env_2 = \{n : t = v \mid n \in Env_1 \text{ et } n \notin Env_2 \} \cup Env_2$$

On constate que cette opération à pour effet de ne garder que les liaisons les plus récentes de chaque nom. Par exemple :

$\{$`lon : int = 10`$\} \oplus \{$`lon : int = 15, lar : int = 5`$\} = \{$`lon : int = 15, lar : int = 5`$\}$

2.3 Expressions

Une expression est une construction Caml destinée à être évaluée. Toute expression a un type et une valeur. Le type d'une expression est déterminé (on dit aussi synthétisé) par Caml avant son évaluation.

Le type et la valeur d'une expression Caml dépendent de l'environnement dans lequel elle est évaluée. Si *e* est une expression et *Env* un environnement nous noterons

type(*e*, *Env*)

et

val(*e*, *Env*)

le type et la valeur de *e* dans *Env*. Par la suite nous dirons qu'une expression *e* est *valide* dans un environnement *Env*, si *type*(*e*, *Env*) est calculable.

Les types et les expressions valides en Caml seront étudiés pas à pas tout au long de ce cours. Pour débuter nous considérerons un sous-ensemble d'expressions (voir Figure 1.1), que nous appellerons de base, qui sont construites à partir des valeurs, des noms de valeur, et d'opérateurs sur les entiers, les flottants et les chaînes de caractères.

Considérons, par exemple, l'environnement :

Env = {`lon : int = 15, lar : int = 5`}

On a :

type(`2`, *Env*) = `int`
type(`lon`, *Env*) = `int`
type(`lar`, *Env*) = `int`
type(`lon + lar`, *Env*) = `int`
type(`2 * (lon + lar)`, *Env*) = `int`

et :

val(`2 * (lon + lar)`, *Env*)
= *val*(`2`, *Env*) * *val*(`(lon + lar)`, *Env*)
= `2 *` *val*(`lon + lar`, *Env*)
= `2 *` (*val*(`lon`, *Env*) `+` *val*(`lar`, *Env*))
= `2 * (15 + 5)`

```
= 40
```

2.4 Alternative

Il est souvent nécessaire d'effectuer des calculs qui dépendent d'une condition. Pour cela Caml dispose d'une construction particulière : l'*alternative*. Une expression alternative a la forme suivante :

`if` e_1 `then` e_2 `else` e_3

où e_1 est une expression de type `bool` et e_2 et e_3 sont des expressions de même type. Par exemple :

```
#if 10 > 5 then "Evidemment !" else "Bizarre !";;
- : string = "Evidemment !"
```

Les règles de typage et d'évaluation d'une alternative sont les suivantes :

TYPE ET VALEUR D'UNE ALTERNATIVE. Si *Env* est un environnement et e_1, e_2, e_3 sont des expressions telles que *type*(e_1, *Env*) = `bool` et *type*(e_2, *Env*) = *type*(e_3, *Env*) = *t* alors :

type(`if` e_1 `then` e_2 `else` e_3) = *t*,

val(`if` e_1 `then` e_2 `else` e_3, *Env*) =
si *val*(e_1, *Env*) = `true` alors *val*(e_2, *Env*) sinon *val*(e_3, *Env*).

On constate que selon la valeur de la condition, soit e_2, soit e_3 sont évaluées mais pas les deux.

Il est possible d'omettre la partie `else`. On a :

`if` e_1 `then` $e_2 \equiv$ `if` e_1 `then` e_2 `else ()`

ce qui implique que l'expression e_2 soit elle même de type `unit`.

Les expressions suivantes sont équivalentes à l'alternative :

e_1 `&` $e_2 \equiv$ `if` e_1 `then` e_2 `else false`,

e_1 `or` $e_2 \equiv$ `if` e_1 `then true else` e_2

Par exemple :

```
#(true or false) & true;;
- : bool = true
```

2.5 Définitions

2.5.1 Définitions globales

Une *définition globale* nomme des valeurs pour la durée d'une session. C'est une phrase qui a la forme suivante :

`let` n_1 `=` e_1 `and ... and` n_k `=` e_k`;;`

où n_1, …, n_k sont des noms de valeurs et e_1, …, e_k des expressions. Elle a pour effet, pour tout i de 1 à n, de donner le nom n_i à la valeur de l'expression e_i. Cette valeur constitue la définition de n_i. Par exemple :

```
#let deux_fois_pi = 2. *. 3.14;;
deux_fois_pi : float = 6.28

#let lon = 15 and lar = 5;;
lon : int = 15
lar : int = 5
```

La définition de `deux_fois_pi` est `6.28`, celle de `lon` est `15` et celle de `lar` est `5`.

Une définition globale a pour effet d'étendre l'environnement courant en cours par les liaisons introduites par la clause `let`. Soit *Env_courant* cet environnement, la définition `let` n_1 **=** e_1 `and` … `and` n_k = e_k le transforme en :

Env_courant ⊕ *Env_def*(`let` n_1 `=` e_1 `and` … `and` n_k `=` e_k, *Env_courant*).

où *Env_def* calcule l'environnement ajouté par l'ensemble des liaisons introduites par une clause `let` selon la règle suivante :

ENVIRONNEMENT AJOUTE PAR UNE DEFINITION NON RECURSIVE. Si *Env* est un environnement et pour tout i de 1 à k, n_i est un nom de valeur et e_i est une expression telle que $type(e_i, Env) = t_i$, alors :

Env_def(`let` n_1 `=` e_1 `and` … `and` n_k `=` e_k, *Env*) =
{$n_1 : t_1 = val(e_1, Env)$, …, $n_k : t_k = val(e_k, Env)$}.

Par exemple, si l'environnement courant est :

Env_init ⊕ {`rayon` : `float` = `10.`, `pi` : `float` = `3.14`}

la définition :

```
let surface = 2. *. pi *. rayon;;
```

le transforme en :

Env_init ⊕ {`rayon` : `float` = `10.`, `pi` : `float` = `3.14`, `surface` : `float` = `62.8`}

La liaison établie par la définition globale d'un nom reste valable pendant toute la session. Pour la changer il faut redéfinir ce nom, ce qui générera une nouvelle liaison qui cachera l'ancienne. De plus cette redéfinition n'aura aucun effet sur les définitions antérieures ayant utilisé ce nom. Par exemple :

```
#let annee_courante = 1998;;
annee_courante : int = 1998

#2000 - annee_courante;;
- : int = 2

#let age = annee_courante - 1981;;
age : int = 17

#let annee_courante = 1999;;
annee_courante : int = 1999

#2000 - annee_courante;;
- : int = 1

#age;;
- : int = 17
```

La valeur de `age` a été calculée avec l'ancienne valeur de l'année courante et n'a pas été affectée par la redéfinition de celle-ci.

Les noms définis dans une même clause `let` doivent être tous différents. Si ce n'est pas le cas Caml le signale. Par exemple :

```
#let lon = 5 and lon = 10;;
> Toplevel input:
>let lon = 5 and lon = 10;;
>    ^^^
> Variable lon is bound twice in this pattern.
```

Notons de plus, que les liaisons établies par une clause `let` ne sont visibles qu'une fois que cette clause a été évaluée. Dans l'exemple suivant :

```
#let cote = 10 and périmètre = 4 * cote;;
> Toplevel input:
>let cote = 10 and périmètre = 4 * cote;;
>                                  ^^^^
> Variable cote is unbound.
```

la liaison `cote = 10` n'est pas visible dans la définition de `périmètre`. Pour qu'elle le soit il faut réaliser deux définitions successives :

```
#let cote = 10;;
cote : int = 10

#let périmètre = 4 * cote;;
périmètre : int = 40
```

2.5.2 Définitions locales

La construction d'une expression peut être facilitée en nommant certaines sous-expressions dans des définitions locales à cette expression. Pour cela, Caml dispose d'un type particulier d'expression, que nous appellerons *expression à définitions locales*, qui a la forme suivante :

`let` n_1 `=` e_1 `and ... and` n_k `=` e_k `in` e

où n_1, …, n_k sont des noms de valeurs, e_1, …, e_k sont des expressions et e est une expression. Par exemple :

```
#8 * (let lon = 15 and lar = 5 in 2 * (lon + lar));;
- : int = 320
```

Les règles de typage et d'évaluation d'une expression à définitions locales sont les suivantes :

TYPE ET VALEUR D'UNE EXPRESSION A DEFINITIONS LOCALES. Si *Env* est un environnement et pour tout *i* de 1 à *k*, n_i est un nom de valeur, e_i est une expression valide dans *Env*, et *e* est une expression telle que *type*(*e*, *Env* ⊕ *Env_def*(`let` n_1 = e_1 `and ... and` n_k = e_k, *Env*)) = *t*, alors :

type(`let` n_1 = e_1 `and ... and` n_k = e_k `in` *e*, *Env*) = *t*,

val(`let` n_1 = e_1 `and ... and` n_k = e_k `in` e, *Env*) =
val(*e*, *Env* ⊕ *Env_def*(`let` n_1 = e_1 `and ... and` n_k = e_k, *Env*))}.

On remarque que l'environnement est étendu par les définitions locales pour la durée de l'évaluation de l'expression *e*.

Concernant la duplication des noms et la visibilité des liaisons établies, les règles sont les mêmes que pour les définitions globales (§1.5.1). Notamment pour définir des noms locaux qui dépendent les uns des autres, il faut d'imbriquer les expressions à définitions locales. Par exemple :

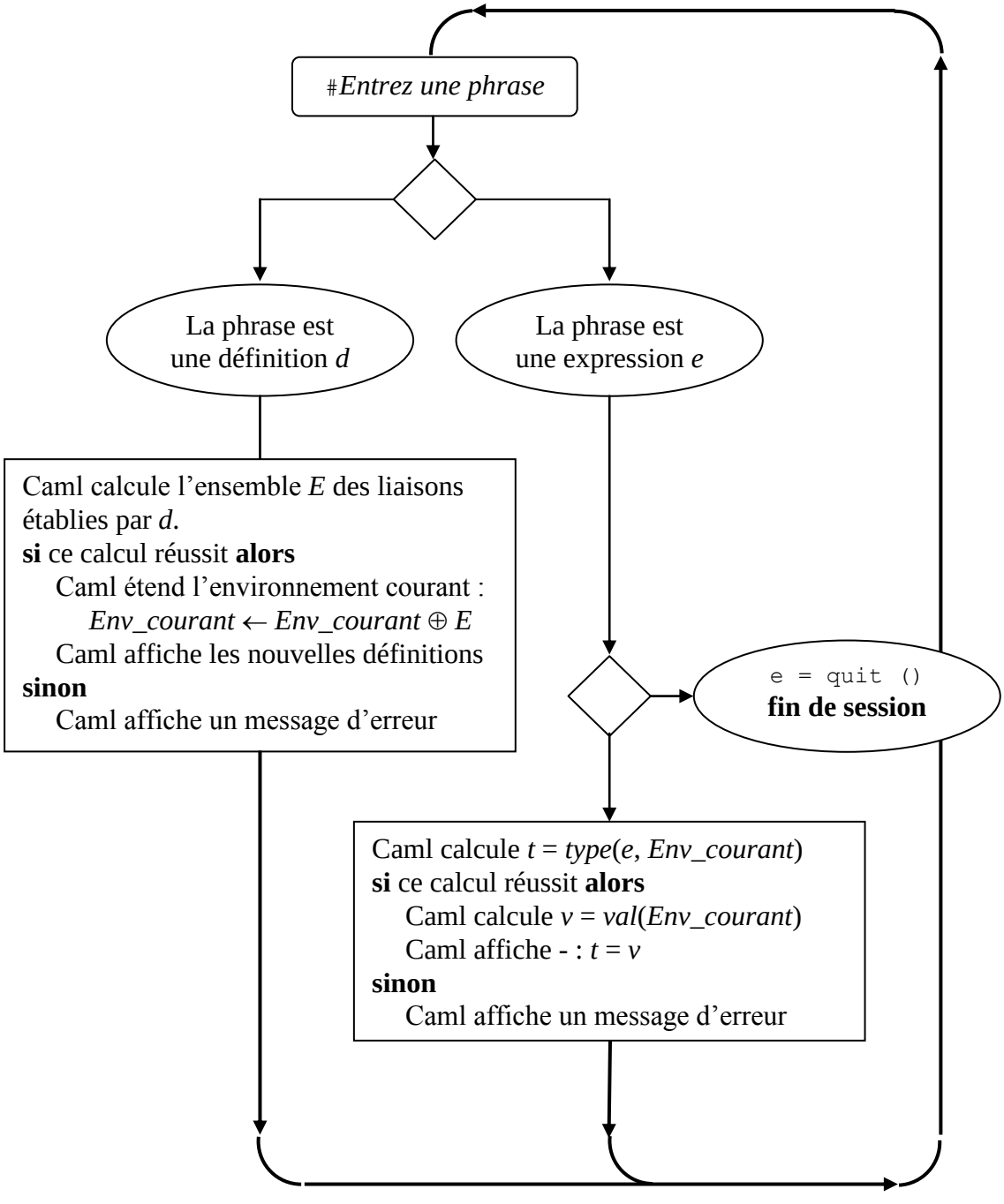

Figure 1.2 : La boucle de l'interprète Caml

```
let lon = 15 and lar = 5 and hauteur = 30 in
   let surface = lon * lar in surface * hauteur;;
- : int = 2250
```

2.6 L'interprète Caml

La figure 1.2 schématise l'interaction entre le programmeur et l'interprète Caml au cours d'une session interactive.

3
Fonctions

Dans ce chapitre nous pénétrons au cœur de Caml, puisque dans ce langage un programme n'est pas autre chose qu'une suite de définitions de valeurs ou de fonctions, suivie d'une expression à évaluer par applications successives d'une fonction à son argument.

En Caml, les fonctions sont des valeurs à part entière : elles peuvent être argument et valeur d'une fonction. Nous étudierons tout d'abord les fonctions à un seul argument : comment les construire et comment les appliquer. Nous montrerons ensuite que les fonctions à plusieurs arguments peuvent être traitées comme une suite de fonctions à un argument. Nous terminerons par l'étude des fonctions récursives.

3.1 Construction et application d'une fonction

En Caml, une fonction est une valeur particulière appelée *fermeture* et symbolisée par le mot-clé `<fun>`. Une fermeture est un triplet que nous noterons *fermeture*(n -> e, E_c) où n est un nom : l'*argument formel* de la fonction, e est une expression : le *corps* de la fonction et E_c est l'environnement dans lequel la fonction a été construite et dans lequel sa valeur devra être calculée.

Le type d'une fonction est noté t_1 -> t_2 où t_1 et t_2 sont des expressions de type qui dénotent respectivement le type de départ et le type d'arrivée de la fonction. Le symbole -> est un constructeur de type.

La construction d'une fonction est une expression qui a la forme suivante :

`function` *x* -> *e*

où *x* est un identificateur et *e* est une expression. La valeur de cette expression est la fermeture :

fermeture(x -> e, *Env_courant)*

La fonction suivante, par exemple, calcule le carré de son argument :

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

Une fermeture n'étant pas une valeur affichable, elle est symbolisée par `<fun>`.

Nous appellerons *expression fonctionnelle*, une expression dont la valeur est une fonction.

Une fonction est destinée à être appliquée. Soit *f* une expression fonctionnelle de type t_1 -> t_2 et *e* une expression de type t_1. L'application de *f* à *e* est une expression de type t_2, ayant la forme suivante :

f e

(il suffit tout simplement de juxtaposer la fonction et son argument). La valeur de l'expression *e* est l'*argument effectif* de la fonction *f*. Par exemple :

```
#(function x -> x * x) 2;;
- : int = 4
```

La fonction construite ci-dessus est anonyme. Comme toute valeur, une fonction peut être nommée par une définition globale ou locale. Par exemple :

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

#let carre = (function x -> x * x) in carre (carre 5);;
- : int = 625
```

3.1.1 Typage d'une fonction

Le typage d'une fonction n'est pas trivial car le type de son argument n'est pas déclaré dans l'expression qui la construit. Il peut être réalisé par la procédure suivante dont l'automatisation nécessite la mise en œuvre de techniques de démonstration automatique.

TYPE D'UNE FONCTION. Soit *Env* un environnement et `function` *x* `->` *e* une fonction à typer dans cet environnement. On désigne par t_1 le type de *x* (type de départ de la fonction) et par t_2 le type de *e* (type d'arrivée de la fonction). Une analyse du type de chaque sous-expression qui compose le corps *e*, permet d'établir un ensemble de contraintes sur t_1 et t_2. Deux cas peuvent alors se présenter :

1) Les contraintes sont compatibles et déterminent complètement t_1 et t_2, c'est à dire qu'il existe deux types Caml *a* et *b* tel que $t_1 = a$ et $t_2 = b$. On a alors *type*(`function` *x* `->` *e*, *Env*) = *a* `->` *b*.

2) Les contraintes sont incompatibles. Cela signifie qu'il y a une erreur de typage.

Considérons, par exemple, la fonction `function x -> x * x`. Soit t_1 le type de `x` et t_2 le type de `x * x`. Puisque `*` est un opérateur sur les entiers qui produit un entier, l'ensemble des contraintes est {t_1 = `int`, t_2 = `int`}. t_1 et t_2 sont complètement déterminés. Le type de cette fonction est donc `int -> int`. Vérifions le :

```
#(function x -> x * x);;
- : int -> int = <fun>
```

3.1.2 Evaluation d'une construction de fonction

Les variables (ou noms de valeur) apparaissant dans le corps d'une fonction peuvent être classées en deux catégories : l'argument formel de la fonction est une *variable liée*, les autres sont des *variables libres*. Par exemple, dans l'expression fonctionnelle :

```
function r -> pi *. r *. r
```

(où `*.` est la multiplication des flottants) `r` est liée et `pi` est libre. Lors de l'application d'une fonction, l'argument formel est lié à la valeur de son argument effectif. Pour les arguments libres deux choix sont possibles. On peut les lier à la valeur qu'ils avaient dans l'environnement de construction de la fonction ou bien, à la valeur qu'ils ont dans son environnement d'application. On parle de langage à *portée statique* dans le premier cas et de langage à *portée dynamique* dans le second.

Caml est un langage à portée statique. Les langages à portée statique assurent la transparence référentielle : quelque soit le contexte où elle apparaît une fonction fournit la même valeur lorsque est appliquée au même argument. Montrons le sur l'exemple suivant :

```
#let pi = 3.14;;
pi : float = 3.14

#let surface_cercle = function r -> pi *. r *. r;;
surface_cercle : float -> float = <fun>

#surface_cercle 10.;;
- : float = 314

#let pi = 3.1416;;
pi : float = 3.1416

#surface_cercle 10.;;
- : float = 314
```

On remarque que la redéfinition de `pi` n'a pas modifié la valeur de l'application de la fonction `surface_cercle` à `10`. Si Caml avait été à portée dynamique la deuxième application aurait eue la valeur `314.16`.

Il faut donc que l'environnement dans lequel une fonction a été construite avec le corps de cette fonction. C'est pourquoi, en Caml, une fonction est une fermeture qui réunit l'argument formel de la fonction, son corps et l'environnement dans lequel elle a été définie. La règle d'évaluation d'une définition de fonction est donc la suivante :

VALEUR D'UNE CONSTRUCTION DE FONCTION. Si *Env* est un environnement et `function` *x* `->` *e* est une fonction valide dans *Env*, alors :

val(`function` *x* `->` e, *Env*) = *fermeture*(*x* `->` *e*, *Env*).

3.1.3 Type et valeur d'une application de fonction

Les règles de typage et d'évaluation d'une application de fonction sont les suivantes :

TYPE ET VALEUR D'UNE APPLICATION DE FONCTION. Si *Env* est un environnement et si e_1, e_2 sont des expressions telles que *type*(e_1, *Env*) = t_1 `->` t_2 et *type*(e_2, *Env*) = t_1 et *val*(e_1, *Env*) = *fermeture*(*x* `->` *e*, Env_c), alors :

type(e_1 e_2, *Env*) = t_2,

val(e_1 e_2, *Env*) = *val*(*e*, $Env_c \oplus$ {x = *val*(e_2, *Env*)}).

Considérons, par exemple, l'application `(function x -> x * x) 2`. On a :

type(`(function x -> x * x) 2`, *Env*) = `int`, car *type*(`function x -> x * x`, *Env*) = `int -> int` et *type*(`2`, *Env*) = `int`.

val(`(function x -> x * x) 2`, *Env*) = `4`, car *val*(`(function x -> x * x)`, *Env*) = *fermeture*(`x -> x * x`, *Env*), *val*(`2`, *Env*) = `2` et *val*(`x * x`, *Env* $\oplus$ {`x = 2`}) = `4`

Vérifions le :

```
#(function x -> x * x) 2;;
- : int = 4
```

3.2 Fonctions à plusieurs arguments

Considérons la fonction à deux arguments *divisible_par*, qui appliquée à deux entiers *x* et *y*, indique si *y* est divisible par *x*. En Caml, cette fonction sera construite comme une fonction qui appliquée à *x* fournira une fonction qui appliquée à *y* fournira le résultat cherché :

```
#let divisible_par =
   function x -> function y -> (y mod x) = 0;;
divisible_par : int -> int -> bool = <fun>

#divisible_par 4 5;;
- : bool = false

#divisible_par 2 4;;
- : bool = true
```

(la fonction prédéfinie `mod` calcule le reste de la division de deux entiers).

Une fonction à plusieurs arguments est donc traitée en Caml comme une suite de fonctions à un seul argument. En Caml, l'application est associative à gauche et le constructeur de type `->` est associatif à droite. On a :

$f\ x_1\ x_2\ \ldots\ x_n \equiv (\ldots((f\ x_1)\ x_2)\ldots)\ x_n$

$t_1 \text{ -> } (t_2 \text{ -> } \ldots(t_n \text{ -> } t)\ldots) \text{ -> } t \equiv t_1 \text{ -> } \ldots \text{ -> } t_n \text{ -> } t$

Soit f une fonction de type t_1 `->` $\ldots$ t_k `->` $\ldots$ t_n `->` t et $x_1, \ldots, x_k, \ldots, x_n$ des valeurs de type respectifs $t_1, \ldots, t_k, \ldots, t_n$. Alors :

- $f\ x_1\ \ldots\ x_n$ est une *application totale* de f qui a pour type t,
- $f\ x_1\ \ldots\ x_k$ est une *application partielle* de f qui a pour type t_{k+1} `->` $\ldots$ `->` t_n `->` t.

Par exemple, l'expression `divisible_par 2 4` est une application totale de la fonction `divisible_par` et `divisible_par 2` en est une application partielle :

```
#divisible_par 2 4;;
- : bool = true

#divisible_par 2;;
- : int -> bool = <fun>
```

La fonction `divisible_par 2` indique si un entier est pair. On peut lui donner le nom `est_pair` :

```
#let est_pair = divisible_par 2;;
est_pair : int -> bool = <fun>

#est_pair 12;;
- : bool = true

#est_pair 17;;
- : bool = false
```

Une fonction à n arguments construite comme une suite d'applications partielles est dite *curryfiée*. Ce terme a été adopté en hommage au logicien Haskell Curry.

Les opérateurs prédéfinis infixés tels que `+ - * /` sont eux mêmes des fonctions à deux arguments. Ils peuvent donc faire l'objet d'applications partielles. Cela nécessite de pouvoir les écrire en position préfixe, ce qui est possible en les faisant précéder du mot-clé `prefix`. En voici un exemple classique :

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

#let successeur = (prefix +) 1;;
successeur : int -> int = <fun>

#successeur 9;;
- : int = 10
```

3.3 Facilités d'écriture

Les deux facilités d'écriture suivantes peuvent rendre plus lisibles la construction d'une fonction à plusieurs arguments ou la définition d'un nom de fonction :

`fun` x_1 … x_n `->` $e \equiv$ `function` x_1 `->` … `function` x_n `->` e

`let` n x_1 … x_k `=` e`;;` $\equiv$ `let` n `=` `function` x_1 `->` … `function` x_k `->` e`;;`

Par exemple :

```
#(fun x y -> (y mod x) = 0) 3 9;;
- : bool = true

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

#let divisible_par x y = (y mod x) = 0;;
divisible_par : int -> int -> bool = <fun>
```

On peut indiquer à Caml, par la directive `#infix` *n*, qu'une fonction de nom *n* s'appliquera en position infixe (une directive est une instruction placée dans un programme à destination du compilateur). Par exemple :

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

##infix "et";;

#2 et 2;;
- : int = 4

#4 et 4;;
- : int = 8
```

(« …huit et huit font seize… Répétez ! dit le maître … »). Pour pouvoir appliquer à nouveau la fonction en position préfixe, il faut envoyer à Caml la directive `#uninfix`.

3.4 Fonctions récursives

Considérons la session Caml suivante :

```
#let fac n = if n = 1 then 1 else n * fac (n - 1);;
> Toplevel input:
>let fac n = if n = 1 then 1 else n * fac (n - 1);;
>                                     ^^^
> Variable fac is unbound.
```

Que s'est il passé ? Une erreur a été signalée car tout nom apparaissant dans une expression doit appartenir à l'environnement dans lequel cette expression est typée puis évaluée. Ce n'est pas le cas du nom `fac` qui est en cours de définition. Il aurait fallu écrire :

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

Le mot-clé `rec` indique à Caml que la définition de `fac` est récursive et donc que le nom `fac` sera rencontré dans sa définition. Nous appellerons *définition récursive* une telle définition.

Pour que les règles de typage et d'évaluation d'une définition de fonction données au §2.1 soient applicables aux fonctions récursives, il faut que l'environnement de définition soit étendu par une liaison entre le nom de la fonction récursive, son type et sa valeur. Ces deux derniers n'étant pas connus à ce moment là, ils seront chacun désignés par une variable.

L'environnement ajouté par une définition d'un nom de fonction récursive peut être calculée par la règle suivante :

ENVIRONNEMENT AJOUTE PAR UNE DEFINITION RECURSIVE. Si *Env* un environnement, si *n* est un nom de valeur et si `function` *x* `->` *e* est une construction de fonction, alors :

Env_def(`let rec` *n* `=` `function` *x* `->` *e*, *Env*) = {*n* : *t* = *f*},

où *t* = *type*(`function` *x* `->` *e*, *Env* ⊕ {*n* : *t* }) et *f* = *fermeture*(*x* `->` *e*, *Env* ⊕ {*n* = *f*}).

Illustrons cette règle en calculant l'environnement ajouté par la définition de la fonction `fac` dans l'environnement Env_d. D'après la règle ci-dessus cet environnement est égal à {`fac` : *t* = *f*} où *t* = *type*(`let rec fac n = if n = 1 then 1 else n * fac (n - 1)`, Env_d ⊕ {`fac` : *t* }) et *f* = *fermeture*(*x* `->` *e*, Env_d ⊕ {`fac` = *f*}).

Rappelons au préalable que `let rec fac n = if n = 1 then 1 else n * fac (n - 1)` est une écriture simplifiée équivalente à `let rec fac = function n -> if n = 1 then 1 else n * fac (n - 1)`.

Il faut d'abord calculer *t* = *type*(`function n -> if n = 1 then 1 else n * fac (n - 1)`, *Env* ⊕ {`fac` : *t*}}). Pour cela on applique la règle de typage d'une fonction (cf. ci-dessus § 2.1.1) :

- 1) Soit t_1 le type de `n` et t_2 le type de l'expression `if n = 1 then 1 else n * fac (n - 1)`. Le système de contraintes initial est donc {*t* = t_1 `->` t_2}.
- 2) Soit t_3 le type de l'expression `n = 1` et t_4 celui de l'expression `n * fac (n - 1)`. Par définition de l'alternative et sachant que le type de l'expression `1` est `int`, on a les contraintes suivantes t_3 = `bool` et t_2 = t_4 = `int`. Le système de contraintes devient {*t* = t_1 `-> int`, t_3 = `bool`, t_2 = t_4 = `int`}.
- 3) L'expression `n = 1` est équivalente à `(prefix =) n 1`. L'opérateur `(prefix =)` est de type `int -> int -> bool`. Sachant que le type de `n` est t_1 et que celui de `1` est `int`, on en déduit les contraintes t_1 = `int` et t_3 = `bool`. Le système de contraintes devient {*t* = `int -> int`, t_3 = `bool`, t_1 = t_2 = t_4 = `int`}.
- 4) L'expression `n * fac (n - 1)` est équivalente à `(prefix *) n (fac (n - 1))`. Soit t_5 le type de l'expression `(fac (n - 1))`. Sachant que l'opérateur `(prefix *)` est de type `int -> int -> int` et que le type de `n` est t_1, on en déduit les contraintes t_1 = `int` (déjà présente), t_5 = `int` et t_4 = `int` (déjà présente). Le système de contraintes devient {*t* = `int -> int`, t_3 = `bool`, t_1 = t_2 = t_4 = t_5 = `int` }.
- 5) L'expression `(fac (n - 1))` est équivalente à `fac (n - 1)`. Soit t_6 le type de `(n - 1)`. Sachant que type de `fac` est égal à *t* qui est égal à `int -> int`, on en déduit les nouvelle contraintes t_6 = `int` et t_5 = `int` (déjà présente). Le système de contraintes devient {*t* = `int -> int`, t_3 = `bool`, t_2 = `int`, t_4 = `int`, t_5 = `int`, t_6 = `int`}.
- 6) L'expression `(n - 1)` est équivalente à `(prefix -) n 1`. Sachant que type de l'opérateur prédéfini `(prefix -)` est égal à `int -> int -> int`, que le type de `n` est t_1 et que celui de `1` est `int` on en déduit les contraintes t_1 = `int` et t_6 = `int` (déjà présentes). Le système de contraintes reste donc inchangé.

Le système de contraintes est sans contradiction et contient la contrainte *t* = `int -> int`. On a donc :

Env_def(`let rec fac = …`, *Env*) = {`fac` : `int -> int` = *f*}
où *f* = *fermeture*(`n -> if n = 1 then 1 else …`, *Env* ⊕ {`fac` = `f`})

L'application d'une fonction récursive se déroule de la même façon que celle d'une fonction non récursive. Montrons le en évaluant `fac 2` dans un environnement *Env* contenant la liaison `fac` : `int -> int` = *f* où *f* = *fermeture*(`n -> if n = 1 then 1 else n * fac (n - 1)`, $Env_d \oplus \{$`fac` = *f*$\}$). On a :

val(`fac 2`, *Env*)
= *val*(`if n = 1 then 1 else n * fac (n - 1)`, $Env_d \oplus \{$`fac` = *f*, `n` = `2`$\}$)
= *val*(`2 * fac 1`, $Env_d \oplus \{$`fac` = *f*, `n` = `2`$\}$)
= `2 *` *val*(`fac 1`, $Env_d \oplus \{$`fac` = *f*, `n` = `2`$\}$)
= `2 *` *val*(`if n = 1 then 1` …, $Env_d \oplus \{$`fac` = *f*, `n` = `1`$\}$)
= `2 * 1`
= `2`

3.5 Fonctions mutuellement récursives

Plusieurs fonctions sont dites mutuellement récursives lorsqu'elles s'appellent de façon croisée c.-à-d. lorsque le nom de l'une peut apparaître dans le corps de l'autre et inversement. La définition de *n* fonctions mutuellement récursives de noms f_1, …, f_n et de corps respectifs e_1, …, e_n est une phrase qui a la forme suivante :

`let rec` f_1 `=` e_1 `and ... and` f_n `=` e_n `;;`

que nous appellerons *définition mutuellement récursive*.

La règle calculant l'environnement ajouté par une définition récursive (§2.3) se généralise aisément à une définition mutuellement récursives. La fermeture obtenue comme valeur de chaque fonction f_i enfermera les liaisons associées aux noms f_1, …, f_n.

Un bon exemple est constitué par les suites de Fibonacci qui sont définies par les équations suivantes :

$u_0 = 1$; $v_0 = 1$; $u_n = u_{n-1} + 2 * v_{n-1}$; $v_n = 3 * u_{n-1} + v_{n-1}$

En Caml elles pourront être définies de la façon suivante :

```
#let rec
     u = function n -> if (n = 0) then 1 else u(n - 1) + 2 * v(n - 1)
 and v = function n ->
            if (n = 0) then 1 else 3 * u(n - 1) + v(n - 1);;
u : int -> int = <fun>
v : int -> int = <fun>

#u 3;;
- : int = 37

#v 3;;
- : int = 46
```

4
Valeurs structurées, polymorphisme et filtrage

Une valeur structurée est formée par assemblage de valeurs qui sont soit de même type, une liste par exemple, soit de types différents, un n-uplet par exemple.

Caml autorise la définition de fonctions polymorphes dont le type de l'argument est variable. Ce *polymorphisme* est particulièrement pratique pour construire des fonctions génériques sur des valeurs structurées : par exemple, le calcul de la longueur d'une liste qui ne nécessite pas la connaissance du type des éléments de cette liste ou bien l'extraction du i^{e} élément d'un n-uplet, qui ne nécessite pas la connaissance du type de cet élément.

Une technique classique d'accès aux éléments d'une valeur structurée est le *filtrage* qui consiste à faire passer cette valeur au travers d'une structure de même forme, appelée *filtre*, dans laquelle chaque composante à accéder est désignée par un nom auquel elle sera liée. Par exemple, le filtrage de la paire (1, « un ») par le filtre (*c*, *m*) produira les deux liaisons : $c = 1$ et $m =$ « un ».

Nous étudierons tout d'abord les n-uplets, ce qui nous permettra d'introduire le polymorphisme puis le filtrage et nous terminerons par l'étude des listes.

4.1 N-uplets

Rappelons que le *produit cartésien* de n ensembles $E_1, \ldots, E_n$, habituellement noté $E_1 \times \ldots \times E_n$, est l'ensemble des n-uplets $(e_1, \ldots, e_n)$ tels que, pour tout i de 1 à n, e_i appartient à E_i. Par exemple, le produit cartésien des ensembles {Jean} et {Pierre, Paul, Marie} est l'ensemble {(Jean, Pierre), (Jean, Paul), (Jean, Marie)}.

En Caml le produit cartésien de n types $t_1, \ldots, t_n$ est le type t_1 `*` … `*` t_n (une expression de type) dont les éléments sont les n-uplets $v_1, \ldots, v_n$ tels que pour tout i de 1 à n, v_i est une valeur de type t_i. On a donc :

$$\text{ext}(t_1 \texttt{ * } \ldots \texttt{ * } t_n) = \{v_1, \ldots, v_n \mid v_1 \in \text{ext}(t_1), \ldots, v_n \in \text{ext}(t_n)\}$$

Par exemple `"Jean", 35, true` est un n-uplet de type `string * int * bool`. On notera que les parenthèses ne sont pas obligatoires pour encadrer un n-uplet, elles sont cependant conseillées pour améliorer la lisibilité.

Les n-uplets à deux éléments sont appelés des *paires*.

4.1.1 Construction d'un n-uplet

Pour construire un n-uplet, Caml offre le constructeur multi-infixé « , ». Par exemple :

```
#("Jean" ^ "-Paul", 25 + 1, true);;
- : string * int * bool = "Jean-Paul", 26, true
```

La sémantique de ce constructeur est la suivante :

CONSTRUCTION D'UN N-UPLET. Si *Env* est un environnement et e_1, …, e_n sont des expressions telles que pour tout *i* de 1 à *n*, *type*(e_i, *Env*) = t_i et *val*(e_i, *Env*) = v_i, alors e_1, …, e_n est une expression telle que :

type(e_1, …, e_n, *Env*) = t_1 * … * t_n,

val(e_1, …, e_n, *Env*) = v_1, …, v_n.

Attention ! l'expression « e_1, e_2 » construit une paire, l'expression « e_1, e_2, e_3 » construit un triplet, et l'expression « e_1, e_2, e_3, …, e_n » construit un n-uplet. Ce ne sont pas des valeurs de même type. Les constructeurs `*` et `,` ne sont donc pas associatifs comme le montre le programme suivant :

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

La première expression a pour valeur une paire formée d'une paire d'entiers et d'un entier, la seconde une paire formée d'un entier et d'une paire d'entiers et la troisième un triplet d'entiers.

4.1.2 Egalité structurelle de n-uplets

En Caml l'égalité n'est définie que sur des valeurs de même type. Deux n-uplets de même type sont structurellement égaux si leurs éléments de même rang sont structurellement égaux.

4.1.3 Accès aux éléments d'un n-uplet

L'accès aux éléments d'un n-uplet se fait normalement par filtrage comme nous le verrons au §3.3. Cependant, dans le cas des paires, deux fonctions prédéfinies sont disponibles `fst` et `snd` qui permettent d'accéder respectivement à la première et à la seconde composante. En voici un exemple :

```
#fst ("un", "deux");;
- : string = "un"

#snd ("un", "deux");;
- : string = "deux"
```

Ces deux fonctions étant polymorphes nous les étudierons au §4.2 consacré au polymorphisme

4.2 Polymorphisme

En Caml il existe un type particulier appelé *paramètre de type* (ou *variable de type*) noté `'`*i* où *i* est un identificateur (en général une lettre minuscule `a`, `b`, …). Si une valeur *v* est de type « paramètre de type » alors quelque soit le type *t*, *v* peut être considérée comme étant de type *t*.

Un *type polymorphe* (ou *type paramétré*) est un type dont l'expression contient au moins un paramètre de type. Par exemple `'a` et `(int * 'a)` sont des types polymorphes. Le premier a

pour extension l'ensemble des valeurs Caml et le deuxième, l'ensemble des paires d'un entier et d'une valeur de type quelconque.

Une *fonction polymorphe* est une fonction dont le type de départ est polymorphe. Un exemple classique est la fonction `id` qui appliquée à une valeur retourne une valeur égale :

```
#let id = function x -> x;;
id : 'a -> 'a = <fun>

#id 1;;
- : int = 1

#id "un";;
- : string = "un"
```

Un autre exemple est l'opérateur prédéfini d'égalité :

```
#prefix =;;
- : 'a -> 'a -> bool = <fun>
```

On notera que les arguments doivent être du même type, comme le prouve à contrario le programme suivant :

```
#1 = true;;
> Toplevel input:
>1 = true;;
>    ^^^^
> Expression of type bool
> cannot be used with type int
```

Les fonctions d'accès aux composantes d'une valeur structurée sont polymorphes, de façon à les rendre indépendantes des types de ces composantes. C'est le cas notamment des fonctions d'accès au premier et au deuxième élément d'une paire que nous avons présentées au §3.1.3 :

```
#fst;;
- : 'a * 'b -> 'a = <fun>

#snd;;
- : 'a * 'b -> 'b = <fun>
```

4.3 Filtrage

Le filtrage est une technique très utilisée dans les langages de programmation fonctionnels ou logiques, car elle améliore énormément la lisibilité des programmes.

Un filtre visualise la structure d'une valeur et nomme certaines de ses composantes. Le filtrage est l'opération consistant à faire passer une valeur au travers d'un filtre. Si elle réussit, elle produit un environnement qui lie les noms (on dit aussi les variables) du filtre aux composantes correspondantes de la valeur.

Commençons par un exemple :

```
let (nom, _, age, ((ville, "France") as adresse)) =
   ("Dupont", "Jean", 12, ("Marseille", "France"));;
ville : string = "Marseille"
adresse : string * string = "Marseille", "France"
age : int = 12
nom : string = "Dupont"
```

La construction `(nom, _, age, (ville, "France") as adresse)` est un filtre. Le filtrage par ce filtre du quadruplet `("Dupont", "Jean", 12, ("Marseille", "France"))` déclenche :

- le filtrage de `"Dupont"` par le filtre `nom` qui produit la liaison `nom : string = "Dupont"`,
- le filtrage de `"Jean"` par le filtre _ qui réussit car il filtre n'importe quelle valeur,
- le filtrage de `12` par le filtre `age` qui produit la liaison `age : int = 12`,
- le filtrage de la paire `("Marseille", "France")` par le filtre `(ville, "France") as adresse` qui déclenche :
  - le filtrage de `"Marseille"` par le filtre `ville` qui produit la liaison `ville : string = "Marseille"`,
  - le filtrage de `"France"` par le filtre `"France"` qui réussit,
- puis produit la liaison `adresse : string * string = "Marseille", "France"`.

Soit *F* un filtre et *v* une valeur, nous noterons *env_filtré*(*F*, *v*) l'environnement produit en liant les variables de *F* aux composantes correspondantes de *v*. La fonction *env_filtré* définit la sémantique du filtrage.

Tout filtre a un type qui est celui des valeurs qu'il filtre.

Nous considérerons tout d'abord un sous-ensemble des filtres de Caml :

- Le filtre _ filtre toute valeur. Si *v* est une valeur on a *env_filtré*(_, *v*) = {}.
- Un identificateur filtre toute valeur. Si *i* est un identificateur et *v* une valeur de type *t* alors a *env_filtré*(*i*, *v*) = {*i* : *t* = *v*}. *i* devient le nom de la valeur *v*.
- Une valeur filtre toute valeur qui lui est égale. Si *v* est une valeur alors *env_filtré*(*v*, *v*) = {}.
- Si F_1, …, F_n sont des filtres alors F_1, …, F_n filtre tout n-uplet v_1, …, v_n tel que pour tout *i* de 1 à n, F_i filtre v_i. On a : *env_filtré*(F_1, …, F_n, v_1, …, v_n) = *env_filtré*(F_1, v_1) ⊕ … ⊕ *env_filtré*(F_n, v_n).
- Si *F* est un filtre et *n* est un identificateur, alors *F* `as` *n* est un filtre qui lie n à la valeur filtrée. Si *v* est une valeur de type *t* et *F* est un filtre de même type, alors on a : *env_filtré*(*F* `as` *n*, *v*) = *env_filtré*(*F*, *v*) ⊕ {*n* : *t* = *v*}.
- Si *F* est un filtre alors `(`*F*`)` est un filtre équivalent à *F*.

Nous verrons par la suite, qu'à chaque type structuré est associé un filtre dont la syntaxe est identique à celle du constructeur des valeurs de ce type.

4.3.1 Noms définis par filtrage

Les filtres peuvent être utilisés dans les clauses `let` des définitions globales ou locales. Les noms définis sont les identificateurs présents dans les filtres et les liaisons sont obtenues par filtrage.

Soit F_1, …, F_n des filtres, une définition globale a la forme générale suivante :

`let` F_1 `=` e_1 `and ... and` F_n `=` e_n `;;`

et une expression à définitions locales a la forme générale suivante :

`let` F_1 `=` e_1 `and ... and` F_n `=` e_n `in` e`;;`

Par exemple :

```
#let (lon, lar, haut) = (15, 5, 20);;
haut : int = 20
lar : int = 5
lon : int = 15
```

```
#let (lon, lar, haut) = (15, 5, 20) in lon * lar * haut;;
- : int = 1500
```

On notera que ces nouvelles clauses `let` généralisent celles étudiées au §1.5, puisqu'un identificateur est un cas particulier de filtre. Les règles d'évaluation et de typage sont elles-mêmes généralisées de la façon suivante :

ENVIRONNEMENT AJOUTE PAR UNE DEFINITION NON RECURSIVE. Si *Env* est un environnement et pour tout *i* de 1 à *n*, F_i est un filtre et e_i est une expression valide dans *Env*, alors :

Env_def(`let` F_1 `=` e_1 `and` … `and` F_n `=` e_n, *Env*) =

env_filtré(F_1, val(e_1, Env)) ⊕ … ⊕ env_filtré(F_n, val(e_n, Env)).

TYPE ET VALEUR D'UNE EXPRESSION A DEFINITIONS LOCALES. Si *Env* est un environnement et pour tout *i* de 1 à *n*, F_i est un filtre et e_i est une expression valide dans *Env*, et *e* est une expression valide dans *Env* ⊕ *Env_def*(`let` F_1 = e_1 `and` … `and` F_n = e_n, *Env*), alors :

type(`let` F_1 = e_1 `and` … `and` F_n = e_n `in` *e*, *Env*) =

type(*e*, *Env* ⊕ *Env_def*(`let` F_1 = e_1 `and` … `and` F_n = e_n, *Env*)),

val(`let` F_1 = e_1 `and` … `and` F_n = e_n `in` *e*, *Env*) =

val(e, *Env* ⊕ *Env_def*(`let` F_1 = e_1 `and` … `and` F_n = e_n, *Env*)).

4.3.2 Fonctions définies par filtrage

Le filtrage fournit un moyen élégant de définir une fonction : la *définition par cas*, qui généralise le mode de définition que nous avons présenté au chapitre 2. Si F_1, …, F_n sont des filtres et e_1, …, e_n sont des expressions, l'expression :

`function` F_1 `->` e_1 `|` … `|` F_n `->` e_n

définit une fonction qui appliquée à un argument *x* rend la valeur de l'expression e_i associée au premier filtre F_i qui filtre *x*, en essayant successivement les filtres F_1, …, F_n dans cet ordre. Pour des raisons de symétrie, on peut placer une `|` devant le premier cas. Par exemple :

```
#let nul_ou_non_nul =
   function
      | 0 -> "nul"
      | _ -> "non nul";;
nul_ou_non_nul : int -> string = <fun>

#nul_ou_non_nul 0;;
- : string = "nul"

#nul_ou_non_nul 1;;
- : string = "non nul"
```

On remarquera que le mode de définition que nous avons présenté au chapitre 2 (`function` *x* `->` *e*) n'est qu'un cas particulier de définition par cas puisqu'un nom de variable est un filtre.

Lorsque la définition ne comporte qu'un seul cas on peut écrire

`let` *n* *F* `=` *e*

au lieu de

`let` *n* `=` `function` *F* `->` *e*

Par exemple :

```
#let moyenne (x, y) = (x +. y) /. 2.;;
moyenne : float * float -> float = <fun>
```

Les règles de typage et d'évaluation d'une définition de fonction énoncées au §2.1, sont généralisées de la façon suivante :

TYPE D'UNE DEFINITION DE FONCTION. Soit *Env* un environnement et `function` F_1 `->` e_1 `|` … `|` F_n `->` e_n une fonction à typer dans cet environnement. On désigne par *f* la fonction définie par cette expression, par *t* et *t*' les types de départ et d'arrivée de *f*, par t_1, …, t_n les types de F_1, …, F_n et par t'_1, …, t'_n, les types de e_1, …, e_n. On pose les contraintes $t = type(F_1, Env) = \ldots = type(F_n, Env)$ et $t' = type(e_1, Env) = \ldots = type(e_n, Env)$. Une analyse du type de chaque sous-expression qui compose les expressions e_1, … e_n, permet d'établir un ensemble de contraintes sur *t* et *t*'. Trois cas peuvent alors se présenter :

1) Les contraintes sont compatibles et déterminent complètement *t* et *t*'. On a alors $type(f, Env) = t \text{ -> } t'$.

2) Les contraintes sont compatibles mais ne déterminent pas *t* et/ou *t*'. Ils resteront indéterminés et seront remplacés par des paramètres de type, en général `'a`, `'b`, …

3) Les contraintes sont incompatibles. Cela signifie que la fonction est mal définie.

VALEUR D'UNE DEFINITION DE FONCTION. Si *Env* est un environnement et `function` F_1 `->` e_1 `|` … `|` F_n `->` e_n est une définition de fonction valide dans cet environnement alors :

$val($`function` F_1 `->` e_1 `|` … `|` F_n `->` $e_n, Env) = fermeture(F_1 \text{ -> } e_1 \mid \ldots \mid F_n \text{ -> } e_n, Env)$.

VALEUR D'UNE APPLICATION DE FONCTION. Soit *Env* un environnement et soit *e* et *e*' deux expressions telles que $val(e, Env) = fermeture(F_1 \text{ -> } e_1 \mid \ldots \mid F_n \text{ -> } e_n, Env_d)$ et $val(e', Env) = a$. On recherche le premier des filtres F_1, …, F_n, parcourus dans cet ordre, qui filtre *a*. S'il en existe un, soit *i* son rang, alors :

$$val(e\ e', Env) = val(e_i, Env_d \oplus env\_filtré(F_i, a))$$

sinon l'application est impossible.

Une opération classique, dans la plupart des langages de programmation, est la sélection qui choisit l'action à réaliser en fonction de la valeur d'une expression (par exemple l'instruction « switch » de C). En Caml, cette opération se traduit par l'application d'une fonction définie par cas :

`(function` F_1 `->` e_1 `|` … `|` F_n `->` e_n`)` e

qui s'écrit aussi :

`match` e `with` F_1 `->` e_1 `|` … `|` F_n `->` e_n

La valeur calculée dépend de la valeur de *e* et les filtres F_1, …, F_n jouent le rôle de sélecteur.

Par exemple la fonction suivante donne la traduction en anglais des 4 saisons :

```
#(match x with
   | "printemps" -> "spring"
   |       "été" -> "summer"
   |   "automne" -> "autumn"
```

```
    |     "hiver" -> "winter"
    |           _ -> "Ce n'est pas une saison !") automne;;
- : string = "autumn"
```

La construction `if...then...else` que nous avons étudiée au §1.4 n'est pas une construction de base de Caml. Elle est un cas particulier d'application d'une fonction de type de départ `bool`. On a :

`if` e_1 `then` e_2 `else` e_3
≡ `(function true ->` e_2 `| false ->` e_3`)` e_1
≡ `match` e_1 `with true ->` e_2 `| false ->` e_3

4.3.3 Exhaustivité du filtrage

Lorsqu'une fonction est définie par filtrage, toutes les occurrences du type de départ de cette fonction doivent être prises en compte. Si l'on ne le fait pas, la définition sera acceptée mais le message suivant sera affiché :

```
> Warning: pattern matching is not exhaustive.
```

Par exemple :

```
#let un_deux_trois = function
      | 1 -> "un"
      | 2 -> "deux"
      | 3 -> "trois";;
Toplevel input:
>.....................function
>       | 1 -> "un"
>       | 2 -> "deux"
>       | 3 -> "trois"..
Warning: this matching is not exhaustive.
un_deux_trois : int -> string = <fun>
```

On veut définir une fonction applicable à tout entier mais seuls les entiers `1`, `2` et `3` ont été considérés. Il faut donc définir la valeur de cette fonction pour les autres entiers. Par exemple :

```
#let un_deux_trois = function
      | 1 -> "un"
      | 2 -> "deux"
      | 3 -> "trois"
      | _ -> "pas un, pas deux, pas trois";;
un_deux_trois : int -> string = <fun>
```

4.4 Listes

Rappelons qu'une liste est une suite finie, éventuellement vide, d'éléments. Il est habituel de construire une nouvelle liste en ajoutant un élément *e* en tête d'une liste *l* : *e* est la *tête* de la nouvelle liste et *l* sa *queue*. Par exemple :

cons(printemps, *liste*(été, automne, hiver)) = *liste*(printemps, été, automne, hiver)

tête(*liste*(printemps, été, automne, hiver)) = printemps

queue(*liste*(printemps, été, automne, hiver)) = *liste*(été, automne, hiver)

Inversement on parcourt et l'on traite les éléments d'une liste par une opération récursive qui consiste à accéder et à traiter la tête de la liste puis à relancer l'opération sur la queue de la liste jusqu'à atteindre une liste vide.

En Caml les listes sont homogènes, ce qui signifie qu'elles sont constituées de valeurs de même type. Le type *t* `list` a pour éléments les listes de 0, 1, 2, … valeurs de type *t*. On a donc :

ext(*t* `list`) = `[]` ∪ {`[`v_1`]` | $v_1 \in ext(t)$} ∪ {`[`v_1`;` v_2`]` | $v_1 \in ext(t), v_2 \in ext(t)$} ∪ …

Les éléments d'une liste sont encadrés par des crochets et séparés par un point-virgule. Le symbole `[]` désigne la liste vide. Quelque soit le type *t*, la liste vide appartient au type *t* `list`. La liste vide est donc une valeur polymorphe. Par exemple, les listes `[]` et `[1; 3; 5; 7; 9]` sont de type `int list`.

4.4.1 Construction d'une liste

Le constructeur de liste est l'opérateur infixé et associatif à droite `::` qui construit une liste à partir de sa tête et de sa queue (c'est l'opérateur *cons* de Lisp). Par exemple :

```
#"printemps" :: "été" :: "automne" :: "hiver" :: [];;
- : string list = ["printemps"; "été"; "automne"; "hiver"]

#("un", 1) :: ("deux", 1 + 1) :: ("trois", 2 + 1) :: [];;
- : (string * int) list = ["un", 1; "deux", 2; "trois", 3]
```

La sémantique du constructeur de liste est la suivante :

CONSTRUCTION D'UNE LISTE. Si *Env* est un environnement et si $e_1$et e_2 sont deux expressions telles que *type*(e_1, *Env*) = *t* , *val*(e_1, *Env*) = *v*, *type*(e_2, *Env*) = *t* `list` et *val*(e_2, *Env*) = *l*, alors e_1 `::` e_2 est une expression telle que :

type(e_1 `::` e_2, *Env*) = *t* `list`,

val(e_1 `::` e_2, *Env*) =

`[`*v*`]` si *l* = `[]`,

`[`*v*, v_1, … , v_n `]` si *l* = `[`v_1, … , v_n`]`.

L'équivalence syntaxique suivante permet d'améliorer la lisibilité de la construction d'une liste de longueur connue :

`[`e_1`; … ;` e_n`]` ≡ e_1 `::` … `::` e_n `::` `[]`

Par exemple :

```
#[("un", 0 + 1); ("deux", 1 + 1); ("trois", 2 + 1)];;
- : (string * int) list = ["un", 1; "deux", 2; "trois", 3]
```

4.4.2 Egalité structurelle des listes

Deux listes vides sont structurellement égales. Deux listes non vides de même type, sont structurellement égales si leurs têtes et leurs queues sont structurellement égales.

4.4.3 Tête, queue et concaténation

La tête et la queue d'une liste peuvent être extraites par les fonctions prédéfinies `hd` et `tl` ou bien par filtrage comme nous le verrons ci-dessous. La concaténation de deux listes est réalisée par l'opérateur prédéfini et infixé `@`. Par exemple :

```
#let les_saisons = ["printemps"; "été"; "automne"; "hiver"];;
les_saisons : string list = ["printemps"; "été"; "automne"; "hiver"]

#hd;; (* tête *)
- : 'a list -> 'a = <fun>
#hd les_saisons;;
- : string = "printemps"

#tl;; (* queue *)
- : 'a list -> 'a list = <fun>

#tl les_saisons;;
- : string list = ["été"; "automne"; "hiver"]

#prefix @;; (* concatenation *)
- : 'a list -> 'a list -> 'a list = <fun>

#[hd les_saisons] @ (tl les_saisons);;
- : string list = ["printemps"; "été"; "automne"; "hiver"]
```

4.4.4 Filtrage des listes

Trois filtres sont disponibles pour les listes : le premier pour la liste vide, le second pour accéder à la tête et à la queue d'une liste et le troisième pour accéder aux éléments d'une liste de longueur connue :

- 1) `[]` filtre `[]`. On a *env_filtré*(`[]`, `[]`) = {}.
- 2) F_1 `::` F_2 filtre toute liste l dont la tête t est filtrée par F_1 et la queue q est filtrée par F_2. On a *env_filtré*(F_1 `::` F_2, l) = *env_filtré*(F_1, t) ⊕ *env_filtré*(F_2, q).
- 3) `[`F_1`; …; `F_n`]` filtre toute liste `[`v_1`; … ; `v_n`]` telle que, pour tout i de 1 à n, F_i filtre v_i. On a *env_filtré*(`[`F_1`; …; `F_n`]`, l) = *env_filtré*(F_1, v_1) ⊕ … ⊕ *env_filtré*(F_n, v_n).

Par exemple :

```
#match ["Le"; "langage"; "Caml"] with
   | [] -> "Liste vide !"
   | m::_ -> "Le premier mot est " ^ m;;
- : string = "Le premier mot est Le"

#match les_saisons with
   | [] -> "Y a plus de saisons !"
   | [s1; s2; s3; s4] ->
        "Le " ^ s1 ^ ", l'" ^ s2 ^ ", l'" ^ s3 ^ " et l'" ^ s4;;
- : string = "Le printemps, l'été, l'automne et l'hiver"
```

4.4.5 Exemples de manipulations de listes

Les listes sont associées aux langages fonctionnels depuis la naissance de ceux-ci (le nom du premier des langages fonctionnels, Lisp, est un acronyme pour « List processing »). Un grand nombre d'opérations sur cette structure de données sont devenues des classiques. La plupart sont d'ailleurs prédéfinies en Caml (dans le module `list` de la *bibliothèque de base*). Nous allons en étudier quelques unes afin de mettre en évidence la technique de traitement des listes qui est basée sur la récurrence structurelle :

PRINCIPE DE RECURRENCE STRUCTURELLE SUR LES LISTES. Si une propriété *P* est vraie pour `[]` et si dès que *P* est vraie pour *l* alors *P* est vraie pour *x*::*l*, alors *P* est vraie pour toutes les listes.

Compter les éléments d'une liste

Pour compter les éléments d'une liste on utilise la propriété suivante : le nombre d'éléments d'une liste vide est égal à 0 et le nombre d'éléments d'une liste non vide est égal à 1 plus le nombre d'éléments de la queue de cette liste. Ce qui donne :

```
#let rec compter l =
   match l with
      | [] -> 0
      | _ :: q -> 1 + (compter q);;
compter : 'a list -> int = <fun>

#compter les_saisons;;
- : int = 4
```

Appartenance d'un élément à une liste

Pour vérifier qu'une valeur appartient à une liste on utilise la règle suivante : si cette liste est vide, cette valeur ne lui appartient pas, si elle n'est pas vide cette valeur lui appartient soit si elle est égale à la tête de cette liste, soit si elle appartient à la queue de cette liste. Ce qui donne :

```
#let rec membre v l =
   match l with
      | [] -> false
      | t::q -> if t = v then true else membre v q;;
membre : 'a -> 'a list -> bool = <fun>

#membre "automne" les_saisons;;
- : bool = true

#membre "winter" les_saisons;;
- : bool = false
```

Select…from…where

Le langage SQL a popularisé l'opérateur « select…from…where », qui appliqué à un ensemble de tables : (i) en construit le produit cartésien, (ii) le restreint aux lignes qui vérifient une condition donnée, et enfin (iii) le projette sur certaines colonnes. Nous en donnons ici une version simplifiée, la fonction `select_from_where` qui appliquée à une fonction *f*, à une liste *l* et à un prédicat *p* extrait la liste formée par l'application de la fonction *f* à chaque élément de *l* qui vérifie le prédicat *p*.

Pour définir la fonction `select_from_where` nous utiliserons deux fonctions intermédiaires `filter` et `map`. La fonction `filter` élimine d'une liste les éléments qui ne vérifient pas un prédicat donné. Par exemple :

```
filter [1; 2; 3; 4; 5; 6; 7; 8; 9] est_pair = [2; 4; 6; 8]
```

Elle peut se définir de de la façon suivante :

– le filtrage d'une liste vide produit une liste vide ;
– le filtrage d'une liste *l* de tête *x* par un prédicat *p* produit :
  • la liste ayant pour tête *x* et pour queue le résultat du filtrage de la queue de *l* par *p*, si *x* vérifie *p* (on garde *x*) ;
  • le résultat du filtrage de la queue de *l* par *p*, si *x* ne vérifie pas *p* (on élimine *x*).

La fonction `map` applique une fonction à chaque élément d'une liste. Par exemple :

```
map [1; 2; 3] est_pair = [false; true; false]
```

Elle peut se définir de la façon suivante :

– le « mapping » d'une liste vide produit une liste vide ;
– le « mapping » d'une liste *l* de tête *x* par une fonction *f* produit la liste ayant pour tête *f x* et pour queue le « mapping » de la queue de *l* par *f*.

En utilisant ces deux fonctions on obtient :

```
select_from_where f l p  = map (filter l p) f.
```

D'où la définition complète de `select_from_where` en Caml :

```
let select_from_where f l p =
   let rec
      filter l p =
         match l with
            | [] -> []
            | t::q -> if p t then t::(filter q p) else (filter q p)
   and
      map l f =
         match l with
            | [] -> []
            | t::q -> (f t)::(map q f)
   in
      map (filter l p) f;;
  select_from_where : ('a -> 'b) -> 'a list -> ('a -> bool) -> 'b list =
  <fun>
```

Considérons maintenant une liste d'enfants, où chaque enfant est représenté par son prénom et son âge :

```
#let les_enfants =
   [("Claire", 13); ("Alain", 8); ("Paul", 12); ("Marie", 5)];;
les_enfants : (string * int) list = ["Claire", 13; "Alain", 8; "Paul",
12; "Marie", 5]
```

La requête « Quels sont les prénoms des enfants de plus de 10 ans » peut s'exprimer de la façon suivante :

```
#let prénom = function (p, a) -> p
    and plus_de_10_ans = function (p, a) -> a > 10
 in
    select_from_where prénom les_enfants plus_de_10_ans;;
- : string list = ["Claire"; "Paul"]
```

Remarquons enfin, que la fonction `select_from_where` aurait pu être définie directement de la façon suivante :

```
#let rec select_from_where f l p =
   match l with
      | [] -> []
      | t::q -> if p t then
                   (f t)::(select_from_where f q p)
                else
                   (select_from_where f q p);;
select_from_where : ('a -> 'b) -> 'a list -> ('a -> bool) -> 'b list =
<fun>
```

5
Types définis, types paramétrés

Un langage de programmation évolué doit offrir la possibilité de définir les types nécessaires à la gestion d'une application informatique particulière. Par exemple, la gestion d'une bibliothèque implique la manipulation Par exemple, les types livre, abonné ou emprunt pour une application de gestion de bibliothèque.

Caml permet de construire de nouveaux types par produit ou union de types existants : les premiers sont appelés types *enregistrement* et les seconds, types *union*. Caml permet de plus la définition de types *paramétrés* ou *polymorphes*. Par exemple : le type « pile de valeurs de type quelconque », qui pourra ensuite être instancié selon les besoins, en « pile d'entiers », « piles de caractères », etc.

Nous étudierons successivement les types enregistrements, les types unions et les types paramétrés et nous présenterons de nombreux exemples les utilisant.

5.1 Types enregistrement

Un enregistrement est un ensemble de valeurs nommées. Par exemple, une adresse composée d'une rue, d'un code postal et d'une ville.

Un type enregistrement est défini par la phrase :

`type` *t* `= {`C_1`:` t_1`; …; `C_n`:` t_n`};;`

où *t*, C_1, …, C_n sont des identificateurs et t_1, … t_n sont des expressions de types. Cette phrase définit un type enregistrement de nom *t*, dont les instances sont les enregistrements `{`C_1 `=` v_1`; …;` C_n `=` v_n`}` tels que pour tout *i* de 1 à *n*, v_i est de type t_i. On a donc :

$$ext(t) = \{\{C_1 = v_1; \ldots; C_n = v_n\} \mid v_1 \in ext(t_1), \ldots, v_n \in ext(t_n)\}$$

Les identificateurs C_1, …, C_n sont les noms des *champs* du type *t* et v_1, …, v_n sont les valeurs de ces champs. Les noms de champ C_1, …, C_n sont spécifiques au type *t* et ne peuvent donc pas être des noms de champs d'un autre type enregistrement. Dans un enregistrement l'ordre des champs n'a pas d'importance.

Par exemple :

```
#type enfant = {Prénom: string; Age: int};;
Type enfant defined.
```

Le type `enfant` est un type enregistrement qui a deux champs `Prénom` et `Age`. La valeur `{Age = 12; Prénom = "Amandine"}`, par exemple, est une valeur de type `enfant`.

5.1.1 Construction d'un enregistrement

La construction d'un enregistrement de champs C_1, …, C_n est réalisée par le constructeur `{`C_1 `= ; …;` C_n `= }`.

Par exemple :

```
#{Prénom = "Jean-" ^ "Paul"; Age = 5 + 1};;
- : enfant = {Prénom = "Jean-Paul"; Age = 6}
```

La sémantique du constructeur d'enregistrement est la suivante :

CONSTRUCTION D'UN ENREGISTREMENT. Si *Env* est un environnement contenant la définition `type` t `=` $\{C_1 : t_1; \ldots; C_n : t_n\}$ et si pour tout i de 1 à n, e_i est une expression telle que $type(e_i, Env) = t_i$ alors $\{C_1 : e_1; \ldots; C_n : e_n\}$ est une expression telle que :

$type(\{C_1 : e_1; \ldots; C_n : e_n\}, Env) = t$,

$val(\{C_1 = e_1; \ldots; C_n = e_n\}, Env) = \{C_1 = val(e_1, Env); \ldots; C_n = val(e_n, Env)\}$.

5.1.2 Egalité structurelle

Deux enregistrements de même type sont structurellement égaux si les valeurs des champs de même nom sont structurellement égales.

5.1.3 Accès à la valeur d'un champ

L'accès à la valeur d'un champ est réalisée par l'opérateur « . ». Si $v = \{C_1 = v_1; \ldots; C_n = v_n\}$ alors pour tout i de 1 à n, $v.C_i = v_i$. Par exemple :

```
#let jp = {Prénom = "Jean-Paul"; Age = 6};;
jp : enfant = {Prénom = "Jean-Paul"; Age = 6}

#jp.Age;;
- : int = 6
```

5.1.4 Filtrage

Si t est un type défini par $t = \{C_1 : t_1; \ldots; C_n : t_n\}$ et si, pour tout i de 1 à n, v_i est une valeur de type t_i et F_i est un filtre de type t_i, alors :

$\{C_1 = F_1; \ldots; C_n = F_n\}$

est un filtre de type t tel que :

$env\_filtré(\{C_1 = v_1; \ldots; C_n = v_n\}, \{C_1 = F_1; \ldots; C_n = F_n\}) =$
$\quad env\_filtré(v_1, F_1) \oplus \ldots \oplus env\_filtré(v_n, F_n)$

Par exemple :

```
#let age = function {Prénom = _; Age = a} -> a;;
age : enfant -> int = <fun>

#age jp;;
- : int = 6
```

Puisqu'un nom de champ est attaché à un et un seul type, il est possible, dans un filtre, d'omettre les champs non concernés. La définition de `age` dans l'exemple précédent peut alors s'exprimer par :

```
#let age = function {Age = a} -> a;;
age : enfant -> int = <fun>
```

5.2 Types union

Un type union rassemble sous un même nom des valeurs de différents types. Par exemple, le type « véhicule » rassemblant les vélos, les motos et les autos.

Un type union est défini par la phrase :

`type` t `=` C_1 `of` t_1 `|` … `|` C_n `of` t_n`;;`

où t, C_1, …, C_n sont des identificateurs et t_1, …, t_n sont des expressions de type. Cette phrase définit un type union de nom t dont les instances sont les valeurs C_i v telles que $C_i \in \{C_1, \ldots, C_n\}$ et v est de type t_i. On a donc :

$$ext(t) = \{C_1\ v \mid v \in ext(t_1)\} \cup \ldots \cup \{C_n\ v \mid v \in ext(t_n)\}$$

Les identificateurs C_1, … , C_n sont les noms des constructeurs des valeurs de type t. Ces constructeurs sont spécifiques au type t et ne peuvent donc pas être des noms de champs d'un autre type union.

Pour des raisons de symétrie une barre (« `|` ») peut être placée devant le premier constructeur.

Par exemple :

```
#type forme =
   | Rectangle of float * float
   | Carré of float
   | Cercle of float;;
Type forme defined.
```

Le type `forme` est un type union qui a trois constructeurs `Rectangle`, `Carré` et `Cercle`. Les valeurs `Rectangle (15.5, 20.0)`, `Carré 12.3` et `Cercle 10.25` sont de type `forme`.

Un type union peut avoir une seule composante. Par exemple :

```
#type pile = Pile of int list;;
Type pile defined.
```

ou bien avoir certaines composantes constantes (c'est à dire réduites à leur constructeur). Par exemple :

```
#type couleur_primitive = Rouge | Jaune | Bleu;;
Type couleur_primitive defined.
```

Le type prédéfini `bool` est un type union défini de la façon suivante :

```
type bool = false | true;;
```

5.2.1 Construction d'une valeur de type union

Une valeur d'un type union est construite par application de l'un des constructeurs de ce type. Par exemple :

```
#Carré (27. +. 0.5);;
- : forme = Carre 27.5

#Pile [3; 3 - 1; 3 - 2];;
- : pile = Pile [3; 2; 1]

#[Rouge; Jaune; Bleu];;
- : couleur_primitive list = [Rouge; Jaune; Bleu]
```

La sémantique d'un constructeur de valeur de type union est la suivante :

CONSTRUCTION D'UNE VALEUR DE TYPE UNION. Si *Env* est un environnement contenant la définition `type` $t = C_1\ t_1 \mid \ldots \mid C_n\ t_n$ et si e est une expression telle que $type(e, Env) = t_i$ et $t_i \in \{t_1, \ldots, t_n\}$ alors :

$type(C_i\ e, Env) = t$,

$val(C_i\ e, Env) = C_i\ val(e, Env)$.

5.2.2 Egalité structurelle

Deux valeurs d'un même type union sont structurellement égales si elles sont produites par l'application du même constructeur à deux valeurs structurellement égales.

5.2.3 Filtrage

Si t un type union défini par $t = C_1\ t_1 \mid \ldots \mid C_n\ t_n$ et si F est un filtre de type $t_i \in \{t_1, \ldots, t_n\}$ alors :

$C_i\ F$

est un filtre de type t tel que $env\_filtré(C_i\ F, C_i\ v) = env\_filtré(F, v)$.

Voici quelques exemples de manipulation de valeurs de type `forme` ou `pile` définis ci-dessus :

```
#let surface = function
   | Rectangle (lon, lar) -> lon *. lar
   | Carré côté -> côté *. côté
   | Cercle rayon -> 3.14 *. rayon *. rayon;;
surface : forme -> float = <fun>

#surface (Cercle 10.);;
- : float = 314

#let empiler i (Pile p) = Pile (i::p);;
empiler : int -> pile -> pile = <fun>

#let dépiler = function
   | Pile (i::p) -> i
   | Pile [] -> failwith "pile_vide";;
dépiler : pile -> int = <fun>

#dépiler (empiler 3 (Pile [1; 2]));;
- : int = 3
```

La fonction `failwith` appliquée à une chaîne de caractères *m* génère une exception, expliquée par le message *m*. Le traitement des exceptions est l'objet du chapitre suivant (chapitre 5).

Il est naturel que le corps d'une fonction s'appliquant à une valeur de type union comporte un cas pour chaque type de valeur composant ce type. Cette fonction ne pouvant être définie que par filtrage, si un cas n'est pas prévu, Caml le signalera (voir ci-dessus chapitre 3, §3.3). Par exemple :

```
#let périmètre = function
   | Rectangle (lon, lar) -> 2. *. (lon +. lar)
   | Carré côté -> 4. *. côté;;
> Toplevel input:
>...............function
>   Rectangle (lon, lar) -> 2. *. (lon +. lar)
>   | Carré côté -> 4. *. côté..
> Warning: pattern matching is not exhaustive
```

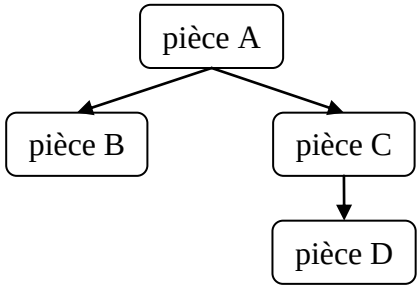

Figure 5.1. *Pièce composée*

```
périmètre : forme -> float = <fun>
```

Le cas `Cercle` n'a pas été prévu. La fonction `périmètre` n'est donc applicable qu'aux rectangles et aux carrés :

```
#périmètre (Carré 4.);;
- : float = 16
```

mais pas aux cercles :

```
#périmètre (Cercle 10.);;
Uncaught exception: Match_failure ("", 1151, 1235)
```

(l'exception `Match_failure` … a été déclenchée par le système qui n'a pu filtrer la valeur `Cercle 10.`).

5.3 Types récursifs

Il est possible de définir des types récursifs ou mutuellement récursifs. Un type *t* est récursif lorsque *t* intervient dans sa propre définition. *n* types sont mutuellement récursifs, si chacun d'eux apparaît directement ou indirectement dans la définition des *n* – 1 autres.

Considérons par exemple des pièces (mécaniques, électriques…) comme celle de la figure 5.1, identifiées par leur nom et pouvant être simples ou composées d'autres pièces. Ces pièces peuvent être représentée par un enregistrement du type `pièce` défini de la façon suivante :

```
#type pièce = {Nom: string; Composants: pièce list};;
Type pièce defined.
```

Le type `pièce` est récursif puisqu'il apparaît dans sa propre définition.

Par exemple, la pièce *A* de la figure 5.1 qui est composée d'une pièce simple *B* et d'une pièce *C* elle-même composée d'une pièce simple *D*, pourra être représentée par la valeur suivante, de type `pièce` :

```
{Nom = "A";
 Composants = [{Nom = "B"; Composants = []};
               {Nom = "C"; Composants = [{Nom = "D"; Composants = []}]}]}
```

Le défaut de cette représentation est de ne pas mettre clairement en évidence le fait qu'une pièce soit simple ou composée : une pièce simple est caractérisée par une liste de composants vide. On peut le faire en stipulant explicitement qu'une pièce est soit simple, soit composée, qu'une pièce simple est décrite par son nom et qu'une pièce composée est décrite par un enregistrement à deux champs : son nom et la liste des pièces qui la compose. On représentera donc les pièces comme des valeurs du type `pièce` défini par :

```
type pièce = Simple of string | Composée of pièce_composée
```

et les descriptions de pièces composées comme des valeurs du type `pièce_composée` défini par :

```
type pièce_composée = {Nom: string; Composants: pièce list}
```

Les types `pièce` et `pièce_composée` sont mutuellement récursifs puisque le type `pièce_composée` apparaît dans la définition du type `pièce` et que le type `pièce` apparaît dans le type `pièce_composée`. A l'instar des fonctions mutuellement récursives (voir chapitre 3, §3.5) , ils devront être définis simultanément de la façon suivante :

```
#type pièce = Simple of string | Composée of pièce_composée
and pièce_composée = {Nom: string; Composants: pièce list};;
Type pièce defined.
```

La pièce *A* de la figure 5.1 pourra alors être représentée par la valeur suivante de type `pièce` :

```
Composée {Nom = "A";
          Composants = [Simple "B";
                        Composée {Nom = "C";
                                  Composants = [Simple "D"]}]}
```

5.4 Types paramétrés

Certaines des opérations réalisées sur des valeurs structurées telles que des listes, des piles ou bien encore des arbres, sont indépendantes du type des composantes (ou de certaines composantes) de ces valeurs. Par exemple, le calcul de la longueur d'une liste ne nécessite pas de connaître le type des éléments de cette liste. De même, l'empilement d'un élément sur une pile, ne nécessite pas de connaître le type des éléments de cette pile.

Afin de définir de telles opérations, que l'on peut qualifier de génériques, il est nécessaire que les types des composantes (ou de certaines des composantes) de ces valeurs structurées soit des paramètres et donc que ces valeurs structurées aient un type paramétré.

Reprenons l'exemple des piles, on définira tout d'abord le type paramétré « pile de *t* » où *t* est un paramètre de type qui désigne le type des éléments de la pile puis l'opération « empiler une valeur de type *t* sur une pile de type "pile de *t*" ». A partir de ce type et de cette opération, on pourra par exemple engendrer le type « pile de caractères » et l'opération « empiler un caractère sur une pile de caractères » en instanciant *t* par le type « caractère ».

Rappelons (voir chapitre 4, §4.2) qu'un paramètre de type est noté `'`*i* où *i* est un identificateur et qu'un type paramétré est un type dont l'expression contient au moins un paramètre de type.

Un type paramétré est défini par la phrase :

`type (`$'a_1$`, ..., `$'a_n$`) t = d ;;`

où $'p_1$, …, $'p_n$ sont des paramètres de type, *t* est le nom du type et *d* est sa définition qui peut être, soit une définition de type enregistrement, soit une définition de type union, contenant les paramètres $'p_1$, …, $'p_n$. Lorsque le type défini ne possède qu'un seul paramètre les parenthèses peuvent être omises. L'objet *t* ainsi défini est un constructeur de type, c'est à dire une fonction qui appliquée aux types t_1, …, t_n a pour valeur le type obtenu en remplaçant dans *d*, $'a_1$ par t_1, …, $'a_n$ par t_n.

Un premier exemple de type paramétré est le type prédéfini `list` que nous avons étudié au chapitre 3. Il est défini de la façon suivante :

```
type 'a list = [] | prefix :: of 'a * 'a list
```

Le paramètre de type `'a` représente le type des éléments de la liste. Il faut bien faire attention à la signification de cette définition. Elle stipule que les éléments d'une liste doivent avoir le même type et non que chaque élément peut avoir un type quelconque. Voici, par exemple, ce que répondra Caml si l'on essaie de construire une liste dont les éléments n'ont pas tous le même type :

```
#1::"x"::[];;
Toplevel input:
>1::"x"::[];;
>   ^^^^^^^
This expression has type string list,
but is used with type int list.
```

Lorsque Caml rencontre `1`, le premier élément de la liste en construction, il lie le paramètre de type `'a` au type `int` et en déduit que la liste en construction doit être une liste d'entiers. Il signale donc une erreur lorsqu'il rencontre le deuxième élément `"x"` qui est de type `string`.

Allons un peu plus loin. Considérons, par exemple, la fonction `appartient` qui teste si une valeur `v` appartient à une liste `l`. Elle peut être définie de la façon suivante :

```
#let rec appartient v l =
   match l with
      | [] -> false
      | t::q -> if t = v then true else appartient v q;;
appartient : 'a -> 'a list -> bool = <fun>
```

Son type est le type paramétré : `'a -> 'a list -> bool` qui signifie que l'élément dont on teste l'appartenance à la liste doit avoir le même type `'a` que les éléments de cette liste. Si cette condition n'est pas vérifiée, voici ce que répondra Caml :

```
#appartient "x" 1::2::[];;
Toplevel input:
>appartient "x" 1::2::[];;
>               ^
This expression has type int,
but is used with type string list.
```

Lorsque Caml rencontre `"x"`, il lie le paramètre de type `'a` au type `string` et en déduit que l'on veut tester l'appartenance à une liste de chaînes de caractères. Il signale donc une erreur lorsqu'il rencontre la liste `1::2::[]` qui est de type `int list`.

5.5 Sémantique d'une définition de type

Jusqu'ici nous avons considéré qu'un environnement était constitué par un ensemble de liaisons « nom-valeur ». Lorsque qu'un nouveau type est défini il faut que son nom, ainsi que les constructeurs des valeurs de ce type soient insérés dans l'environnement de la session afin de pouvoir être utilisés par le synthétiseur de types. Pour les mêmes raisons, les types prédéfinis et leurs constructeurs de valeur sont présents dans l'environnement initial.

Une définition de type `type` *t* `=` *d* étend l'environnement courant en :

Env_courant ⊕ *Env_def*(type *t* = *d*, *Env_courant*).

où *Env_def* calcule l'environnement ajouté par cette définition de type, selon les règles suivantes :

ENVIRONNEMENT AJOUTE PAR UNE DEFINITION DE TYPE. Si *Env* est un environnement, alors :

Env_def(`type` t `=` `{`C_1`:` t_1`;` … `;` C_n`:` t_n`}`, *Env*) =
  Env ⊕ {`type` t, `cons` `{`C_1 `=` t_1`;` … `;` C_n `=` t_n`}` `->` t},

Env_def(`type` t `=` C_1 `of` t_1 `|` … `|` C_n `of` t_n, *Env*) =
  Env ⊕ {`type` t, `cons` C_1 : t_1 `->` t , …, `cons` C_n : t_n `->` t}.

Les noms de types et les noms de constructeurs ont été préfixés respectivement par les mots-clés `type` et `cons` afin de ne pas les confondre avec les noms de valeur, car en Caml ces trois ensembles de noms sont disjoints (c'est-à-dire qu'une valeur, un type ou un constructeur peuvent avoir le même nom).

5.6 Exemples

5.6.1 Calculette

A titre d'exemple, nous allons programmer une calculette permettant d'évaluer des expressions arithmétiques construites à partir des nombres flottants et des cinq opérations classiques d'addition, de soustraction, de multiplication, de division et de calcul de l'opposé.

Il faut tout d'abord choisir une représentation des expressions soumises à la calculette. Nous les représenterons comme des valeurs du type récursif `expr` défini de la façon suivante :

```
#type expr =
   | Nb of float
   | Add of expr * expr
   | Soust of expr * expr
   | Mult of expr * expr
   | Div of expr * expr
   | Opp of expr;;
Type expr defined.
```

Par exemple, l'expression arithmétique –(5,2 × (35,0 + 19,3)) sera représentée par la valeur de type `expr` suivante :

```
Opp (Mult (Nb 5.2, Add (Nb 35.0, Nb 19.3)))
```

Programmons maintenant notre calculette. L'évaluation d'une expression arithmétique représentée par une valeur de type `expr` sera réalisée par application à cette expression de la fonction `eval` définie de la façon suivante :

```
#let rec eval = function
   | Nb n -> n
   | Add (e1, e2) -> (eval e1) +. (eval e2)
   | Soust (e1, e2) -> (eval e1) -. (eval e2)
   | Mult (e1, e2) -> (eval e1) *. (eval e2)
   | Div (e1, e2) -> (eval e1) /. (eval e2)
   | Opp n -> -. (eval n);;
eval : expr -> float = <fun>
```

Remarquons encore une fois la souplesse apportée par la programmation par filtrage. Il suffit de prévoir un cas pour chaque constructeur du type `expr`.

Essayons :

```
#let mon_expression = Opp (Mult ((Nb 5.2), (Add (Nb 35.0, Nb 19.3))));;
mon_expression : expr = Opp (Mult (Nb 5.2, Add (Nb 35.0, Nb 19.3)))
```

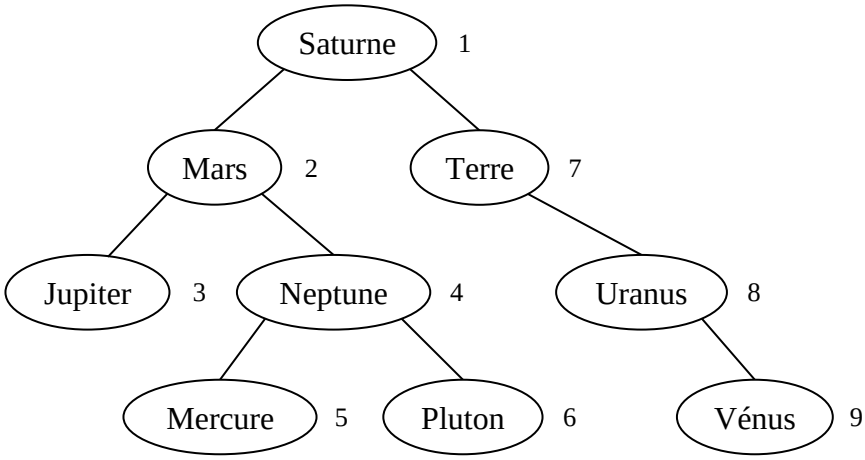

Figure 5.2. *Un arbre binaire de recherche*

```
#eval mon_expression;;
- : float = -282.36
```

5.6.2 Arbres binaires de recherche

Afin d'illustrer l'utilisation des types paramétrés, nous allons travailler sur une structure de données très classique : celle des arbres binaires de recherche.

Un arbre binaire est un ensemble fini de nœuds qui peut être soit un ensemble vide, soit un ensemble d'au moins un nœud tel que :

– un des nœuds constitue la *racine* de l'arbre ;
– l'ensemble des nœuds restants (excluant la racine) est partitionné en deux sous-ensembles dont chacun est un arbre binaire et qui sont appelés *sous-arbre gauche* et *sous-arbre droit* de la racine.

Chaque nœud contient une information appelée *étiquette* de ce nœud.

La figure 5.2 présente un exemple d'arbre binaire constitué de 9 nœuds numérotés de 1 à 9 et d'étiquettes respectives : « Saturne », « Mars », « Jupiter », « Neptune », « Mercure », « Pluton », « Terre », « Uranus » et « Vénus », qui sont les noms des neuf planètes du soleil. L'arbre binaire {1, 2, 3, 4, 5, 6, 7, 8, 9} a pour racine le nœud 1 dont le sous-arbre gauche est {2, 3, 4, 5, 6} et le sous-arbre droit est {7, 8, 9}. L'arbre binaire {2, 3, 4, 5, 6} a pour racine le nœud 2 dont le sous-arbre gauche est {3} et le sous-arbre droit {4, 5, 6}. L'arbre binaire {4, 5, 6} a pour racine le nœud 4 dont le sous-arbre gauche est {5} et le sous-arbre droit est {6}. L'arbre binaire {7, 8, 9} a pour racine le nœud 7 dont le sous-arbre gauche est vide et le sous-arbre droit est l'arbre binaire {8, 9}. L'arbre binaire {8, 9} a pour racine le nœud 8 dont le sous-arbre gauche est vide et le sous-arbre droit est l'arbre {9}. Les arbres binaires {3}, {5}, {6} et {9} ont pour racines respectives les nœuds 3, 5, 6 et 9 dont les sous-arbres gauche et droit sont vides.

Une catégorie intéressante d'arbres binaires est celle des *arbres binaires de recherche* car ils offrent une représentation des ensembles permettant de tester l'appartenance d'un élément à un ensemble plus rapidement qu'en parcourant séquentiellement les éléments de cet ensemble.

Un arbre binaire de recherche est un arbre binaire dont les étiquettes sont les éléments d'un ensemble muni d'une relation d'ordre total et qui est tel que pour chaque nœud d'étiquette *e*, les étiquettes des nœuds de son sous-arbre gauche sont toutes inférieures à *e* et celles des nœuds de son sous-arbre droit sont toutes supérieures à *e*. L'arbre binaire de la figure 5.2, par exemple, est un arbre binaire de recherche.

Pour chercher si un élément *x* appartient à l'ensemble *E* des étiquettes d'un arbre binaire de recherche *R*, il suffit d'appliquer l'algorithme suivant : « si *R* est vide alors *x* n'appartient pas à *E*, sinon, si *x* est égal à l'étiquette de *r* alors *x* appartient à *E*, sinon, si *x* est inférieur à l'étiquette de *r* alors *x* appartient à *E* s'il appartient à l'ensemble des étiquettes du sous-arbre gauche de *r*, sinon, si *x* est supérieur à l'étiquette de *r* alors *x* appartient à *E* s'il appartient à l'ensemble des étiquettes du sous-arbre droit de *R* ».

Testons, par exemple, si « Neptune » appartient à l'ensemble *Noms_planètes* des étiquettes de l'arbre binaire de recherche de la figure 5.2. On se place sur le nœud 1 racine de cet arbre : « Neptune » est inférieur à l'étiquette « Saturne » du nœud 1. On descend donc sur le nœud 2 racine du sous-arbre gauche du nœud 1 : « Neptune » est supérieur à l'étiquette « Mars » du nœud 2. On descend donc sur le nœud 4 racine du sous-arbre droit du nœud 2 : « Neptune » est égal à l'étiquette du nœud 2, il appartient donc à l'ensemble *Noms_planètes*.

Testons maintenant si « Io » appartient à l'ensemble *Noms_planètes*. On se place sur le nœud 1 : « Io » est inférieur à l'étiquette « Saturne » du noeud 1. On descend donc sur le nœud 2 : « Io » est inférieur à l'étiquette « Mars » du nœud 2. On descend donc sur le nœud 3 : « Io » est inférieur à l'étiquette « Jupiter » du nœud 3. Mais le sous-arbre gauche du nœud 3 est vide : « Io » n'appartient donc pas à l'ensemble *Noms_planètes*.

Montrons maintenant comment représenter de tels arbres en Caml et comment les manipuler au travers de trois opérations : insertion d'un élément, test d'appartenance d'un élément à un ensemble et tri par ordre croissant des éléments de l'ensemble.

On définit tout d'abord le type `'a arbre` dont les instances sont des arbres binaires de recherche dont les nœuds ont pour étiquette des valeurs de type `'a` :

```
#type 'a arbre =
   | Noeud of 'a arbre * 'a * 'a arbre
   | Vide;;
Type arbre defined.
```

L'insertion d'un nœud dans un arbre binaire nécessite la connaissance de la relation d'ordre dont est muni l'ensemble des étiquettes de cet arbre. Nous représenterons cette relation par une fonction qui appliquée à deux étiquettes e_1 et e_2 retourne `Egal` si e_1 est égal à e_2, `Inf`, si e_1 est inférieur à e_2 et `Sup` si e_1 est supérieur à e_2. Les valeurs `Inf`, `Egal` et `Sup` sont les instances du type `position` défini par :

```
type position = Inf | Egal | Sup;;
#Type position defined.
```

L'insertion d'un nœud d'étiquette *e* dans un arbre binaire de recherche *B* produit un nouvel arbre binaire de recherche obtenu de la façon suivante :

– si *B* est vide, on produit un arbre binaire de recherche dont la racine a l'étiquette *e* et un sous-arbre gauche et un sous-arbre droit vide ;

– sinon, soit *r* l'étiquette de la racine de *B*, *G* le sous-arbre gauche de la racine de *B* et *D* son sous-arbre droit :

  • si *e* est égal à *r*, alors l'arbre produit est *B* puisque l'ensemble de ses étiquettes contient déjà *e* ;

- sinon, si *e* est inférieur à *r* alors l'arbre produit a une racine dont l'étiquette est *r* ; le sous-arbre gauche de cette racine est le résultat de l'insertion du nœud d'étiquette *e* dans *G* et son sous-arbre droit est *D* ;
- sinon, si *e* est supérieur à *r* alors l'arbre produit a une racine dont l'étiquette est *r* ; le sous-arbre gauche de cette racine est *G* et son sous-arbre droit est le résultat de l'insertion du nœud d'étiquette *e* dans *D*.

D'où la définition de la fonction `insérer_noeud` qui retourne l'arbre binaire de recherche obtenu en insérant le nœud d'étiquette `e` dans l'arbre binaire de recherche `arbre` muni de la relation d'ordre `comp` :

```
#let rec insérer_noeud comp arbre e =
   match arbre with
      | Vide -> Noeud (Vide, e, Vide)
      | Noeud (g, r, d) ->
           (match (comp e r) with
              | Inf -> Noeud ((insérer_noeud comp g e), r, d)
              | Egal -> arbre
              | Sup -> Noeud (g, r, (insérer_noeud comp d e)));;
insérer_noeud : ('a -> 'a -> position) -> 'a arbre -> 'a -> 'a arbre =
<fun>
```

En examinant le type de la fonction `insérer` on constate qu'il impose bien que le type `'a` de l'étiquette du nœud à insérer soit le même que le type des étiquettes de l'arbre dans lequel cet élément est inséré et le même que le type des éléments auxquels est appliquée la fonction `comp`.

Testons maintenant cette fonction, en construisant un arbre binaire de recherche `planètes` dont les étiquettes sont les noms des neuf planètes du soleil. Cet arbre sera muni de la relation d'ordre alphabétique définie par la fonction `comp_chaînes` définie de la façon suivante :

```
let comp_chaînes n1 n2 =
   if n1 < n2 then
      Inf
   else
      if n1 = n2 then
         Egal
      else
         Sup;;
#comp_chaînes : 'a -> 'a -> position = <fun>
```

L'arbre de recherche `planètes` est construit en insérant itérativement dans un arbre binaire de recherche initialement vide les nœuds d'étiquettes : « Saturne », « Mars », « Neptune », « Jupiter », « Terre », « Uranus », « Mercure », « Pluton » et « Vénus ». Ce qui donne :

```
#let planètes =
   let rec insérer_liste_noeuds comp arbre le =
      match le with
         | [] -> arbre
         | e::le -> insérer_liste_noeuds comp
                                         (insérer_noeud comp arbre e)
                                         le
   in
      insérer_liste_noeuds comp_chaînes
                           Vide
                           ["Saturne"; "Mars"; "Neptune"; "Jupiter";
                           "Terre"; "Uranus"; "Mercure"; "Pluton";
                           "Vénus"];;
planètes : string arbre =
 Noeud
  (Noeud
    (Noeud (Vide, "Jupiter", Vide), "Mars",
```

```
      Noeud
       (Noeud (Vide, "Mercure", Vide), "Neptune",
        Noeud (Vide, "Pluton", Vide))),
   "Saturne",
   Noeud (Vide, "Terre", Noeud (Vide, "Uranus", Noeud (Vide, "Vénus",
Vide))))
```

On vérifiera facilement que l'on obtient l'arbre binaire de recherche `planètes` est celui de la figure 5.2.

Pour rechercher si un élément *e* appartient à l'ensemble *E* des étiquettes d'un arbre binaire de recherche *B*, on applique la règle suivante, qui dérive directement de la règle de construction des arbres binaires de recherche :

– si *B* est vide, alors *e* n'appartient pas à *E* ;
– sinon, soit *r* l'étiquette de la racine de *B* :
  • si *e* est égal à *r* alors *e* appartient à *E* ;
  • sinon, si *e* est inférieur à *r* alors rechercher *e* dans le sous-arbre gauche de la racine de *B* ;
  • sinon, si *e* est supérieur à *r* alors rechercher *e* dans le sous-arbre droit de la racine de *B*.

D'où la définition de la fonction `appartient` qui retourne `true` si l'élément `e` appartient à l'arbre binaire de recherche `arbre` ou `false` sinon:

```
#let rec appartient comp arbre e =
   match arbre with
      | Vide -> false
      | Noeud (g, r, d) ->
           (match (comp e r) with
              | Inf -> appartient comp g e
              | Egal -> true
              | Sup -> appartient comp d e);;
appartient : ('a -> 'b -> position) -> 'b arbre -> 'a -> bool = <fun>
```

On peut tester cette fonction en recherchant « Neptune » puis « Io » dans l'ensemble des étiquettes de l'arbre binaire de recherche `planètes` :

```
#appartient comp_chaînes planètes "Neptune";;
- : bool = true

#appartient comp_chaînes planètes "Io";;
- : bool = false
```

On vérifie bien que « Neptune » appartient à cet ensemble et que « Io » n'y appartient pas.

Pour finir, nous allons définir une fonction `trier`, qui permet de construire la liste triée par ordre croissant des étiquettes de l'arbre binaire de recherche `arbre`. Pour réaliser ce tri, il suffit de parcourir récursivement cet arbre dans l'ordre sous-arbre gauche – racine – sous-arbre droit. La fonction `trier` peut donc être définie de la façon suivante :

```
#let rec trier arbre =
   match arbre with
      | Vide -> []
      | Noeud (g, r, d) ->
           (trier g) @ [r] @ (trier d);;
trier : 'a arbre -> 'a list = <fun>
```

Appliquons la fonction `trier` à l'arbre binaire de recherche `planètes` :

```
#trier planètes;;
- : string list =
 ["Jupiter"; "Mars"; "Mercure"; "Neptune"; "Pluton"; "Saturne"; "Terre";
  "Uranus"; "Vénus"]
```

Cette application a bien pour valeur la séquence des neuf noms de planètes triée par ordre alphabétique.

6
Traitement des exceptions

En programmation fonctionnelle, les fonctions sont totales, c'est à dire qu'elles sont applicables à tout argument qui appartient à leur type de départ. Il faut donc être capable de traiter les cas, appelés *exceptions*, où cet argument n'est pas acceptable. Par exemple : une division par zéro ou bien la recherche de la tête d'une liste vide. Ceci peut être fait par des tests préventifs placés dans le corps des fonctions, mais ce mécanisme est très lourd car il implique un travail important pour le programmeur et il altère la lisibilité d'un programme en masquant son fonctionnement normal. C'est pourquoi les langages de programmation modernes comportent un mécanisme spécifique pour le traitement des exceptions. Dans ce chapitre nous étudions celui qui est offert par Caml.

6.1 Déclenchement d'exceptions sans récupération

Considérons le programme suivant :

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

#let g x y z = (x + y) / z;;
g : int -> int -> int -> int = <fun>

#(f 4 2) + (g 3 1 0);;
Uncaught exception: Division_by_zero
```

L'expression `(f 4 2) + (g 3 1 0)` n'a pas pu être calculée car Caml a rencontré une division par 0 lors de l'application de `g`. Il a alors *déclenché* (on dit aussi « *levé* ») l'exception `Division_by_zero` qui a été interceptée au niveau le plus haut (« top level » en anglais) ce qui a provoqué une interruption du programme signalée par le message `Uncaught exception: Division_by_zero`.

Ce mécanisme de détection des exceptions est tout à fait insuffisant pour les deux raisons suivantes :

– Le message généré par Caml ne permet pas de savoir en quel point du programme l'exception a été déclenchée. Dans l'exemple ci-dessus on ne sait pas si la division par zéro s'est produite lors de l'application de `f` ou de celle de `g`.
– Le programme est interrompu brutalement sans qu'il soit possible de remédier à la cause de l'erreur pour que l'exécution puisse se poursuivre.

6.2 Déclenchement d'exceptions avec récupération

Le mécanisme de traitement des exceptions de Caml peut être schématisé de la façon suivante (cf. Figure 6.1) :

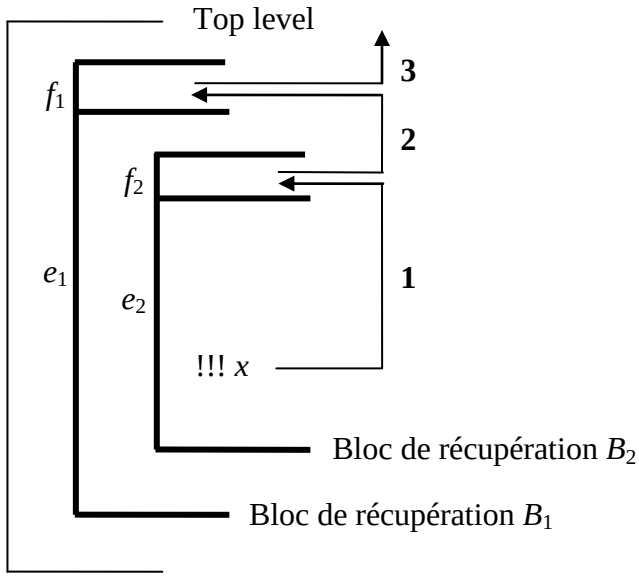

Figure 5.1 : Le mécanisme de traitement des exceptions de Caml

1. Lorsque le programmeur désire remédier aux erreurs pouvant se produire lors de l'évaluation d'une expression il peut encapsuler celle-ci dans un bloc de *récupération d'exception* et déclencher une exception (!!!) si une erreur se produit.
2. Un bloc de récupération d'exceptions (B_1 et B_2) est composé de l'expression à évaluer et d'une fonction applicable aux exceptions pouvant être déclenchées dans ce bloc (e_1 et f_1 pour le bloc B_1, e_2 et f_2 pour le bloc B_2).
3. Lorsqu'une exception x est déclenchée (**1**) l'évaluation en cours est interrompue. La fonction de récupération f du bloc englobant le plus imbriqué est appliquée à x. La valeur de cette application remplace celle de l'expression dont l'évaluation a été interrompue.
4. Si f n'est pas applicable à x alors x est à nouveau déclenchée (**2**) et le processus recommence à l'étape 3. Si le niveau le plus haut est atteint Caml interrompt l'évaluation et indique que cette interruption a été causée par une exception non récupérée (`Uncaught exception`) (**3**).

6.2.1 Définition d'une exception

Caml offre un type union prédéfini `exn` dont les instances sont des exceptions. Les constructeurs d'exceptions sont définis incrémentalement par le programmeur au fur et à mesure de ses besoins.

Une exception est définie par la phrase :

```
exception C of t;;
```

où *C* est le nom du constructeur de l'exception et *t* est une expression de type. Par exemple :

```
#exception Pile_vide;;
Exception Pile_vide defined.

#exception Erreur of string;;
Exception Erreur defined.
```

```
#Pile_vide;;
- : exn = Pile_vide

#Erreur "Division par zéro en appliquant g";;
- : exn = Erreur "Division par zéro en appliquant g"
```

Les valeurs `Pile_vide` et `Erreur "Division par zéro en appliquant g"` sont des instances du type `exn`.

Par ailleurs Caml possède un certain nombre d'exceptions prédéfinies : `Failure`, `Match_Failure`, `Invalid_argument`, etc.

6.2.2 Déclenchement d'une exception

Le déclenchement d'une exception est réalisé :

– soit par l'évaluateur de Caml, c'est le cas des exceptions prédéfinies comme `Division_by_zero` dont nous avons donné un exemple au §6.1,

– soit par l'évaluation de l'expression :

```
raise e
```

– où `raise` est une fonction prédéfinie et *e* est une expression de type `exn`.

Dans le programme suivant, par exemple, on déclenche une exception lorsqu'une pile devient vide :

```
#type 'a pile = Pile of 'a list;;
Type pile defined.

#let depiler = function
   | Pile (x::p) -> x
   | Pile [] -> raise Pile_vide;;
depiler : 'a pile -> 'a = <fun>

#depiler (Pile [1; 2]);;
- : int = 1

#depiler (Pile []);;
Uncaught exception: Pile_vide
```

Il est important de noter que le type d'arrivée de la fonction `raise` est un paramètre de type. Vérifions le :

```
#raise;;
- : exn -> 'a = <fun>
```

Ceci, afin que l'insertion d'un déclenchement d'exception dans une expression n'ait pas d'influence sur le type de cette expression. Par exemple, si dans la fonction suivante :

```
#function x -> 365 / x;;
- : int -> int = <fun>
```

on insère un déclenchement d'exception en cas de division par zéro, on constate que le type de la fonction n'est pas changé :

```
#function x ->
   if x <> 0 then
      365 / x
   else
      raise (Erreur "Division par zéro");;
- : int -> int = <fun>
```

Il existe des exceptions prédéfinies de la forme `Failure` *m* (où *m* est une chaîne de caractères) qui sont déclenchées par l'application de la fonction prédéfinie `failwith` au messsage *m*. Par exemple :

```
#let debiter compte montant =
   if compte > montant then
      compte - montant
   else
      failwith "Credit insuffisant";;
debiter : int -> int -> int = <fun>

#debiter 23 54;;
Uncaught exception: Failure "Credit insuffisant"
```

6.2.3 Récupération d'une exception

La récupération d'une exception déclenchée lors de l'évaluation d'une expression e peut être réalisée en encapsulant cette expression dans une expression try..with qui a la forme suivante :

`try` *e* `with` F_1 `->` e_1 `|` … `|` F_n `->` e_n

où *e*, e_1, …, e_n sont des expressions de même type et F_1, …, F_n sont des filtres de type `exn`.

La valeur de cette expression est celle de *e* si aucune exception n'est déclenchée, sinon elle est celle de l'expression associée au premier des filtres F_1, …, F_n qui filtre l'exception déclenchée.

La sémantique d'une récupération d'exception est la suivante.

TYPE ET VALEUR D'UNE RECUPERATION D'EXCEPTION. Si *Env* est un environnement et si *e*, e_1, …, e_n sont des expressions telles que *type*(e, *Env*) = *type*(e_1, *Env*) = … = *type*(e_n, *Env*) = *t* alors :

type(`try` *e* `with` F_1 `->` e_1 `|` … `|` F_n `->` e_n, *Env*) = *t*.

Si aucune exception n'a été déclenchée lors de l'évaluation de *e* alors :

val(`try` *e* `with` …, *Env*) = *val*(*e*, *Env*)

sinon, si une exception *x* a été déclenchée :

val(`try` *e* `with` F_1 `->` e_1 `|` … `|` F_n `->` e_n, *Env*) =
val(`match` *x* `with` F_1 `->` e_1 `|` … `|` F_n `->` e_n `|` `_ ->` `raise` *x*, *Env*).

En clair,

Par exemple, la fonction `ajouter` définie par :

```
#let ajouter n p = n + try depiler p with Pile_vide -> 0;;
ajouter : int -> int pile -> int = <fun>
```

ajoute à `n` le nombre placé au sommet de la pile d'entiers `p`. Dans le cas où la pile est vide, c'est zéro qui est ajouté à *n*. Vérifions le :

```
#ajouter 5 (Pile [1]);;
- : int = 6

#ajouter 5 (Pile []);;
- : int = 5
```

7
Programmation impérative

Jusqu'ici nous avons étudié un sous-ensemble de Caml que l'on peut qualifier de *purement fonctionnel*. La seule opération mise en jeu, dans un programme purement fonctionnel, est l'application d'une fonction. Il n'existe pas d'opérations permettant de modifier explicitement l'état de la mémoire. Malgré son élégance, la programmation purement fonctionnelle atteint ses limites quand il s'agit :

– de communiquer avec le monde extérieur. Par exemple, lire des données, imprimer des résultats ;
– de modifier explicitement le contenu de certaines cases mémoires, ce qui peut être nécessaire pour optimiser des programmes ou bien lorsque les changements d'état sont inhérents à l'application traitée ;
– de spécifier l'ordre dans lequel doivent être évaluées certaines expressions.

On appelle *effet de bord* « une modification d'une case de la mémoire ou bien une interaction avec le monde extérieur (impression ou lecture) »[1]. Un programme qui opère par une suite d'effets de bord est appelé programme *impératif*. Caml est un langage qui combine harmonieusement programmation fonctionnelle et programmation impérative.

C'est le sous-ensemble impératif de Caml qui fait l'objet de ce chapitre. Nous étudierons successivement les entrées-sorties, le séquencement des expressions, les valeurs modifiables et les boucles.

7.1 Entrées-sorties

La gestion des entrées-sorties en Caml est classique. Pour lire ou écrire sur un fichier on utilise des canaux d'entrée ou de sortie qui sont des valeurs Caml de type `in_channel` ou `out_channel`.

Un *canal d'entrée* est créé par la fonction prédéfinie `open_in` appliquée au nom du fichier à ouvrir. Il est fermé en lui appliquant la fonction prédéfinie `close_in`. La lecture sur un canal d'entrée est réalisée en lui appliquant les fonctions prédéfinies `input_char` qui lit le prochain caractère et `input_line` qui lit la prochaine ligne ; l'exception `End_of_file` est déclenchée lorsque la fin du fichier est atteinte.

Un *canal de sortie* est créé par la fonction prédéfinie `open_out` appliquée au nom du fichier à ouvrir. Il est fermé en lui appliquant la fonction prédéfinie `close_out`. L'écriture sur un canal de sortie est réalisée en lui appliquant les fonctions prédéfinies `output_char` qui écrit un caractère et `output_string` qui écrit une chaîne de caractères. Par exemple, le programme suivant écrit dans un fichier puis relit ce qu'il a écrit :

[1] Définition extraite du livre « Le langage Caml » de Pierre Weis et Xavier Leroy.

```
#let sortie = open_out "mon_fichier";;
sortie : out_channel = <abstract>

#output_string sortie "un\n";;
- : unit = ()

#output_string sortie "deux\n";;
- : unit = ()

#output_string sortie "12";;
- : unit = ()

#close_out sortie;;
- : unit = ()

#let entree = open_in "mon_fichier";;
entree : in_channel = <abstract>

#input_line entree;;
- : string = "un"

#input_line entree;;
- : string = "deux"

#input_char entree;;
- : char = `1`

#input_char entree;;
- : char = `2`

#input_char entree;;
Uncaught exception: End_of_file

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

On notera que la valeur d'une ouverture est `<abstract>` et que le type d'arrivée de la fonction `input_line` est `string`.

Classiquement, la lecture de caractères saisis au clavier est un cas particulier de lecture de fichier. Elle se fait sur le canal prédéfini `std_in` (ou `stdin`). Il en est de même pour l'affichage à l'écran qui se fait sur le canal prédéfini `std_out` (ou `std_out`). Cependant par souci de simplicité Caml offre les fonctions prédéfinies suivantes :

```
read_line ()      ≡ input_line std_in
print_string s    ≡ output_string std_out s
print_char c      ≡ output_char std_out c
print_int i       ≡ output_string std_out (string_of_int i)
print_float f     ≡ output_string std_out (string_of_float f)
print_newline ()  ≡ output_string std_out "\n"
```

Voici, par exemple, deux fonctions dont nous nous servirons par la suite :

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

#let a_la_ligne () = print_newline ();;
a_la_ligne : unit -> unit = <fun>
```

7.2 Séquencement

L'opérateur `;` permet d'imposer l'ordre dans lequel sont évaluées deux d'expressions. Si e_1 et e_2 sont des expressions, alors :

e_1 `;` e_2

est une expression qui a pour effet de bord d'évaluer e_1 puis d'évaluer e_2 et qui a pour valeur la valeur de e_2. Il est évident qu'une telle expression n'a d'intérêt que si e_1 est une expression à effet de bord.

L'opérateur `;` est associatif à droite. Pour marquer clairement le début et la fin d'une séquence d'expressions, cette séquence peut être encadrée par les mots-clés `begin` et `end`.

Par exemple, le programme suivant calcule la surface d'un rectangle dont la longueur et la largeur sont demandées à l'utilisateur.

```
#let lon =
    print_string "Donnez la longueur du rectangle en mètres ? ";
    int_of_string (read_line ())
 and lar =
    print_string "Donnez la largeur du rectangle en mètres ? ";
    int_of_string (read_line ())
 in
    begin
       print_string "Ce rectangle a une surface de ";
       print_int (lon * lar);
       print_string " m2";
       a_la_ligne ()
    end;;
Donnez la longueur du rectangle en mètres ? 15
Donnez la largeur du rectangle en mètres ? 5
Ce rectangle a une surface de 75 m2
- : unit = ()
```

7.3 Valeurs modifiables

7.3.1 Références

Pour permettre au programmeur de modifier explicitement le contenu de la mémoire, Caml manipule des pointeurs comme en Pascal ou en C. En Caml un pointeur vers un objet de type *t* est appelé *référence* et est une instance du type *t* `ref`. Une référence est construite en appliquant le constructeur `ref` à l'objet à référencer. Dans le programme suivant, `horloge` référence un triplet constitué de l'heure, des minutes et des secondes :

```
#let horloge = ref (23, 21, 59);;
horloge : (int * int * int) ref = ref (23, 21, 59)
```

L'opérateur d'affectation `:=` permet de remplacer l'objet référencé et l'opérateur `!` permet d'y accéder :

```
#prefix :=;;
- : 'a ref -> 'a -> unit = <fun>

#prefix !;;
- : 'a ref -> 'a = <fun>
```

On remarque que l'opérateur `:=` a pour type d'arrivée `unit` puisqu'il se réduit à un effet de bord : la modification de la mémoire. Le programme suivant fait avancer l'horloge d'une seconde :

```
#let avancer_horloge () =
    horloge := match !horloge with
                  | (h, 59, 59) -> (h + 1, 0, 0)
                  | (h, m, 59) -> (h, m + 1, 0)
                  | (h, m, s) -> (h, m, s + 1);;
avancer_horloge : unit -> unit = <fun>
```

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

#horloge;;
- : (int * int * int) ref = ref (23, 22, 0)
```

Comme les autres valeurs Caml, les références peuvent être filtrées. Si *F* est un filtre de type *t* alors `ref` *F* est un filtre de type *t* `ref`. Par exemple :

```
#match horloge with ref (h, _, _) -> h;;
- : int = 23
```

7.3.2 Tableaux

Un *tableau* (ou *vecteur*) est une suite modifiable, de longueur fixe et non nulle, de valeurs du même type que nous appellerons *éléments* du tableau. Un tableau de valeurs de type *t* est une instance du type *t* `vect`. Il est noté en séparant chacun de ses éléments par un `;` et en encadrant cette suite par les signes `[|` et `|]`. Par exemple le tableau `[|15.0; 5.5; 10.0|]` est de type `float vect`.

Un tableau peut être vu comme une suite de *n* cases numérotées de 0 à *n* – 1 dont chacune contient une valeur. Le contenu des cases est initialisé à la construction du tableau et peut être modifié par la suite. Un tableau peut être construit de deux façons différentes :

1. soit en donnant la liste de ses éléments. Par exemple :

   ```
   #[|1 + 9; 2 + 8; 3 + 7; 4 + 6; 5 + 5|];;
   - : int vect = [|10; 10; 10; 10; 10|]
   ```

2. soit à l'aide du constructeur `make_vect` en donnant le nombre d'éléments et une valeur initiale qui sera affectée à chacun de ces éléments. Par exemple :

   ```
   #make_vect 4 "";;
   - : string vect = [|""; ""; ""; ""|]
   ```

La fonction prédéfinie `vect_length` calcule la longueur du tableau auquel elle s'applique. Par exemple :

```
#let saisons = make_vect 4 "";;
saisons : string vect = [|""; ""; ""; ""|]

#vect_length saisons;;
- : int = 4
```

L'opérateur `.()` permet d'extraire la valeur contenue dans une case d'un tableau et l'opérateur `<-` d'affecter une valeur à une case d'un tableau. L'expression *t*`.(`*i*`)` a pour valeur la valeur contenue dans la i^e case du tableau *t* et l'expression *t*`.(`*i*`) <-` *v* a pour effet de bord d'affecter la valeur *v* à la i^e case du tableau *t*. Par exemple :

```
#begin
   saisons.(0) <- "printemps";
   saisons.(1) <- "été";
   saisons.(2) <- "automne";
   saisons.(3) <- "hiver";
end;;
- : unit = ()

#saisons.(3);;
- : string = "hiver"
```

Nous verrons au §6.4 comment se servir des boucles pour parcourir les éléments d'un tableau.

7.3.3 Enregistrements à champs modifiables

Un enregistrement peut avoir des champs modifiables, c'est à dire des champs dont on peut changer la valeur. Il suffit que dans la définition du type de cet enregistrement ces champs soient précédés du mot clé `mutable`. Par exemple :

```
#type personne = {Nom: string; Prénom: string; mutable Age: int};;
Type personne defined.
```

Le champ `Age` d'un enregistrement de type `personne` est modifiable.

Pour remplacer la valeur d'un champ modifiable *c* d'un enregistrement *e* on utilise la construction *e*`.`*c* `<-` *v* où *v* est la nouvelle valeur du champ *c*. Par exemple le programme suivant augmente l'âge d'une personne :

```
#let dupont = {Nom = "Dupont"; Prénom = "Jean"; Age = 42};;
dupont : personne = {Nom="Dupont"; Prénom="Jean"; Age=42}

#dupont.Age <- dupont.Age + 1;;
- : unit = ()

#dupont;;
- : personne = {Nom="Dupont"; Prénom="Jean"; Age=43}
```

7.4 Boucles

Caml offre deux sortes de boucles pour répéter l'évaluation d'expressions à effet de bord : la boucle `while` et la boucle `for`.

Si e_1 est une expression booléenne et e_2 une expression quelconque, l'expression :

`while` e_1 `do` e_2 `done`

a pour effet de bord de répéter l'évaluation de l'expression e_2 tant que l'expression e_1 est vraie. L'expression e_2 est évaluée au début de chaque itération. Par exemple, le programme suivant affiche la suite des chiffres de 0 à 9 :

```
#let i = ref(0) in
   while !i < 10 do
      print_int !i;
      un_espace ()
   done;
   a_la_ligne ();;
0 1 2 3 4 5 6 7 8 9
- : unit = ()
```

Si *i*, *d* et *f* sont des entiers et *e* est une expression quelconque, l'expression :

`for` *i* `=` *d* `to` (resp. `downto`) *f* `do` *e* `done`

a pour effet de répéter l'évaluation de l'expression *e* avec $i = d$, $i = d + 1$, …, $i = f$ (resp. $i = d$, $i = d - 1$, …, $i = f$). Si $f < d$ (resp. $f > d$) l'expression *e* n'est jamais évaluée. Par exemple, le programme suivant calcule la somme des éléments d'un tableau d'entiers `t` :

```
#let t = [|2; 4; 6; 8; 10; 12|] and s = ref 0 in
   for i = 0 to (vect_length t) - 1 do
      s := !s + t.(i)
   done;
   !s;;
- : int = 42
```

8
Programmation modulaire

Jusqu'ici nous avons programmé en Caml en considérant que nous disposions d'un ensemble de fonctions et de types prédéfinis chargés à l'ouverture d'une session interactive et constituant l'environnement initial de la session, puis que nous définissions au fur et à mesure des besoins un certain nombre de noms de valeurs, de types ou d'exceptions. Cette façon de faire est impraticable pour développer de véritables logiciels. Il faut pouvoir découper un programme en plusieurs parties qui puissent être sauvegardées et bien entendu réutilisées.

Caml offre différents outils pour cela. Nous étudierons tout d'abord une technique simple qui est l'*inclusion de fichiers*. Elle permet d'enregistrer un programme Caml dans un ou plusieurs fichiers et de charger ces fichiers en mémoire au fur et à mesure des besoins. Cette technique n'est pas suffisante pour deux raisons : (i) les programmes doivent être recompilés à chaque chargement, (ii) ces fichiers ne peuvent pas être développés indépendamment puisqu'il faut s'assurer que les ensembles de noms définis dans chacun soient disjoints. Pour pallier à cette insuffisance, Caml permet de découper une application en *modules* compilables séparément. Les modules offrant un degré suffisant de généralité peuvent être insérés dans une bibliothèque partageable par plusieurs applications.

Dans ce chapitre nous appellerons :

– *entité* : un type, une exception, un constructeur ou une valeur.
– programme Caml : une suite de phrases qui sont soit des expressions, soit des définitions de valeurs, de type ou d'exception, soit des directives.

8.1 Inclusion de fichiers

Un programme Caml peut être enregistré dans un fichier qui doit avoir l'extension `.ml`. Pour charger un programme Caml en mémoire, il suffit d'appliquer la fonction prédéfinie `include` au nom du fichier qui le contient. Cette application a pour effet de lire chaque phrase du programme et de l'évaluer. Tout se passe comme si ces phrases avaient été entrées au clavier.

Supposons, par exemple, que le fichier `"un_deux_trois.ml"` du répertoire courant ait le contenu :

```
let un = 1;;
let deux = un + 1;;
let trois = deux + 1;;
```

son inclusion se déroulera de la façon suivante :

```
#include "un_deux_trois.ml";;
un : int = 1
deux : int = 2
trois : int = 3
- : unit = ()
```

Si le fichier n'appartient pas au répertoire courant il faut préfixer le nom de ce fichier par le nom du répertoire auquel il appartient. Si l'extension `.ml` est omise elle est automatiquement rajoutée. On aurait donc pu écrire `include "un_deux_trois"` au lieu de `include "un_deux_trois.ml"`.

8.2 Modules

Un module est un morceau de programme Caml qui forme un tout et qui est susceptible d'être réutilisé. Le texte d'un module est enregistré dans un fichier appelé *fichier source* du module, qui a le même nom que celui du module et l'extension `.ml`. Par exemple, le module `"cercle"` de texte :

```
type cercle = {Rayon: float};;
let pi = 3.14;;
let périmètre {Rayon = r} = 2. *. pi *. r;;
let surface {Rayon = r} = pi *. r *. r;;
```

enregistré dans le fichier `"cercle.ml"`.

On distingue trois types de modules :

– Les modules de la *bibliothèque standard* de Caml qui contiennent notamment les valeurs et les types prédéfinis. Par exemple, le module `io` consacré aux entrées-sorties (`open_in`, `input_line`, …) ou bien le module `int` consacré aux opérations sur les entiers (`prefix +`, `minus`, …).
– Les modules écrits par le programmeur.
– Le module `top` qui est un peu particulier. Il est composé de toutes les phrases entrées au clavier ou incluses à partir d'un fichier au cours d'une session interactive. Il n'a donc pas de fichier source à proprement parler.

8.2.1 Qualification des noms d'un module

Les expressions contenues dans un module peuvent faire appel soit à des noms internes, définis dans ce module, soit à des noms externes définis dans d'autres modules. Rien n'interdisant à des modules différents de contenir des entités de même nom, une entité ne peut être identifiée que par un couple « nom du module, nom interne au module ». En Caml, un tel couple est appelé *nom complet* (ou *nom qualifié*) et est noté *m__n* où *m* est le nom du module et *n* est le nom interne au module appelé *nom abrégé* (ou *nom non qualifié*). Par exemple, `int__minus` désigne le nom `minus` du module `int` qui est un module de la bibliothèque de base de Caml.

On pourrait imposer que seuls les noms complets soient employés dans les programmes, mais ce serait très lourd. Pour l'éviter Caml offre un mécanisme d'ouverture de module qui permet le plus souvent de n'employer que les noms abrégés, leur complétion étant automatiquement effectuée par Caml. Ce mécanisme est le suivant :

– Pour utiliser le nom abrégé d'une entité dans le texte source d'un module il faut au préalable ouvrir le module qui contient cette définition.
– Un module est explicitement ouvert par la directive `#open` *m* où *m* est le nom du module et fermé par la directive `#close` *m*. Pour faciliter la tâche du programmeur, Caml ouvre automatiquement avant l'analyse d'un module, le module à analyser ainsi que les modules de la bibliothèque qui sont le plus utilisés.

– Lorsque Caml rencontre un nom abrégé, il le complète par le nom du premier module ouvert qui définit ce nom, en parcourant ces modules dans l'ordre suivant : le module en cours d'analyse, les modules explicitement ouverts dans l'ordre de leur ouverture, les modules de bibliothèque ouverts dans un ordre défini lors de l'installation de Caml.

Les règles d'emploi des noms abrégés dans le texte d'un module, impliquées par ce mécanisme sont les suivantes.

– Toutes les entités du module en cours d'écriture peuvent être désignées par leur nom abrégé. Cette règle est tout à fait naturelle. Elle a été appliquée implicitement, dans tous les chapitres précédents, pour le module `top`.

– Toutes les entités d'un module ouvert peuvent être désignées par leur nom abrégé, s'il n'existe pas d'entité de même nom dans un module explicitement ouvert préalablement. Il y a donc intérêt à ouvrir en premier le module le plus utilisé.

– Toutes les entités d'un module de bibliothèque peuvent être désignées sous leur nom abrégé, s'il n'existe pas d'entité de même nom dans les modules explicitement ouverts ou bien dans les modules de bibliothèque ouverts préalablement. Une conséquence de cette règle est la possibilité de redéfinir les primitives de la bibliothèque tout en conservant leur désignation. La fonction prédéfinie `min`, par exemple, choisit le plus petit de deux entiers. On peut la redéfinir dans le module `top` pour qu'elle choisisse le plus petit entier d'une liste non vide d'entiers :

```
#min;;
- : int -> int -> int = <fun>

#min 2 1;;
- : int = 1

#let rec min = function
   | [] -> failwith "min est inapplicable"
   | [n] -> n
   | n::l -> int__min n (min l);;
min : int list -> int = <fun>

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

– Dans les autres cas, les définitions devront être désignées par leur nom complet. C'est le cas de `int__min` dans le programme précédent.

8.2.2 Interface de module

Les entités définies dans un module peuvent être classées en deux catégories : les entités *publiques* qui sont destinées à être utilisées par d'autres modules et les entités *privées* qui servent uniquement à définir les premières et qui sont cachées aux autres modules. Considérons, par exemple, un module contenant les formules permettant de calculer le volume de divers objets géométriques (sphères, cylindres, prismes, etc.). Pour définir ces formules il peut être nécessaire d'utiliser des constantes (π par exemple) ou des définitions intermédiaires (la surface de base d'un cylindre, par exemple) qui ne doivent pas être visibles dans les autres modules.

Pour distinguer entre entités publiques et entités privées, Caml décompose un module en deux parties :

– l'*interface*, qui est formée par les signatures des noms de valeurs et par les définitions des types et des exceptions publiques. La signature d'un nom de valeur est une phrase de la forme :

```
value n : t;;
```

où *n* est le nom de la valeur et *t* son type.

– l'*implémentation*, qui est formée par les définitions des noms de valeur et celles des types et des exceptions privées.

Le fichier contenant l'implémentation d'un module de nom *m* doit avoir pour nom *m*`.ml` et celui contenant son interface *m*`.mli`.

Par exemple, dans le module `cercle` on peut vouloir cacher la valeur `pi` et exporter le type `cercle` et les fonctions `périmètre` et `surface`. En ce cas l'interface, enregistrée dans le fichier `cercle.mli` contiendra les phrases :

```
type cercle = {Rayon: float};;
value périmètre cercle -> float;;
value surface cercle -> float;;
```

et l'implémentation, enregistrée dans le fichier `cercle.ml`, les phrases :

```
let pi = 3.14;;
let périmètre {Rayon = r} = 2. *. pi *. r;;
let surface {Rayon = r} = pi *. r *. r;;
```

L'interface par défaut d'un module est constituée de toutes les définitions contenues dans ce module.

8.2.3 Compilation d'un module

Interface et implémentation d'un module sont destinés à être compilés. Il faut pour cela appliquer la fonction prédéfinie `compile` au nom du fichier correspondant. Par exemple :

```
#compile "cercle.mli";;
- : unit = ()
#compile "cercle.ml";;
- : unit = ()
```

La compilation de l'interface d'un module *m* produit le fichier *m*`.zi`, dit *fichier d'interface compilée* et celle de l'implémentation produit le fichier *m*`.zo`, dit *fichier de code objet*. Par exemple `cercle.zi` et `cercle.zo` pour le module `cercle`.

Le fichier d'interface compilée contient les informations sur les types des entités déclarées dans le fichier d'interface. Ce fichier est indispensable pour compiler les modules qui font appel aux entités publiques de *m*. Quant au fichier de code objet, il contient le code compilé des valeurs définies dans le fichier d'implémentation.

8.2.4 Chargement d'un module

Pour charger en mémoire un module non compilé et l'exécuter il faut appliquer la fonction prédéfinie `load` au nom de son fichier source. Par exemple :

```
#load "un_deux_trois";;
un : int = 1
deux : int = 2
trois : int = 3
- : unit = ()
```

Pour faire de même avec un module compilé, il faut appliquer la fonction prédéfinie `load_object` au nom de son fichier de code objet. Par exemple :

```
#load_object "cercle.zo";;
- : unit = ()
```

(si l'extension `.zo` est omise elle automatiquement rajoutée).

Attention ! pour que les définitions des modules chargés soient visibles il faut au préalable ouvrir le module :

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

```
##open "un_deux_trois";;
#un;;
- : int = 1
```

Travailler en mode interactif est équivalent à charger de façon incrémentale le module `top` qui, comme nous l'avons vu, contient toutes les définitions entrées au clavier ou par une inclusion de fichier (`include`).

Au démarrage d'une session interactive, Caml charge automatiquement tous les modules de la bibliothèque standard. Il ouvre aussi un certain nombre de modules comme nous l'avons expliqué au §7.2.1.

8.3 Exemple complet

Cet exemple est consacré à la manipulation de volumes géométriques : sphères, cylindres, prismes, etc. Il s'agit de faire des calculs sur leurs surfaces et sur leurs volumes. Dans un premier temps nous constituons une bibliothèque d'objets bi et tri-dimensionnels : cercles, carrés, cubes, sphères, etc. A chaque type d'objet est associé un module.

Nous donnons ci-dessous les modules relatifs aux cercles, aux rectangles et aux cylindres, avec pour chacun son interface et son implémentation :

– Module `"cercle"` :
  - *Interface*

```
type cercle = {Rayon: float};;
value périmètre : cercle -> float;;
value surface : cercle -> float;;
```

  - *Implémentation*

```
let pi = 3.14;;
let périmètre {Rayon = r} = 2. *. pi *. r;;
let surface {Rayon = r} = pi *. r *. r;;
```

– Module `"rectangle"` :
  - *Interface*

```
type rectangle = {Longueur: float; Largeur: float};;
value périmètre : rectangle -> float;;
value surface : rectangle -> float;;
```

  - *Implémentation*

```
let périmètre {Longueur = x; Largeur = y} =
   2. *. (x +. y);;
let surface {Longueur = x; Largeur = y} = x *. y;;
```

– Module `"cylindre"` :
  - *Interface*

```
type cylindre = {Rayon: float; Hauteur: float};;
value surface : cylindre -> float;;
value volume : cylindre -> float;;
```

- *Implémentation*

```
#open "cercle";;
#open "rectangle";;
let périmètre_base {Rayon = r} =
   cercle__périmètre {cercle__Rayon = r};;
let surface_base {Rayon = r} =
   cercle__surface {cercle__Rayon = r};;
let surface_droite {Rayon = r; Hauteur = h} =
   rectangle__surface
      {Longueur = périmètre_base {Rayon = r; Hauteur = h};
       Largeur = h};;
let surface c =
   (2. *. surface_base c) +. surface_droite c;;
let volume ({Hauteur = h} as c) =
   (surface_base c) *. h;;
```

Remarquons, dans la définition de la fonction `périmètre_base` du module `"cylindre"`, l'utilisation du nom complet `cercle__Rayon` pour désigner le champ `Rayon` d'un cercle et ne pas le confondre avec le champ `Rayon` d'un cylindre. De même le nom complet `rectangle__surface` est utilisé pour distinguer la fonction `surface` d'un cylindre de celle d'un rectangle.

Voici maintenant une session de travail sur les cylindres. Cette session se déroule de la fàçon suivante :

1. On se place dans le répertoire contenant les modules `"rectangle"`, `"cercle"` et `"cylindre"` :

   ```
   cd "…";
   ```

2. On compile ces modules :

   ```
   #compile "rectangle.mli";;
   - : unit = ()
   #compile "rectangle.ml";;
   - : unit = ()
   #compile "cercle.mli";;
   - : unit = ()
   #compile "cercle.ml";;
   - : unit = ()
   #compile "cylindre.mli";;
   - : unit = ()
   #compile "cylindre.ml";;
   File "cylindre.ml"
   Warning: useless #open on module "cercle".
   - : unit = ()
   ```

3. Le message « `Warning: useless #open on module "cercle".` » indique qu'il n'était pas indispensable d'ouvrir le module `cercle` puisque toutes les entités de ce module sont désignées par leur nom complet. Remarquons de plus que l'on a compilé les interfaces des modules `"rectangle"` et `"cercle"` avant de compiler le module `"cylindre"` qui fait appel à ces deux modules. Si l'on avait commencé par compiler le module `"cylindre"` Caml aurait envoyé le message suivant :

   ```
   #compile "cylindre.ml";;
   Cannot find the compiled interface file cercle.zi.
   ```

   indiquant qu'il n'a pas trouvé l'interface compilée du module `"cercle"` dans lequel il aurait recherché la définition du constructeur `cercle__Rayon`.

4. On charge les codes objets :

```
#load_object "rectangle.zo";;
- : unit = ()
#load_object "cercle.zo";;
- : unit = ()
#load_object "cylindre.zo";;
- : unit = ()
```

5. On ouvre le module `"cylindre"` :

```
##open "cylindre";;
```

6. On définit un cylindre de rayon `10` et de hauteur `20`, puis on en calcule sa surface et son volume :

```
#let mon_cylindre = {Rayon = 10.; Hauteur = 20.};;
mon_cylindre : cylindre = {Rayon = 10.0; Hauteur = 20.0}
```

```
#surface mon_cylindre;;
- : float = 1884.0
```

```
#volume mon_cylindre;;
- : float = 6280.0
```

9
Conclusion

Dans ce cours nous avons étudié les principales fonctionnalités du langage Caml. Cependant pour construire des applications en vraie grandeur il faut en plus savoir maîtriser :

- Les instructions d'installation de Caml sur diverses machines : Macintosh, PC, Unix, …
- La bibliothèque standard composée d'une bibliothèque de base, d'une *bibliothèque d'utilitaires* et d'une *bibliothèque graphique*. La bibliothèque de base contient un ensemble de fonctions opérant sur les types de Caml. La bibliothèque d'utilitaires contient entre autres des fonctions de tri, un générateur de nombres aléatoires, des fonctions pour construire et manipuler des tables de hachage, etc.. La bibliothèque graphique contient un ensemble de primitives graphiques portables.
- Des outils divers dont les analyseurs lexicaux et syntaxiques `camllex` et `camlyacc`.

On trouvera tout cela dans le *Manuel de référence du langage Caml* cité dans l'introduction.

Il faut aussi savoir qu'une version orientée objet de Caml appelée Objective Caml a été développée, elle-même téléchargeable à partir du site de l'INRIA (voir Introduction).