Université Paris-Sud
Centre d'Orsay

Formation au langage Caml

Claude Marché
Ralf Treinen

Adresses des auteurs:

Claude Marché
L.R.I., Bâtiment 490
Université Paris-Sud
91405 Orsay cedex
France

Tél : 01 69 15 64 85
Bureau 118
Email : Claude.Marche@lri.fr
WWW : http://www.lri.fr/~marche/

Ralf Treinen
L.R.I., Bâtiment 490
Université Paris-Sud
91405 Orsay cedex
France

Tél : 01 69 15 65 92
Bureau 152
Email : Ralf.Treinen@lri.fr
WWW : http://www.lri.fr/~treinen/

Nous remercions également Srečko Brlek, Serena Cerrito, Judicaël Courant, Quentin El Haïk, Jean-Christophe Filliâtre, Bruno Guillaume, et Benjamin Monate pour leurs contributions et remarques.

Table des matières

Introduction

Le langage Caml fait partie de la famille des langages dits *fonctionnels*, qui se caractérisent par le fait que les fonctions y sont des *objets de première classe*, ce qui signifie qu'une fonction est un type de données au même titre que les entiers, booléens, enregistrements, etc. En particulier, nous verrons qu'une fonction peut naturellement prendre en argument une autre fonction ou bien retourner une fonction.

Caml peut être utilisé aussi bien à la manière d'un *interpréteur*, où l'utilisateur tape des commandes et obtient des réponses immédiates ; ou bien comme *compilateur*, où un fichier source en Caml est traduit en programme exécutable. Pour apprendre les différents aspects du langage, nous allons utiliser l'interpréteur.

Nous verrons dans un second temps comment on utilise le compilateur, les modules, et enfin les outils CamlLex et CamlYacc.

Il faut savoir que le langage Caml est en constante évolution, et quelques différences apparaissent entre chaque version. Il s'agit essentiellement de différence de syntaxe pour la partie du langage exposée dans ce document : celui-ci est réalisé avec la version Objective Caml 3.01 mais la plupart des aspects présentés fonctionnent avec les versions 2.04 à 3.02.

Chapitre 1

L'interpréteur Caml et les expressions simples

1.1 La boucle d'interprétation de Caml

La *boucle d'interprétation* de Caml suit un principe très simple et incontournable :

– L'utilisateur tape une *requête* (on dit encore une *phrase*) Caml : une *expression* terminée par deux points-virgules consécutifs.

– Caml analyse la syntaxe, affiche un message d'erreur si cette syntaxe est incorrecte.

– Si la syntaxe est correcte, l'interpréteur *infère* (c'est-à-dire calcule) le *type* de l'expression, affiche un message d'erreur si l'expression est mal typée.

– Si l'expression est bien typée, l'interpréteur *évalue* l'expression, puis affiche le type calculé et la valeur obtenue.

Voici un exemple d'expression entière :

```
#3*(4+1)-7;;
- : int = 8
```

`int` est le type de l'expression, ici entier, et `8` est sa valeur. Le « `-` » qui précède le type sera expliqué dans la section 1.6.

Notez que les espaces et saut de lignes sont ignorés à l'intérieur d'une expression. Notez également que des commentaires, qui seront également ignorés, peuvent être insérés entre `(*` et `*)`.

Voici maintenant un exemple d'expression syntaxiquement incorrecte :

```
#3*4+;;
Characters 4-6:
Syntax error
```

et voici un exemple d'expression syntaxiquement correcte mais mal typée :

```
#1+true;;
Characters 2-6:
This expression has type bool but is here used with type int
```

1.2 Expressions entières

Les opérateurs arithmétiques classiques sont disponibles, la division entière est notée / et le reste de la division entière est noté `mod`.

```
#  1+2*3-4;;
- : int = 3
```

```
#  7/3 + 5 mod 2;;
- : int = 3
```

Exercice 1
Déterminer le type et la valeur affichés par Caml pour chacune des requêtes suivantes :

```
#  1+2;;

#  4/2*2;;

#  -4/(2*2);;

#  5/2;;

#  5 mod 2;;

#  1/0;;
```

1.3 Expressions réelles

Les constantes réelles (type `float`) doivent absolument être tapées avec le point (représentant la virgule) et les chiffres de la partie décimale derrière. Optionnellement, la notation scientifique $x \times 10^y$ peut être utilisée, en écrivant `e` (ou `E`) suivi de l'exposant : `1.5e3` représente le réel $1,5 \times 10^3$, et `2.4e-5` représente $2,4 \times 10^{-5}$.

Les entiers et les réels sont deux types disjoints : `1` est l'entier 1 de type `int`, et `1.0` est le réel 1, de type `float`, et *ne sont pas* compatibles à l'intérieur d'une même expression arithmétique : `1+1.` est incorrect. Les opérations arithmétiques sur les réels doivent être notées avec un point derrière l'opérateur : addition « `+.` », multiplication « `*.` », etc. :

```
#  1.0 +. 2.5;;
- : float = 3.5
```

```
#  1.0 /. 2.5;;
- : float = 0.4
```

Exercice 2
Déterminer le résultat affiché par Caml pour chacune des requêtes suivantes :

```
#  1.1e-3 +. 2.5 *. 2.0 -. 1e+2;;

#  -. 1.0 /. 2.5;;

#  1 +. 3.6;;

#  1.2 + 3.4;;
```

Fonctions mathématiques usuelles

Caml dispose de toutes les fonctions réelles usuelles : `sqrt`, `exp` (exponentielle de base e), `log` (logarithme népérien), `sin`, `cos`, `tan`, `asin`, `acos`, `atan` (ces six dernières fonctions travaillant en radians), etc...

```
#  sqrt(2.0);;
- : float = 1.41421356237
```

Fonctions de conversion et de troncature

Il existe des fonctions de conversion pour passer d'un type entier à réel et vice-versa : `float_of_int`, `int_of_float`, cette dernière supprimant la partie décimale, donc arrondit vers le bas les nombres positifs et vers le haut les négatifs. Enfin, les fonctions `floor` (sol) et `ceil` (plafond) permettent d'arrondir un nombre réel respectivement vers l'entier inférieur et l'entier supérieur (mais le résultat reste de type réel).

Exercice 3
Déterminer le résultat affiché par Caml pour chacune des requêtes suivantes :

```
#  sqrt(4.0);;

#  sqrt(4);;

#  float_of_int(4) /. 2.0 ;;

#  int_of_float(2.6) ;;

#  int_of_float(-3.7) ;;

#  floor(2.6) ;;

#  floor(-3.7) ;;

#  ceil(2.6) ;;

#  ceil(-3.7) ;;
```

1.4 Expressions caractères et chaînes de caractères

Les constantes caractères (type `char`) se notent entre apostrophes, et les chaînes de caractères de longueur quelconque forment également un type de base de Caml (type `string`), et se notent entre guillemets :

```
#  'a';;
- : char = 'a'

#  "bcd";;
- : string = "bcd"
```

L'opération de base sur les chaînes est la concaténation (c'est-à-dire la mise bout-à-bout) , notée « ^ ».

```
#"bonj" ^ "our";;
- : string = "bonjour"
```

Caml fournit les fonctions d'usage courant suivantes sur les caractères

- `(Char.code` c`)` donne le code ASCII du caractère c ;
- `(Char.chr` n`)` donne le caractère dont le code ASCII est l'entier n ;
- `(String.make` n c`)` renvoie une chaîne de longueur n remplie avec n fois le caractère c. En particulier, cette fonction permet de convertir un caractère c en une chaîne par `(String.make 1` c`)`.

et sur les chaînes :

- `(String.length` s`)` donne la longueur de la chaîne s ;
- `(String.get` s n`)` donne le n-ième caractère de la chaîne s ;
- `(String.sub` s d l`)` renvoie la sous-chaîne de longueur l de la chaîne s qui commence à la position d de s.

Ces fonctions déclenchent une « exception » (voir chapitre 4) dès qu'un argument n'est pas valide. La numérotation des positions dans une chaîne commence à 0, l'expression pour obtenir le dernier caractère d'une chaîne s est donc `(String.get s (String.length s - 1))`.

Il existe également des fonctions de conversion de chaîne vers entier ou réel, et vice-versa : `int_of_string`, `float_of_string`, `string_of_int` et `string_of_float`.

Exercice 4
Déterminer le résultat affiché par Caml pour chacune des requêtes suivantes :

```
#  "Le resultat est : " ^ string_of_int(7*3);;
#  float_of_string("4.7") +. 2.1;;
#  String.get "bonjour" 0;;
#  String.get "bonjour" 3;;
#  String.get "bonjour" 8;;
#  String.sub "bonjour" 0 3;;
#  String.sub "bonjour" 3 (String.length "bonjour" - 3);;
#  String.make 3 'a';;
```

Dans les constantes du type `char` ou du type `string` le caractère `\` sert de caractère « d'échappement ». Les caractères spéciaux suivants ne peuvent être représentés qu'à l'aide de ce caractère d'échappement :

Séquence à écrire en Caml	Caractère représenté
`\\`	antislash (`\`)
`\n`	saut de ligne (line feed)
`\r`	retour chariot (carriage return)
`\t`	tabulation
`\`*ddd*	le caractère avec le code ASCII *ddd* en décimal
`\'`	apostrophe (`'`), seulement dans les constantes de caractères
`\"`	guillemet (`"`), seulement dans les constantes de chaînes

Ces caractères ne servent vraiment que pour les entrées-sorties que nous verrons au chapitre 5.

1.5 Expressions booléennes

Les constantes `true` et `false` sont les constantes booléennes de base. La conjonction (« et » logique) est notée `&` (ou `&&`), la disjonction (« ou » logique) est notée `or` (ou `||`) et la négation est notée `not`. Attention, le mot clef `and` ne désigne pas la conjonction booléenne mais est utilisé pour les définitions simultanées (voir les sections 1.6, 1.7 et 1.8.3).

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

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

Note : `not` est prioritaire sur le `&` qui lui-même est prioritaire sur le `or`.

Les opérations de comparaison sont notées classiquement par >, <, >=, <=, = et <>. Elles peuvent comparer aussi bien des entiers, des réels, des caractères ou des chaînes (ordre alphabétique). Attention, il existe aussi un opérateur == qui a une sémantique différente de l'opérateur de comparaison =.

```
#  1<2;;
- : bool = true

#  3.0 > 5.6;;
- : bool = false

#  'a' <= 'b' ;;
- : bool = true

#  "avant" < "après" ;;
- : bool = false
```

Exercice 5
Déterminer le résultat affiché par Caml pour chacune des requêtes suivantes :

```
#  1 < 2 or 1.1 <> 3.2;;

#  2 = 3 & 1.34e+6 -. 4.9e-4 > 23.4e5;;

#  2>1 or 3<4 & 5>=6 ;;

#  (2>1 or 3<4) & 5>=6 ;;

#  not 2<1 or 3<4 ;;

#  not (2<1 or 3<4) ;;
```

Une des particularités de Caml et des langages fonctionnels en général est l'expression conditionnelle « `if ... then ... else ...` ». Celle-ci est véritablement une expression, avec une valeur, et non pas une « instruction ».

```
#  if 3>5 then 4.0 else 6.7;;
- : float = 6.7
```

À ce titre, une telle conditionnelle peut apparaître à l'intérieur d'une autre expression :

```
#  1 + (if true then 2 else 3);;
- : int = 3
```

Nous rappellons qu'une expression dénote toujours une valeur. De fait de ce statut d'expression, la partie `else` est obligatoire : sinon, une expression comme `if false then 1` n'aurait aucune valeur.

Exercice 6
Déterminer le résultat affiché par Caml pour chacune des requêtes suivantes :

```
#  if 4<5 then "oui" else "non";;

#  3 * (if 3=4 then 2 else 5);;

#  (if 3=4 then 5.0 else 6.1) +. (if false then (sqrt 2.0) else 2.0);;

#  (if 1<2 then "ca" else "pas") ^ ( if 5>6 then "cal" else "ml");;

#  if if 3=4 then 5>6 else 7<8
#    then if 9<10 then "un" else "deux"
#    else if 11>12 then "trois" else "quatre";;
```

1.6 Définitions de variables globales

Il est possible d'affecter la valeur d'une expression à une variable par l'utilisation d'une requête Caml de la forme `let` *identificateur = expression*. La variable ainsi définie est alors utilisable dans les requêtes suivantes. Par exemple :

```
#  let x=2+4;;
val x : int = 6

#  x*7;;
- : int = 42
```

Notez que dans la réponse de Caml, le tiret est maintenant remplacé par le nom de la variable ainsi définie.

L'utilisation d'une variable non définie déclenche une erreur :

```
#  z;;
Characters 2-3:
Unbound value z
```

Il est possible de définir plusieurs variables dans le même `let` en séparant ces définitions par un `and`:

```
#  let x=3 and y=4;;
val x : int = 3
val y : int = 4
```

Cette syntaxe est indispensable pour définir des *fonctions mutuellement récursives* (voir section 1.8.3).

Exercice 7
Dans la suite de requêtes suivante, dessiner des liens qui associent chaque occurrence de variable à l'endroit où elle est définie. En déduire le résultat affiché par Caml.

```
#  let a=3 and b=4;;

#  a+b;;

#  let c=d+1 and d=2;;

#  let c=3 and d=c+2;;

#  let c=d+1 and d=c+2;;
```

Pour finir, signalons que les noms de variables autorisés en Caml sont formés de lettres majuscules ou minuscules, de chiffres et des caractères _ (souligné) et ' (apostrophe), et doivent commencer par une lettre minuscule. Les lettres accentuées sont également autorisées, mais peuvent poser des problèmes de portabilité entre différents systèmes d'exploitation, aussi nous les déconseillons.

1.7 Définitions locales et règles de portée

Une définition de variable peut-être locale à une expression : on utilise alors une syntaxe de la forme

`let` *identificateur = expression1* `in` *expression2*

Par exemple :

```
#  let z=3 in z+4;;
- : int = 7
```

Attention : la valeur de la variable est alors connue dans *expression2* et *seulement* dans *expression2* :

```
#  let t=2 in t*t;;
- : int = 4
```

```
#  t;;
Characters 2-3:
Unbound value t
```

Contrairement au `let` global, un `let` local est lui-même une expression, dont la valeur est celle de *expression2*. Ainsi on peut écrire :

```
#  4+(let x=2 in x*x);;
- : int = 8
```

```
#  let x=1 in let y=x+1 in y*2;;
- : int = 4
```

```
#  let x=let y=3 in y+1 in x*2;;
- : int = 8
```

Exercice 8
Dans chacune des requêtes suivantes, dessiner des liens qui associent chaque occurrence de variable à l'endroit où elle est définie. En déduire le résultat affiché par Caml.

```
#  let a=3 and b=4 in a+b;;

#  let a=3 in let b=a+2 in a+b;;

#  let a=3 and b=a+2 in a+b;;

#  let a=3 in let b=a+2 and a=5 in a+b;;

#  let a=3 in let a=5 and b=a+2 in a+b;;
```

De même avec la suite de requêtes suivante :

```
#  let a=3;;

#  let b=a;;

#  let a=5 in a*a;;

#  let a=b in a*a;;
```

1.8 Définitions de fonctions

1.8.1 Les fonctions à un argument

Pour définir une fonction on utilise de nouveau le mot-clé `let`, que l'on fait suivre du nom de la fonction puis du nom de son paramètre, sans mettre de parenthèses. Par exemple la phrase Caml

```
#  let f x = 2*x+1;;
val f : int -> int = <fun>
```

définit la fonction f, qui prend un entier et retourne un entier, ceci en multipliant par deux et ajoutant un. L'application de cette fonction à une valeur se fait alors en juxtaposant le nom de la fonction et l'argument voulu, là encore sans mettre de parenthèse. Par exemple :

```
#  f 1;;
- : int = 3
```

```
#  f 4;;
- : int = 9
```

Du fait de l'absence de parenthèses, les expressions peuvent être parfois ambiguës. Il faut savoir que l'application d'une fonction est prioritaire sur les autres opérations. Ainsi, on utilisera parfois des parenthèses, pour lever les ambiguïtés. On aura ainsi

```
#  f 1 + 2;;
- : int = 5
```

```
#  (f 1) + 2;;
- : int = 5
```

```
#  f (1 + 2);;
- : int = 7
```

Exercice 9
Écrire une fonction `nieme_lettre` qui étant donné un entier n entre 1 et 26 donne la n-ième lettre de l'alphabet. On utilisera le fait que les caractères de `'A'` à `'Z'` sont numérotés consécutivement dans le codage ASCII. On utilisera des fonctions de base de la section 1.4.

1.8.2 Fonctions à plusieurs arguments

Pour définir une fonction à plusieurs arguments, on continue à utliser la juxtaposition de ceux-ci. Par exemple :

```
#(* (norme x y) retourne la norme du vecteur (x,y) *)
#let norme x y = sqrt (x*.x+.y*.y);;
val norme : float -> float -> float = <fun>
```

Nous reviendrons en détail sur l'écriture du type d'une fonction à plusieurs arguments au paragraphe 2.2. Noter sur cet exemple les commentaires Caml, mis entre `(*` et `*)`.

L'application d'une telle fonction se fait encore en juxtaposant les arguments (et toujours sans parenthèses) :

```
#  norme 3.0 4.0;;
- : float = 5
```

Exercice 10
Écrire une fonction à trois arguments réels a, b et c qui retourne le nombre de solutions réelles de l'équation $ax^2 + bx + c$.

1.8.3 Fonctions récursives

Il est possible de définir une fonction récursive en rajoutant le mot-clé `rec` après le `let`. La fonction factorielle se définit ainsi par :

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

#fact 4;;
- : int = 24
```

Le `let` local peut bien sûr être utilisé pour des fonctions, et l'on doit préciser `rec` dans le cas d'une fonction locale récursive :

```
#  let rec coef_bin n m =
#    let rec fact x = if x<=1 then 1 else x*fact(x-1)
#    in fact(n)/(fact(m)*fact(n-m));;
val coef_bin : int -> int -> int = <fun>

#  coef_bin 5 3;;
- : int = 10
```

Tous les noms de fonctions utilisés dans la définition d'une nouvelle fonction se rapportent aux fonctions déjà définies. Quand on veut définir deux fonctions où la première utilise la deuxième et la deuxième utilise la première on ne peut pas simplement définir une après l'autre:

```
#let rec pair x = if x = 0 then true else (impair (x-1));;
Characters 42-48:
Unbound value impair

#let rec impair x = if x = 0 then false else (pair (x-1));;
Characters 45-49:
Unbound value pair
```

Une telle définition de fonctions *mutuellement récursives* s'écrit obligatoirement à l'aide du mot clef `and`:

```
#let rec pair x = if x = 0 then true else impair (x-1)
#and impair x = if x = 0 then false else pair (x-1);;
val pair : int -> bool = <fun>
val impair : int -> bool = <fun>

#impair 42;;
- : bool = false

#pair 42;;
- : bool = true
```

Exercice 11

1. Programmer une fonction calculant une puissance entière d'un réel : $\texttt{puiss}(x, n) = x^n$, en raisonnant par récurrence sur n.

2. Programmer la fonction `fib` définie par $\texttt{fib}(0) = \texttt{fib}(1) = 1$ et $\texttt{fib}(n+2) = \texttt{fib}(n+1) + \texttt{fib}(n)$

3. Évaluer `fib` pour quelques valeurs. Que peut-on dire du temps de calcul utilisé, par exemple pour $\texttt{fib}(30)$?

4. Écrire une fonction donnant, en fonction de n, la différence entre `fib`(n) et $\frac{\tau^{n+1}}{\sqrt{5}}$ où $\tau = \frac{1+\sqrt{5}}{2}$ est le nombre d'or. Calculer cette différence pour quelques valeurs.

Exercice 12
Programmer une fonction `pgcd` qui étant donné deux entiers retourne leur plus grand diviseur commun. On se servira des relations

$$\begin{aligned} pgcd(x,0) &= x \text{ pour tout } x \\ pgcd(x,y) &= pgcd(y, x \bmod y) \text{ si } y \neq 0 \end{aligned}$$

Exercice 13

1. Programmer une fonction qui étant donné un entier (positif ou nul) retourne une chaîne représentant le codage binaire de cet entier.

2. Écrire la fonction réciproque. On donne les fonctions suivantes qui permettent respectivement d'obtenir le dernier caractère d'une chaîne et toute la chaîne sauf le dernier caractère :

```
#  let dernier s = String.get s (String.length s - 1)
#  and debut s = String.sub s 0 (String.length s - 1);;
val dernier : string -> char = <fun>
val debut : string -> string = <fun>

#  dernier "abcd";;
- : char = 'd'

#  debut "abcd";;
- : string = "abc"
```

Exercice 14
Écrire une fonction `doubler` qui prend une chaîne s et qui renvoie une chaîne ou tous les caractères de s sont doublés. Par exemple, `doubler "caml"` doit donner `"ccaammll"`.

Exercice 15

1. Écrire une fonction qui étant donné un entier (positif ou nul) retourne une chaîne représentant cet entier écrit en chiffres romains.

2. Écrire la fonction réciproque.

1.9 Structure de liste

La liste est un type prédéfini en Caml, on énumère les éléments entre crochets, séparés par des points-virgules :

```
#  [1;4;7;3];;
- : int list = [1; 4; 7; 3]

#  [3.5;7.8;-9.78];;
- : float list = [3.5; 7.8; -9.78]

#  ["a";"bcd";"bonjour"];;
- : string list = ["a"; "bcd"; "bonjour"]
```

Attention ! Dans une liste, tous les éléments doivent être de même type :

```
#  [1;"a"];;
Characters 5-8:
This expression has type string but is here used with type int
```

Les fonctions qui définissent abstraitement le type de données liste sont prédéfinies : la liste vide se note naturellement `[]`, l'ajout d'un élément `x` en tête d'une liste `l` se note `x::l`, le premier élément d'une liste est obtenu avec la fonction `List.hd` et le reste de la liste par `List.tl`. Enfin on teste si la liste est vide en testant son égalité à `[]`.

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

#  4::l;;
- : int list = [4; 1; 2; 3]

#  List.hd l;;
- : int = 1

#  List.tl l;;
- : int list = [2; 3]

#  if l=[] then "l est vide" else "l n'est pas vide";;
- : string = "l n'est pas vide"
```

Les fonctions sur les listes se définissent facilement à partir de ces fonctions et par récurrence. La somme des éléments d'une liste d'entiers peut ainsi se calculer par

```
#  let rec sum l =
#    if l=[] then 0 else (List.hd l)+(sum (List.tl l));;
val sum : int list -> int = <fun>

#  sum [2;6;9;3];;
- : int = 20
```

La forme de définition ci-dessus, où une fonction est définie par récurrence en utilisant le cas de la liste vide, puis le cas des listes non vides et en utilisant `hd` et `tl`, est très fréquente en pratique. Pour abréger cette construction systématique, il existe une construction appelée *décomposition par cas* (« pattern-matching » en anglais) de la forme

`match` *expression de type liste* `with`
| `[]` $\rightarrow$ *cas de base*
| var_1 `::` var_2 $\rightarrow$ *cas de récurrence*

où dans le cas de récurrence, var_1 sera liée à la tête de la liste et var_2 au reste de la liste. (Remarque : le | de la première ligne est optionnel). Ainsi la fonction `sum` peut s'écrire

```
#let rec sum l =
#  match l with
#  | [] -> 0
#  | x::r -> x+(sum r);;
val sum : int list -> int = <fun>
```

Exercice 16

1. Écrire une fonction qui calcule le produit des éléments d'une liste de réels.

2. Écrire une fonction étant donnée une liste de réels $[x_1; \ldots; x_n]$ retourne $\sum_{i=1}^{n} \frac{x_i}{i}$. Indication : on définira au préalable la fonction qui à un entier k et une liste $[x_k; \ldots; x_n]$ associe la somme $\sum_{i=k}^{n} \frac{x_i}{i}$

Certaines fonctions sur les listes ne dépendent pas du type des éléments contenus dans la liste. C'est la cas par exemple de la fonction qui calcule la longueur d'une liste :

```
#  let rec longueur l =
#    match l with
#     | [] -> 0
#     | x::r -> 1+(longueur r);;
val longueur : 'a list -> int = <fun>
```

Le type calculé par Caml contient `'a` qui est une *variable de type*. Ceci signifie que la fonction pourra être utilisée avec n'importe quel type à la place de `'a` :

```
#  longueur [2; 6; 4];;
- : int = 3
```

```
#  longueur ["abc";"de"];;
- : int = 2
```

Nous reviendrons en détail sur ce type de fonctions dans le paragraphe 2.2.

Exercice 17

1. Écrire une fonction `concat` qui concatène deux listes.

   Note : cette fonction `concat` est en fait un opérateur prédéfini en Caml noté `@` :

```
#  [1;7;3] @ [6; 2 ; 5 ; 6];;
- : int list = [1; 7; 3; 6; 2; 5; 6]
```

2. Écrire une fonction qui calcule la liste inverse d'une liste donnée, par exemple `inv [4;7;2]` retourne `[2;7;4]`. On utilisera l'opérateur de concaténation `@`.

3. Écrire une autre version de la fonction `inv` en définissant d'abord une fonction auxiliaire `inv2` spécifiée de la façon suivante :

   inv2 : liste, liste $\rightarrow$ liste
   { (inv2 l_1 l_2) retourne l'inverse de l_1, concaténé avec l_2 }

4. Comparer l'efficacité des deux versions de `inv`

Exercice 18

Quelle est la réponse de Caml sur les requêtes suivantes :

```
#  1::[2;3];;
```

```
#  [1;[2;3]];;
```

```
#  [[1];[2;3]];;
```

Exercice 19

Écrire une fonction de tri d'une liste d'entiers dans l'ordre croissant par *insertion* : on utilise une fonction auxiliaire spécifiée par

insert : entier, liste $\rightarrow$ liste
{ (insert x l), lorsque l est triée dans l'ordre croissant, retourne la liste l où x a été inséré à la bonne place }

Exercice 20
Un autre algorithme pour trier une liste est celui du *tri rapide* (*quicksort* en anglais). La méthode utilisée est connue sous l'expression « diviser pour régner » : elle consiste en effet à résoudre un problème en le réduisant à des problèmes identiques mais plus petits, jusqu'à parvenir à des cas de base que l'on sait résoudre simplement.

Plus précisément, on procède de la façon suivante : étant donnée une liste

$$[a_1; a_2; \ldots; a_n]$$

à trier, on choisit arbitrairement un élément a_p de la liste appelé *pivot* (par exemple le premier) et on sépare la liste en deux sous-listes, celle des éléments inférieurs ou égaux au pivot et celle des éléments strictement supérieurs au pivot.

$$\underbrace{[a_{i_1}; a_{i_2}; \ldots; a_{i_k}]}_{\leq a_p} \qquad a_p \qquad \underbrace{[a_{i_{k+1}}; a_{i_{k+2}}; \ldots; a_{i_{n-1}}]}_{> a_p}$$

On applique alors récursivement le même algorithme aux deux sous-listes. Le cas de base se résout trivialement lorsque l'on arrive à une liste ayant au plus un élément : elle est triée.

1. Écrire une fonction `avant` ayant la spécification suivante

   avant : entier, liste $\rightarrow$ liste
   { (avant x l) retourne les éléments de l inférieurs ou égaux à x }

   De même, écrire une fonction `apres` rendant les éléments de l strictement supérieurs à x.

2. En déduire une fonction `tri_rapide` implantant la méthode de tri décrite ci-dessus.

3. Critiquer le programme précédent du point de vue algorithmique.

4. Critiquer le programme précédent du point de vue Génie Logiciel.

Exercice 21
On peut représenter une matrice à coefficients entiers comme une liste de ses lignes, donc comme une liste de listes d'entiers. Par exemple, on peut représenter la matrice

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

en Caml comme `[[1;2;3];[4;5;6]]`.

1. Écrire une fonction Caml qui teste si une liste de listes d'entiers est bien une matrice, c.-à-d. si toutes les lignes ont la même longueur.

2. Écrire une fonction Caml qui calcule la somme de deux matrices, à l'aide d'une fonction auxiliaire qui calcule la somme de deux vecteurs.

3. Écrire une fonction Caml qui calcule le transposé d'une matrice, on utilisant des fonctions auxiliaires bien choisies.

4. Écrire une fonction Caml qui calcule le produit de deux matrices.

Chapitre 2

Aspects fonctionnels du langage

2.1 Fonctions d'ordre supérieur

2.1.1 Les fonctions comme objets de première classe

Comme expliqué dans l'introduction, une des caractéristiques des langages fonctionnels est que les fonctions sont des objets de première classe. Pour construire un objet de type fonction, on écrit

`function` *var* `->` *expression*

qui signifie « la fonction qui à *var* associe *expression* ». Par exemple

```
#  function x -> x+2;;
- : int -> int = <fun>
```

Le type d'un objet fonctionnel est noté t_1->t_2 où t_1 est le type de l'argument et t_2 le type du résultat. Une fonction, définie sans lui donner de nom, s'appelle fonction *anonyme*.

Sur un objet fonctionnel, l'opération principale est bien entendu l'application. Comme pour les fonction nommée, cette application se note en juxtaposant la fonction et son argument. Comme pour les fonctions nommées, il n'y a pas besoin de parenthèses autour de l'argument, par contre pour des raisons de priorités, il est quasiment toujours nécessaire de parenthèser la définition de la fonction. On écrira par exemple

```
#  (function x -> x+3) 2;;
- : int = 5
```

Comme tout autre objet, un objet fonctionnel peut être nommé par un `let` :

```
#  let f = function x -> 2*x+1;;
val f : int -> int = <fun>
```

```
#  f 4;;
- : int = 9
```

On remarque que la fonction ainsi définie est équivalente à la forme du chapitre 1 :

```
#  let f x = 2*x+1;;
val f : int -> int = <fun>
```

```
#  f 4;;
- : int = 9
```

De manière générale, l'écriture

```
let f x = ...
```

est une abréviation de

```
let f = function x -> ...
```

Exercice 22
Déterminer les réponses de Caml aux requêtes suivantes :

```
#  let f = function x -> x*x + 3*x + 1;;

#  f 1;;

#  f 2;;

#  f;;

#  let g = function x -> f(x+1)-f(x-1);;

#  g 3;;

#  g;;

#  let h = function x -> if x<4 then f x else g x;;

#  h 3;;

#  h 5;;

#  h;;
```

2.1.2 Fonctions à plusieurs arguments

Les fonctions anonymes à plusieurs arguments se définissent à l'aide du mot-clé `fun` de la manière suivante :

`fun` $var_1 \ldots var_n$ `->` *expression*

qui signifie « la fonction qui à $var_1,\ldots,var_n$ associe *expression* » (n peut être égal à 1, la déclaration étant alors équivalente à `function`; mais n ne peut pas être nul). Par exemple :

```
#(fun x y -> sqrt (x*.x+.y*.y)) 3.0 4.0;;
- : float = 5
```

De même que pour les fonctions anonymes à un argument, on peut nommer les fonctions anonymes à plusieurs argument, et de manière générale

```
let f x1 .. xn = ...
```

est une abréviation de

```
let f = fun x1 .. xn -> ...
```

2.1.3 Fonctions s'appliquant à des fonctions

Le langage des types est donné (partiellement) par la grammaire suivante :

Type → Type de base | Type `->` Type
Type de base → `int` | `float` | `char` | `string` | `bool`

Cela implique naturellement que l'on peut construire par exemple des expressions de type `(int -> int) -> int`, c'est-à-dire des fonctions s'appliquant à des fonctions :

```
#  let g = function f -> (f 0)+1;;
val g : (int -> int) -> int = <fun>
```

```
#  g (function n -> n*n+2);;
- : int = 3
```

Exercice 23

Écrire une fonction `croit` qui prend en argument une fonction f de type `int -> int` et qui retourne vrai si $f(0) \leq f(1)$.

Exercice 24
Quels sont les types des fonctions suivantes :

```
#  function f -> (f 0.0) ^ "a";;
```

```
#  let g = function f -> (f 0)+.1.2;;
```

Quelles sont alors les valeurs renvoyées par :

```
#  let h = function n -> if n>0 then 1.2 else 3.4;;
```

```
#  g h;;
```

2.1.4 Fonctions retournant des fonctions — application partielle

En mathématiques, il arrive parfois qu'une fonction ait comme résultat une fonction. Par exemple, on peut chercher à résoudre l'équation « paramétrée » suivante :

$$\lambda x^2 + (2\lambda + 1)x + 7 = 0$$

Il s'agit de résoudre une équation de la forme $f(\lambda, x) = 0$ mais où c'est x la variable, c'est-à-dire de manière équivalente $f_\lambda(x) = 0$ ou encore $(f(\lambda))(x) = 0$, c'est-à-dire que f est une fonction de λ dont le résultat est une fonction de x.

Cette équivalence entre une fonction à deux arguments et une fonction à un argument retournant une autre fonction à un argument se retrouve en Caml : il est possible également de définir des fonctions qui retournent des fonctions :

```
#  let f = function lambda -> (function x -> lambda*x*x+(2*lambda+1)*x+7);;
val f : int -> int -> int = <fun>
```

Le type de `f` affiché par Caml n'a pas de parenthèses : `int -> int -> int`, autrement dit il s'agit du type d'une fonction à deux arguments entiers retournant un entier. Cela vous montre effectivement qu'en Caml, une fonction à deux arguments est vue exactement comme une fonction du premier argument qui retourne une fonction du deuxième argument. Autrement dit encore, la syntaxe

```
fun x y -> ...
```

est simplement une abréviation pour

```
function x -> function y -> ...
```

La définition de `f` ci-dessus est donc parfaitement équivalente à

```
#  let f = fun lambda x -> lambda*x*x+(2*lambda+1)*x+7;;
val f : int -> int -> int = <fun>
```

De même, toutes les définitions suivantes sont équivalentes :

```
#  let delta = function f -> function x -> f (x+1) - 1;;

#  let delta = fun f x -> f (x+1) - 1;;

#  let delta f = function x -> f (x+1) - 1;;

#  let delta f x = f (x+1) - 1;;
```

Règle générale : un type t_1 `->` t_2 `->` t_3 est vu exactement comme t_1 `->` $(t_2$ `->` $t_3)$: on dit que `->` *associe à droite*.

La fonction `f` en tant que fonction qui retourne une fonction, peut parfaitement être appelée sur un seul argument, ce que l'on nomme *application partielle*. Cette application partielle donne alors une fonction qui peut être à nouveau appliquée :

```
#  f 2;;
- : int -> int = <fun>

#  (f 2) 4;;
- : int = 59
```

On a vu que les fonctions à deux arguments peuvent être appliquées à plusieurs arguments sans mettre de parenthèses : l'expression `(f 2) 4` est parfaitement équivalente à

```
#  f 2 4;;
- : int = 59
```

On dit que l'application *associe à gauche*.

Exercice 25
Déterminer les réponses de Caml aux requêtes suivantes :

```
#  let delta = function f -> function x -> f (x+1) - 1;;

#  let g=function x -> 3*x+2;;

#  delta g;;

#  (delta g) 4;;

#  delta (g 4);;

#  delta g 4;;
```

Exercice 26
Définir des expressions ayant pour type

```
(bool -> bool) -> bool
string -> string -> string
(float -> float) -> (float -> float)
(bool -> string) -> int
```

Exercice 27

1. Écrire une fonction `deriv` qui étant donnés une fonction f de $\mathbb{R}$ dans $\mathbb{R}$, un réel ε et un réel x, retourne la dérivée de f en x, approchée à l'ordre 2 par le taux d'accroissement sur $[x-\varepsilon, x+\varepsilon]$, c'est-à-dire
$$\frac{f(x+\varepsilon) - f(x-\varepsilon)}{2\varepsilon}$$
Déterminer le type de cette fonction avant de vérifier ce que Caml répond.

2. Utiliser la fonction `deriv` et l'application partielle pour définir en Caml, sans utiliser `fun` ou `function`, la fonction f' dérivée de $f(x) = x^2 + 3x + 4$ à 10^{-6} près. Calculer la valeur de cette dérivée en 0, 1 et 2.

2.2 Fonctions polymorphes

Il peut arriver que certaines définitions ne contraignent pas une variable à être d'un type particulier. L'exemple le plus simple est la fonction identité :

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

`'a` s'appelle une *variable de type*. Une telle fonction peut être appliquée à une valeur de n'importe quel type à la place de `'a` :

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

#  id 4;;
- : int = 4

#  id "bonjour";;
- : string = "bonjour"

#  id 5.6;;
- : float = 5.6
```

Une telle fonction s'appelle une fonction *polymorphe*. Nous avons déjà vu que les fonctions `longueur`, `concat` et `inv` sont polymorphes sur les listes. Nous avons également pu constater que les fonctions de comparaisons était polymorphes, mais attention : la comparaison d'expressions de type autre que les entiers, réels, caractères ou chaînes donne un résultat non spécifié, et est donc fortement déconseillé.

Exercice 28
Déterminer la réponse de Caml aux requêtes suivantes :

```
#  let iter f = fun x -> f (f x);;

#  let succ n = n+1;;

#  iter succ;;

#  iter succ 3;;

#  let compose f g = fun x -> f (g x);;

#  let g n = sqrt (float_of_int n);;

#  compose g succ;;

#  compose g succ 35;;
```

Exercice 29

1. Écrire une fonction `applique_liste` qui étant données une fonction f et une liste $[x_1; \ldots; x_n]$ retourne $[f(x_1); \ldots; f(x_n)]$. Quel est son type ?

2. Quel est le résultat de `applique_liste (fun x -> x*x) [1;2;4]` ?

   Note : cette fonction `applique_liste` est en fait prédéfinie en Caml, et s'appelle `List.map`.

3. Écrire une fonction qui étant donnée une liste $l = [x_1; \cdots; x_n]$ considérée comme un ensemble, retourne la liste de ses sous-ensembles (peu importe l'ordre) :

$$[[]; [x_1]; \cdots; [x_n]; [x_1; x_2]; [x_1; x_3]; \cdots; [x_1; x_n]; [x_2; x_3]; \cdots; [x_{n-1}; x_n]; \cdots; [x_1; \cdots; x_n]]$$

   Utiliser une méthode directe puis une méthode utilisant `List.map`.

4. Écrire une fonction `iter_liste` qui étant données une fonction f, une valeur a et une liste $[x_1; \ldots; x_n]$ retourne $f(\cdots f(f(a, x_1), x_2) \ldots, x_n)$. Quel est son type ?

5. Quel est le résultat de `iter_liste (fun x y -> x+y) 3 [4;5;8]` ?

   Note : cette fonction `iter_liste` est en fait prédéfinie en Caml, et s'appelle `List.fold_left`.

6. En utilisant `List.fold_left`, écrire deux définitions des fonctions qui calculent respectivement la somme d'une liste d'entiers et le produit d'une liste de réels.

Exercice 30

1. On désire écrire une fonction de tri qui classe en ordre décroissant, mais afin de suivre les principes de base du génie logiciel, on ne veut pas *dupliquer* la fonction de l'exercice 19, mais au contraire écrire une fonction *réutilisable*, c'est-à-dire qui pourra servir dans les deux cas. Pour cela, on code une fonction d'ordre supérieur : on va donner un paramètre supplémentaire qui sera une fonction de type `int -> int -> bool`, qui retourne `true` si son premier paramètre doit être mis avant le deuxième et `false` sinon. Ainsi, on doit pouvoir obtenir les résultats suivants :

```
#  insert_sort (fun x y -> x < y) [4;7;2;-6;9];;
- : int list = [-6; 2; 4; 7; 9]

#  insert_sort (fun x y -> x > y) [4;7;2;-6;9];;
- : int list = [9; 7; 4; 2; -6]
```

2. Utiliser la fonction de tri précédente pour trier des listes de chaînes de caractères par ordre alphabétique, ou bien par ordre croissant de longueur.

Exercice 31

1. Écrire une fonction `map_fun` qui étant donnée une liste de fonctions $[f_1; \cdots; f_n]$ et une variable x retourne la liste $[f_1(x); \cdots; f_n(x)]$. Prévoir le type de cette fonction avant de regarder la réponse de Caml.

2. Quel est le type de `map_fun [(fun x -> x+1); (fun x-> 2*x+3)]` ?

3. Écrire une fonction `it_fun` qui étant donnée une liste de fonctions $[f_1; \cdots; f_n]$ et une variable x retourne $f_1(f_2(\cdots f_n(x) \cdots))$. Prévoir le type de cette fonction avant de regarder la réponse de Caml.

4. Quel est le type de `it_fun [(fun x -> x+1); (fun x-> 2*x+3)]` ?

Exercice 32
Le but de cet exercice est d'écrire une fonction `permutations` qui étant donnée une liste $l = [x_1; \cdots; x_n]$, retourne la liste de ses permutations (peu importe l'ordre). Par exemple, `permutations [1;2;3]` doit renvoyer

$$[[1; 2; 3]; [1; 3; 2]; [2; 1; 3]; [2; 3; 1]; [3; 1; 2]; [3; 2; 1]]$$

1. Coder la fonction `insert_all` telle que (`insert_all` x $[y_1; ...; y_n]$) retourne $[[x; y_1; ...; y_n]; [y_1; x; y_2; ...; y_n]; ...; [y_1; ...; y_n; x]]$. Utiliser la fonction `List.map`.
2. Coder la fonction `concat_all` telle que (`concat_all` $[l_1; ..; l_n]$) retourne l_1`@`...`@`l_n. Utiliser la fonction `List.fold_left`.
3. En déduire un codage de la fonction `permutations`. Utiliser à nouveau la fonction `List.map`.

Chapitre 3

Types définis par l'utilisateur

Nous nous intéressons maintenant aux différentes constructions disponibles pour créer des types de données *complexes*, c'est-à-dire *composés* à partir des types simples vus au chapitre 1.

3.1 Couples et n-uplets

La première construction consiste à grouper des types pour former des couples et plus généralement des n-uplets :

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

#  let x = (3.5,"oui");;
val x : float * string = 3.5, "oui"

#  let y = (1,true,"sd");;
val y : int * bool * string = 1, true, "sd"
```

De manière générale, le type d'un n-uplet est noté t_1 `*` $\cdots$ `*` t_n où $t_1, \ldots, t_n$ sont les types des éléments. Caml fournit deux fonctions `fst` et `snd` qui permettent respectivement d'extraire le premier et le deuxième élément d'un couple :

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

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

Les fonctions anonymes sur les couples ou les n-uplets se définissent de la manière suivante :

`function` (var$_1$,...,var$_n$) -> *expression*

avec les abréviations habituelles.

Voici par exemple une fonction pour additionner les éléments d'un triplet d'entiers :

```
#  let add (x,y,z) = x+y+z;;
val add : int * int * int -> int = <fun>

#  add (3,5,9);;
- : int = 17
```

et une fonction polymorphe qui permute les éléments d'un couple :

```
#  let permute (x,y) = (y,x);;
val permute : 'a * 'b -> 'b * 'a = <fun>
```

```
#  permute (3,"a");;
- : string * int = "a", 3
```

Exercice 33
Écrire une fonction qui extrait le premier élément d'un triplet.

La construction `let` peut être utilisée pour définir directement des variables liées aux composants d'un n-uplets de la manière suivante :

> `let` (var$_1$,...,var$_n$) = *expression de type n-uplet*

Cette construction évite l'utilisation de `fst` et `snd`, ou même de définir des fonctions comme celle de l'exercice précédent.

Exercice 34

1. Écrire une fonction récursive qui étant donné un entier n retourne le couple $(fib(n), fib(n+1))$ mais ceci sans utiliser fib.
2. En déduire une méthode de calcul de fib.
3. Tester sur $fib(30)$, comparer avec l'exercice 11.

Exercice 35
Programmer à nouveau le tri rapide de l'exercice 20 en remplaçant les fonctions `avant` et `après` par une unique fonction spécifiée par :

> sépare : entier, liste $\rightarrow$ liste$\times$liste
> { (sépare x l) retourne un couple (l_1, l_2) où l_1 sont les éléments de l inférieurs ou égaux à x et l_2 sont les autres éléments. }

Exercice 36
Le *tri à bulles* est un algorithme de tri dont le principe est : tant que la liste donnée n'est pas triée, prendre deux éléments consécutifs qui ne sont pas dans le bon ordre, et les permuter.

1. Coder la fonction spécifiée par :

   > permute : liste d'entiers $\rightarrow$ liste d'entiers $\times$ booléen
   > { (permute l) recherche dans la liste l si deux éléments consécutifs x et y sont tels que $x > y$. Si oui, retourne le couple $(l', \texttt{true})$ où l' est la liste obtenue à partir de l en inversant x et y, et retourne $(l, \texttt{false})$ sinon. }

2. Coder alors la fonction de tri à bulles.
3. Généraliser ce tri en mettant une fonction de comparaison en paramètre. Que se passe-t-il si la fonction fournie est une comparaison au sens large, par exemple avec

   ```
   tri_bulle (fun x y -> x <= y) [2;1];;
   ```

Remarque : cette méthode de tri est très inefficace, et est donc déconseillée en pratique.

3.2 Enregistrements

La deuxième construction de type complexe est classique dans les langages de programmation : les enregistrements. On déclare un type enregistrement sous la forme :

type *nom_de_type* =
{
  étiquette$_1$: *type*$_1$;
  étiquette$_2$: *type*$_2$;
  ⋮
}

Une valeur de ce type est alors construite sous la forme :

{
  étiquette$_1$ = *valeur*$_1$;
  étiquette$_2$ = *valeur*$_2$;
  ⋮
}

Le i-ième composant peut alors être obtenu en suffixant par .*étiquette*$_i$. Voici par exemple un type enregistrement pour représenter une date :

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

#  let aujourd'hui = { jour=17; mois=10; annee=1996 };;
val aujourd'hui : date = {jour=17; mois=10; annee=1996}

#  aujourd'hui.jour;;
- : int = 17

#  aujourd'hui.mois;;
- : int = 10

#  aujourd'hui.annee;;
- : int = 1996
```

Exercice 37

1. Définir un type `complexe` représentant un nombre complexe par sa partie réelle et sa partie imaginaire.
2. Construire les complexes $1 - 2i$ et $-3.5 + 4i$.
3. Définir les fonctions d'addition, de soustraction, de multiplication et de division de nombres complexes.
4. Tester les fonctions précédentes sur quelques valeurs bien choisies.

Exercice 38

1. Définir un type `rat` représentant un nombre rationnel par un numérateur et un dénominateur.

2. Construire les rationnels $\frac{1}{2}$, $-\frac{3}{7}$.

3. Écrire une fonction qui normalise un rationnel, c.-à-d. qui retourne le rationnel équivalent où le dénominateur est positif et premier avec le dénominateur. Utiliser l'exercice 12.

4. Définir les fonctions d'addition, de soustraction, de multiplication et de division de rationnels.

5. Tester les fonctions précédentes sur quelques valeurs bien choisies.

Exercice 39

1. Définir un type `personne` caractérisé par un nom et une date de naissance.

2. Écrire une fonction qui teste si une personne d'un nom donné appartient à une liste de personnes.

3. Écrire une fonction qui ajoute une personne à une liste si celle-ci n'y appartient pas déjà.

3.3 Types énumérés

Un type énuméré est déclaré sous la forme :

`type` *nom_de_type* = $\mathit{Valeur}_1 \mid \cdots \mid \mathit{Valeur}_n$

les valeurs ainsi déclarées étant alors des constantes utilisables dans la suite. Par exemple :

```
#  type couleur =
#    Blanc | Noir | Rouge | Vert | Bleu | Jaune | Cyan | Magenta;;
type couleur =
    Blanc
  | Noir
  | Rouge
  | Vert
  | Bleu
  | Jaune
  | Cyan
  | Magenta

#  let c1 = Blanc;;
val c1 : couleur = Blanc

#  let c2 = Vert;;
val c2 : couleur = Vert
```

Notez que les noms ainsi définis doivent obligatoirement commencer par une majuscule.

Pour définir une fonction s'appliquant à un type énuméré, on peut le faire grâce à un pattern-matching, comme pour les listes : si *expression* est une expression ayant le type énuméré considéré, l'expression

`match` *expression* `with`
$\mid \mathit{Valeur}_1$ -> $\mathit{expression}_1$
$\mid \mathit{Valeur}_2$ -> $\mathit{expression}_2$
$\vdots$
$\mid \mathit{Valeur}_n$ -> $\mathit{expression}_n$

s'évalue en *expression*$_i$ si la valeur calculée de *expression* est *Valeur*$_i$. Le | de la première ligne est optionnel, mais c'est une bonne habitude de le mettre car cela simplifie les chose quand on insère ou on supprime des cas.

Par exemple on écrira :

```
#  let est_colore c = match c with
#    | Blanc -> false
#    | Noir -> false
#    | Rouge -> true
#    | Vert -> true
#    | Bleu -> true
#    | Jaune -> true
#    | Cyan -> true
#    | Magenta -> true;;
val est_colore : couleur -> bool = <fun>
```

```
#  let complementaire c = match c with
#    | Blanc -> Noir
#    | Noir -> Blanc
#    | Rouge -> Cyan
#    | Vert -> Magenta
#    | Bleu -> Jaune
#    | Jaune -> Bleu
#    | Cyan -> Rouge
#    | Magenta -> Vert;;
val complementaire : couleur -> couleur = <fun>
```

Pour abréger, il est possible d'utiliser la valeur « générique » _ qui signifie « autre cas » :

```
#  let est_colore c = match c with
#    | Blanc -> false
#    | Noir -> false
#    | _ -> true;;
val est_colore : couleur -> bool = <fun>
```

Exercice 40

1. Écrire un type représentant les villes suivantes : Berlin, Lyon, Londres, Marseille, New York, Paris, San Francisco et Tokyo.

2. Écrire un type représentant les pays suivants : Allemagne, États-Unis, France, Grande-Bretagne et Japon.

3. Écrire une fonction qui à chaque ville associe le pays où elle se trouve.

4. Écrire une fonction qui à chaque pays associe la liste des villes qui s'y trouvent.

Remarque : si l'on définit un pattern-matching où il manque des cas, Caml accepte la définition et se contente de donner un avertissement :

```
#  let est_colore c =
#    match c with
#    | Blanc -> false
#    | Noir -> false;;
```

```
Characters 26-80:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(Magenta|Cyan|Jaune|Bleu|Vert|Rouge)
val est_colore : couleur -> bool = <fun>
```

Dans un tel cas, si l'on appelle cette fonction sur un cas non prévu, l'évaluation s'arrête d'une façon analogue à une division par zéro : une exception « `Match_failure` » est levée :

```
#  est_colore Blanc;;
- : bool = false
```

```
#  est_colore Rouge;;
Uncaught exception: Match_failure ("", 26, 80).
```

3.4 Types sommes

Les types énumérés sont en fait des cas particuliers des *types sommes* que l'on introduit maintenant. Les types sommes sont des constructions particulières qui peuvent s'apparenter aux enregistrements avec variante de PASCAL ou aux unions de C. Alors qu'un n-uplet ou un enregistrement est utilisé pour représenter un produit cartésien au sens mathématique, les types sommes sont utilisés pour représenter des unions disjointes.

Une déclaration d'un type somme a la forme suivante :

`type` *nom_de_type* =
| $\mathit{Constructeur}_1$ `of` type_1
| $\mathit{Constructeur}_2$ `of` type_2
| $\mathit{Constructeur}_3$ `of` type_3
⋮

Une valeur de ce type est alors construite sous l'une des formes :

$\mathit{Constructeur}_i(\mathit{expr})$

où *expr* est une expression de type type_i. Notez qu'il est possible de mettre également des constructeurs sans argument, ce qui revient aux types énumérés décrits précédemment. Notez encore que, comme les types énumérés, les noms de constructeurs ainsi définis doivent obligatoirement commencer par une majuscule.

Voici par exemple un type `carte` qui permet de représenter les cartes d'un jeu de cartes :

```
#  type couleur = Pique | Coeur | Carreau | Trefle;;
type couleur = Pique | Coeur | Carreau | Trefle
```

```
#  type carte =
#   | As of couleur
#   | Roi of couleur
#   | Dame of couleur
#   | Valet of couleur
#   | Numero of int * couleur;;
type carte =
    As of couleur
  | Roi of couleur
  | Dame of couleur
  | Valet of couleur
  | Numero of int * couleur
```

La dame de cœur et le 9 de pique sont alors représentés par

```
#  let c1 = Dame(Coeur);;
val c1 : carte = Dame Coeur
```

```
#  let c2 = Numero(9,Pique);;
val c2 : carte = Numero (9, Pique)
```

Les fonctions sur les types sommes se définissent également par pattern-matching :

`match` *expression* `with`
| *Constructeur*$_1(x_1)$ -> *expression*$_1$
| *Constructeur*$_2(x_2)$ -> *expression*$_2$
| *Constructeur*$_3(x_3)$ -> *expression*$_3$
⋮

Voici par exemple une fonction qui à une carte donnée associe vrai si celle-ci est une figure :

```
#  let est_une_figure carte =
#  match carte with
#   | Roi(c) -> true
#   | Dame(c) -> true
#   | Valet(c) -> true
#   | _ -> false;;
val est_une_figure : carte -> bool = <fun>
```

```
#  est_une_figure c1;;
- : bool = true
```

```
#  est_une_figure c2;;
- : bool = false
```

De façon générale, une définition par pattern-matching est de la forme

`match` *expression* `with`
| *motif*$_1$ -> *expression*$_1$
| *motif*$_2$ -> *expression*$_2$
| *motif*$_3$ -> *expression*$_3$
⋮

où *motif* est n'importe quelle expression construite à l'aide de constructeurs (ou constantes) de types, de variables et du motif générique « _ ». Voici par exemple une fonction qui à une carte associe sa valeur à la belote, le premier argument de la fonction étant la couleur de l'atout :

```
#  let valeur_carte atout carte =
#  match carte with
#   | As(_) -> 11
#   | Roi(_) -> 4
#   | Dame(_) -> 3
#   | Valet(c) -> if c=atout then 20 else 2
#   | Numero(10,_) -> 10
#   | Numero(9,c) -> if c=atout then 14 else 0
#   | _ -> 0;;
val valeur_carte : couleur -> carte -> int = <fun>
```

1. Les variables d'un motif sont définies par ce motif, comme pour un let.
2. Une variable ne peut apparaître qu'une fois dans un motif.
3. Les motifs sont reconnus dans l'ordre de haut en bas.

FIG. 3.1 – *Les trois règles d'or du pattern-matching*

```
#  valeur_carte Carreau c1;;
- : int = 3
#  valeur_carte Carreau c2;;
- : int = 0
#  valeur_carte Pique c2;;
- : int = 14
```

Il existe trois règles très importantes à respecter pour le pattern-matching, qui sont présentées figure 3.1. Pour illustrer la première règle, voici une définition fausse de `valeur_carte` :

```
#  let valeur_carte atout carte =
#  match carte with
#  | As(_) -> 11
#  | Roi(_) -> 4
#  | Dame(_) -> 3
#  | Valet(atout) -> 20
#  | Valet(_) -> 2
#  | Numero(10,_) -> 10
#  | Numero(9,c) -> if c=atout then 14 else 0
#  | _ -> 0;;
Characters 129-137:
Warning: this match case is unused.
val valeur_carte : couleur -> carte -> int = <fun>
```

en effet le motif `Valet(atout)` filtre toutes les cartes qui sont des valets, en *définissant* une nouvelle variable `atout` différente du premier paramètre de la fonction. Ceci est illustré par le « warning » donné par Caml ainsi que par l'exemple :

```
#  valeur_carte Carreau (Valet(Pique));;
- : int = 20
```

Exercice 41

1. Écrire un type `nombre` qui permette de représenter des nombres, aussi bien entiers que réels.
2. Écrire les fonctions d'addition de multiplication sur les objets de type `nombre`.

3.5 Types récursifs

Les types peuvent être définis récursivement : par exemple, le type des arbres binaires d'entiers peut être défini par

```
#  type arbre =
#  | Feuille of int
```

```
#   | Noeud of int * arbre * arbre;;
type arbre = Feuille of int | Noeud of int * arbre * arbre
```

L'expression pour représenter l'arbre

```
        3
       / \
      1   4
     / \
    7   9
```

est

```
#  Noeud(3,Noeud(1,Feuille(7),Feuille(9)),Feuille(4));;
- : arbre = Noeud (3, Noeud (1, Feuille 7, Feuille 9), Feuille 4)
```

Exercice 42

1. Écrire une fonction qui retourne le nombre de nœuds d'un arbre.

2. Écrire une fonction qui fait la somme des valeurs contenues dans un arbre.

Exercice 43

1. Définir un type `expr` qui représente abstraitement les expressions arithmétiques définies par la grammaire
$$expr \to entier \mid expr + expr \mid expr * expr$$

2. Construire la représentation de $1 + 2 * 3$.

3. Écrire une fonction qui compte le nombre d'opérations dans une expression.

4. Écrire une fonction qui évalue une expression.

Exercice 44

1. Construire un type Caml qui permet de représenter des arbres binaires dont les nœuds sont les opérateurs binaires *plus, moins, mult et div*, et dont les feuilles sont des entiers. On appelera ces arbres, des arbres arithmétiques car ils représentent des expressions arithmétiques.

2. Donner le codage des expressions suivantes :

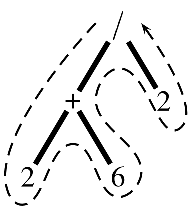
   – `3+2*5;`

   – `(3+2)*5;`

   – `2*3*(5+4)-2.`

3. On peut parcourir les arbres binaires de gauche à droite de trois façons différentes :

   – GRD (Gauche - Racine - Droite) : lors du parcours GRD, pour chaque nœud, on explore d'abord le fils gauche, puis la racine et enfin le fils droit ;



- GDR : Gauche - Droite - Racine ;
- RGD : Racine - Gauche - Droite.

Donner les fonctions Caml qui transforment un arbre binaire en liste (liste arithmétique) en fonction des trois parcours possibles.

4. Donner une fonction Caml qui évalue un arbre arithmétique, c'est-à-dire qui donne le résultat de l'expression représentée dans l'arbre.

5. On dira qu'un arbre arithmétique A est inférieur à un arbre arithmétique B si et seulement si l'évaluation de A est inférieure à l'évaluation de B. Donner les fonctions Caml de comparaisons d'arbres arithmétiques.

6. Donner une fonction Caml qui trie par insertion une liste d'arbres arithmétiques.

7. Donner les fonctions Caml qui évaluent les listes arithmétiques obtenues par un parcours RGD.

Exercice 45

1. Décrire un type `boolexpr` pour représenter des expressions booléennes construites sur les constantes `True` et `False`, et les opérateurs `And`, `Or` et `Not`.

2. Écrire une fonction qui évalue une expression booléenne.

3.6 Types polymorphes

Il est parfois utile de définir soi-même des types polymorphes. De tels types peuvent être déclarés en faisant précéder le nom du type d'une variable de type :

`type 'a` *nom_de_type* =
  déclaration de type utilisant `'a`

ou d'un n-uplet de variables de types :

`type ('a,'b,...)` *nom_de_type* =
  déclaration de type utilisant `'a`, `'b`, *etc.*

Ainsi, si les listes n'étaient pas prédéfinies dans le langage, on pourrait les définir soi-même par

```
#  type 'a liste =
#  | Liste_vide
#  | Cons_liste of 'a * 'a liste;;
type 'a liste = Liste_vide | Cons_liste of 'a * 'a liste
```

ce qui donne par exemple :

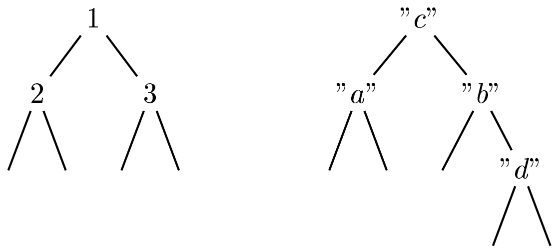
```
#  Cons_liste(1,Liste_vide);;
- : int liste = Cons_liste (1, Liste_vide)
```

```
#  Cons_liste("a",Cons_liste("b",Liste_vide));;
- : string liste = Cons_liste ("a", Cons_liste ("b", Liste_vide))
```

Exercice 46

1. Définir un type arbre binaire contenant des objets de type arbitraire, en autorisant l'arbre vide.

2. Donner des expressions Caml pour les arbres suivants



3. Définir des fonctions calculant la taille (nombre de nœuds) et la profondeur (nombre maximal de nœuds sur une branche) d'un arbre.

4. Définir une fonction qui calcule la somme des éléments d'un arbre de réels.

Exercice 47

1. Écrire une fonction d'insertion dans un arbre, de manière à avoir toujours un arbre binaire de recherche : la valeur à chaque nœud de l'arbre doit être plus grande que les valeurs du sous-arbre gauche et plus petite que les valeurs du sous-arbre droit. La fonction de comparaison de valeurs devra être passée en argument.

2. Écrire une fonction qui étant donnée une liste et une fonction de comparaison des éléments de cette liste retourne un arbre binaire de recherche contenant les éléments de la liste.

3. Écrire une fonction qui étant donné un arbre retourne la liste de ses éléments, parcouru dans l'ordre infixe : chaque sous-arbre est parcouru dans l'ordre sous-arbre gauche, racine, sous-arbre droit.

4. Écrire une fonction de tri par l'intermédiaire d'un arbre binaire de recherche.

5. En utilisant l'application partielle, écrire une fonction de tri de dates.

Chapitre 4

Les exceptions

Il arrive souvent que l'on veuille écrire des fonctions qui ne sont pas définies partout. Dans de tels cas, comme toute expression Caml doit avoir une valeur, il nous faut un moyen pour signaler qu'une fonction ne peut pas être calculée : on appelle cela des *exceptions*.

4.1 Exceptions prédéfinies de Caml

L'exemple le plus simple est la fonction de division, qui n'est pas définie lorsque le diviseur est nul :

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

On dit que l'exception de nom « `Division_by_zero` » a été *levée*.

Un autre cas d'exception est par exemple obtenu pour les fonctions `List.hd` et `List.tl`, lorsque la liste en argument est vide :

```
#  List.hd [];;
Uncaught exception: Failure "hd".
```

```
#  List.tl [];;
Uncaught exception: Failure "tl".
```

On dit que l'exception de nom « `Failure` » et d'argument `"hd"` (respectivement `"tl"`) a été levée.

Il faut remarquer que lorsqu'une exception a été levée, l'évaluation normale de l'expression en cours est stoppée, et l'exécution s'arrête en affichant l'exception qui a été levée (le calcul ne peut pas se poursuivre si une expression a levé une exception, sauf... voir 4.3) :

```
#  let x=3+6*(4+5/0);;
Uncaught exception: Division_by_zero.
```

Le calcul de l'expression n'ayant pas abouti, le let n'a pas eu lieu :

```
#  x;;
Characters 2-3:
Unbound value x
```

4.2 Exceptions déclarées par le programmeur

Il est possible de déclarer de nouvelles exceptions, grâce à une déclaration de la forme :

`exception` *Nom d'exception*`;;`

Notez que les noms d'exceptions doivent obligatoirement commencer par une majuscule.

Ensuite, une fonction peut lever une telle exception grâce à l'expression spéciale

`raise` *Nom d'exception*

Voici par exemple la fonction factorielle, qui teste si son argument est bien positif :

```
#  exception Argument_invalide;;
exception Argument_invalide

#  let rec fact n =
#    if n<0
#      then raise Argument_invalide
#      else if n=0
#             then 1
#             else n*(fact (n-1));;
val fact : int -> int = <fun>

#fact 4;;
- : int = 24

#fact (-2);;
Uncaught exception: Argument_invalide.
```

Exercice 48
Écrire en Caml la fonction *associe* dont la spécification est la suivante :

associe : `'a -> ('a * 'b) list -> 'b`

(`associe` a l) retourne b tel que le couple (a, b) appartient à l. Lève l'exception « `Non_trouvé` » si ce n'est pas possible. Faire attention à bien respecter les trois règles d'or du pattern-matching.

Remarque : en fait, cette fonction est prédéfinie en Caml, s'appelle `List.assoc`, et lève l'exception prédéfinie `Not_found` quand elle ne trouve pas.

Il est possible de définir aussi des exceptions avec argument par une déclaration de la forme

`exception` *nom d'exception* `of` *type*`;;`

Ainsi l'exception `Failure` mentionnée précédemment a été déclarée avec

```
exception Failure of string;;
```

Le programmeur peut lever lui-même cette exception prédéfinie de Caml :

```
#  let rec fact n =
#    if n<0
#      then raise (Failure "argument non autorise pour fact")
#      else if n=0
#             then 1
#             else n*(fact (n-1));;
val fact : int -> int = <fun>

#  fact 4;;
- : int = 24

#  fact (-2);;
Uncaught exception: Failure "argument non autorise pour fact".
```

Il existe même une fonction prédéfinie `failwith` à argument `string` dont le rôle consiste simplement à lever cette exception :

```
#failwith "essai de failwith";;
Uncaught exception: Failure "essai de failwith".
```

4.3 Capture d'une exception

Nous avons vu que la levée d'une exception interrompait l'évaluation normale. Ce comportement est parfois gênant : on voudrait pouvoir parfois appeler des fonctions qui risquent de lever des exceptions, et dans le cas où une exception est levée, faire un traitement particulier. Ceci peut se faire à l'aide de la construction de *capture* d'une ou plusieurs exceptions, qui a la forme analogue au pattern-matching suivante :

`try` *expression* `with`
| $exception_1$ `->` $expression_1$
| $exception_2$ `->` $expression_2$
⋮
| $exception_n$ `->` $expression_n$

et qui s'évalue en $expression_i$ si *expression* lève l'exception $exception_i$. Toutes ces expressions doivent avoir le même type.

Voici par exemple une fonction qui traduit un mot donné d'anglais en français à l'aide d'un petit dictionnaire. Les mots non trouvés dans le dictionnaire sont laissés tels quels.

```
#  let mon_dico = [("a","un"); ("called","appelé"); ("dog","chien");
#  ("hands","mains"); ("is","est"); ("language","langage");
#  ("my","mon"); ("us","nous"); ("wonderful","magnifique")];;
val mon_dico : (string * string) list =
  ["a", "un"; "called", "appelé"; "dog", "chien"; "hands", "mains";
   "is", "est"; "language", "langage"; "my", "mon"; "us", "nous";
   "wonderful", "magnifique"]

#  let traduire mot =
#    try (List.assoc mot mon_dico)
#    with Not_found -> mot;;
val traduire : string -> string = <fun>

#  let traduire_phrase = List.map traduire;;
val traduire_phrase : string list -> string list = <fun>

#  traduire_phrase ["Caml";"is";"a";"wonderful";"language"];;
- : string list = ["Caml"; "est"; "un"; "magnifique"; "langage"]

#  traduire_phrase ["my";"dog";"is";"called";"Snoopy"];;
- : string list = ["mon"; "chien"; "est"; "appelé"; "Snoopy"]

#  traduire_phrase ["let";"us";"shake";"hands"];;
- : string list = ["let"; "nous"; "shake"; "mains"]
```

Exercice 49

1. Décrire un type `expr` pour représenter des expressions arithmétiques avec des variables (syntaxe *abstraite* des expressions). Par exemple

```
#  Plus(Cte 1,Var "x");;
```

```
- : expr = Plus (Cte 1, Var "x")
#  Mult(Var "y",Plus(Cte 4,Var "z"));;
- : expr = Mult (Var "y", Plus (Cte 4, Var "z"))
```

2. Écrire une fonction qui évalue une expression dans un certain *environnement*, c'est-à-dire une liste de couples (nom de variable,valeur). Par exemple :

```
#  evalue [("x",4)] (Plus(Cte 1,Var "x"));;
- : int = 5
#  evalue [("y", 5);("z",6)] (Mult(Var "y",Plus(Cte 4,Var "z")));;
- : int = 50
```

Si l'expression contient une variable qui n'est pas dans l'environnement, la fonction devra lever une exception appropriée avec un argument indiquant le nom de la variable non définie :

```
#  evalue [("y", 5)] (Mult(Var "y",Plus(Cte 4,Var "z")));;
Uncaught exception: Variable_non_definie "z".
```

Exercice 50
On prolonge l'exercice 45

1. Décrire un type `boolexpr` pour représenter des expressions booléennes construites sur les constantes `True` et `False`, les opérateurs `And`, `Or` et `Not`, et les variables booléennes.
2. Écrire une fonction qui évalue une expression dans un *environnement de valeurs booléennes*, c'est-à-dire une liste de couples (nom de variable, booléen). Utiliser la fonction `List.assoc`.
3. Donner une fonction qui prend un `boolexpr` et qui retourne la liste de toutes ses variables. Une variable qui se trouve plusieurs fois dans l'expression doit apparaître une seule fois dans la liste.
4. Écrire une fonction qui retourne, pour une liste de variables, la liste de tous les environnements avec valeurs booléennes pour ces variables.
5. Écrire une fonction `pour_tous` de type `'a list -> ('a -> bool) -> bool` tel que `(pour_tous l f)` retourne `true` si et seulement si `f x` retourne `true` pour tous les éléments `x` de la liste `l`.
6. Écrire une fonction qui prend une expression du type `boolexpr` et qui retourne `true` si cette expression est une *tautologie*, c'est-à-dire si l'évaluation de cette expression retourne `true` pour tous les environnements.
7. En utilisant le fait que

$$\begin{array}{rcl} not(and(x,y)) & = & or(not(x),not(y)) \\ not(or(x,y)) & = & and(not(x),not(y)) \\ not(not(x)) & = & x \end{array}$$

écrire une fonction qui prend une expression du type `boolexpr` et retourne une expression du type `boolexpr` équivalente où il n'y a plus de `Not` sauf si appliqué directement à une variable.

Par exemple, la fonction appliqué à l'expression $not(and(x,or(y,z))$ doit retourner $or(not(x),and(not(y),not(z)))$.

Chapitre 5

Entrées-sorties

5.1 Le type unit

Il existe un type de base que l'on n'a pas encore rencontré : le type `unit`. Ce type ne possède qu'une seule valeur possible, notée `()`. Ce type sert à simuler des procédures avec des fonctions : une procédure sera simplement une fonction qui retourne `()`.

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

#  let test x = if x=0 then () else failwith "pas nul !";;
val test : int -> unit = <fun>

#  test 0;;
- : unit = ()

#  test 1;;
Uncaught exception: Failure "pas nul !".
```

Bien entendu, ces procédures ne sont pas très utiles tant que l'on ne sait pas faire des « effets de bord » visibles, comme les entrées-sorties.

5.2 Entrées au clavier et sorties à l'écran

La fonction `read_line` permet de lire une chaîne de caractères au clavier et retourne la chaîne lue comme résultat. Comme cette fonction n'a pas besoin d'argument, elle est définie ayant comme argument un objet de type `unit` :

```
#  read_line;;
- : unit -> string = <fun>
```

Voici un exemple d'utilisation :

```
#  let x = read_line ();;
bonjour, ca va ? (ceci est tapé au clavier)
x : string = "bonjour, ca va ?"
```

L'affichage d'une chaîne de caractères à l'écran peut se faire à l'aide de la fonction `print_string`, qui est une « procédure », c'est-à-dire une fonction qui retourne `()`.

```
#  print_string;;
```

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

Remarquer que cette fonction ne passe pas à la ligne automatiquement, ce qui explique que la réponse Caml `- : unit = ()` soit « collée » au message affiché. Pour forcer un passage à la ligne il faut utiliser

```
# print_newline;;
- : unit -> unit = <fun>

# print_newline();;

- : unit = ()
```

L'affichage d'un entier ou d'un réel se fait à l'aide des fonctions suivantes :

```
# print_int;;
- : int -> unit = <fun>
# print_float;;
- : float -> unit = <fun>
# print_int (4+6);;
10- : unit = ()
# print_float (sqrt 2.0);;
1.41421356237- : unit = ()
```

Pour passer à la ligne, il existe une autre méthode : afficher le caractère spécial « `\n` ». Ainsi :

```
# print_string "premiere ligne\ndeuxieme ligne\n";;
premiere ligne
deuxieme ligne
- : unit = ()
```

De même, on peut afficher une tabulation avec « `\t` », ce qui permet des affichages alignés par colonnes :

```
# print_string "1 \t : 34\n45 \t : 67\n";;
1       : 34
45      : 67
- : unit = ()
```

Voir section 1.4 pour les autres caractères spéciaux.

5.3 Enchaînement d'entrées-sorties

Lorsque l'on fait des entrées-sorties, on est rapidement amenés à vouloir faire plusieurs sorties de suite. Pour ce faire, il existe une syntaxe spéciale

begin
  expression$_1$;
  expression$_2$;
  $\vdots$
  expression$_{n-1}$;
  expression$_n$;
end

Comme tout ce qu'on écrit en Caml, ceci est une expression, avec donc une valeur, et pas une « instruction ». La valeur de l'expression est tout simplement la valeur de *expression*$_n$, les valeurs des autres expressions étant perdues. En pratique, toutes ces expressions seront de type `unit`.

Cette construction permet de faire des affichages intéressants. Voici une fonction qui demande de rentrer un nombre au clavier et affiche sa factorielle.

```
#let rec fact n = if n=0 then 1 else n*(fact(n-1));;
val fact : int -> int = <fun>
```

```
#let calc_fact () =
#  try
#    begin
#    print_string "Entrez un nombre : ";
#    let x = int_of_string (read_line ())
#    in begin
#       print_string "La factorielle de ";
#       print_int x;
#       print_string " est ";
#       print_int (fact x);
#       print_newline()
#       end
#    end
#  with Failure "int_of_string" ->
#        begin
#          print_string "Ceci n'est pas un nombre entier !";
#          print_newline()
#        end;;
val calc_fact : unit -> unit = <fun>
```

ce qui donne à l'exécution :

```
#  calc_fact ();;
Entrez un nombre : 4
La factorielle de 4 est 24
- : unit = ()
#  calc_fact ();;
Entrez un nombre : x
Ceci n'est pas un nombre entier !
- : unit = ()
```

5.4 Expression `if` sans `else`

Dans le contexte d'enchaînement des expressions avec des effets de bord il peut être utile d'avoir une instruction conditionnelle sans partie `else`. En fait, en OCaml une expression `if` *cond* `then` *expr* sans partie `else` est équivalente à `if` *cond* `then` *expr* `else` (). Noter que par conséquent, l'expression *expr* dans une conditionnelle sans partie `else` doit être du type `unit`, ou être une expression du type `raise` *exception*. Cela peut servir par exemple pour faire un affichage dans certains cas, ou aussi pour lever une exception.

```
#  exception Nombre_negatif
#  let compte_nombres n =
#    if n <= 0 then raise Nombre_negatif;
```

```
#    print_string "il y a ";
#    print_int n;
#    print_string " nombre";
#    if n > 1 then print_char 's';
#    print_newline ()
#;;
exception Nombre_negatif
val compte_nombres : int -> unit = <fun>

#compte_nombres 1;;
il y a 1 nombre
- : unit = ()

#compte_nombres 42;;
il y a 42 nombres
- : unit = ()

#compte_nombres (-3);;
Uncaught exception: Nombre_negatif.
```

Exercice 51
Reprendre le type des expressions de l'exercice 49 et faire une fonction d'affichage d'une expression abstraite sous forme complètement parenthésée. Par exemple,

```
#  affiche_expr (Mult(Var "y",Plus(Cte 4,Var "z")));;
(y*(4+z))- : unit = ()
```

5.5 Lecture et écriture de fichiers

En Caml, il existe deux types de données prédéfinis respectivement pour les *canaux* d'entrée et de sortie : `in_channel` et `out_channel`. Pour ouvrir un fichier en lecture on utilise

```
open_in : string -> in_channel
```

`(open_in s)` ouvre le fichier de nom `s` en lecture et retourne le canal correspondant. L'exception `Sys_error` est levée si le fichier n'existe pas, cette exception a un argument de type `string` contenant un message d'erreur précis.

Pour ouvrir un fichier en écriture, on utilise

```
open_out : string -> out_channel
```

`(open_out s)` ouvre le fichier de nom `s` en écriture, et retourne le canal correspondant. Si le fichier existe déjà, il est préalablement effacé.

Pour fermer un fichier, on utilise

```
close_in : in_channel -> unit
close_out : out_channel -> unit
```

qui ferment les canaux donnés en argument.

La lecture depuis un fichier fonctionne de manière analogue à l'entrée au clavier :

```
input_line : in_channel -> string
```

`(input_line f)` retourne une chaîne formé d'une ligne lue depuis le canal `f`. Lève l'exception `End_of_file` s'il n'y a plus rien à lire.

L'écriture dans un fichier se fait avec la fonction

```
output_string : out_channel -> string -> unit
```

`(output_string f s)` écrit la chaîne `s` dans le canal `f`.

Exercice 52
Écrire une fonction qui prend en argument deux noms de fichier f_1 et f_2, lit le fichier f_1 qui devra contenir un nombre entier par ligne, et écrire le fichier f_2 qui contiendra sur chaque ligne les valeurs lues suivies de la factorielle des nombres du fichier f_1. Gérer proprement tous les cas d'erreurs :

– fichier f_1 inexistant ;
– $f_1 = f_2$;
– une ligne de f_1 ne contient pas un nombre ;
– un de ces nombres est négatif.

Pour ces deux derniers cas, le fichier f_2 devra contenir le message approprié. Exemple de fichier d'entrée

```
1
2
3
4
5
-1
x
```

et le fichier de sortie correspondant :

```
1       1
2       2
3       6
4       24
5       120
-1      nombre negatif
x       n'est pas un nombre entier
```

La lecture et l'écriture peuvent aussi se faire de manière plus directe en lisant ou écrivant un seul caractère à la fois, avec les fonctions suivantes :

```
output_char : out_channel -> char -> unit
```

`(output_char ch c)` écrit le caractère `c` dans le canal `ch`.

```
input_char : in_channel -> char
```

`(input_char ch)` lit un caractère depuis le canal `ch`.

Exercice 53
Écrire une fonction qui prend en argument deux noms de fichier f_1 et f_2 et qui copie le ficher f_1 sur le fichier f_2. Dans un premier temps, ne pas tenir compte des cas d'erreurs. Dans un second temps, gérer les cas d'erreurs suivants :

– fichier f_1 inexistant ;
– $f_1 = f_2$;

5.6 Remarque sur l'ordre d'évaluation des expressions

Quand les expressions peuvent avoir des effets de bord l'ordre d'évaluation des expressions devient important. C'est-à-dire, étant donnée une application d'une fonction $(f e_1 \dots e_n)$, dans quel ordre les expressions $e_1, \dots, e_n$ sont-elles évaluées ? Voici un exemple où l'ordre d'évaluation est crucial :

```
#(* compter le nombre d'octets (les sauts de ligne exclus) depuis un canal *)
#let rec nb_octets ch =
#  try String.length (input_line ch) + nb_octets ch
#  with End_of_file -> 0
#;;
val nb_octets : in_channel -> int = <fun>
```

Le bon fonctionnement de cette fonction dépend de l'ordre dans lequel les deux arguments du $+$ sont évalués. Si l'argument de gauche est évalué le premier alors la fonction fait ce qu'on attend. Par contre si l'argument de droite est évalué le premier alors la fonction ne se termine pas.

Le documentation Objective Caml dit à ce propos que l'ordre d'évaluation des expressions *n'est pas spécifié*. Par conséquent si l'ordre est important, comme dans l'exemple au-dessus, il faut imposer l'ordre à l'aide d'un définition locale:

```
#(* compter le nombre d'octets (les sauts de ligne exclus) sur un canal *)
#let rec nb_octets input =
#  try let n = String.length (input_line input)
#      in n + nb_octets input
#  with End_of_file -> 0
#;;
val nb_octets : in_channel -> int = <fun>
```

En fait, la version 3.01 de Objective Caml évalue les expressions de droite à gauche, c.-à-d. que l'exécution de notre première version de la fonction `nb_octets` ne se termine pas.

Le cas est différent pour les opérateurs booléens : ils commencent l'évaluation de leurs arguments toujours avec l'argument gauche et évaluent l'argument droite seulement quand il est nécessaire pour déterminer le résultat.

```
#true && failwith "Coucou";;
Uncaught exception: Failure "Coucou".
```

```
#false && failwith "Coucou";;
- : bool = false
```

Par conséquent, `if` *cond* `then` expr `else false` est parfaitement équivalente à *cond* `&&` *expr*.

5.7 Graphisme

5.7.1 Disponibilité

Le graphisme n'est à priori pas disponible dans l'interpréteur interactif `ocaml`. Pour travailler avec le graphisme dans un interpréteur interactif il faut créer une version de l'interpréteur qui contient le graphisme :

```
ocamlmktop -o ocaml-graphisme graphics.cma
```

Cette commande effectuée vous pouvez lancer votre nouvel interpréteur `ocaml-graphisme` soit dans un interpréteur de commande (shell), soit dans le mode *tuareg* de l'éditeur *Emacs*.

Pour le compilation des programmes OCaml utilisant le graphisme l'outil `creemake` (voir Section 6.3) tient compte des options nécessaires.

Pour en savoir plus voir la chapitre *The graphics library* dans le manuel OCaml.

5.7.2 Initialisation

Le graphisme est disponible en Caml à l'aide du module `graphics`. Les commandes pour ouvrir une fenêtre graphique sont différentes pour les systèmes Windows et UNIX.

Windows : `open_graph "`lxh+x+y`"` ouvre une fenêtre graphique de largeur l et hauteur h où le point inférieur gauche se trouve la position (x, y) de l'écran. Un exemple des valeurs « raisonnables » est `open_graph "600x400+10+10"`. Le graphisme est disponible seulement dans l'interpréteur Caml mais pas dans les programmes compilés pour l'exécution autonome. Il peut être nécessaire de sélectionner la fenêtre dans le menu « Windows » de l'interpréteur Caml.

UNIX : (après avoir fait `open Graphics;;`)`open_graph " `lxh`"` ouvre une fenêtre graphique de largeur l et hauteur h (l'espace au début de l'argument est impératif !). Un exemple de valeurs « raisonnables » est `open_graph " 600x400"`. Pour utiliser le graphisme dans l'interpréteur Caml il faut lancer l'interpréteur `ocamlgraph`. Le graphisme est également disponible dans les programmes compilés pour l'exécution autonome quand le programme est compilé à l'aide d'outil `creemake` (voir section 6.3).

Pour en savoir plus sur les options de `open_graph` consulter le manuel Caml. Par ailleurs il y a les fonctions suivantes :

– `close_graph: unit -> unit` pour fermer la fenêtre graphique.

– `clear_graph: unit -> unit` pour effacer le contenu de la fenêtre graphique.

Toutes les expressions suivantes renvoie comme valeur `()` du type `unit` sauf dans les cas où c'est spécifié autrement. L'exception `Graphic_failure` avec un argument du type `string` est déclenchée quand les arguments d'une fonction ne sont pas valides.

5.7.3 Points et lignes

– `(plot x y)` dessine un point à la position (x, y) et positionne le curseur graphique en ce point ;

– `(moveto x y)` positionne le curseur graphique en (x, y) ;

– `(lineto x y)` dessine un trait du curseur graphique à (x, y) et positionne le curseur graphique en (x, y) ;

– `(set_line_width n)` sélectionne n comme épaisseur des lignes.

5.7.4 Textes

– `(draw_char c)` affiche le caractère c à la position actuelle du curseur graphique ;

– `(draw_string s)` affiche la chaîne s à la position actuelle du curseur graphique ;

– `(set_font_size n)` sélectionne n comme taille de fonte ;

– `(text_width s)` renvoie la paire $(largeur, hauteur)$ de la chaîne s quand elle est affichée dans la fonte courante.

5.7.5 Couleurs

Il y a un type `color` représentant les couleurs. Les constantes prédéfinies du type `color` sont `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan`, `magenta`.

– `(rgb r v b)` renvoie la couleur (du type `color`) avec les composantes rouges r, verte v et bleue b. Les valeurs légales pour les arguments sont de 0 à 255.

– `(set_color c)` sélectionne c comme la couleur courante ;

– `(fill_rect x y l h)` remplit le rectangle de largeur l, de hauteur h et de point inférieur gauche (x, y) par la couleur courante.

Par exemple, pour afficher un rectangle rose :

```
#let shocking_pink = rgb 255 105 180;;
shocking_pink : color = 16738740
#set_color shocking_pink;;
- : unit = ()
#fill_rect 100 100 200 200;;
- : unit = ()
```

5.7.6 Événements

Un *événement* se produit quand l'utilisateur clique sur un bouton de la souris, déplace la souris ou presse une touche du clavier. Le type `event` contient les formes différentes des événements :

```
type event = Button_down | Button_up | Key_pressed | Mouse_motion ;;
```

La fonction `wait_next_event` prend comme argument une liste l d'événements et attend le prochain événement appartenant à la liste l (les autres événements seront ignorés). Quand le premier événement se produit une description détaillée est renvoyée, du type `status` :

```
type status =
{
mouse_x    : int;  (* coordonnée x de la souris                *)
mouse_y    : int;  (* coordonnée y de la souris                *)
button     : bool; (* vrai si un bouton de la souris est enfoncé  *)
keypressed : bool; (* vrai si une touche du clavier a été pressée *)
key        : char; (* touche pressée du clavier le cas échéant    *)
}
```

Remarque : il n'y a aucune distinction entre les boutons de la souris.

Exemple : le programme suivant ouvre une fenêtre graphique et affiche un petit rectangle chaque fois qu'un bouton de la souris est pressé. Le programme se termine quand la touche `q` du clavier est pressée.

`waitevent.ml`

```
open Graphics;;

(* la fonction eventhandler traite les évenements *)

let rec eventhandler () =
  let eve = wait_next_event [Button_down;Key_pressed]
  in if eve.button           (* souris cliquée ?                *)
     then begin
```

```
       fill_rect eve.mouse_x eve.mouse_y 4 4;
       eventhandler()       (* attendre l'événement suivant     *)
     end
  else                      (* donc, c'était le clavier         *)
    if eve.key <> 'q'
    then eventhandler ()    (* attendre l'événement suivant     *)
    else ()                 (* terminer la récurrence           *)
;;

open_graph " 500x500";;
eventhandler ();;
close_graph ();;
```

fin de `waitevent.ml`

Exercice 54

1. Écrire un programme de peinture. Quand l'utilisateur bouge la souris et le bouton de la souris est pressé la trace de la souris doit être affichée sur la fenêtre graphique. La couleur actuelle peut être changée par pressant les touches `r` (rouge), `v` (vert), `b` (bleue) et `g` (gris). La touche `q` termine la séance.

2. Ajouter un changement de l'épaisseur du pinceau, par pressant une touche entre 1 et 9.

Exercice 55

1. Écrire une fonction qui prend comme argument une fonction f du type `float -> float` et quatre valeurs réelles $xmin$, $xmax$, $ymin$, $ymax$ et qui affiche le graphe de la fonction f sur l'intervalle $[xmin, xmax]$ où on suppose que les valeurs tombent dans l'intervalle $[ymin, ymax]$.

2. Écrire un programme principal qui affiche dans la même fenêtre et en des couleurs différentes les fonctions $\sin$ et $\cos$ sur l'intervalle $[0, 2\pi]$.

Chapitre 6

Compilation et programmation modulaire

6.1 Utilisation de Caml comme un compilateur

Si des phrases Caml sont stockées dans un fichier (portant l'extension `.ml`) alors on peut *compiler* ce fichier en tapant la commande suivante :

`ocamlc -o` *nom_programme fichier*`.ml`

Cela produit un fichier exécutable appelé *nom_programme*.(Sous MS-DOS, il faut que le nom de ce fichier se termine par `.exe` pour pouvoir être exécuté.)

La commande *nom_programme*, tapée sous l'interpréteur de commandes, lance alors l'exécution du programme de la même façon que s'il était évalué sous l'interpréteur, mais ni les types, ni les valeurs des expressions calculées ne sont affichés, aussi seules les entrées-sorties ont un effet visible. Par contre, il est parfois pratique de voir les types des expressions à la compilation, ce qui peut se faire grâce à l'option `-i` du compilateur :

`ocamlc -i -o` *nom_programme fichier*`.ml`

Recommendations importantes

Lorsque l'on exécute un programme Caml compilé, il faut respecter deux petits trucs simples :

- tous les programmes doivent se terminer par un `print_newline ()` : car sinon, la dernière ligne incomplète n'apparait pas à l'écran ;
- si le programme lève une exception et que celle-ci n'est pas capturée, l'exécution s'arrête en affichant simplement « `Uncaught exception` » sans que l'on sache de quelle exception il s'agit : pour le savoir, la meilleure méthode consiste à mettre les instructions à exécuter dans un procédure principale `main` et à appeler la fonction Caml prédéfinie `Printexc.catch` qui exécute une fonction en capturant et affichant toutes les exceptions. Pour que cela fonctionne parfaitement, il faut compiler avec l'option de *debugging* `-g`.

Ainsi, la forme générale d'un programme est

```
(* définitions des fonctions *)
...
(* procedure principale *)

let main () =
  begin
    ....
    print_newline ()
```

```
  end;;

Printexc.catch main ();;
```

et il vaut mieux le compiler avec les options :

`ocamlc -g -i -o` *nom_programme fichier*`.ml`

Exercice 56
Écrire un programme exécutable qui permet le calcul de factorielle suivant la méthode de la page 43.

6.2 Écriture de modules en Caml

En Caml, l'interface et le corps d'un module doivent être écrits dans des fichiers séparés. Le fichier contenant l'interface doit avoir l'extension `.mli`, alors que le corps doit avoir l'extension `.ml`. Le contenu du corps est formé d'expressions Caml habituelles. Par contre, l'interface contient des déclarations :

- déclarations de types au sens habituel ;
- déclarations de types *abstraits* : ce sont des déclarations de la forme simple

  `type` *identificateur* `;;`

  qui déclarent que le type « *identificateur* » existe mais que l'on ne connaît pas son codage; la définition complète de ce type devra être donnée dans le corps du module ;
- déclarations d'exceptions ;
- déclarations de valeurs, en général des fonctions, de la forme :

  `val` *identificateur* `:` *type* `;;`

  D'autre part, il faut donner la spécification de chacune des fonctions dans des commentaires.

La compilation d'un module se fait en compilant séparément l'interface et le corps. L'interface se compile avec la commande

`ocamlc -g -i -c` *fichier*`.mli`

ce qui produit un fichier *fichier*`.cmi`. Le corps se compile avec la commande

`ocamlc -g -i -c` *fichier*`.ml`

ce qui produit un fichier *fichier*`.cmo`.

Pour utiliser, dans un module `module2`, une fonction (ou un type, ou une exception, etc.) définie dans un autre module `module1`, il suffit de préfixer le nom de cette fonction par « `Module1.` ». Par exemple, la fonction déjà rencontrée `List.map` désigne en fait la fonction `map` du module `List`.

Lorsqu'un module `module2` utilise de nombreuses fois des éléments d'un module `module1`, il est pratique « d'ouvrir » le module `module1`, en utilisant la déclaration

```
open Module1;;
```

à l'endroit approprié (l'interface ou le corps) de `module2`. Pour que cette déclaration soit acceptée, il faut que le fichier *module1*`.cmi` existe.

Attention : il faut que les noms de module soient des identificateurs Caml valides, et qui doivent commencer par une majuscule. Le nom de fichier associé doit être le même, mais commençant en minuscule. Remarquer que cela oblige le nom du fichier a être un identificateur Caml valide, donc en particulier ne pas utiliser le caractère tiret − dans les noms de fichiers, utiliser plutôt le souligné _.

Voici par exemple le programme qui calcule la factorielle où la fonction factorielle est placée dans un module à part.

`fact.mli`

```
(*
 (fact n) retourne la factorielle de l'entier n.
 Lève l'exception Failure "argument invalide pour fact" si n est
 négatif
*)
val fact : int -> int ;;
```

fin de `fact.mli`

Le corps de ce module est

`fact.ml`

```
let rec fact n =
  if n<0
    then failwith "argument invalide pour fact"
  else if n=0
    then 1
  else n*(fact (n-1));;
```

fin de `fact.ml`

Le programme principal est alors :

`prog.ml`

```
open Fact;;

let main () =
  try
    begin
    print_string "Entrez un nombre : ";
    let x = int_of_string (read_line ())
    in begin
       print_string "La factorielle de ";
       print_int x;
       print_string " est ";
       print_int (fact x);
       print_newline()
       end
    end
  with
      Failure "int_of_string" ->
        begin
          print_string "Ceci n'est pas un nombre entier !";
          print_newline()
        end
    | Failure "argument invalide pour fact" ->
        begin
          print_string "Ce nombre est negatif !";
          print_newline()
```

```
        end
;;

Printexc.catch main ();;
```

fin de `prog.ml`

La compilation de ce programme à deux modules se fait en tapant

```
ocamlc -g -i -c fact.mli
ocamlc -g -i -c fact.ml
ocamlc -g -i -c prog.ml
```

Il reste alors à reconstituer un seul programme à partir de ces modules compilés : c'est la phase *d'édition des liens*. Elle est effectuée à l'aide de la commande

```
ocamlc -g -o prog fact.cmo prog.cmo
```

qui produit alors le fichier exécutable `prog` (fabriquer `prog.exe` sous MSDOS).

6.3 Les utilitaires `make` et `creemake`

On peut constater que pour un projet avec un grand nombre de modules, il est très fastidieux de taper toutes les commandes de compilation nécessaires.

Pour cela, la commande `make` permet de lancer automatiquement les commandes de compilation nécessaires pour compiler tous les modules d'un programme et de faire l'édition de liens. Les dates de modifications des fichiers sont prises en compte pour ne recompiler que ce qui a besoin de l'être.

Pour fonctionner, on doit fournir à `make` un fichier (appelé obligatoirement `makefile`) de description des modules. L'écriture d'un fichier `makefile` adapté à un projet n'est pas triviale, aussi il existe un deuxième outil `creemake` qui permet de créer automatiquement un fichier `makefile` convenant à un projet en plusieurs modules écrits en Caml.

6.3.1 Utilisation de `creemake`

On l'utilise en donnant en argument le nom du fichier principal du projet, sans l'extension `.ml`. Optionnellement, on peut donner aussi le répertoire des sources, ce qui est utile surtout sous MSDOS si les sources sont sur la disquette et que l'on désire compiler sur le disque `C:`.

`creemake` analyse le fichier principal, détecte automatiquement la liste des modules dont il a besoin, puis crée un fichier `makefile` correspondant dans le répertoire courant. Il suffit alors de taper `make` pour compiler le projet.

`creemake` crée aussi un fichier `include.ml` utile si l'on désire travailler sous l'interpréteur : il suffit d'inclure ce fichier en tapant

```
#use "include";;
```

pour charger tous les modules compilés et le programme principal dans l'interpréteur.

6.3.2 Exemple d'utilisation sous MSDOS

Le répertoire par défaut pour les fichiers sources est `a:`, c.-à-d. le répertoire courant de la disquette.

Le plus pratique consiste à écrire ses fichiers sources sur la disquette dans un répertoire (avec l'éditeur multi-fenêtre de Windows), puis, dans une fenêtre MSDOS, de se placer dans `c:\temp` et de taper la commande `creemake` nécessaire. Ainsi, si le fichier principal est `main.ml` dans le répertoire `a:\projet` de la disquette :

```
creemake main a:\projet
```

ou plus simplement

```
creemake main
```

si `projet` est déjà le répertoire courant de la disquette `a:`. Ensuite taper

```
make
```

pour compiler.

Note : sous MSDOS, `creemake` crée également un fichier `modules` qui contient la liste des modules du projet, ceci afin d'avoir une ligne de commande pour l'édition de liens qui ne dépasse jamais la limite imposée par MSDOS.

6.3.3 Exemple d'utilisation sous UNIX

Le répertoire par défaut des sources est le répertoire courant, et a priori il n'est pas utile de compiler dans un répertoire différent donc on se contentera de taper, dans le répertoire des sources :

```
creemake main
```

puis

```
make
```

pour compiler.

6.3.4 Exemple d'utilisation dans l'éditeur Emacs

Sous l'éditeur Emacs, le menu `Tuareg` permet de lancer directement dans l'éditeur les commandes de création du `makefile` (également touche `F5`) et de compilation (touches `F6` et `F7`). De plus, en cas d'erreur de compilation, on peut se positionner automatiquement à l'endroit où l'erreur s'est produite (touche `F8`).

6.3.5 Mise à jour du fichier `makefile`

Le fichier `makefile` a besoin d'être mis à jour à chaque fois qu'un nouveau module est créé ou bien quand une dépendance supplémentaire entre modules a été ajoutée par un `open`.

En dehors de ces cas, il suffit de taper `make` pour compiler à nouveau le projet après une modification.

Exercice 57

La compagnie HAL vous demande de réaliser en Caml un module qui permette de construire des ensembles finis d'entiers, la taille de ces ensembles pouvant être arbitrairement grande.

Dans le cahier des charges, la compagnie HAL précise qu'elle a besoin de construire ces ensembles élément par élément, en collectant des entiers dès qu'ils arrivent par une ligne de communication. Elle a aussi besoin ensuite d'afficher à l'écran les éléments d'un ensemble d'entiers, dans l'ordre croissant.

1. Spécifiez un module qui fournit les fonctionnalités désirées par la compagnie HAL.

Pour la réalisation du module promis à HAL, et comme il est nécessaire de pouvoir afficher les ensembles dans l'ordre croissant, vous décidez d'utiliser une structure de données appropriée : les arbres binaires de recherche (voir exercice 47).

Vous engagez un sous-traitant pour partager le travail. Le sous-traitant va réaliser les arbres binaires que vous utilisez.

2. Spécifiez à votre sous-traitant le module que vous désirez.

3. Écrire les modules correspondants en Caml.

4. Écrire un programme principal qui teste ces modules en construisant un ensemble à partir de valeurs lues au clavier, et l'affiche.

5. Créer un fichier `makefile` à l'aide de `creemake`, compiler et tester.

Exercice 58
On désire écrire un module permettant d'utiliser des *dictionnaires*. Un dictionnaire est une structure de données qui permet de stocker des informations de type quelconque, indexées par des noms (chaînes de caractères). Plus précisément ce module doit fournir les fonctionnalités suivantes :

– le type dictionnaire, polymorphe vis-à-vis du type des informations ;

– une fonction donnant un dictionnaire vide ;

– une fonction d'ajout d'une nouvelle entrée dans le dictionnaire ;

– une fonction qui recherche une information d'index donné dans un dictionnaire.

Voici un exemple d'utilisation de ce module :

```
let dico =
  ajout "Chameau" "Animal fétiche des programmeurs en Caml"
   (ajout "Touareg" "Surnom adapté pour un programmeur Caml"
      dictionnaire_vide);;
- : dictionnaire = <abstr>
recherche "Chameau" dico;;
- : string = "Animal fétiche des programmeur en Caml"
recherche "Dromadaire" dico;;
Uncaught exception : Index_absent "Dromadaire"
```

1. Écrire l'interface de ce module.

2. Écrire le corps de ce module, en représentant un dictionnaire par une liste de couples.

3. Écrire un autre version du corps de ce module, en représentant un dictionnaire par un arbre binaire de recherche (cf. exercice 47).

Chapitre 7

Aspect impératifs du langage

On a vu que Caml est classé parmi les langages fonctionnels. Mais l'interprétation et le compilateur OCaml fournit aussi des extensions de type impératifs. Nous décrivons dans ce chapitre une partie de ces extensions.

Il faut noter que ces aspects impératifs doivent être utilisés de manière très attentive, en ayant une bonne connaissance de leur sémantique. Une mauvaise utilisation peut provoquer des erreurs très surprenantes, en particulier à cause de partage de structure de données.

Dans un premier temps, il est conseillé de n'utiliser que les références, définies globalement (donc pas comme argument ou variable locale de fonction, en particulier dans le cas d'une fonction récursive), ainsi que la boucle `for` qui est peu risquée.

7.1 Les références

Une référence est un nouveau type polymorphe Caml : un objet de type `'a ref` est une référence sur un objet de type `'a`. On peut voir cela comme une flèche (ou un « pointeur ») qui montre l'objet référencé. On crée une référence avec le mot clé `ref` suivi de l'objet à référencer. Le cas le plus simple est une référence sur un entier, par exemple :

```
#let r = ref 4;;
val r : int ref = {contents=4}
```

qui définit `r` comme une référence sur l'entier 4. Note : la réponse de Caml `{contents=4}` est justifié dans la section 7.3.

Les deux opérations de base sur les références sont d'une part le *déférenciation*, c'est-à-dire l'accès à l'objet pointé, notée avec le point d'exclamation :

```
#!r;;
- : int = 4
```

et d'autre part l'opération de modification de la référence, notée `:=` à cause de sa ressemblance avec l'affectation de PASCAL.

```
#r:=5;;
- : unit = ()
```

```
#!r;;
- : int = 5
```

L'objet référencé peut être de n'importe quel type (même une fonction !). Par exemple on peut écrire

```
#let r = ref [1;2;3];;
```

```
val r : int list ref = {contents=[1; 2; 3]}

#r := 0 :: !r;;
- : unit = ()

#!r;;
- : int list = [0; 1; 2; 3]
```

7.2 Structures de contrôle impératives

La boucle `for` de Caml a la structure suivante :

`for` v= expr1 `to` expr2 `do`
expr3
`done`

où expr1 et expr2 sont des expressions entières, et expr3 une expression de type `unit`. La sémantique est d'exécuter $expr3$ successivement pour v prenant les valeurs de expr1 à expr2 incluses. Rien n'est exécuté si expr1 est supérieure à expr2. Par exemple

```
#for i = 1 to 10 do print_int i done;;
12345678910- : unit = ()
```

De manière analogue, il existe une boucle qui parcourt en descendant :

`for` v= expr1 `downto` expr2 `do`
...
`done`

Par exemple

```
#for i = 9 downto 0 do print_int i done;;
9876543210- : unit = ()
```

Il faut bien remarquer que la variable de boucle est une nouvelle variable introduite, comme avec `let`.

La boucle *tant que* a la forme

`while` expression booléenne `do`
...
`done`

Le contenu de la boucle est exécuté tant que la condition est vraie. Contrairement à la boucle `for`, la boucle `while` ne peut être utile que si l'on utilise aussi des références, qui font que la condition passe de vrai à faux.

7.3 Enregistrements avec champs modifiables

Il est possible de modifier sur place les champs d'un enregistrement, à condition de mettre dans la définition du type enregistrement le mot-clé `mutable` devant chaque champ que l'on veut être modifiable. L'instruction de modification de la valeur d'un champ s'écrit alors x`.`c `<-` v où x est une variable de type enregistrement, c un de ses champs modifiables et v la nouvelle valeur que l'on veut y affecter. Par exemple :

```
#type point = { mutable x:float; mutable y:float};;
type point = { mutable x : float; mutable y : float; }

#let p = {x=0.0;y=1.0};;
val p : point = {x=0; y=1}

#p.x <- 2.0;;
- : unit = ()

#p;;
- : point = {x=2; y=1}
```

Remarque : les références sont en fait un cas particulier d'enregistrement avec champs modifiables : il y a un unique champ `content` qui est modifiable.

7.4 Tableaux

Les tableaux sont analogues aux listes, mais en plus ils permettent d'accéder directement à un élément particulier à l'aide de son indice, et les cases de tableaux sont modifiables. Ils s'écrivent entre `[|` et `|]`. La première case a toujours l'indice 0.

```
#let table = [|1;2;5;7|];;
val table : int array = [|1; 2; 5; 7|]

#table.(3);;
- : int = 7

#table.(2) <- 8;;
- : unit = ()

#table;;
- : int array = [|1; 2; 8; 7|]
```

Les fonctions usuelles sur les tableaux sont définies dans le module `Array`. En particulier, la fonction `Array.length` donne le nombre d'éléments d'un tableau.

Voici par exemple une fonction qui ajoute 1 à tous les éléments d'un tableau :

```
#let ajoute table =
#  for i=0 to (Array.length table)-1 do
#    table.(i) <- table.(i) + 1
#  done
#;;
val ajoute : int array -> unit = <fun>

#let table =[|1;5;2;4;7|];;
val table : int array = [|1; 5; 2; 4; 7|]

#ajoute table;;
- : unit = ()

#table;;
- : int array = [|2; 6; 3; 5; 8|]
```

Autre exemple, voici une fonction qui trouve l'indice d'un élément donné dans un tableau :

```
#let cherche elem table =
#  let i = ref 0
#  and trouve = ref false
#  in
#    while (not !trouve & !i < Array.length table) do
#      if elem = table.(!i)
#      then
#        trouve := true;
#      i := !i +1
#    done;
#    if !trouve
#    then !i
#    else raise Not_found
#;;
val cherche : 'a -> 'a array -> int = <fun>
```

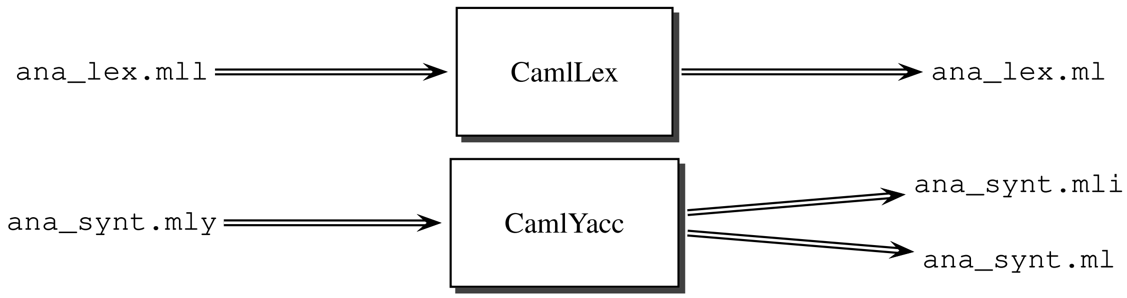
```
#cherche 4 [|1;5;2;4;7|];;
- : int = 4
```

```
#cherche 6 [|1;5;2;4;7|];;
Uncaught exception: Not_found.
```

Chapitre 8

Analyse syntaxique avec CamlLex et CamlYacc

Ces outils permettent d'engendrer automatiquement des procédures d'analyse lexicale et d'analyse syntaxique. Ils prennent tous deux en paramètre un fichier de description du langage à reconnaître et engendrent un module Caml, comme illustré par le schéma suivant :



Sur ce diagramme, le fichier `ana_lex.mll` est un *fichier de description Lex* que l'utilisateur doit écrire, et l'outil CamlLex fabrique automatiquement le fichier `ana_lex.ml`. De même, le fichier `ana_synt.mly` est un *fichier de description Yacc* que l'utilisateur doit écrire, et l'outil CamlYacc fabrique automatiquement les fichiers `ana_synt.ml` et `ana_synt.mli`.

8.1 Analyse lexicale avec CamlLex

L'exemple classique d'utilisation de Lex (mais ce n'est pas le seul) qui va nous intéresser est l'analyse lexicale. Le but de l'analyse lexicale d'un texte (chaîne de caractères, fichier d'un langage de programmation, etc.) est de décomposer ce texte en *unités lexicales*, qui sont les éléments de base du langage.

Par exemple, une analyse lexicale des expressions arithmétiques consiste à décomposer une expression en ces éléments de base : constantes, variables, symboles opératoires, parenthèses, etc.

Par exemple, l'expression `"12 + ( x*673 + y1)"` devra être décomposée en la séquence suivante : `"12"`, `"+"`, `"("`, `"x"`, `"*"`, `"673"`, `"+"`, `"y1"`, `")"`. Remarquer que dans cette décomposition, les espaces sont ignorées.

Une difficulté de l'analyse lexicale est de savoir décider si un mot est bien formé d'une seule unité lexicale, ou bien s'il doit être décomposé. Il n'existe pas de définition purement formelle de la notion d'unité lexicale : c'est un choix que l'on fait dans la définition d'un langage. Néanmoins, il existe un principe à toujours respecter : on ne s'autorise à décomposer un mot en plusieurs unités lexicales que si cela ne change pas sa signification. Ainsi, dans l'exemple des expressions arithmétiques, le mot `"y1"`, qui désigne la variable de nom y_1, ne peut pas être décomposé en la séquence `"y"`, `"1"` qui désigne la variable de nom y suivi de la constante 1. Par conséquent, le choix des unités lexicales d'un langage devra toujours être effectué de manière à ce qu'il soit autorisé d'insérer des espaces (ou des tabulations, ou des retours à la ligne) entre les unités lexicales, mais pas à l'intérieur de celles-ci. Par exemple, dans `"12+(x*673+y1)"`, les endroits où il est autorisé d'insérer des espaces sans changer la signification de l'expression sont donnés par `"12 + ( x * 673 + y1 )"`. L'insertion d'espaces à l'intérieur des unités lexicales provoque en général des erreurs de syntaxe.

8.1.1 Syntaxe des fichier de description Lex

Un fichier de description Lex contient une liste d'expressions rationnelles, où à chacune de ces expressions rationnelles est associée une *action*, c'est-à-dire une expression Caml à calculer lorsque qu'un mot est reconnu par l'automate associé à l'expression rationnelle.

Un tel fichier doit toujours avoir l'extension `.mll`. L'outil CamlLex prend en entrée ce fichier et produit comme résultat un fichier de même nom mais avec l'extension `.ml`. Ce fichier résultat contient un programme Caml réalisant un automate déterministe pour l'ensemble des expressions rationnelles du fichier.

La fonction principale du programme, dont le nom est choisi par l'utilisateur dans le fichier de description, prend en argument un *buffer d'analyse*, structure de données qui permet de mémoriser la chaîne ou le fichier en cours d'analyse, avec la position courante d'analyse. À chaque appel à cette fonction, elle essaye de reconnaître si *un préfixe* de la chaîne en cours d'analyse est accepté par l'automate, c'est-à-dire par l'une des expressions rationnelles du fichier de description. Dans ce cas, la fonction retourne la valeur de l'action associée. Par exemple, l'analyse lexicale d'une expression pourra prendre la forme suivante :

```
# let b = Lexing.from_string "12 + x1*673";;
b : lexbuf = <abstr>
# analyse b;;
- : string = "12"
# analyse b;;
- : string = "+"
# analyse b;;
- : string = "x1"
# analyse b;;
- : string = "*"
# analyse b;;
- : string = "673"
# analyse b;;
exception Fin_d_analyse
```

Les fichiers de description `.mll` ont le format suivant :

`{`
  déclarations Caml préliminaires
`}`
`let` nom_1 = $exprat'_1$
`let` nom_2 = $exprat'_2$
⋮
`rule` *nom de l'analyseur lexical* = `parse`
  $exprat_1$ `{` *action1* `}`
| $exprat_2$ `{` *action2* `}`
  ⋮

`'` *char* `'`	reconnaît le caractère donné
`_`	reconnaît n'importe quel caractère
`eof`	reconnaît la fin de l'entrée
`"` *string* `"`	reconnaît la chaîne de caractères donnée
`[` *set* `]`	reconnaît n'importe lequel des caractères de l'ensemble *set*. L'ensemble de caractères à reconnaître peut être donné par juxtaposition des caractères cherchés, et également par intervalles (par exemple `['a'-'z''_''\'']` reconnaît une lettre minuscule, un souligné ou une apostrophe).
`[^` *set* `]`	reconnaît n'importe lequel des caractères *qui n'appartient pas* à l'ensemble *set*.
exprat `*`	reconnaît toute concaténation (zéro ou plus) de mots reconnus par *exprat*
exprat `+`	reconnaît toute concaténation d'un mot ou de plusieurs mots reconnus par *exprat*
exprat `?`	reconnaît le mot vide ou un mot reconnu par *exprat*
$exprat_1 \mid exprat_2$	reconnaît tout mot reconnu soit par $exprat_1$, soit par $exprat_2$
$exprat_1\, exprat_2$	reconnaît tout mot qui est la concaténation d'un mot reconnu par $exprat_1$ et d'un mot reconnu par $exprat_2$
`(` *exprat* `)`	reconnaît les mots reconnus par *exprat*.
nom	correspond à l'expression rationnelle associée à *nom* dans les déclarations `let`.

FIG. 8.1 – *syntaxe CamlLex des expressions rationnelles*

```
;;
```

Les déclarations Caml préliminaires sont les fonctions auxiliaires utiles pour les actions. Pour nous, ce sera essentillement des `open` des modules où sont définis les types utilisés.

Les expressions rationnelles sont écrites à l'aide des constructions données dans le tableau de la figure 8.1. L'ordre de priorité des opérateurs est le suivant : `*`, `+` et `?` ont la plus haute priorité, puis la concaténation, et enfin $\mid$. Ainsi, $e_1 e_2$`*` signifie la même chose que $e_1(e_2$`*`$)$ et pas $(e_1 e_2)$`*`, et $e_1 e_2 \mid e_3 e_4$ signifie $(e_1 e_2) \mid (e_3 e_4)$ et pas $e_1(e_2 \mid e_3)e_4$, Les parenthèses permettent de forcer une autre priorité.

Une action est n'importe quelle expression Caml, mais toutes les actions doivent retourner une valeur du même type. On peut y utiliser l'expression `(Lexing.lexeme lexbuf)` qui retourne la chaîne qui a été reconnue.

Les `let` préliminaires permettent de donner des noms à expressions rationnelles utilisées dans la suite. Voici par exemple un fichier de description Lex reconnaissant des nombres entiers (séquences non vides de chiffres) :

```
let chiffre = ['0'-'9']
rule nombre_entier = parse
  chiffre+        { action }
```

Exercice 59
Donner des expressions rationnelles, dans la syntaxe Caml et en utilisant les `let` dans la mesure du possible, pour les langages suivants :

1. les entiers avec signe $+$ ou $-$ éventuel ;
2. les identificateurs PASCAL (séquences non vides de chiffres, lettres minuscules ou majuscules, ou _, se commençant par une lettre) ;

3. les chaînes de caractères (entre guillemets, contenant n'importe quelle séquence même vide de caractères autres que les guillemets) ;
4. les réels ;
5. les réels avec exposant éventuel ;
6. (difficile) les chaînes de caractères dans lesquelles on autorise d'écrire `\"` pour y mettre un caractère guillemets (par exemple `"le caractere \" est le guillemet"`;
7. (très difficile) les commentaires commençant par `(*` et finissant par `*)` (non imbriqués).

8.1.2 Le fichier Caml engendré par CamlLex

La commande

```
ocamllex ana_lex.mll
```

va alors créer un fichier `ana_lex.ml` qui sera un fichier source Caml définissant une fonction dont le nom est celui donné dans le fichier de description et de type `lexbuf ->` *type* où *type* est le type des actions. `lexbuf` est un type de données abstrait défini dans le module `Lexing` (défini dans la bibliothèque Caml) qui permet de mémoriser la chaîne ou le fichier en cours d'analyse, avec la position courante d'analyse. Un objet de ce type doit être créé à l'aide de l'une des deux fonctions suivantes définies dans le module `Lexing`:

```
val from_channel : in_channel -> lexbuf
```

`(Lexing.from_channel c)` crée une structure d'analyseur lexical à partir du canal d'entrée `c`, normalement créé par `open_in`.

```
val from_string : string -> lexbuf
```

`(Lexing.from_string s)` crée une structure d'analyseur lexical à partir de la chaîne `s`.

8.1.3 Un exemple complet

Voici par exemple un fichier Lex qui décrit une fonction de remplacement de `aabab` par `bb`:

`remplace.mll`

```
{
  (* pas de declaration preliminaire *)
}

rule remplace  = parse
  "aabab"     { "bb" }
| _           { (Lexing.lexeme lexbuf) }
| eof       { raise End_of_file }
;;
```

fin de `remplace.mll`

CamlLex sur ce fichier engendre une fonction

```
remplace : lexbuf -> string
```

On peut donc construire l'interface suivante

remplace.mli

```
(*
  (remplace buf) retourne à chaque appel la suite du contenu de buf,
  où le cas échéant, "aabab" est remplacé par "bb". Lève l'exception
  End_of_file s'il n'y a plus rien à lire.
*)

val remplace : Lexing.lexbuf -> string;;
```

fin de remplace.mli

Cette fonction peut alors servir à faire une fonction de remplacement en itérant la lecture :

test_remplace.ml

```
(*
  (boucle_remplace buf) itere le remplacement sur le buffer d'analyse
  buf et retourne la chaine ainsi produite.
*)

let rec boucle_remplace buf =
  try
    let res = (Remplace.remplace buf)
    in res ^ (boucle_remplace buf)
  with End_of_file -> ""
;;

(*
  (remplace_string s) retourne la chaine obtenue à partir de s en
  remplaçant toutes les occurrences de "aabab" par "bb"
*)

let remplace_chaine s =
  boucle_remplace (Lexing.from_string s);;

(*
  Un test
*)

let main () =
  begin
    print_string (remplace_chaine "ababbbabbaaababbbabbab");
    print_newline();
  end;;

Printexc.catch main ();;
```

fin de test_remplace.ml

Pour compiler ce programme, utiliser `creemake`, qui sait gérer les fichier de description Lex. L'exécution de ce programme donne alors :

```
ababbbabbabbbbabbab
```

Exercice 60
On désire réaliser un analyseur lexical pour les expressions arithmétiques avec constantes entières, variables, opérations $+$ et $\times$, et parenthèses. Pour les identificateurs de variables, on acceptera n'importe quelle séquence non vide de lettres majuscules, minuscules ou chiffres, ne commençant pas par un chiffre.

Écrire un fichier de description Lex permettant de réaliser cette analyse lexicale. La fonction réalisé devra, sur la donnée `"12 + ( x1*345+ yy)"`, produire la séquence de chaînes suivante :

```
constante 12
operateur +
parenthese ouvrante
variable x1
operateur *
constante 345
operateur +
variable yy
parenthèse fermante
```

Pour ignorer les espaces, tabulations et passages à la ligne, on placera dans le fichier Lex une expression rationnelle reconnaissant une séquence non vide de tels caractères, et on lui associera comme action un appel récursif à la fonction d'analyse lexicale.

8.1.4 Règles de priorité

Dans le cas où un mot analysé est accepté par plusieurs des expressions régulières données dans le fichier, le choix de l'action qui sera effectuée répond aux deux règles suivantes :

1. on doit toujours reconnaître la plus longue unité lexicale possible ;
2. à longueur égale, c'est la première action dans l'ordre donné dans le fichier qui est effectuée.

Pour illustrer ces règles, voici par exemple un fichier définissant deux expressions rationnelles :

```
{}
let lettre = ['a'-'z']
rule exemple = parse
  "var"  { "mot cle var reconnu" }
| lettre+ { "identificateur <" ^ (Lexing.lexeme lexbuf) ^ "> reconnu" }
;;
```

avec un tel fichier de description, la chaîne `"variable"` sera reconnue comme un identificateur en vertu de la règle 1, alors que la chaîne `"var"` sera reconnue comme un mot clé. Si l'on échangeait l'ordre des deux premières règles, `"var"` serait reconnue comme un identificateur.

Exercice 61
Écrire à l'aide de CamlLex une fonction qui lit un fichier contenant un programme en PASCAL et écrit (à l'écran) le programme obtenu en mettant les mots clés `begin`, `end`, `if`, etc. en majuscules, en laissant les autres identificateurs inchangés. Il faut faire attention aux identificateurs contenant des mots clés, et utiliser les règles de priorités. Par exemple le programme

```
BegIn
  endx := 3;
eNd.
```

devra être transformé en

```
BEGIN
  endx := 3;
END.
```

Exercice 62
Écrire une fonction qui prend deux noms de fichiers et qui dit si les deux fichiers contiennent le même text à des différences dans l'espacement près. Donc, un fichier peut avoir par example deux espaces entre deux mots là où l'autre a un seul espace, ou peut couper les lignes différemment.

Exercice 63
Écrire une fonction qui prend deux noms de fichiers f_1 et f_2 et qui copie le texte qui se trouve dans f_1 sur f_2, où la longueur des lignes de f_2 est bornée (disons, par 40). Les lignes de f_2 doivent être remplies au maximum et les séquences de plusieurs espaces doivent être compressées. Utiliser Lex pour l'analyse lexicale.

8.1.5 Notion de *token*

Un token est un nom pour désigner un ensemble d'unités lexicales de même nature : constantes, variables, etc. On dira par exemple que `"12"` est une unité lexicale de token `Cte` et de valeur entière 12. Le type suivant définit tous les tokens possibles des expressions arithmétiques :

```
type token =
  CTE of int     (* constantes *)
| ID of string   (* identificateurs de variables *)
| PLUS           (* operateur + *)
| MULT           (* operateur * *)
| PAR_GAUCHE     (* parenthese ouvrante *)
| PAR_DROITE;;   (* parenthese fermante *)
```

Exercice 64
Modifier le fichier de description Lex de l'exercice 60 afin de réaliser un analyseur lexical qui retourne des tokens.

8.2 Analyse syntaxique avec Yacc

Yacc est un outil plus puissant que Lex dans la mesure où il permet de reconnaître un langage reconnu par une grammaire non nécessairement régulière (mais quand même non-ambiguë). Ceci est donc utile pour reconnaître par exemple les expressions arithmétiques. Néanmoins, afin de ne pas engendrer une fonction d'analyse trop complexe, les portions simples du langage comme les entiers, les identificateurs, etc. resteront reconnus par un analyseur Lex, et la grammaire sera donc définie sur le vocabulaire formé par les tokens.

8.2.1 Syntaxe des fichier de description Yacc

Un fichier de description Yacc est en fait une description d'une grammaire, et également d'actions associées aux règles de grammaires. Il doit avoir l'extension `.mly` et le format suivant :

```
%{
   déclarations Caml préliminaires
%}
   déclarations préliminaires de la grammaire
%%
   règles de la grammaire
```

Les déclarations Caml préliminaires sont en général les `open` des modules dont on se sert dans les actions. Il est aussi possible de placer ici quelques fonctions auxiliaires pour ces actions.

Les déclarations de la grammaire permettent de définir les tokens sur lesquels la grammaire va travailler, et également de donner l'axiome de la grammaire. L'axiome de la grammaire est donné par

`%start` *non-terminal*

Les tokens sans valeurs sont déclarés par

`%token` id_1 id_2 ...

et les tokens avec valeurs sont déclarés par

`%token` <*type*> id_1 id_2 ...

où *type* est le type des valeurs de id_1, id_2, etc. Il y a encore une déclaration nécessaire :

`%type` <*type*> *non-terminal*

qui déclare que les actions associées au *non-terminal* sont des expressions de type *type*. En pratique, il est seulement obligatoire de déclarer le type de l'axiome, car Caml est capable de calculer alors le type des autres non-terminaux.

Les règles de grammaire sont données sous la forme

non terminal **:**
  $alternative_1$ `{` *action1* `}`
| $alternative_2$ `{` *action2* `}`
  ⋮
;

Les actions sont des expressions Caml qui seront évaluées à chaque fois que la règle de syntaxe correspondante sera utilisée. On peut utiliser les expressions spéciales \$1, \$2, etc. dans les actions, qui signifie la chose suivante : pour une règle $N \to N_1 \cdots N_k$, $\$i$ est la valeur de l'expression évaluée dans l'action qui a engendré N_i si N_i est un non-terminal, et la valeur du token si c'est un terminal.

8.2.2 Le module Caml engendré par CamlYacc

La commande

```
ocamlyacc ana_synt.mly
```

engendre deux fichiers, `ana_synt.mli` et `ana_synt.ml`.

Attention : si la grammaire donnée est ambiguë, CamlYacc donne des messages "shift/reduce conflicts" ou "reduce/reduce conflicts". Les fichiers Caml sont engendrés quand même, avec un choix arbitraire pour lever les ambiguïtés. **Pour être sûr de ce qui va se passer à l'analyse, il faut absolument se débrouiller pour avoir une grammaire non ambiguë.**

Le type `token` est alors défini à partir des déclarations de tokens donnés dans `ana_synt.mly`, et sa définition placée dans `ana_synt.mli` afin d'être utilisée par le fichier Lex qui reconnaîtra les tokens.

`ana_synt.ml` contient une fonction dont le nom est celui de l'axiome, et de type `(lexbuf -> token) -> lexbuf ->` *type* où *type* est le type des expressions utilisées comme actions dans la grammaire. Le premier argument de cette fonction est la fonction de reconnaissance des tokens. Si le langage analysé n'est pas reconnu par la grammaire, l'exception `Parse_error` (définie dans le module `Parsing` de la bibliothèque Caml) est levée.

8.2.3 Exemple des expressions arithmétiques

Supposons que l'on veuille écrire un programme qui lise des expressions arithmétiques données, disons pour simplifier avec `+`, `*` et les constantes entières, et les évalue. Pour ce faire, et afin de bien séparer l'analyse syntaxique de l'évaluation, on se définit d'abord un type `expr` qui décrit abstraitement les expressions arithmétiques :

`express.mli`

```
type expr =
  Cte of int
| Add of expr * expr
| Mul of expr * expr;;

val print_expr : expr -> unit;;
```

fin de `express.mli`

On va alors construire un analyseur syntaxique qui retourne une expression de type `expr`. Voici un fichier Yacc qui définit la grammaire des expressions et retourne une telle expression abstraite.

`ana_synt.mly`

```
%{
  open Express;;
%}

/* liste des tokens */

%token <int> ENTIER
%token PLUS MULT PARG PARD FIN

/* l'axiome, et son type */
%start entree
%type <Express.expr> entree

%%
/* liste des regles */
entree:
  expr FIN      { $1 }
;
expr:
  expr PLUS terme    { Add($1,$3) }
| terme              { $1 }
;
terme:
  terme MULT facteur { Mul($1,$3) }
| facteur            { $1 }
;
facteur:
  ENTIER             { Cte($1) }
| PARG expr PARD     { $2 }
;
```

fin de `ana_synt.mly`

Remarquer que dans la déclaration du type du non-terminal `entree`, on utilise la notation qualifiée `Express.expr`. Ceci est obligatoire car la déclaration `open Express;;` n'est copiée que dans le fichier `ana_synt.ml`, pas dans le fichier `ana_synt.mli`.

Le fichier Lex qui reconnaît les tokens est le suivant.

`ana_lex.mll`

```
{
  (* Le type des tokens est defini dans ana_synt.mli *)
  open Ana_synt;;
}

let chiffre = ['0'-'9']
let sep = [' ''\t''\n']

rule token_suivant = parse
  chiffre+   { ENTIER(int_of_string(Lexing.lexeme lexbuf)) }
| '+'        { PLUS }
| '*'        { MULT }
| '('        { PARG }
| ')'        { PARD }
| sep+       { token_suivant lexbuf }
| _      { failwith "caractere non reconnu" }
| eof       { FIN }
;;
```

fin de `ana_lex.mll`

La figure 8.2 montre quelles sont les dépendances entre tous ces fichiers. Une flèche double $f_1 \Rightarrow f_2$, signifie que le fichier f_2 est engendré automatiquement à partir de f_1 par CamlLex ou CamlYacc. Une flèche simple $f_1 \rightarrow f_2$ signifie que le fichier f_1 dépend de f_2 et donc doit être compilé après. Ces dépendances se retrouvent dans le fichier `MAKEFILE` engendré par `creemake`.

Voici par exemple le calcul de l'expression associée à $1 + 2 * 3$, qui vous montre comment le calcul est fait par l'analyse syntaxique *ascendante*. Sous chaque token ou non-terminal, on place sa valeur.

ENTIER	PLUS	ENTIER	MULT	ENTIER	$\rightarrow$
1		2		3	
facteur	PLUS	ENTIER	MULT	ENTIER	$\rightarrow$
`Cte(1)`		2		3	
terme	PLUS	ENTIER	MULT	ENTIER	$\rightarrow$
`Cte(1)`		2		3	
expr	PLUS	ENTIER	MULT	ENTIER	$\rightarrow$
`Cte(1)`		2		3	
expr	PLUS	facteur	MULT	ENTIER	$\rightarrow$
`Cte(1)`		`Cte(2)`		3	
expr	PLUS	terme	MULT	ENTIER	$\rightarrow$
`Cte(1)`		`Cte(2)`		3	
expr	PLUS	terme	MULT	facteur	$\rightarrow$
`Cte(1)`		`Cte(2)`		`Cte(3)`	
expr	PLUS	terme			$\rightarrow$
`Cte(1)`		`Mul(Cte(2),Cte(3))`			
expr					
`Add(Cte(1),Mul(Cte(2),Cte(3)))`					

Les fichiers créés par CamlLex et CamlYacc fournissent alors les fonctions

```
token_suivant : lexbuf -> token
entree : (lexbuf -> token) -> lexbuf -> expr
```

Voici à titre d'exemple un programme principal utilisant ces fonctions :

`test_expr.ml`

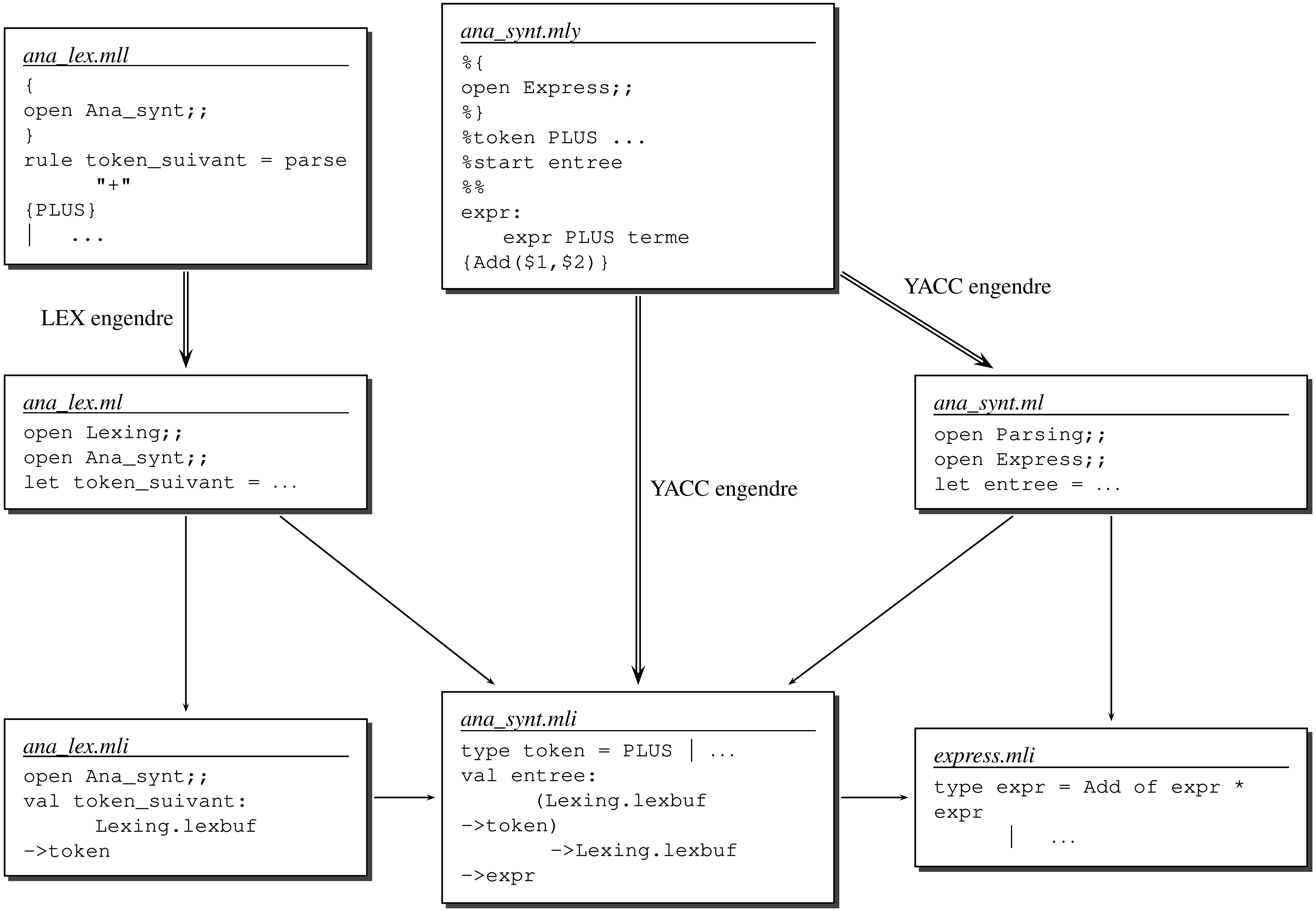


FIG. 8.2 – *Dépendances entre les fichiers de l'exemple des expressions arithmétiques.*

```
let expr_of_string s =
  let buf = Lexing.from_string s
  in Ana_synt.entree Ana_lex.token_suivant buf
;;

let test s =
 try
   let e = expr_of_string s
   in begin
        Express.print_expr e;
        print_newline ()
      end

 with Parsing.Parse_error ->
        begin
          print_string "Erreur de syntaxe";
          print_newline ()
        end
     | Failure "caractere non reconnu" ->
        begin
          print_string "Caractere non reconnu";
          print_newline ()
        end
;;

let main () =
  begin
    test "1*2+3*4";
    test "(1+2)*3";
    test "1+";
    test "tout faux";
  end
;;

Printexc.catch main ();;
```

fin de `test_expr.ml`

La compilation se fait en tapant `creemake test_expr` puis `make`, car creemake sait gérer les fichier Yacc et Lex.

Le résultat de l'exécution est alors :

```
Add(Mul(Cte(1),Cte(2)),Mul(Cte(3),Cte(4)))
Mul(Add(Cte(1),Cte(2)),Cte(3))
Erreur de syntaxe
Caractere non reconnu
```

Exercice 65

1. Écrire un module Caml définissant un type de donnée pour les expressions arithmétiques abstraites sur les entiers, sans variables, et contenant les opérations $+$, $-$, $\times$, div et mod. Fournir des fonctions d'affichage parenthésé et d'évaluation.

2. Écrire un programme principal qui crée quelques expressions, les affiche et affiche leur valeur. Compiler ce programme (avec `creemake` et `make`) et tester.

3. Écrire une grammaire non ambiguë pour ce langage. Décrire les tokens par des expressions régulières.

4. Écrire le fichier Lex et le fichier Yacc correspondant. Écrire l'interface du module engendré par CamlLex.

5. Écrire un programme principal avec une fonction qui prend en argument une chaîne de caractères, l'analyse pour en construire une expression abstraite, puis l'affiche sous forme parenthésée et enfin affiche sa valeur. Compiler et tester.

6. Écrire un programme qui lit un fichier contenant des séquences d'expressions séparées par des points-virgules, et affiche à l'écran les expressions parenthésées correspondantes et leurs valeurs.

Exercice 66

1. Reprendre l'exercice précédent en ajoutant les variables et les expressions « `let ... in ...` ». S'inspirer de l'exercice 49.

2. Rajouter les constantes `true` et `false`, les opérateurs de comparaison <, >, =, <>, >= et <=, ainsi que les opérateurs booléens `and`, `or` et `not`. Les priorités des opérateurs doit être les mêmes que dans Caml.

Exercice 67

1. Reprendre l'exercice précédent en complétant les actions de la grammaire pour retourner un couple, la première composante étant l'expression abstraite et la deuxième étant la liste de tous les variables de cette expression. La liste des variables d'une expression doit donc être calculée au fur et à mesure de l'analyse syntaxique.

2. Écrire un programme principal avec une fonction qui prend en argument une chaîne de caractères, l'analyse pour en construire une expression abstraite, puis affiche l'expression sous forme parenthésée ainsi que la liste de toutes les variables de cette expression.