un “CAML primer” pour l’option informatique des classes préparatoires

Alain Jean-Pierre Bèges
email : `alain.beges@enst-bretagne.fr`

version du document : 4 (janvier 1999)

Table des matières

I LE LANGAGE 5

1 premiers concepts 5
1.1 nombres entiers/flottants et définitions globales de variables . . . 5
1.2 définitions locales de variables . . . 6
1.3 expressions fonctionnelles et définitions de fonction; clôtures . . . 7
1.4 application partielle . . . 9
1.5 multiplets et produits cartésiens . . . 10
1.6 un peu de programmation fonctionnelle . . . 11
1.6.1 la composition des fonctions . . . 11
1.6.2 curryfication et dé-curryfication . . . 12
1.6.3 un exemple “algébrique” . . . 13
1.7 fonctions récursives . . . 13
1.7.1 définitions récursives . . . 13
1.7.2 notion de récursion terminale . . . 14
1.7.3 le jeu des tours de Hanoï . . . 16
1.8 listes . . . 17
1.9 quelques mots sur les motifs et le filtrage . . . 18
1.10 filtrage sur les listes . . . 19
1.11 quelques fonctions sur les listes . . . 21
1.12 le type ’unit’ . . . 22
1.13 le type ’bool’ (booléens) et les opérateurs associés . . . 22
1.14 le type ’char’ (caractère) . . . 24

2 les autres structures de contrôle 24
2.1 séquencement . . . 24
2.2 boucles . . . 25
2.2.1 la boucle “for” . . . 26
2.2.2 la boucle “while” . . . 27
2.3 exceptions . . . 27
2.3.1 distinguer déclenchement et évaluation d’une exception . . . 27
2.3.2 le type ’exn’ des exceptions . . . 28
2.3.3 rattrapage d’exceptions . . . 28

3 construction et usage de nouveaux types 29
3.1 produit cartésien et somme disjointe d’ensembles . . . 29
3.1.1 produit cartésien . . . 29
3.1.2 somme disjointe . . . 29
3.1.3 remarque . . . 30
3.2 types produit . . . 30
3.3 types produit paramétrés . . . 31
3.4 types somme . . . 32
3.5 les énumérations sont des sommes dégénérés . . . 32

3.6 types somme paramétrés . . . 33
3.7 types récursifs . . . 33
3.7.1 un type récursif non paramétré . . . 33
3.7.2 un type somme récursif paramétré . . . 34
3.7.3 remarque . . . 35
3.7.4 un type produit récursif paramétré . . . 35
3.7.5 type produit et somme mutuellement récursifs et paramétrés . . . 35
3.8 abréviations de type . . . 36

4 les types mutables 36
4.1 types produit mutables (“records” modifiables) . . . 36
4.2 types somme mutables . . . 37
4.3 les références . . . 37
4.4 les vecteurs . . . 38
4.5 les chaînes de caractères . . . 38

5 l’égalité en caml 39

II EXEMPLES 41

6 programmation “fonctionnelle” avec les nombres 41
6.1 calcul de pgcd et bibliothèque des grands nombres . . . 41
6.1.1 deux calculs du pgcd sur les entiers (¶) . . . 41
6.1.2 la librairie “num” . . . 41
6.1.3 quelques fonctions avec les grands nombres . . . 44
6.2 accumulation . . . 44
6.3 dichotomie (¶) . . . 47
6.4 point fixe et méthode de Newton . . . 47
6.5 fractions continues et séries hypergéométriques . . . 49

7 structures algébriques et types caml 52
7.1 la fonction puissance (¶) . . . 52
7.2 l’exponentiation “indienne” (¶) . . . 52
7.3 digression : la fonction puissance dans les monoïdes . . . 53
7.4 digression : structure de groupe . . . 53
7.5 ordre lexicographique (¶) . . . 54
7.5.1 première version . . . 54
7.5.2 généralisation . . . 54

8 encapsulation de plusieurs types : un exemple 55
8.1 approche fonctionnelle . . . 55
8.2 une autre approche . . . 57
8.3 dernière approche . . . 58

9 utilisation des “effets de bord” 58
9.1 fonctions à “variables rémanentes” . . . 58
9.2 un exemple amusant : une fonction qui s’use . . . 59
9.3 un autre exemple (moins amusant) : gestion de comptes bancaires . . . 59
9.4 un dernier exemple : comptage d’appels . . . 60

10 itérations bornée et non bornée **60**
10.1 itération bornée . . . 60
10.1.1 trois manières de voir les choses (¶) . . . 60
10.1.2 ces trois points de vue sont équivalents . . . 62
10.2 itération non bornée . . . 62
10.2.1 deux manières de voir les choses (¶) . . . 63
10.2.2 ces deux points de vue sont équivalents . . . 63
10.2.3 le “while” est plus “puissant” que le “for” (¶) . . . 63
10.3 passage à la récursivité terminale (quand cela est possible) . . . 64

11 structure de liste **65**
11.1 définition du type liste (¶) . . . 65
11.2 itérateurs sur les listes . . . 66
11.3 quelques fonctions sur les listes (¶) . . . 68
11.3.1 longueur d’une liste . . . 68
11.3.2 calcul du minimum . . . 68
11.3.3 n-ième élément d’une liste . . . 69
11.3.4 concaténation de deux listes . . . 69
11.3.5 image miroir . . . 69
11.3.6 insertion d’un élément . . . 70
11.3.7 le tri par insertion sur les listes . . . 70
11.3.8 suppression d’un élément . . . 70
11.3.9 le tri par sélection sur les listes . . . 71
11.3.10 le tri rapide sur les listes . . . 71
11.3.11 quelques versions (purement) itératives . . . 72

12 procédures de tri (¶) **72**
12.1 quelques utilitaires . . . 72
12.2 le tri bulle (bubble sort) . . . 73
12.3 le tri par sélection (selection sort) . . . 74
12.4 le tri par insertion (insertion sort) . . . 74
12.5 le tri rapide (quick sort), ou tri de Hoare, ou tri par segmentation . . . 74
12.6 le tri fusion (merge sort) . . . 75

13 deux utilisations des listes (¶) **75**
13.1 polynômes creux . . . 75
13.2 matrices creuses . . . 76

14 structure de pile (¶) **78**
14.1 les fonctions indispensables . . . 78
14.2 le “code” de ces fonctions . . . 78

15 manipulation de termes **79**
15.1 déclarations de type (¶) . . . 79
15.2 fonctions d’accès (¶) . . . 80
15.3 ce terme est-il valide?(¶) . . . 80
15.4 substitutions . . . 80

15.5 évaluation (¶) . . . 80
15.6 une fonctionnelle générale d'itération sur les termes . . . 81
15.7 des termes aux expressions postfixées (¶) . . . 81
15.8 évaluation des expressions postfixées à l'aide d'une pile (¶) . . . 81

16 analyse lexicale et analyse syntaxique : une introduction **82**
16.1 les flots caml . . . 82
16.1.1 le type "stream" des flots caml . . . 82
16.1.2 les flots sont paresseux . . . 83
16.1.3 regarder un flot, c'est le tuer (presque) . . . 83
16.1.4 motifs de filtrage "fonctionnels" . . . 85
16.1.5 construction du flot infini des nombres premiers par le crible d'Ératosthène 85
16.2 analyse lexicale et syntaxique . . . 87
16.2.1 mise en place formelle . . . 87
16.2.2 distinction du niveau lexical et du niveau syntaxique . . . 88
16.2.3 analyse lexicale et syntaxique d'un langage simple . . . 88
16.2.4 analyse lexicale de EAR . . . 90
16.2.5 analyse syntaxique de EAR . . . 91

17 un mini-langage fonctionnel **92**
17.1 présentation du mini-langage ($\mathcal{ML}$) . . . 92
17.2 les types CAML correspondants aux termes $\mathcal{ML}$ . . . 93
17.3 analyse lexicale de $\mathcal{ML}$ . . . 93
17.4 analyse syntaxique de $\mathcal{ML}$ . . . 93
17.5 l'interprète de $\mathcal{ML}$ . . . 94
17.6 amélioration des clôtures et donc de `eval` . . . 96
17.7 pretty-print de $\mathcal{ML}$ . . . 96
17.8 une boucle read-eval-print pour $\mathcal{ML}$ . . . 97
17.9 utilisation de $\mathcal{ML}$ . . . 98
17.9.1 utilisation de $\mathcal{ML}$: les booléens . . . 98
17.9.2 utilisation de $\mathcal{ML}$: les couples . . . 99
17.9.3 utilisation de $\mathcal{ML}$: les entiers . . . 100
17.9.4 utilisation de $\mathcal{ML}$: la récursivité . . . 101

III MISCELLANÉES 103

18 les versions 0.73 et 0.74 de caml **103**

19 bibliographie **104**

Partie I
LE LANGAGE

Ce document reprend un peu l'idée des "CAML primer"s, écrits (en anglais) pour d'anciennes versions de CAML *par Guy Cousineau et Gérard Huet. De larges emprunts ont également été faits à d'autres ouvrages ([1] et [6] en particulier); voir la bibliographie donnée à la fin de ce poly. On utilise la version 0.73 (Mac.3) de* CAML *(en version anglaise); les différences avec la version 0.74 (peu significatives, au niveau d'utilisation de* CAML *où se place ce poly) seront signalées au fil du texte. Ce document n'est pas un aide-mémoire des constructions et des fonctions disponibles en* CAML*; pour cet usage, reportez-vous à la "petite référence de* CAML*" (par Alain Bèges et Laurent Chéno), ou, en beaucoup plus complet, aux manuels de référence du langage [2] (en anglais).*

1 premiers concepts

1.1 nombres entiers/flottants et définitions globales de variables

CAML se présente le plus souvent sous la forme d'un interprète (ou interpréteur); vous avez l'habitude d'un interprète si vous utilisez le logiciel MAPLE ou même une simple calculatrice de poche : quand vous tapez '`1 + 3 ;;`' (suivi d'un "enter") vous obtenez la réponse. Lançons un CAML tout frais et essayons :

```
>       Caml Light version 0.73/Mac.3

#1 + 2 ;; (* ceci est un commentaire *)
- : int = 3
```

Le caractère '`#`' n'est pas à taper, c'est le "prompt" ou "signal d'invite" qui indique que CAML vous écoute; il sera désormais supprimé. La fin d'une phrase CAML est signalée par '`;;`'. Tout interprète est une boucle (infinie) lecture-évaluation-affichage.

Vous pouvez saupoudrer votre "code" CAML de commentaires : ces commentaires ne sont pas lus par l'interprète. Un commentaire commence par '`(*`' et se termine par '`*)`'. Les commentaires peuvent courir sur plusieurs lignes et ils peuvent être emboîtés.

Ceci dit, on voit que CAML indique simultanément le résultat et son type ('`int`' pour integer). On distinguera les entiers (`int`) des flottants (`float`) ou nombres "à virgules" :

```
1.0 +. 2.0 ;;
- : float = 3.0 (* la virgule en question est un point, Caml est anglophone *)
```

Noter que l'addition sur les entiers ('`+`') ne se note pas comme l'addition sur les flottants ('`+.`'); la même convention est appliquée aux autres opérations.

On peut, bien sûr, lier des variables dans l'environnement global (i.e. donner des définitions) :

```
let pi = 3.141592654 ;; (* ce qui se traduit par : "soit pi = 3.141592654" *)
pi : float = 3.141592654
```

On peut alors évaluer sans erreurs des expressions du genre :

```
pi *. 2.0 *. 2.0 *. 3.0 ;;
- : float = 37.699111848
```

On peut également donner des définitions multiples :

```
let h = 3.0 and r = 2.0 ;;

h : float = 3.0
r : float = 2.0
```

Puis évaluer l'expression suivante.

```
let v = pi *. r *. r *. h ;;
v : float = 37.699111848
```

1.2 définitions locales de variables

Les définitions introduites (à l'aide de la forme '`let ... and ...`') dans le paragraphe précédent sont permanentes : elles restent valides tant que vous n'abandonnez pas la session CAML en cours; ces *liaisons variable-valeur* sont aussi qualifiées de *globales*. On peut aussi introduire des *liaisons locales* à l'aide de la forme '`let ... in ...`' (ou '`let ... and ... in ...`') :

```
let y = (3 + 3) in (y + y) ;;
- : int = 12
```

Cette liaison de la variable '`y`' à la valeur `(3 + 3)` est dite locale car sa *portée* est limitée à l'expression `(y + y)`. L'identificateur '`y`' retrouvera son ancienne valeur (si elle existe) après l'évaluation de la forme '`let ... in ...`'.
On peut maintenant tester si '`y`' est lié de manière globale en essayant de l'évaluer :

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

Ce n'est donc pas le cas ici (unbound = non lié).
Bien sûr, les liaisons globales sont disponibles à l'intérieur de la forme '`let ... in ...`' :

```
let y = 3.0 in pi *. 2.0 *. 2.0 *. y ;;
- : float = 37.699111848
```

Nota Bene. Dans les formes

`let v_1 = e_1 and v_2 = e_2 and ...` (forme globale)

et

`let v_1 = e_1 and v_2 = e_2 and ... in ...` (forme locale)

les expressions `e_1`, `e_2`, etc. sont évaluées simultanément (en parallèle, si vous préférez) avant toute liaison :

```
let a = 4 ;;
a : int = 4

let a = a * a and b = a + 3 ;;
a : int = 16
b : int = 7
```

Pour effectuer des liaisons séquentiellement (celles de niveau k dépendant de celles des niveaux $k - i$) il suffit d'emboîter les formes `let` :

```
let x = 2 and x' = 3
in let y = 3 * x * x'
   in let z = 4 * (y + 1)
      in (x + y + z) ;;
- : int = 96
```

Enfin, il est important de remarquer que l'introduction de variables locales permet de partager des évaluations parfois coûteuses : il vaut mieux écrire (c'est plus lisible et plus économique)

```
let x = long_à_calculer(40)
in (x + 1) * (x + 2) * (x + 3) ;;
```

que l'idiotie suivante :

```
(long_à_calculer(40) + 1) *
   (long_à_calculer(40) + 2) *
      (long_à_calculer(40) + 3) ;;
```

1.3 expressions fonctionnelles et définitions de fonction; clôtures

En CAML les fonctions sont des valeurs comme les autres; on verra que l'on peut donner une (des) fonction(s) en argument à une fonction et qu'une fonction peut renvoyer une (des) fonction(s) en résultat. De telles fonctions "d'ordre supérieur" sont parfois appelées "fonctionnelles".

Pour l'instant, contentons-nous de remarquer qu'il n'est pas nécessaire de nommer une fonction pour qu'elle existe :

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

On constate que CAML est capable d'inférer (i.e. de déduire) le type de la fonction à partir de sa définition; ici cette fonction va des entiers vers les entiers : `int -> int`. Le '`= <fun>`' vous indique que cette valeur est une fonction (en anglais, function); *il sera supprimé par la suite (sauf nécessité ou inattention).*

Si l'on veut appliquer cette fonction à un argument (ici 2), on peut écrire :

```
(function x -> (x * x * x)) 2 ;;
- : int = 8
```

Si cette fonction est destinée à servir plusieurs fois, mieux vaut la nommer :

```
let cube = function x -> (x * x * x) ;;
cube : int -> int
```

Remarquer que cette syntaxe est en tout point analogue à une définition de variable comme '`let hauteur = 14.0 ;;`' (en fait, il n'y a pas de différence). Mais, variante syntaxique agréable, on peut aussi écrire

```
let cube x = x * x * x ;;
cube : int -> int
```

Pour appliquer cette fonction à un argument on écrira *indifféremment* '`cube 3`' ou '`cube(3)`' :

```
cube 3 ;;
- : int = 27
```

Voici une définition de fonction comportant une définition locale de fonction et une définition locale de variable (re-définition inutile de `pi`) :

```
let volume_cylindre r h =
   let carré x = x *. x and pi = 3.141592654
   in pi *. (carré r) *. h ;;

volume_cylindre : float -> float -> float (* il est bizarre ce type *)
```

Mais on aurait pu écrire aussi (`pi` existe déjà au niveau global)

```
pi ;; (* pi existe-t-il au niveau global ? *)
- : float = 3.141592654 (* oui *)

let volume_cylindre r h = pi *. r *. r *. h ;;

volume_cylindre : float -> float -> float
```

On peut maintenant calculer le volume d'un cylindre (droit à base circulaire) de rayon `r = 2.0` et de hauteur `h = 3.0` par

```
volume_cylindre 2.0 3.0 ;;
- : float = 37.699111848
```

Attention, important : Remarquer que si, maintenant, on modifie la valeur de `pi`...

```
let pi = 0.0 ;;
pi : float = 0.0

volume_cylindre 2.0 3.0 ;;
- : float = 37.699111848 (* c'est à n'y rien comprendre... mais si *)
```

...la fonction `volume_cylindre` n'est pas altérée. Cela mérite vraiment quelques mots d'explications. Dans une expression de la forme $x \mapsto f(x,a)$, la variable x est dite liée, et la variable a libre. Pour que cette expression ait un sens, il faut bien sûr que a (la variable libre) ait une valeur A donnée par un niveau englobant cette expression. Il en est de même en CAML. Mais CAML fait plus : quand il rencontre une expression de ce genre, vous pouvez considérer qu'elle est stockée en mémoire sous la forme d'un couple du genre

$$(\, a = A \, , \, x \mapsto f(x,a) \,)$$

qui mémorise la valeur de toutes les variables libres de l'expression (ici, il n'y en a qu'une seule, c'est a). On dit que l'on obtient une *clôture fonctionnelle*. Notons au passage que, dans une même expression, la même variable peut apparaître libre et liée. C'est par exemple le cas de l'expression suivante :

$$x \mapsto (a + ((a \mapsto a^2)\, x))$$

qui est une écriture contournée de $x \mapsto (a + x^2)$. *Ce mécanisme de clôture est général et essentiel en* CAML. *Lui seul permet de travailler correctement avec des fonctions admettant des fonctions en argument et produisant des fonctions en résultat : c'est ce mécanisme qui fait de* CAML *un "langage fonctionnel".*

On donnera ultérieurement (en partie **II**) le texte CAML de l'évaluateur d'un langage fonctionnel proche de CAML lui-même; nous y retrouverons ce mécanisme de clôture.

Enfin, le type de la fonction `volume_cylindre`, à savoir '`float -> float -> float`', vous inquiète peut-être un peu. C'est l'objet du paragraphe suivant que de dissiper cette inquiétude. Attention, lui non plus ne s'accomode pas d'une lecture superficielle.

1.4 application partielle

Une application partielle est l'application d'une fonction à quelques uns de ses arguments, mais pas à tous. Considérons la fonction `f` définie par :

```
let f x = function y -> (2 * x * y) ;;
f : int -> int -> int
```

L'expression `(f 3)` (ou '`f(3)`', ou '`f 3`') dénotera alors la fonction qui, étant donné un argument 'y', renvoie la valeur `(2 * 3 * y)`. L'application `(f x)` est qualifiée de partielle car `f` attend deux arguments successifs et n'en reçoit qu'un seul dans `(f x)`; la valeur de `(f x)` est encore une fonction...

```
f 3;;
- : int -> int
f 3 4;;
- : int = 24
let g = f 3 in g 4 ;;
- : int = 24
(f 3) 4 ;;
- : int = 24
```

On comprend maintenant le type de la fonction `f` (`f : int -> int -> int`); ce type doit se lire comme '`int -> (int -> int)`'; en effet, la fonction `f` attend un entier pour fabriquer une fonction des entiers vers les entiers. On retiendra également que '`f x y`' est compris par CAML comme '`(f x) y`'. Insistons :

Nota-Bene : Retenir que

```
F x y z
```

est compris (par CAML) comme

```
((F x) y) z
```

Retenir également que le type

```
'a -> 'b -> 'c -> 'd
```

doit être compris (par nous) comme

```
'a -> ('b -> ('c -> 'd))
```

Tant que nous y sommes, précisons un point de syntaxe.

Nota-Bene : La forme syntaxique

```
(function x -> (function y -> (function z -> expression)))
```

est équivalente à la suivante :

```
(fun x y z ->  expression)
```

Voici un autre exemple. Nous pouvons définir une fonction `addition` par

```
let addition x y = x + y ;;
addition : int -> int -> int
```

qui pourrait être écrite

```
let addition = function x -> (function y -> x + y)
```

ou encore

```
let addition = fun x y -> x + y
```

On peut alors définir une fonction `successeur` par

```
let successeur = addition 1;;
successeur : int -> int
```

Fonction `successeur` que l'on peut maintenant utiliser

```
successeur (successeur 1) ;;
- : int = 3
```

1.5 multiplets et produits cartésiens

Vous connaissez déjà deux types de base de CAML : les entiers (`int`) et les flottants (`float`); les autres types de base sont le type singleton (`unit`); les booléens (`bool`); les caractères (`char`).

Dans une certaine mesure (mais un peu moins que les types précédents) les types vecteurs (`'a vect`); chaînes de caractères (`string`; ce sont des vecteurs de caractères); couples/multiplets et listes (`'a list`) peuvent être considérés comme des types de base. Commençons par nous intéresser aux couples et multiplets.

Il est possible en CAML de combiner des valeurs de types *identiques ou différents* en couples ou multiplets. Le *constructeur* de multiplet est la virgule '`,`'; on utilise, la plupart du temps, une paire de parenthèses pour ensérer couples et multiplets mais cela n'est pas nécessaire : c'est une question de lisibilité.

```
1,2 ;;
- : int * int = 1, 2

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

(1,2,3,4,5) ;;
- : int * int * int * int * int = 1, 2, 3, 4, 5

let bizarre = (13, function x -> x + 1) ;;
bizarre : int * (int -> int) = 13, <fun>
```

L'identificateur '`*`' dénote le produit cartésien; par exemple, le type '`int * (int -> int)`' correspond au produit cartésien du type '`int`' et du type '`(int -> int)`'.

Noter que les multiplets s'évaluent "normalement" ('`succ`' est la fonction successeur) :

```
(succ 0,succ 1,succ 2,succ 3, succ 4) ;;
- : int * int * int * int * int = 1, 2, 3, 4, 5
```

Noter aussi que '`T1 * T2 * T3`', '`T1 * (T2 * T3)`' et '`(T1 * T2) * T3`' sont trois types distincts.

On peut extraire les deux composantes d'un couple par les deux projections `fst` (pour first) et `snd` (pour second) :

```
fst ;;
- : 'a * 'b -> 'a
snd ;;
- : 'a * 'b -> 'b
```

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

snd (1,2) ;;
- : int = 2

(snd bizarre) (fst bizarre) ;;
- : int = 14
```

Les fonctions '`fst`' et '`snd`' pourrait être définies par :

```
let first (x,y) = x and second (x,y) = y ;;

first  : 'a * 'b -> 'a
second : 'a * 'b -> 'b
```

1.6 un peu de programmation fonctionnelle

1.6.1 la composition des fonctions

Essayons de définir en CAML la composition des fonctions; on se propose d'écrire la fonction `compose` qui à un couple (f, g) de fonction associe la fonction $f \circ g$. Ceci peut être fait de la manière suivante en CAML :

```
let compose (f,g) = function x -> (f (g x)) ;;
compose : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

On constate (une fois de plus) que CAML a été capable d'inférer le type le plus général correspondant à la définition de `compose` :

- on donne à `x` le type polymorphe `'c` (arbitrairement);
- le type le plus général de la fonction `g` doit être alors `('c -> 'a)` (on introduit un nouveau type polymorphe `'a`);
- étant donné que le programme comporte l'expression `(f (g x))`, le type le plus général de `f` doit être `('a -> 'b)` (on introduit un nouveau type polymorphe `'b`);
- le type de `compose` doit donc bien être `('a -> 'b) * ('c -> 'a) -> 'c -> 'b` qui doit se lire `(('a -> 'b) * ('c -> 'a)) -> ('c -> 'b) .`

Essayons d'appliquer notre fonction `compose` :

```
compose (succ,succ) 1 ;;
- : int = 3
```

Ne pas oublier que

```
compose (succ,succ) 1
```

est compris par CAML comme

```
(compose (succ,succ)) 1
```

Nous aimerions pouvoir écrire la composition des fonctions comme en mathématiques : '`f o g`' (quitte à sacrifier l'identificateur '`o`' à cet usage); c'est possible en CAML en utilisant la directive '`#infix`' :

```
(* 'o' est la version curryfiée de 'compose', cf paragraphe suivant *)
let o f g x = f (g x) ;;
o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b

#infix "o" ;; (* il faut taper le '#' *)
```

Et, maintenant,

```
(succ o succ o succ) 0 ;;
- : int = 3
```

1.6.2 curryfication et dé-curryfication

Revenons vers notre fonction `volume_cylindre` :

```
let volume_cylindre r h =
  let pi = 2.0 *. (asin 1.0)
  in (pi *. r *. r *. h) ;;

volume_cylindre : float -> float -> float
```

Maintenant que nous disposons des couples, on peut en écrire une variante :

```
let volume_cylindre_variante (r, h) =
  let pi = 2.0 *. (asin 1.0)
  in (pi *. r *. r *. h) ;;

volume_cylindre_variante : float * float -> float
```

Ces deux fonctions diffèrent uniquement par la manière dont elles prennent leurs arguments : la fonction `volume_cylindre_variante` attend un couple de flottants alors que la fonction `volume_cylindre` attend un flottant et fabrique une fonction des flottants vers les flottants.

La fonction `volume_cylindre` est dite la version curryfiée (en l'honneur du mathématicien/logicien anglais Haskell Curry) de la fonction `volume_cylindre_variante`. Les fonctionnelles de curryfication et de décurryfication s'écrivent facilement en CAML :

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

curry   : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

ou, de manière plus élégante :

```
let curry f x y = f (x,y) and uncurry f (x,y) = f x y ;;
```

Finissons par une petite vérification de type :

```
curry volume_cylindre_variante ;;
- : float -> float -> float

uncurry volume_cylindre ;;
- : float * float -> float
```

1.6.3 un exemple “algébrique”

Vous savez probablement définir en mathématiques l'addition de deux fonctions $\mathbb{R} \to \mathbb{R}$: si f et g sont deux fonctions de ce type, la fonction $(f+g)$ est la fonction $\mathbb{R} \to \mathbb{R}$ définie (pour tout $x \in \mathbb{R}$) par $(f+g)(x) = f(x) + g(x)$. Essayons de programmer ceci en CAML; les flottants (`float`) nous servirons de (pauvre) approximation des réels ($\mathbb{R}$) :

```
let add_fonction (f,g) = function x -> f(x) +. g(x) ;;
add_fonction : ('a -> float) * ('a -> float) -> 'a -> float
```

Nous pourrions, de la même manière, définir la multiplication des fonctions $\mathbb{R} \to \mathbb{R}$. Plutôt que de travailler ainsi “au coup par coup” nous pouvons essayer de passer l'opérateur binaire (soit $+$, soit $\times$, soit ...) en argument à une fonction `étendre_loi` :

```
let étendre_loi opération_binaire =
    function (f,g) -> (function x -> opération_binaire(f x, g x)) ;;

étendre_loi : ('a * 'b -> 'c) ->
                ('d -> 'a) * ('d -> 'b) ->
                   'd -> 'c
```

Nous constatons alors que nous obtenons plus que prévu; en effet, le type de cette fonction s'interprète comme suit : `opération_binaire` peut, en fait, être n'importe quelle application de type $A \times B \to C$; la fonction f a pour type le plus général $D \to A$ tandis que la fonction g a pour type le plus général $D \to B$; l'algorithme fabrique alors une nouvelle fonction de type $D \to C$.

Avant de tester cette fonction, mentionnons que l'on peut obtenir la forme curryfiée des opérateurs *infixes* par le mot-clef `prefix` :

```
prefix + ;;
- : int -> int -> int

prefix +. ;;
- : float -> float -> float

let add_float = uncurry (prefix +.)
and mul_float = uncurry (prefix *.) ;;

add_float : float * float -> float
mul_float : float * float -> float
```

Testons, maintenant :

```
let add_étendue = étendre_loi add_float
and mul_étendue = étendre_loi mul_float ;;

add_étendue : ('_a -> float) * ('_a -> float) -> '_a -> float
mul_étendue : ('_a -> float) * ('_a -> float) -> '_a -> float
```

Vous pouvez vous-même poursuivre les investigations. (Ne vous inquiétez pas du '`_`' (underscore) devant le 'a', c'est la “cuisine interne” de CAML : cela nous entraînerait trop loin d'expliquer. Vous devez considérer que '`_a`' est un type au même titre que 'a'.)

1.7 fonctions récursives

1.7.1 définitions récursives

Lors de la création d'une liaison (globale ou locale), c'est à dire lorsque l'on utilise la forme '`let x = E`', les variables qui figurent dans l'expression '`E`' doivent avoir été définies à un niveau

englobant. Si la variable 'x' elle-même apparaît dans 'E' cela fait référence à une définition de 'x' pré-existente :

```
>       Caml Light version 0.73/Mac.3 (* un Caml tout frais *)

let x = x + 2 ;;
Toplevel input:
>let x = x + 2 ;;
>        ^
The value identifier x is unbound.

let x = 1 ;;
x : int = 1

let x = x + 2 ;;
x : int = 3
```

Essayons de définir la factorielle par

```
let factorielle n = if (n = 0)
                       then 1
                       else n * factorielle (n - 1) ;;
Toplevel input:
>                       else n * factorielle (n - 1) ;;
>                                ^^^^^^^^^^^
The value identifier factorielle is unbound.
```

Cette définition est rejetée car la définition de `factorielle` fait intervenir une liaison `factorielle` inexitante au niveau englobant.

Heureusement, il y a, en CAML, deux constructions spéciales pour introduire les définitions récursives : c'est

(1) la forme '`let rec ... and ...`' pour les définitions globales, et

(2) la forme '`let rec ... and ... in ...`' pour les liaisons récursives utilisées localement.

```
let rec factorielle n =
   if n = 0
      then 1
      else n * factorielle (n - 1) ;;

factorielle : int -> int

factorielle 6 ;;
- : int = 720
```

On peut également définir des fonctions mutuellement récursives comme dans

```
let rec pair n = if (n = 0) then true  else impair (n - 1)
and impair n = if (n = 0) then false else pair (n - 1) ;;

pair   : int -> bool
impair : int -> bool
```

1.7.2 notion de récursion terminale

Notre définition de la factorielle était :

```
let rec fact1 n =
   if n = 0
      then 1
      else n * fact1 (n - 1) ;;
```

Essayons une autre définition :

```
let rec fact2 n r = (* 'r' comme résultat *)
   if n = 0
      then r
      else fact2 (n - 1) (n * r) ;;

fact2 : int -> int -> int

fact2 6 1 ;;
- : int = 720
```

Essayons d'imaginer, dans les deux cas, le calcul de 3! :

```
(fact1 3) = 3 * (fact1 2)
(fact1 2) = 2 * (fact1 1)
(fact1 1) = 1 * (fact1 0)
(fact1 0) = 1
=> (fact1 1) = 1 * 1 = 1
=> (fact1 2) = 2 * 1 = 2
=> (fact1 3) = 3 * 2 = 6
```

Alors que

```
(fact2 3 1) = (fact2 (3-1) (3*1)) = (fact2 2 3)
(fact2 2 3) = (fact2 (2-1) (2*3)) = (fact2 1 6)
(fact2 1 6) = (fact2 (1-1) (1*6)) = (fact2 0 6)
(fact2 0 6) = 6
```

Nous constatons que le calcul de 3! par '`fact2`' n'a nécessité aucune sauvegarde intermédiaire des calculs "en attente", contrairement au calcul par '`fact1`'. La version '`fact2`' de la factorielle est dite *récursive terminale*; elle est à préférer à la version '`fact1`' car la version '`fact2`' *est aussi efficace qu'une version itérative*. À titre d'exercice, le lecteur peut chercher une version récursive terminale de la suite de Fibonacci; là le gain est d'importance, car, par rapport à la version récursive naïve, le temps de calcul, d'exponentiel devient linéaire. Voici une version récursive naïve :

```
let rec fib n =
   if n < 2
      then n
      else fib(n - 1) + fib(n - 2) ;; (* temps exponentiel *)
```

Ceci dit, il est désagréable de devoir fournir deux arguments à la fonction '`fact2`', alors que la factorielle n'en admet qu'un. Pour pallier cet inconvénient, on peut utiliser une définition locale de fonction récursive (noter, dans la fonction suivante, que c'est `aux` qui est récursive, et non `fact`) :

```
let fact n =
   let rec aux k r =
      if k = 0
         then r
         else aux (k - 1) (k * r)
   in aux n 1 ;;
```

Signalons enfin, qu'hélas (pas forcément hélas, en fait...), il n'est pas toujours possible de transformer une fonction récursive arbitraire en itération. (Un contre-exemple célèbre est donné par Ackermann :

```
let rec Ackermann = function
   | (0,n) -> n+1
   | (m,0) -> Ackermann(m-1,1)
   | (m,n) -> Ackermann(m-1,Ackermann(m,n-1)) ;;
```

"mais ceci est une autre histoire".)

1.7.3 le jeu des tours de Hanoï

Le jeu des tours de Hanoï consiste en une plaquette sur laquelle sont plantées trois tiges (`A`, `B`, `C`). Sur la première de ces tiges (`A`) sont enfilés, dans l'état initial du jeu, `n` disques dont les diamètres sont strictement décroissants (le plus grand est en bas). Le but du jeu est de transférer les `n` disques de la tige de départ (`A`) vers la tige but (`C`) en respectant les trois règles suivantes :

- toutes les tiges peuvent être utilisées comme intermédiaire;
- on ne peut transférer qu'un disque à la fois;
- on n'a pas le droit de poser un grand disque sur un petit disque.

Une légende, inventée par le mathématicien français Édouard Lucas en 1883, raconte que ce jeu occupe les moines d'un temple (de Hanoï) qui passent théologiquement leur temps à le résoudre pour le nombre étrange `n = 64`.

Le jeu est très facile à résoudre à l'aide d'une fonction récursive. On va commencer par écrire une fonction de déplacement des disques (elle se contente de donner les ordres de déplacement). La voici, assortie d'un exemple; elle fait appel au type `string` des chaînes de caractères; le chapeau (`^`) est la concaténation des chaînes de caractères :

```
let déplace x y =
   print_string ("Déplacer le disque au sommet de "^x^" vers le sommet de "^y);
   print_newline() ;;

déplace : string -> string -> unit

déplace "A" "C" ;;
Déplacer le disque au sommet de A vers le sommet de C
- : unit = ()
```

Et voici la solution (assortie d'un exemple); c'est un bon exercice que d'essayer de la comprendre :

```
(*
   n est le nombre de disques;
   d est la tige de départ;
   i est la tige intermédiaire;
   b est la tige but.
*)

let rec hanoï n d i b =
   if n = 1
      then déplace d b
    else begin
             hanoï (n - 1) d b i;
             déplace d b;
```

```
            hanoï (n - 1) i d b
          end ;;

hanoï : int -> string -> string -> string -> unit

hanoï 3 "A" "B" "C" ;;

Déplacer le disque au sommet de A vers le sommet de C
Déplacer le disque au sommet de A vers le sommet de B
Déplacer le disque au sommet de C vers le sommet de B
Déplacer le disque au sommet de A vers le sommet de C
Déplacer le disque au sommet de B vers le sommet de A
Déplacer le disque au sommet de B vers le sommet de C
Déplacer le disque au sommet de A vers le sommet de C
- : unit = ()
```

1.8 listes

Les listes sont des séquences de valeurs de même type (contrairement aux multiplets); voici une liste d'entiers (type `int list`) :

```
[1; 2; 3] ;;
- : int list = [1; 2; 3]
```

On dispose de la liste vide `[]`, c'est une liste "polymorphe" :

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

Étant donnée une liste `t` de type `'a list` et un objet `h` de type `'a` on peut toujours former la liste `h :: t` qui aura `h` comme premier élément et les éléments de `t` comme éléments suivants (dans le même ordre) :

```
1 :: [] ;;
- : int list = [1]
2 :: [1] ;;
- : int list = [2; 1]
1 :: (2 :: (3 :: [])) ;;
- : int list = [1; 2; 3]
1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
```

On verra que le type "liste" est en fait définissable en CAML; il est défini par deux *constructeurs* : '`[]`' (prononcer *"nil"*) d'arité 0, et '`::`' (prononcer *"quatres-points"*) d'arité 2. Ce vocabulaire deviendra plus clair dans la suite, rassurez-vous.

On voit que l'on peut construire des listes d'entiers; mais on peut aussi construire des listes de fonctions (et plus généralement de n'importe quoi) :

```
let multiplier_par x y = x * y
in [multiplier_par 1; multiplier_par 2; multiplier_par 3] ;;
- : (int -> int) list = [<fun>; <fun>; <fun>]
```

Tout ce qui peut se programmer sur les listes peut se construire en termes des deux *constructeurs* '`[]`' et '`::`', et des deux fonctions '`hd`' (prononcer head (tête)) et '`tl`' (prononcer tail (queue)) :

```
hd [1; 2; 3] ;;
- : int = 1

tl [1; 2; 3] ;;
- : int list = [2; 3]
```

On voit que si `h = hd L` et `t = tl L` , alors `L = h :: t` . L'application de `hd` ou `tl` à `[]` déclenche une erreur (une exception, pour être plus précis) :

```
hd [] ;;
Uncaught exception: Failure "hd"
tl [] ;;
Uncaught exception: Failure "tl"
```

Avant de poursuivre, insistons sur le fait que tous les éléments d'une liste doivent être de même type :

```
[1; 1.1] ;;
Toplevel input:
>[1; 1.1] ;;
>^^^^^^^^
This expression has type float list,
but is used with type int list.
```

On constate que CAML a obstinément refusé de construire une liste contenant l'entier '`1`' et le flottant '`1.1`'.

1.9 quelques mots sur les motifs et le filtrage

Les motifs CAML sont des expressions construites à partir de *constructeurs*, de variables et du joker '`_`' (joker = traduction de "wildcard symbol"). On connaît déjà les constructeurs '`,`', '`[]`' et '`::`'. Les valeurs des types de base (`unit`, `bool`, `int`, `float`, `char`, `string`) peuvent être considérés comme des constructeurs constants. Toute variable apparaissant dans un motif CAML n'y doit apparaître qu'une fois (les motifs CAML sont dit *"linéaires"*).

Pour parler bref, les conventions syntaxiques sont *grosso-modo* les mêmes dans les motifs que dans les expressions ordinaires.

Par exemple, `(x, 1::_)` est un motif qui filtrera tous les couples dont la seconde composante est une liste d'entiers commençant par `1`; car le joker '`_`' filtre tout et n'importe quoi. Ce motif va lier la première composante du couple à `x`. L'expression `(x,x)` n'est pas un motif admissible car la variable '`x`' apparaît deux fois dans ce motif.

Les motifs sont principalement utilisés dans les définitions de fonction pour discriminer différents cas. Ils peuvent être aussi utilisés dans l'expression '`match ... with ...`'. Pour plus de précision, se reporter, par exemple, à la "petite référence de CAML". *On va continuer en supposant le filtrage assimilé par le lecteur : en fait, quelques exemples simples vaudront sans doute mieux qu'une théorie formalisée du filtrage.*

La fonction factorielle déjà définie par

```
let rec factorielle n = if (n = 0) then 1 else n * factorielle(n - 1) ;;
```

peut être commodément définie par filtrage :

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

La suite de Fibonacci peut se programmer de la façon suivante :

```
let rec fib n =
   if (n = 0)
      then 0
      else if (n = 1)
           then 1
           else fib(n - 1) + fib(n - 2) ;;
```

Mais la définition suivante est bien plus lisible :

```
let rec fib = function
   | 0 -> 0
   | 1 -> 1
   | n -> fib(n - 1) + fib(n - 2) ;;
```

Ou avec une expression '`match ... with ...`' (mais c'est, dans le cas présent, plus lourd) :

```
let rec fib n =
   match n with
      | 0 -> 0
      | 1 -> 1
      | n -> fib(n - 1) + fib(n - 2) ;;
```

Voici un autre exemple. Si `1` représnte la valeur "vrai" et `0` la valeur "faux", on peut penser définir l'implication par :

```
let implique = function
   | (1,1) -> 1
   | (1,0) -> 0
   | (0,1) -> 1
   | (0,0) -> 1 ;;
```

Mais il y a plus simple :

```
let implique = function
   | (1,x) -> x
   | (0,_) -> 1 ;;
```

Et mieux encore :

```
let implique = function
   | (1,0) -> 0
   | _     -> 1 ;;
```

Enfin, on a vu que l'on pouvait définir '`first`' et '`second`' par

```
let first(x,y) = x and second(x,y) = y ;;
```

mais il vaut mieux écrire

```
let first(x,_) = x and second(_,y) = y ;;
```

1.10 filtrage sur les listes

Nous avons vu plus haut que les deux fonctions d'accès aux éléments d'un liste sont `hd` et `tl`. En fait, ces deux fonctions ne sont guère employées car on préfère souvent programmer sur les listes en utilisant le filtrage :

```
let est_liste_vide = function
   | [] -> true
   | _  -> false ;;

est_liste_vide : 'a list -> bool
```

En fait, les deux fonctions `hd` et `tl` peuvent se définir en CAML en utilisant le filtrage; on peut penser en donner les définitions suivantes :

```
let head = function
   |  []  -> []
   | h::t -> h;;

head : 'a list list -> 'a list

let tail = function
   |  []  -> []
   | h::t -> t;;

tail : 'a list -> 'a list
```

Ces définitions sont valides mais ne correspondent pas aux fonctions `hd` et `tl` de CAML; en effet, en CAML, la liste vide `[]` n'a "ni queue ni tête". Il faut écrire, si l'on veut que `head` (resp. `tail`) corresponde à `hd` (resp. `tl`),

```
let head = function
   |  []  -> raise (Failure "head")
   | h::t -> h;;

let tail = function
   |  []  -> raise (Failure "tail")
   | h::t -> t;;
```

La notion *d'exception* (formes '`raise ...`' et '`try ... with ...`') sera vu ultérieurement.

On constate que la programmation par filtrage sur les listes suit très naturellement la définition *inductive* du type liste : une liste de type `'a list` est soit la liste vide `[]`, soit une liste notée `(h :: t)` où `h` est un objet de type `'a` et où `t` est un objet de type `'a list`. Une fonction sur les liste suivra donc souvent la structure suivante :

```
let rec une_fonction_sur_les_listes = function
   |  []  -> expression1
   | h::t -> expression2;;
```

Ne pas oublier de traiter tous les cas possibles de listes : une liste est formée d'un élément (la tête) et d'une liste (la queue), mais peut être également la liste vide. Si vous pensez définir l'équivalent de la fonction `hd` par

```
let mauvaise_tête = function h::t -> h;;
> Toplevel input:
>...................function
>h::t -> h..
> Warning: pattern matching is not exhaustive
mauvaise_tête : 'a list -> 'a
```

CAML vous avertit que vous avez oublié d'examiner le cas correspondant à la liste vide : *warning: pattern matching is not exhaustive*, ce qui se traduit par *avertissement : le filtrage n'est pas exhaustif.*

1.11 quelques fonctions sur les listes

À cause de l'importance du type liste (qui tient à sa commodité d'emploi) on va donner ici le "code" de quelques fonctions sur les listes; filtrage et récursivité seront largement employés. On trouvera d'autres exemples (parfois plus simples) en partie **II**.

Voici deux versions (équivalentes) d'une fonction d'accès aux éléments d'une liste :

```
let rec list_nth k = function
   |  []  -> raise (Failure "list_nth")
   | h::t -> if k = 0 then h else list_nth (k - 1) t ;;

let rec list_nth = fun
   | _    []   -> raise (Failure "list_nth")
   | 0 (h::t) -> h
   | k (h::t) -> list_nth (k - 1) t ;;

list_nth : int -> 'a list -> 'a

list_nth 3 [0;1;2;3] ;; (* attention, les indices commencent à zéro *)
- : int = 3
```

Cherchons maintenant à réaliser la fonction `map` (cette fonction existe également en MAPLE) telle que `(map f [a1; a2; a3; ...])` renvoie la liste `[f a1; f a2; f a3; ...]`.

```
let map f =
   let rec aux = function
      |  []  -> []
      | h::t -> (f h)::(aux t)
   in aux ;;

map : ('a -> 'b) -> 'a list -> 'b list

map (function x -> x*x) [2; 3; 4; 5] ;;
- : int list = [4; 9; 16; 25]
```

Comparer `map` à

```
let do_list f =
   let rec aux = function
      |  []  -> ()
      | h::t -> begin (f h) ; (aux t) end
   in aux ;;

do_list : ('a -> 'b) -> 'a list -> unit
```

Cette fonction fait partie du standard CAML; mais, en version 0.74, elle est proposée avec un typage plus restrictif :

```
do_list : ('a -> unit) -> 'a list -> unit
```

Voici une manière de *forcer* ce typage :

```
let (do_list : ('a -> unit) -> 'a list -> unit) =
   function f ->
      let rec aux = function
         |  []  -> ()
         | h::t -> begin (f h) ; (aux t) end
      in aux ;;
```

Concluons ce paragraphe avec deux fonctions utiles qui, étant donné un prédicat (fonction à résultat booléen) `p`, indique si tous les éléments d'une liste y satisfont (c'est `for_all`) ou si au moins un élément de la liste y satisfait (c'est `exists`). Ces deux fonctions font elles aussi partie du standard CAML.

```
let for_all p =
   let rec aux = function
      |  []  -> true
      | h::t -> (p h) && (aux t) (* && = et *)
   in aux ;;

let exists p =
   let rec aux = function
      |  []  -> false
      | h::t -> (p h) || (aux t) (* || = ou *)
   in aux ;;

for_all : ('a -> bool) -> 'a list -> bool
exists  : ('a -> bool) -> 'a list -> bool
```

Les deux opérateurs '`||`' (le 'ou') et '`&&`' (le 'et') ne sont pas si anodins qu'ils en ont l'air; on en dira bientôt un peu plus.

1.12 le type 'unit'

C'est le plus simple de tous les types CAML : le type `unit` ne contient qu'un seul élément qui est `()` (prononcer *"void"*). Ce type correspond en mathématiques au "singleton canonique" (canonique veut dire idéal) qui pourrait être quelque chose comme $\{\emptyset\}$.

Il est principalement utilisé comme résultat des instructions *impératives* qui opèrent des *effets de bord* (c'est à dire qui effectuent des actions ou qui modifient des états mémoire; plus généralement : qui modifient des états) plutôt que de calculer une valeur. Par exemple, la fonction `quit`, utilisée pour clore une session CAML, a le type '`unit -> unit`'. Cette fonction s'invoque par '`quit()`'.

En anticipant un peu, l'analogue du type `unit` pourrait être défini par :

```
type singleton = Vide ;;
```

1.13 le type 'bool' (booléens) et les opérateurs associés

Les valeurs booléennes `true` et `false` forment une autre type de base, le type `bool`; l'analogue du type `bool` pourrait aussi être défini en CAML par

```
type booléen = Vrai | Faux ;;
```

On dispose de l'opérateur '`&&`' qui réalise le "et" et de l'opérateur '`||`' qui réalise le "ou". La syntaxe de la conditionnelle est de la forme suivante

```
if test
   then expression_1
   else expression_2
```

Attention. La forme `test` doit être à résultat booléen, et, attention, `expression_1` et `expression_1` *doivent être de même type.*

```
if true then 1 else 2;;
- : int = 1
```

```
if true then 1 else [];;
Toplevel input:
>if true then 1 else [];;
>                    ^^
This expression has type 'a list,
but is used with type int.
```

Remarquer en outre que `expression_2` (resp. `expression_1`) n'est évaluée que si `test` vaut `false` (resp. `true`) :

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

if true then 1 else 1/0 ;;
- : int = 1
```

Ceci nous mène à une remarque un peu fine mais intéressante : on peut penser définir le "ou" et le "et" par les deux *fonctions* CAML suivantes :

```
let OU x y = if x then true else y
and ET x y = if x then y else false ;;

OU : bool -> bool -> bool
ET : bool -> bool -> bool
```

Ces définitions sont logiquement correctes mais les fonctions ainsi définies ne sont pas équivalentes aux opérateurs CAML '`&&`' et '`||`'. En effet, ces derniers n'évaluent leurs arguments qu'au fur et à mesure de leurs besoins :

```
(1>0) || ((1/0)>0) ;;
- : bool = true

OU (1>0) ((1/0)>0) ;;
Uncaught exception: Division_by_zero
```

alors que, quand on applique une fonction `f` aux arguments (`arg_1`, `arg_2`, ..., `arg_i`, ...) les arguments `arg_i` sont évalués avant mise en œuvre de l'algorithme `f`. En bref, les opérateurs CAML '`&&`' et '`||`' sont de véritables structures de contrôle et ne peuvent pas être définis en tant que fonctions CAML.

Les conditionnelles peuvent être emboîtées comme dans

```
if test_1
   then expression_1
   else if test_2
           then expression_2
           else expression_3
```

que l'on présentera plutôt

```
if test_1 then expression_1
else if test_2 then expression_2
else expression_3
```

mais ceci n'est guère utile : en pratique, cette forme peut souvent être avantageusement remplacée par un filtrage.

Attention. Signalons que l'on peut omettre la clause '`else`' d'une forme '`if`', mais qu'alors, le résultat de la clause '`then`' doit être de type '`unit`' :

```
if test
   then expression
```

est équivalent à

```
if test
   then expression
   else ()
```

Voici enfin un petit complément amusant (amusant au niveau où ce place ce poly) : il se trouve que l'on pourrait presque se passer du type `bool` et de la conditionnelle; la programmation fonctionnelle suffit (au problème d'évaluation des arguments près); en voici la preuve. Définissons

```
let vrai = fun x y -> x and faux = fun x y -> y ;;
```

Et définissons le `if` de la manière suivante :

```
let SI predicat alors sinon = predicat alors sinon ;;
```

On peut alors évaluer l'expression suivante

```
let pair n = if (n mod 2) = 0 then vrai else faux
in (SI (pair 8) 1 0 , SI (pair 7) 1 0) ;;
- : int * int = 1, 0
```

1.14 le type 'char' (caractère)

Il faut savoir qu'ils sont codés, en CAML, sur 8 bits (i.e. un octet). Cela nous donnes $256 = 2^8$ caractères possibles (numérotés de 0 à 255). Les $128 = 2^7$ premiers sont codés suivant le code ASCII. On dispose des deux fonctions de conversion `char_of_int : int -> char` et `int_of_char : char -> int` :

```
char_of_int 77;;
- : char = ‘M‘

int_of_char ‘M‘;;
- : int = 77

char_of_int 0;;
- : char = ‘\000‘

char_of_int 255;;
- : char = ‘\255‘

char_of_int 256;;
Uncaught exception: Invalid_argument "char_of_int"
```

À part ça, rien de bien passionant à dire sur ce type; vous êtes re-dirigés vers la “petite référence de CAML”.

2 les autres structures de contrôle

2.1 séquencement

Pour exécuter successivement une suite '`E_1`', ..., '`E_N`' d'instructions, on utilise la construction :

```
begin
   E_1;
   ...
   E_i;
   ...
   E_N;
end ;;
```

Ce qui peut aussi s'écrire `(E_1 ; ... ; E_i ; ... ; E_N) ;;`.

En fait, et contrairement à des langages comme Pascal, CAML ne fait pas de différences entre la notion d'instruction (i.e. une construction du langage qui effectue une action) et celle d'expression (i.e. une construction du langage qui renvoie une valeur) : en CAML, toute "instruction" possède une valeur. Le séquencement ne fait pas exeption à cette règle : la valeur de `(E_1 ; ... ; E_i ; ... ; E_N)` est celle de la dernière instruction/expression exécutée, i.e. celle de '`E_N`'.

Bien sûr, le séquencement n'est utile que si les instructions/expressions '`E_i`' (de $i = 1$ à $i = N - 1$) réalisent des "effets de bord" (i.e. modifient des "états-mémoire"); c'est le cas des fonctions d'entrée/sortie.

Signalons enfin que, en version 0.74, un avertissement est généré si les expressions `E_i` ne sont pas toutes (de $i = 1$ à $i = N - 1$) de type `unit` :

```
begin 1 ; 2 ; 3 ; () ; 4 end ;;
Toplevel input:
>begin 1 ; 2 ; 3 ; () ; 4 end ;;
>      ^
Warning: this expression has type int,
but is used with type unit.
Toplevel input:
>begin 1 ; 2 ; 3 ; () ; 4 end ;;
>          ^
Warning: this expression has type int,
but is used with type unit.
Toplevel input:
>begin 1 ; 2 ; 3 ; () ; 4 end ;;
>              ^
Warning: this expression has type int,
but is used with type unit.
- : int = 4
```

Bien sûr, la dernière expression `E_N` peut être d'un type quelconque.

2.2 boucles

Revenons une fois encore vers la factorielle pour considérer les couples $(n, n!)$. On voit que l'on peut obtenir la suite de ces couples par la fonction $(x, y) \mapsto (x + 1, xy)$ en prenant la précaution de considérer $(1, 1)$ comme premier élément de cette suite (et non $(0, 1)$). En fait, c'est cette idée qui a donné naissance à la version récursive terminale de la factorielle.

Écrivons une fonction générale d'itération qui, étant donnés un entier n, une fonction f, et un argument x, calcule $f^n(x)$, où $f^0 = (x \mapsto x)$ et où où $f^n = f \circ f^{n-1} = f^{n-1} \circ f$. Voici cette fonction :

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

```
itérer : int -> ('a -> 'a) -> 'a -> 'a
```

La factorielle s'écrit alors :

```
let fact n =
   snd (itérer n
              (function (x, y) -> (x + 1 , x * y))
              (1, 1)) ;;
```

La suite des nombres de Fibonacci s'écrirait de manière analogue en considérant la fonction $(x, y) \mapsto (y, x + y)$.

Nous pouvons généraliser la fonction 'itérer' en passant en argument, non plus un entier 'n' indiquant le nombre d'itération à effectuer, mais un prédicat 'p' permettant de tester si oui ou non l'itération doit se poursuivre :

```
let rec appliquer_tant_que p f x =
   if (p x)
      then appliquer_tant_que p f (f x)
      else x ;;

appliquer_tant_que : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a
```

Remarque importante. Faisons l'hypothèse que le calcul de 'f' termine toujours (quelque soit l'argument donné à 'f'). Une propriété essentielle de (itérer n f x) est alors de toujours terminer; cette propriété n'est plus assurée pour (appliquer_tant_que p f x) (même si le prédicat p termine toujours).

Les deux fonctions 'itérer' et 'appliquer_tant_que' vont respectivement correspondre à ce que l'on peut faire avec une boucle 'for' et avec une boucle 'while'. On reviendra ultérieurement vers ce problème.

2.2.1 la boucle "for"

Un exemple vaudra sans doute mieux qu'un long discours :

```
for i = 0 to 9
   do print_int i;
      print_string ":";
      print_int (i * i);
      print_string "; "
   done ;;
0:0; 1:1; 2:4; 3:9; 4:16; 5:25; 6:36; 7:49; 8:64; 9:81; - : unit = ()
```

CAML connaît aussi la version descendante de la boucle for :

```
for i = 9 downto 0
   do print_int i;
      print_string ":";
      print_int (i * i);
      print_string "; "
   done ;;
9:81; 8:64; 7:49; 6:36; 5:25; 4:16; 3:9; 2:4; 1:1; 0:0; - : unit = ()
```

Remarques. Tout d'abord, l'indice de boucle ('i' dans les deux exemples précédents) est implicitement local : si 'i' est non lié avant le 'for ...', il se retrouve non lié après son exécution (ou retrouve son ancienne valeur). Enfin, la boucle 'for' renvoie toujours la valeur '()' de type 'unit'.

2.2.2 la boucle “while”

Tout comme la boucle 'for', la boucle 'while' renvoie toujours la valeur '()' de type 'unit'. Là aussi, on se contentera d'un exemple; il fait appel à la notion de référence, examinée plus loin dans ce poly :

```
let x = ref 8 ;;
x : int ref = ref 8

while !x > 0
   do print_int !x;
      print_string " ";
      x := !x - 1
   done ;;
8 7 6 5 4 3 2 1 - : unit = ()
```

2.3 exceptions

```
1 + 2/0;;
Uncaught exception: Division_by_zero
(* Exception non rattrapée : division_par_zéro *)
```

Comme on le voit sur cet exemple, des situations embarrassantes peuvent se présenter au cours d'un calcul, comme la division d'un nombre par 0; il est alors inutile de poursuivre le calcul et il serait (en général) faux de renvoyer une valeur. On utilise plutôt la notion d'exception qui permet de prendre en compte les situations exceptionnelles qui peuvent se présenter au cours d'un calcul y compris les interruptions provenant de causes extérieures au calcul.

En bref, le déclenchement (par la forme 'raise ...') d'une exception CAML

(1) interrompt le calcul en cours,

(2) propage l'exception jusqu'à ce qu'elle soit rattrapée (par une forme 'try ... with ...').

(3) Si elle n'est pas rattrapée (Uncaught exception: ...) l'exception apparaît finalement au “toplevel” de l'interprète. Ceci va devenir plus clair dans la suite avec quelques exemples.

2.3.1 distinguer déclenchement et évaluation d'une exception

Certaines opérations illicites déclenchent des exceptions :

```
1 + 2/0;;
Uncaught exception: Division_by_zero

hd [];;
Uncaught exception: Failure "hd"
```

Mais on peut aussi déclencher ces exceptions “à la main” à l'aide de la forme 'raise ...' :

```
raise Division_by_zero ;;
Uncaught exception: Division_by_zero

raise (Failure "abracadabra") ;;
Uncaught exception: Failure "abracadabra"
```

Ceci dit, les exceptions, qui sont toutes des valeurs appartenant au type 'exn', sont évaluées au même titre que les autres expressions CAML :

```
Division_by_zero;;
- : exn = Division_by_zero

Failure ("abra" ^ "cadabra") ;;
- : exn = Failure "abracadabra"
```

2.3.2 le type 'exn' des exceptions

L'ensemble des exceptions définies à un moment donné dans le système constitue un type prédéfini : `exn`. Ce type ressemble beaucoup à un *type somme* (désolé, voir plus loin dans ce poly) mais, contrairement aux types somme ordinaires, le type `exn` est *extensible :* l'utilisateur a la possibilité de lui rajouter des *constructeurs*.

Le type des exceptions comporte (ce qui précède le prouve) en particulier les constructeurs suivants :

```
type exn = ...
         | Division_by_zero
         | Failure of string
         | ...
```

Mais il est possible de lui adjoindre de nouveaux constructeurs : lorsqu'on écrit

```
exception Mon_exception of int;;
Exception Mon_exception defined. (* c'est la réponse de Caml *)
```

ceci a pour effet d'ajouter une clause dans la définition du type `exn` qui devient :

```
type exn = ...
         | Division_by_zero
         | Failure of string
         | Mon_exception of int
         | ...
```

2.3.3 rattrapage d'exceptions

L'utilisateur peut rattraper les exceptions déclenchées par la forme '`raise`' (les opérations illicites y font appel) en utilisant la forme '`try ... with ...`' dont la syntaxe complète est :

```
try expr with
   | m_1 -> e_1
   | m_2 -> e_2
   | ...
   | m_n -> e_n
```

Si l'expression `expr` s'évalue normalement le résultat de l'ensemble de l'expression est la valeur de `expr`; en revanche, si `expr` déclenche une exception, l'ensemble des motifs `m_1`, ..., `m_n` est utilisé pour filtrer cette exception; tous les motifs `m_1`, ..., `m_n` doivent être de type `exn`. Si `m_i` est le premier motif à filtrer l'exception déclenchée par `expr`, la valeur de l'expression tout entière est celle de `e_i`. Il s'en suit que `expr` et les `e_i` doivent être de même type. Si aucun motif ne filtre l'exception déclenchée par `expr`, cette dernière est propagée au-delà de l'expression '`try ...`'.

Donnons quelques exemples :

```
try 1/0 with Division_by_zero -> 666 ;;
- : int = 666

try hd [] with _ -> [] ;;
- : '_a list = []
```

La “petite référence de CAML” vous donne un autre exemple. Signalons que la fonction prédéfinie `failwith` pourrait être définie par

```
let failwith message = raise (Failure message) ;;
```

Et remarquons enfin qu'un des intérêts de disposer d'exceptions typées est de pouvoir en affiner le filtrage.

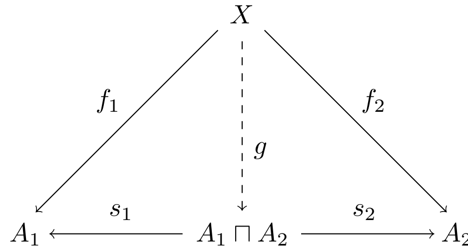
3 construction et usage de nouveaux types

En CAML, l'utilisateur peut définir ses propres structures de donnée. Cette facilité permet une meilleure détection des erreurs de programmation car les types définis par l'utilisateur sont sensés refléter précisément les besoins de ses algorithmes. Les types constructibles en CAML sont de deux genres : types du genre somme et types du genre produit. Nous verrons que les types `unit`, `bool`, `'a list` sont du genre somme; en revanche, et cela ne surprendra personne, le produit cartésien est du genre produit. Avant de voir ce que sont sommes et produits de types en CAML, il est intéressant de formaliser un peu ces deux notions : la symétrie entre les deux est profonde.

3.1 produit cartésien et somme disjointe d'ensembles

3.1.1 produit cartésien

Soit A_1 et A_2 deux ensembles. On sait que l'on peut former leur produit cartésien $A_1 \times A_2$. On notera (temporairement) cet ensemble $A_1 \times A_2 = A_1 \sqcap A_2$. On note s_1 et s_2 les deux projections : $s_1(a_1, a_2) = a_1$ et $s_2(a_1, a_2) = a_2$ pour tout $(a_1, a_2) \in A_1 \sqcap A_2$; ces deux applications sont des surjections. Une (la) propriété fondamentale du produit cartésien est la suivante. Pour tout ensemble X et pour tout couple d'applications $f_1 : X \to A_1$ et $f_2 : X \to A_2$ il existe une seule application $g : X \to A_1 \sqcap A_2$ telle que $s_1 \circ g = f_1$ et $s_2 \circ g = f_2$ (on dit que les deux triangles du diagramme suivant “commutent”.)

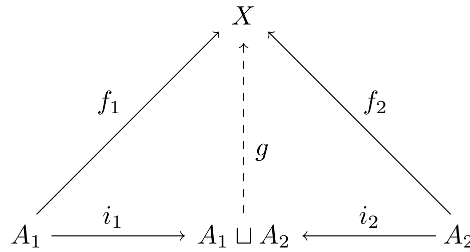


En effet, il suffit de définir g par (pour tout $x \in X$) $g(x) = (f_1(x), f_2(x))$; elle satisfait bien à $s_1 \circ g = f_1$ et $s_2 \circ g = f_2$. C'est bien la seule manière de définir cette application, car la condition $(s_1 \circ g)(x) = f_1(x)$ oblige la première composante de chaque couple $g(x) \in A_1 \sqcap A_2$ à être $f_1(x)$; de même pour la deuxième composante.

3.1.2 somme disjointe

Soit A_1 et A_2 deux ensembles non forcément disjoint. Définissons leur réunion disjointe par $A_1 \sqcup A_2 = (\{1\} \times A_1) \cup (\{2\} \times A_2)$. L'intérêt de former les produits $A'_k = \{k\} \times A_k$ $(k = 1, 2)$ est de “colorier” les deux ensembles A_1 et A_2 pour assurer que $A'_1 \cap A'_2 = \emptyset$. À cette construction, on peut associer deux applications $i_1 : A_1 \to A_1 \sqcup A_2$ et $i_2 : A_2 \to A_1 \sqcup A_2$ définies par (pour tout $a_1 \in A_1$) $i_1(a_1) = (1, a_1)$ et (pour tout $a_2 \in A_2$) $i_2(a_2) = (2, a_2)$. Ces deux applications sont clairement des injections. Une (la) propriété fondamentale de la somme disjointe est la suivante. Pour tout ensemble X et pour tout couple d'applications $f_1 : A_1 \to X$ et $f_2 : A_2 \to X$ il existe une

seule application $g : A_1 \sqcup A_2 \to X$ telle que $g \circ i_1 = f_1$ et $g \circ i_2 = f_2$ (on dit que les deux triangles du diagramme suivant "commutent".)



Il suffit de remarquer que tout élément y de $A_1 \sqcup A_2$ est soit de la forme $(1, a_1)$ où $a_1 \in A_1$, soit de la forme $(2, a_2)$ où $a_2 \in A_2$ et de définir g par $g(1, a_1) = f_1(a_1)$ et $g(2, a_2) = f_2(a_2)$ (pour tout $a_1 \in A_1$ et pour tout $a_2 \in A_2$, bien sûr). Là encore, cette définition est la seule qui soit compatible avec les deux conditions $g \circ i_1 = f_1$ et $g \circ i_2 = f_2$.

3.1.3 remarque

Les notations $A \to B$ et B^A désignent ici toutes deux l'ensemble des applications de A vers B.

- La propriété précédente du produit cartésien exprime que l'application $g \mapsto (s_1 \circ g, s_2 \circ g)$ est une bijection de $(A_1 \sqcap A_2)^X$ vers $A_1^X \sqcap A_2^X$, ou avec les notations à-la-CAML, de $X \to (A_1 \sqcap A_2)$ vers $(X \to A_1) \sqcap (X \to A_2)$.
- La propriété précédente de la somme disjointe exprime que l'application $g \mapsto (g \circ i_1, g \circ i_2)$ est une bijection de $X^{(A_1 \sqcup A_2)}$ vers $X^{A_1} \sqcap X^{A_2}$, ou avec les notations à-la-CAML, de $(A_1 \sqcup A_2) \to X$ vers $(A_1 \to X) \sqcap (A_2 \to X)$.

3.2 types produit

Les types produit sont des "produits cartésiens étiquettés"; cela veut dire que l'on nomme les différentes projections (qui sont des surjections) : elles sont alors appelées des "étiquettes" (labels). Les éléments d'un type du genre produit sont appelés "enregistrements" (records).

En guise d'exemple, supposons que nous voulions définir une structure de donnée contenant diverses informations sur des individus. Nous pourrions définir

```
let x = ("Jean", 18, "étudiant", "Grenoble") ;;
x : string * int * string * string = "Jean", 18, "étudiant", "Grenoble"
```

Nous pourrions alors utiliser le filtrage pour extraire toute information particulière sur `x`. Le problème est que la fonction `nom_de` qui renvoie le nom d'un objet tel que `x` se confond avec la première projection pour les 4-uplets :

```
let nom_de (s, _, _, _) = s ;;
nom_de : 'a * 'b * 'c * 'd -> 'a

nom_de x ;;
- : string = "Jean"
```

Ce type n'est donc pas assez précis puisqu'il permet l'application de la fonction `nom_de` à n'importe quel 4-uplet et non seulement aux objets du genre de `x` :

```
nom_de (1, (function x -> x*x), 7.7, []) ;;
- : int = 1
```

Nous allons donc définir un type de donnée `personne` :

```
type personne =
   {Nom : string ; Age : int ; Occupation : string ; Résidence : string} ;;
```

Le type `personne` est le *produit* des types `string`, `int`, `string`, `string`; les noms des *champs* sont appelés *étiquettes*; ce sont les *projections* de ce produit.

Pour construire un objet de type `personne` vous pouvez spécifier la valeur des champs dans un ordre *quelconque* :

```
let x =
   {Nom = "Jean" ; Résidence = "Grenoble" ; Occupation = "étudiant" ; Age = 18 } ;;

x : personne =
 {Nom="Jean"; Age=18; Occupation="étudiant"; Résidence="Grenoble"}
```

Il y a deux manière d'extraire l'information contenue dans un enregistrement : (1) par filtrage et (2) avec l'opérateur '.' :

```
let nom_de {Nom = s; Age = _; Occupation = _; Résidence = _} = s ;;
nom_de : personne -> string

nom_de x ;;
- : string = "Jean"

x.Nom ;;
- : string = "Jean"
```

3.3 types produit paramétrés

Il est important de pouvoir paramétrer les types (produit ou somme) pour pouvoir créer des structures de donnée *génériques* : le type liste est paramétré, et c'est la raison pour laquelle on peut construire des listes de n'importe quel type. Si nous voulons definir le produit cartésien comme un type produit nous devons, là aussi, le paramétrer car nous voulons pouvoir construire des couples d'objets de n'importe quel type.

```
type ('a, 'b) couple = {Proj_1 : 'a ; Proj_2 : 'b} ;;

let proj_1 c = c.Proj_1 and proj_2 c = c.Proj_2 ;;

proj_1 : ('a, 'b) couple -> 'a
proj_2 : ('a, 'b) couple -> 'b

let un_couple = {Proj_1 = 13 ; Proj_2 = (function x -> 2*x)} ;;
un_couple : (int, int -> int) couple = {Proj_1=13; Proj_2=<fun>}

(proj_2 un_couple) (proj_1 un_couple) ;;
- : int = 26
```

Attention : toute définition de type génère un nouveau type : si on ré-évalue la définition du type `couple` on ne peut plus extraire les composantes de `un_couple` :

```
type ('a, 'b) couple = {Proj_1 : 'a ; Proj_2 : 'b} ;;

un_couple.Proj_1 ;;
Toplevel input:
```

```
>un_couple.Proj_1 ;;
>^^^^^^^^^^
This expression has type (int, int -> int) couple,
but is used with type ('a, 'b) couple.
```

Mais (comprenez-vous pourquoi ?) on peut toujours faire

```
proj_1 un_couple ;;
- : int = 13

(proj_2 un_couple) (proj_1 un_couple) ;;
- : int = 26
```

3.4 types somme

Les types somme sont des "sommes disjointes étiquettés"; cela veut dire que l'on nomme les différentes injections : elles sont alors appelées des *"constructeurs"*.

Imaginons, par exemple, que nous voulions un type `identification` dont les valeurs peuvent être, soit des chaînes de caractères, soit un couple d'entiers (codant le numéro de sécurité sociale); voici la manière de faire cela :

```
type identification = Nom of string
                     | SS of int*int ;;
```

Le type `identification` est l'union disjointe étiquettée des deux types `string` et `int*int`. Les deux injections `Nom` et `SS` sont les deux constructeurs du type `identification`. Elles injectent `string` et `int*int` dans le type unique `identification`.

On peut (heureusement) construire des objets de type `identification` :

```
let id1 = Nom "toto"
and id2 = SS (12345, 12345) ;;

id1 : identification = Nom "toto"
id2 : identification = SS (12345, 12345)
```

Les valeurs `id1` et `id2` sont de mêmes types; elles peuvent, par exemple, faire partie de la même liste :

```
[id1; id2] ;;
- : identification list = [Nom "toto"; SS (12345, 12345)]
```

Le filtrage sera naturellement utilisé pour extraire les composants d'une somme :

```
let type_id = function
   | Nom _  -> "c'est une chaîne"
   | SS  _  -> "c'est une couple d'entiers" ;;

type_id : identification -> string

type_id id1 ;;
- : string = "c'est une chaîne"
```

3.5 les énumérations sont des sommes dégénérés

On a vu comment pourrait être défini le type `unit` :

```
type unit = void ;;
```

et comment pourrait être défini le type `bool` :

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

Du reste, pourquoi ne pas essayer ?

```
type bool = true | false ;;
Type bool defined.

true;;
- : bool = true

1 > 2 ;;
- : builtin__bool = builtin__false
```

On peut facilement trouver à définir des types du même genre :

```
type couleur = Pique | Carreau | Trèfle | Coeur ;;
(* en anglais, si vous préférez *)
type suit = Spade | Diamond | Club | Heart ;;
```

De tels types somme, construits à partir de constructeurs sans arguments (on dira parfois constructeurs constants) ne sont que des énumérations.

3.6 types somme paramétrés

En voici un exemple, un peu artificiel; nous verrons plus intéressant par la suite :

```
type 'a duplication = Copie_1 of 'a | Copie_2 of 'a
```

Cela peut sembler stupide, mais, finalement un type permettant de manipuler les nombres complexes à la fois sous forme polaire ($\rho e^{i\theta}$) et sous forme cartésienne ($x + iy$) pourrait s'écrire :

```
type complexe = Polaire of float*float | Cartésien of float*float
```

3.7 types récursifs

3.7.1 un type récursif non paramétré

Les expressions arithmétiques peuvent être définies par deux types mutuellement récursifs :

```
type expr_arith = Constante of int
                | Variable of string
                | Addition of arguments
                | Soustraction of arguments
                | Multiplication of arguments
                | Division of arguments
and arguments = {A1 : expr_arith ; A2 : expr_arith} ;;
```

Voici comment représenter l'expression ($x + 3y$) avec ce type :

```
Addition {A1= Variable "x";
          A2= Multiplication {A1= Constante 3;
                              A2= Variable "y"}}
```

Mais on peut aussi définir les expressions arithmétiques plus simplement (avec un seul type récursif) :

```
type expr_arith = Constante of int
                | Variable of int
                | Addition of (expr_arith * expr_arith)
                | Soustraction of (expr_arith * expr_arith)
                | Multiplication of (expr_arith * expr_arith)
                | Division of (expr_arith * expr_arith) ;;
```

Les variables sont ici indexées par les entiers (v_1, v_2, ...); voici comment se représente maintenant l'expression $(v_1 + 3v_2)$:

```
Addition (Variable 1, Multiplication (Constante 3, Variable 2))
```

Profitons-en pour faire un petit dérivateur d'expression :

```
let dérive index =
   let rec der = function
      | Constante _ -> Constante 0
      | Variable n -> if (n = index) then (Constante 1) else (Constante 0)
      | Addition(e1,e2) -> Addition(der e1,der e2)
      | Soustraction(e1,e2) -> Soustraction(der e1,der e2)
      | Multiplication(e1,e2) -> Addition(Multiplication(der e1,e2),
                                          Multiplication(e1,der e2))
      | Division(e1,e2) -> Division(Soustraction(Multiplication(der e1,e2),
                                                 Multiplication(e1,der e2)),
                                    Multiplication(e2,e2))
   in der ;;

dérive : int -> expr_arith -> expr_arith

dérive 1 (Addition (Variable 1, Multiplication (Constante 3, Variable 2))) ;;

- : expr_arith =
 Addition
  (Constante 1,
   Addition
    (Multiplication (Constante 0, Variable 2),
     Multiplication (Constante 3, Constante 0)))
```

Notre dérivateur dérive (il calcule même les dérivées partielles), mais il ne simplifie pas !

3.7.2 un type somme récursif paramétré

On peut prendre l'exemple du type `liste` analogue du type CAML prédéfini `list` :

```
type 'a liste = Vide
              | Cellule of 'a * ('a liste) ;;
```

Voici la fonctionnelle `map` ré-écrite pour opérer sur ce type :

```
let map_liste f =
   let rec aux = function
      | Vide -> Vide
      | Cellule(item, reste) -> Cellule(f item, aux reste)
   in aux ;;
```

```
map_liste : ('a -> 'b) -> 'a liste -> 'b liste

let une_liste = Cellule(1, Cellule(2, Cellule(3, Cellule(4, Vide)))) ;;
une_liste : int liste =
 Cellule (1, Cellule (2, Cellule (3, Cellule (4, Vide))))

map_liste (function x -> x*x) une_liste ;;
- : int liste = Cellule (1, Cellule (4, Cellule (9, Cellule (16, Vide))))
```

3.7.3 remarque

Les définitions récursives peuvent mener à des types dont il est difficile ou impossible de d'exhiber des éléments; par exemple :

```
type stupide = {Vas_Y : stupide} ;;
Type stupide defined.

let rec valeur_idiote = {Vas_Y = valeur_idiote} ;;
valeur_idiote : stupide =
 {Vas_Y=
   {Vas_Y=
     {Vas_Y=
       {Vas_Y=
         {Vas_Y=
           {Vas_Y=
             {Vas_Y=
               {Vas_Y=
                 {Vas_Y=
                   {Vas_Y= .}}}}}}}}}}
```

Heureusement, la fonction d'impression standard de CAML a arrêté les dégats. Les définitions récursives de types doivent posséder au moins une clause non récursive...

3.7.4 un type produit récursif paramétré

On peut, par exemple, donner une définition des arbres non vides par :

```
type 'a arbre_non_vide = {Noeud : 'a ; Rameaux : ('a arbre_non_vide) list} ;;
```

3.7.5 type produit et somme mutuellement récursifs et paramétrés

On peut définir un type d'arbres binaires par :

```
type ('a,'b) arbre_binaire = Feuille of 'b
                           | Noeud of ('a,'b) noeud
and ('a,'b) noeud = {Fils_droit : ('a,'b) arbre_binaire;
                     Étiquette : 'a;
                     Fils_gauche : ('a,'b) arbre_binaire} ;;
```

De tels arbres peuvent être utilisés pour représenter des termes ne comportant que des opérateurs binaires et qu'un seul type de feuille.

3.8 abréviations de type

CAML nous permet de définir des synonymes pour des types simples, par exemple

```
type point3d == float * float * float ;;
Type point3d defined.
```

permet d'invoquer par la suite cette abréviation pour définir un nouveau type (qui peut être lui aussi une abréviation) :

```
type chemin3d == point3d list ;;
Type chemin3d defined.
```

ou pour une coercion de type :

```
let Id_point3d (x : point3d) = x ;;
Id_point3d : point3d -> point3d
```

4 les types mutables

Les définitions des types produit peuvent être annotées de manière à permettre la mise à jour physique de ces structures de données; c'est le trait essentiel de la programmation impérative par opposition à la programmation fonctionnelle. Écrire et modifier les valeurs contenues dans un emplacement mémoire est une possibilité essentielle de langages "impératifs" comme C.

4.1 types produit mutables ("records" modifiables)

Revenons vers le type de donnée `personne` :

```
type personne =
   {Nom : string ; Age : int ; Occupation : string ; Résidence : string} ;;
```

Ce type ne permet pas la mise à jour des champs `Age`, `Occupation` et `Résidence`; pour réaliser cela il suffit de redéfinir le type personne par :

```
type personne =
   {Nom : string ;
    mutable Age : int ;
    mutable Occupation : string ;
    mutable Résidence : string} ;;
```

La construction d'une valeur de ce type suit le même schéma qu'avant :

```
let jean =
   {Nom = "Jean" ; Résidence = "Grenoble" ; Occupation = "étudiant" ; Age = 18 } ;;
```

Mais nous pouvons maintenant modifier la valeur des champs mutables de l'objet `jean` :

```
jean.Age <- jean.Age + 1 ;;
- : unit = ()

jean;;
- : personne =
 {Nom="Jean"; Age=19; Occupation="étudiant"; Résidence="Grenoble"}
```

4.2 types somme mutables

Voici un exemple, où l'on définit des points du plan à coordonnées mutables :

```
type point2d = Cartésien of mutable (float * float)
             | Polaire   of mutable (float * float) ;;
```

La fonction suivante met à jour les coordonnées d'un point du plan :

```
let change p nx ny = match p with
   | Cartésien(xy) -> (xy <- (nx, ny))
   | Polaire(xy)   -> (xy <- (nx, ny)) ;;

change : point2d -> float -> float -> unit
```

Exemple :

```
let a = Cartésien(1.,1.);;
a : point2d = Cartésien (1.0, 1.0)
change a 2. 2. ;;
- : unit = ()
a;;
- : point2d = Cartésien (2.0, 2.0)
```

On peut faire la même chose avec des références (voir le paragraphe suivant) :

```
type point2d = Cartésien of (float * float) ref
             | Polaire   of (float * float) ref ;;

let change p nx ny = match p with
   | Cartésien(rxy) -> rxy := (nx, ny)
   | Polaire(rxy)   -> rxy := (nx, ny) ;;

change : point2d -> float -> float -> unit
```

4.3 les références

Comme il est dit dans [6], la notion de référence représente la quintessence de l'idée d'objet modifiable. Elle correspond à la notion de pointeur dans les langages comme Pascal ou C.
On peut créer une référence sur n'importe quel objet en utilisant le constructeur '`ref`' :

```
let r = ref 0 ;;
r : int ref = ref 0
```

La valeur référencée ("pointée") par une référence est accessible par l'opérateur préfixe '`!`' :

```
(r, !r) ;;
- : int ref * int = ref 0, 0
```

Le contenu d'une référence est modifiable par l'opérateur d'affectation '`:=`' :

```
r := !r + 1;;
- : unit = ()
!r;;
- : int = 1
```

Si vous voulez comprendre les interactions références/clôtures, essayez les ordres CAML suivants :

```
let y = 3;;
let foo x = x + y;;
let y = 8;;
foo 3;;
```

Et comparez avec :

```
let y = ref 3 ;;
let foo x = x + !y ;;
y := 8 ;;
foo 3 ;;
```

Les références permettent de fabriquer des fonctions “à états internes”; à cet égard, il est utile de comparer les deux fonctions suivantes :

```
let nouvel_entier =
   let r = ref 0
   in function () -> begin r := !r + 1; !r end ;;

let un () =
   let r = ref 0
   in begin r := !r + 1; !r end ;;
```

L’appel `un()` renvoie toujours `1`; en revanche, l’appel `nouvel_entier()` renvoie à chaque fois un nouvel entier.

Remarque importante. On notera que l’usage des “effets de bord” fait perdre aux fonctions CAML une propriété mathématique essentielle, celle de renvoyer toujours les mêmes résultats lorsqu’elles sont appellées sur des arguments identiques.

Enfin, la “petite référence de CAML” vous dit comment les références pourraient être définies en CAML.

4.4 les vecteurs

Reportez-vous à la “petite référence de CAML” qui vous dit presque tout sur les vecteurs; c’est une structure de donnée mutable car l’on dispose de la fonction

`vect_assign : 'a vect -> int -> 'a -> unit`

La forme ’`vect_assign v i new`’ peut s’écrire aussi ’`v.(i) <- new`’ :

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

v.(0) <- 111 ;;
- : unit = ()

v ;;
- : int vect = [|111; 2; 3|]
```

4.5 les chaînes de caractères

Ce ne sont finalement que des vecteurs de caractères; et c’est aussi une structure de donnée mutable. La fonction responsable est ’`set_nth_char`’ :

```
let s = "asdfg" ;;
s : string = "asdfg"
```

```
set_nth_char s 0 ‘z‘ ;;
- : unit = ()

s ;;
- : string = "zsdfg"
```

Pour pallier l'indigence de ce paragraphe, je reproduis ici une petite fonction, dont nous accorderons la paternité à Xavier Leroy et Pierre Weiss, qui reconnaît si une chaîne de caractères est ou non un palindrome :

```
nth_char : string -> int -> char
string_length : string -> int

let palindrome s =
   let rec palin i j =
      (i >= j) or (nth_char s i = nth_char s j) & palin (i+1) (j-1)
   in palin 0 (string_length(s) - 1) ;;

palindrome : string -> bool

palindrome "toto" ;;
- : bool = false

palindrome "eluparcettecrapule" ;;
- : bool = true

palindrome "tulastropecrasecesarceportsalut" ;;
- : bool = true
```

5 l'égalité en caml

Il y a deux sortes d'égalité en CAML, notées respectivement (de manière infixe) `=` et `==`.

```
(prefix =)  : 'a -> 'a -> bool
(prefix ==) : 'a -> 'a -> bool
```

`(e1 = e2)` teste l'égalité structurelle de `e1` et `e2`. Les structures mutables (au nombre desquelles il faut compter les références, les tableaux et chaînes de caractères) sont '`=`' ssi leurs contenus au moment du test sont structurellement égaux, même si ces deux objets mutables ne sont pas physiquement le même objet. Le test d'égalité `=` entre valeurs fonctionnelles déclenche l'exception `Invalid_argument`. Le test d'égalité `=` entre structures cycliques peut ne pas terminer.

`(e1 == e2)` teste l'égalité physique de `e1` et `e2`. Sur les entiers (`int`) et les caractères (`char`) `==` se confond avec `=`. Sur les structures mutables, `(e1 == e2)` est vrai ssi toute modification physique de `e1` affecte *ipso facto* `e2`. Sur les structures non mutables le comportement de `==` est *"implementation-dependent"*, mais on notera que l'on a toujours `(e1 == e2)` implique `(e1 = e2)`.

En bref, l'egalité structurelle `=` sera plus généralement employée que l'égalité physique `==`, que l'on pourra presque franchement oublier, sauf, par exemple, à manipuler finement des références ou des structures cycliques...

Le standard CAML propose

```
(prefix <>) : 'a -> 'a -> bool
(prefix !=) : 'a -> 'a -> bool
```

`(e1 <> e2 )` est la négation de `(e1 = e2)`, tandis que `(e1 != e2 )` est la négation de `(e1 == e2)`.

On notera que les comparaisons sont désormais génériques en CAML :

```
prefix <  : 'a -> 'a -> bool
prefix <= : 'a -> 'a -> bool
prefix >  : 'a -> 'a -> bool
prefix >= : 'a -> 'a -> bool
```

Ces fonctions réalisent les comparaisons structurelles. Elles coïncident avec l'ordre usuel sur les types `int`, `string`, `float`; elles l'étendent à un ordre total sur tous les autres type : par exemple, sur les multiplets, l'ordre lexicographique est utilisé. Ces comparaisons sont compatibles avec `=`. Les structures mutables sont comparées suivant leurs contenus. Les comparaisons sur les valeurs fonctionnelles déclenchent `Invalid_argument`. Les comparaisons sur les structures cycliques peuvent ne pas terminer.

On dispose de

```
compare: 'a -> 'a -> int
```

telle que `(compare x y)` renvoie 0 si `(x = y)`, un `int` strictement négatif si `(x < y)`, un `int` strictement positif si `(x > y)`. Les restrictions d'utilisation sont les mêmes que pour `=` et les fonctions de comparaison. Cette fonction peut jouer le rôle de la fonction de comparaison requise par les bibliothèques `map` et `set`.

Enfin, dans la même veine, les deux fonctions

```
min : 'a -> 'a -> 'a
max : 'a -> 'a -> 'a
```

renvoient respectivement le plus petit et le plus grand de leurs deux arguments, vous l'aviez deviné.

Partie II
EXEMPLES

Cette partie donne quelques exemples d'utilisation de CAML*. On y trouvera du simple et du compliqué, le tout pouvant se classer sous trois rubriques : (1) les algorithmes au progamme de première année (ils sont signalés par le symbole ¶); (2) des idées qui seront (ou pourraient être) développées en TD de programmation; (3) des idées complémentaires et/ou des présentations d'outils qui pourront servir (ou non) "pour la suite". Les textes* CAML *qui suivent ont donc soit un caractère de complément, soit seront repris en cours et/ou en TD; j'en tire excuse pour, trop souvent, ne pas les commenter autant qu'ils devraient l'être.*

6 programmation "fonctionnelle" avec les nombres

6.1 calcul de pgcd et bibliothèque des grands nombres

6.1.1 deux calculs du pgcd sur les entiers (¶)

La fonction suivante est un algorithme "académique" :

```
let rec pgcd_bête x y =
   if (x > y) then (pgcd_bête y x)
   else if (x > 0) then (pgcd_bête x (y - x))
   else y ;;
```

Il y a bien plus efficace :

```
let rec pgcd x y =
   if (x > y) then (pgcd y x)
   else if (x > 0) then (pgcd x (y mod x))
   else y ;;
```

6.1.2 la librairie "num"

Le langage CAML offre à l'utilisateur de nombreuses librairies (ou bibliothèques, c'est gratuit) dont certaines ne sont disponibles, hélas, que sous UNIX; on citera (entre autres) les bibliothèques

- `Graphics` qui gère le graphique (non ?);
- `stack` qui propose une gestion des *piles*;
- `queue` qui propose une gestion des *files d'attente*;
- `map` qui propose une gestion des *dictionnaires* (ou *tables d'association*);
- `set` qui propose une gestion des *ensembles (totalement ordonnés)*;
- `num` qui propose une gestion des *grands nombres exacts* (c'est à dire des entiers et rationnels de grandeur quelconque).

La directive `#open "num";;` (vous devez taper le `#`) rend disponible (sous l'interprète CAML) la totalité des types et des fonctions que contient la librairie `num`.
Le type `num` est

```
type num = Int of int | Big_int of big_int | Ratio of ratio ;;
```

Il contient donc les entiers courts ordinaires (`int`) les grands entiers signés (`big_int`) et les rationnels que l'on peut construire sur ces entiers (`ratio`). En fait, l'utilisateur "moyen" n'a pas à connaître la structure de ce type car il dispose des fonctions suivantes.

- `num_of_string : string -> num`; on donnera deux exemples :

  ```
  num_of_string "1234567890987654321" ;; (* un big_int *)
  - : num = 1234567890987654321
  num_of_string "123456789/987654321" ;; (* un ratio *)
  - : num = 13717421/109739369
  ```

- `num_of_int : int -> num`; un (seul) exemple suffira :

  ```
  num_of_int 777 ;;
  - : num = 777
  ```

Les opérateurs sur les nombres flottants étaient suffixés par un point (comme dans `+.`) les opérateurs sur les `num` seront suffixés par un slash (comme dans `+/`). On dispose des opérateurs infixes (comme `+/`) et de leurs formes préfixes curryfiées (pour l'addition ce sera `add_num`). Voici une liste de ces opérateurs.

- `(prefix +/)` = `add_num : num -> num -> num`; c'est l'addition;
- `minus_num : num -> num`; c'est le changement de signe unaire;
- `(prefix -/)` = `sub_num : num -> num -> num`; c'est la soustraction;
- `(prefix */)` = `mult_num : num -> num -> num`; c'est la multiplication;
- `square_num : num -> num`; l'élévation au carré;
- `(prefix //)` = `div_num : num -> num -> num`; c'est la division;
- `quo_num : num -> num -> num` et `mod_num : num -> num -> num` sont respectivement le quotient et le reste de la division euclidienne du premier `num` par le second;
- `(prefix **/)` = `power_num : num -> num -> num`; c'est l'exponentiation (ne soyez pas trop gourmand, c'est un vilain péché); un ou deux exemples ?

  ```
  power_num (num_of_string "654")(num_of_string "5") ;;
  - : num = 119643398733024
  power_num (num_of_string "654")(num_of_string "1/5") ;;
  Uncaught exception: Invalid_argument "power_num" (* que croyiez-vous ? *)
  ```

Étant donné que `num` contient $\mathbb{Z}$ et $\mathbb{Q}$, la librairie correspondante procure les fonctions de passage classique de $\mathbb{Q}$ vers $\mathbb{Z}$.

- `is_integer_num : num -> bool`; teste si son argument est un élément de $\mathbb{Z}$;
- `integer_num : num -> num`; approche un $\mathbb{Q}$-`num` par un $\mathbb{Z}$-`num`;
- `round_num : num -> num`; approche un $\mathbb{Q}$-`num` par un $\mathbb{Z}$-`num`; disposer de ces deux fonctions n'a d'intérêt que si l'argument r est du genre $r = n + 1/2$ avec $n \in \mathbb{Z}$; un exemple s'impose :

  ```
  integer_num (num_of_string "7/2"), round_num (num_of_string "7/2") ;;
  - : num * num = 3, 4
  ```

- `floor_num : num -> num` et
  `ceiling_num : num -> num` sont les inévitables fonctions “plancher” et “plafond” sur les `num`.

On dispose bien sûr des comparaisons :

- `sign_num : num -> int`; renvoie -1, 0, ou 1 selon le signe de son argument;
- `compare_num : num -> num -> num`; cette fonction pourrait être définie en CAML :

```
let compare_num n1 n2 = sign_num (sub_num n1 n2) ;;
```

- Les fonctions suivantes (dont les noms sont transparents) sont toutes de type `num -> num -> num`.
  `(prefix =/)` = `eq_num`;
  `(prefix </)` = `lt_num` (less than);
  `(prefix <=/)` = `le_num` (less or equal);
  `(prefix >/)` = `gt_num` (greater than);
  `(prefix >=/)` = `ge_num` (greater or equal);
  `(prefix <>/)` (not equal).

On dispose également de

- `max_num : num -> num -> num`;
- `min_num : num -> num -> num`;
- `abs_num : num -> num`;
- `suc_num : num -> num`;
- `pred_num : num -> num`;
- `incr_num : num ref -> unit`;
- `decr_num : num ref -> unit`.

Là aussi, les noms de ces fonctions sont parfaitement explicites. On dispose enfin de diverses fonctions de “coercion” :

- `string_of_num : num -> string`; c’est la fonction inverse de `num_of_string`;
- `approx_num_fix : int -> num -> string`; renvoie l’écriture décimale (du genre `"s123.321"`, où `s` est ’+’ ou ’−’) de son second argument; le premier argument (entier) est la précision demandée;
- `approx_num_exp : int -> num -> string`; la même chose, mais utilise la notation “scientifique” (mantisse, exposant de 10); le premier argument fixe le nombre de digits de la mantisse.
- `int_of_num : num -> int`; exemple :

```
int_of_num (num_of_string "1234");;
- : int = 1234
int_of_num (num_of_string "123456789123456789123456789");;
Uncaught exception: Failure "int_of_big_int"
```

- `float_of_num : num -> float`; du même poil que la précédente.

6.1.3 quelques fonctions avec les grands nombres

Donnons tout d'abord la version grand nombre du pgcd :

```
let rec big_pgcd x y =
   if (gt_num x y) then (big_pgcd y x)
   else if ((sign_num x) = 1) then (big_pgcd x (mod_num y x))
   else y ;;
```

Ce serait trop bête de ne pas le faire aussi pour la factorielle :

```
let big_fact n =
   let rec fact k r =
      if ((sign_num k) = 0)
      then r
      else fact (pred_num k) (mult_num k r)
   in fact n (num_of_int 1) ;;

big_fact (num_of_int 111) ;;
- : num =
 1762952551090244663872161047107075788761409536026565516041574063347346955
 0872483164365555745984623157731960476628379789131458474971998716233200962
 5414533120000000000000000000000000
```

6.2 accumulation

On cherche à écrire une fonction CAML simple calculant les sommes

$$S(n) = \sum_{k=0}^{k=n} k$$

Ceci est aisée; voici une version récursive naïve de cette fonction :

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

S : int -> int
```

On sait que $S(n) = n(n+1)/2$; on a donc, par exemple, $S(10) = 55$. Vérifions :

```
S 10 ;;
- : int = 55
```

On peut généraliser quelque peu notre fonction S en la redéfinissant par :

$$S(a, b) = \sum_{k=a}^{k=b} k$$

Cela pourrait donner en CAML (`S` est ici la version curryfiée de S)

```
let rec S a b =
   if a > b
     then 0
     else a + S (a + 1) b ;;
```

Mais il est plus efficace d'écrire (récursion terminale)

```
let S a b =
   let rec aux val k =
      if k <= b
        then aux (val + k) (k + 1)
        else val
   in aux 0 a ;;
```

On s'apperçoit alors que l'on fait jouer un rôle inutile à l'élément neutre de l'addition (0); si $a > b$, mieux vaut générer une erreur ou sortir une valeur spéciale (qui peut être 0); nous choisissons ici une sortie avec la valeur `défaut`; cela donne :

```
let S défaut a b =
   let rec aux val k =
      if k <= b
        then aux (val + k) (k + 1)
        else val
   in if a <= b
       then aux a (a + 1)
       else défaut ;;

S : int -> int -> int -> int

S 0 0 10;;
- : int = 55

S 0 2 1;;
- : int = 0

S 0 1 1;;
- : int = 1
```

Si, maintenant, nous voulons calculer la somme des carrés d'entiers en progressant par pas de 2 nous pouvons écrire :

```
let S' défaut a b =
  let rec aux val k =
     if k <= b
       then aux (val + (k * k)) (k + 2)
       else val
      in if a <= b
          then aux (a * a) (a + 2)
          else défaut ;;

S' : int -> int -> int -> int

S' 0 0 10 ;; (* = 2^2 + 4^2 + 6^2 + 8^2 + 10^2 *)
- : int = 220 (* = 4 + 16 + 36 + 64 + 100 *)
```

On constate que le passage de `S` à `S'` bien que simple n'est pas complètement immédiat : il y a dissémination de l'information. En généralisant un peu, mieux vaut passer deux fonctions supplémentaires en argument :

```
let S' défaut incrémente f a b =
  let rec aux val k =
    if k <= b
```

```
      then aux (val + (f k)) (incrémente k)
      else val
  in if a <= b
      then aux (f a) (incrémente a)
      else défaut ;;

S' : int -> ('a -> 'a) -> ('a -> int) -> 'a -> 'a -> int

S' 0 (function x -> x + 2) (function x -> x * x) 0 10 ;;
- : int = 220
```

Poursuivant la généralisation, on "abstrait" l'opération binaire (ce n'est plus forcément '`+`', ce sera '`opération`') et le test d'inférieur-ou-égal (ce n'est plus forcément '`<=`', ce sera '`infeq`'). On écrira finalement :

```
let accumulation défaut infeq opération incrémente f a b =
  let rec aux val k =
    if (infeq k b)
      then aux (opération val (f k)) (incrémente k)
      else val
  in if (infeq a b)
      then aux (f a) (incrémente a)
      else défaut ;;

accumulation :
 'a ->
   ('b -> 'c -> bool) ->
      ('a -> 'a -> 'a) ->
         ('b -> 'b) ->
            ('b -> 'a) ->
               'b -> 'c -> 'a
```

On pourra alors définir (`Id` est l'identité)

```
let somme_int =
  accumulation 0 le_int add_int succ Id
and produit_int =
  accumulation 1 le_int mult_int succ Id ;;

somme_int : int -> int -> int
produit_int : int -> int -> int

let somme_float =
  accumulation 0.0 le_float add_float (function x -> add_float x 1.0) Id
and produit_float =
  accumulation 0.0 le_float mult_float (function x -> add_float x 1.0) Id ;;

somme_float : float -> float -> float
produit_float : float -> float -> float
```

Un petit test :

```
(somme 0 6, produit 1 6) ;; (* doit être égal à (6*7/2 = 21 , 6! = 720) *)
- : int * int = 21, 720 (* c'est le cas *)

produit_float 1.0 100.0 ;;
- : float = 9.33262154439e+157
```

6.3 dichotomie (¶)

On sait que si f est une application continue de $[a, b]$ dans $\mathbb{R}$ telle que $f(a)f(b) \leq 0$ alors il existe un réel c tel que $a \leq c \leq b$ et tel que $f(c) = 0$.

Nous allons donner le programme qui cherche un zéro de f entre a et b en utilisant une méthode classique (¶) qui est utile dans d'autres contextes.

Si $|b - a| > \varepsilon$ (où ε est la précision demandée) il faut poursuivre la recherche... On calcule alors $m = (a + b)/2$ et selon que f change de signe sur l'intervalle $[a, m]$ ou sur l'intervalle $[m, b]$ on ne poursuit la recherche que sur l'un des deux; il est clair que la longueur de l'intervalle de recherche est divisée par 2 à chaque appel récursif. Cela nous donne la fonction suivante :

```
let rec zéro_par_dichotomie f a b epsilon =
   if abs_float(b -. a) >. epsilon
   then let m = (a +. b) /. 2.0
        in if (f a) *. (f m) <=. 0.0
           then zéro_par_dichotomie f a m epsilon
           else zéro_par_dichotomie f m b epsilon
   else (a +. b) /. 2.0 ;;
```

Appliquons cette méthode aux fonctions $x \mapsto x^2 - a$; cela devrait nous donner une approximation de $\sqrt{a}$.

```
let approx_sqrt_à epsilon a =
   let f  = function x -> x *. x -. a
   in zéro_par_dichotomie f 0.0 a epsilon ;;

approx_sqrt_à 1e-5 2.0 ;;
- : float = 1.41421127319
sqrt 2.0 ;;
- : float = 1.41421356237
```

Il est clair que cette recherche de zéro par dichotomie requiert un temps d'exécution (mesuré en nombre d'appel récursif) qui est proportionnel à $\log_2(b-a)/\varepsilon$; de plus, cette méthode de recherche est conceptuellement très générale. Mais la méthode de Newton (dans les cas où elle peut être appliquée) fournira ici beaucoup plus vite une précision beaucoup plus grande (pour des raisons "mathématiques" et non "informatiques").

6.4 point fixe et méthode de Newton

Un nombre réel x_∞ est dit point fixe d'une application $f : \mathbb{R} \to \mathbb{R}$ si l'on a $f(x_\infty) = x_\infty$. Dans certain cas on peut obtenir un point fixe de f par limite de la suite $x_0, x_{n+1} = f(x_n)$ où x_0 est une valeur de départ choisie "plus ou moins" au hasard. On peut, bien sûr, énoncer des résultats plus précis ! Voir en mathématiques les différents énoncés du théorème du point fixe.

On va commencer par définir une fonction `approx_point_fixe` qui prend en argument (1) une fonction `assez_bon` déterminant si oui ou non x_{n+1} est assez proche de x_n; (2) la fonction f (`f`) elle-même; (3) le point de départ x_0 (`x0`) :

```
let approx_point_fixe assez_bon f x0 =
   let rec aux(x, xx) = if (assez_bon x xx)
                        then xx
                        else aux(xx, f xx)
   in aux(x0, f x0) ;;

approx_point_fixe : ('a -> 'a -> bool) -> ('a -> 'a) -> 'a -> 'a
```

À titre d'exercice, vous pouvez écrire cette fonction avec une boucle “while”; pour certains, elle y gagnera peut-être en clarté (pas pour moi). On peut alors déterminer le point fixe de la fonction cosinus :

```
let approx_à epsilon x y = abs_float(y -. x) <. epsilon ;;
approx_à : float -> float -> float -> bool

approx_point_fixe (approx_à 1e-5) cos 1.0 ;;
- : float = 0.739082298522

cos 0.739082298522 ;;
- : float = 0.739087042696
```

Nous allons maintenant utiliser la fonction `approx_point_fixe` pour implémenter la méthode de Newton. Cette méthode dit que, si x est une approximation d'un zéro d'une fonction f, alors une meilleure approximation de ce zéro est donnée par (dans les bon cas) :

$$x - \frac{f(x)}{f'(x)}$$

On dira que la transformée de Newton de (f, g) (g sera f') est la nouvelle fonction $\mathcal{N}(f, g)$ donnée par :

$$\mathcal{N}(f, g) = x \mapsto x - \frac{f(x)}{g(x)}$$

Voici `Newton`, version curryfiée de $\mathcal{N}$:

```
let Newton f g = function x -> x -. ((f x) /. (g x)) ;;
Newton : (float -> float) -> (float -> float) -> float -> float
```

Pour trouver le zéro d'une fonction f dont f' est la dérivée, il suffit alors de chercher un point fixe de $\mathcal{N}(f, f')$. Cela nous donne la fonction CAML suivante :

```
let zéro_par_Newton f f' epsilon estimation =
   approx_point_fixe (approx_à epsilon) (Newton f f') estimation ;;

zéro_par_Newton :
 (float -> float) -> (float -> float) -> float -> float -> float
```

Pour finir, nous allons appliquer cela à

$$f_a : x \mapsto (x^2 - a) \qquad f'_a : x \mapsto 2x$$

ce qui devrait fournir une approximation de $\sqrt{a}$.

```
let approx_sqrt_à epsilon a =
   let f  = function x -> x *. x -. a
   and f' = function x -> 2.0 *. x
   in zéro_par_Newton f f' epsilon a ;;

approx_sqrt_à : float -> float -> float

approx_sqrt_à 1e-6 77.0 ;;
- : float = 8.77496438739

sqrt 77.0 ;;
- : float = 8.77496438739
```

6.5 fractions continues et séries hypergéométriques

Une fraction continue est une expression de la forme :

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \ddots}}$$

Une manière d'exprimer un nombre irrationnel est d'en donner une représentation de cette forme (avec les N_i et D_i dans $\mathbb{Z}$). On obtient une approximation (rationnelle) de ce nombre en tronquant la fraction continue après un certain nombre de termes. On appellera une telle approximation une approximation de rang k si on retient k termes des suites N_i et D_i (on notera $\mathcal{A}_k$) :

$$\mathcal{A}_k(f) = \cfrac{N_1}{D_1 + \cfrac{N_2}{\ddots + \cfrac{N_k}{D_k + 0}}}$$

Donnons un exemple simple de fraction continue. Si on a $N_i = D_i = 1$ pour tout $i > 0$, on obtient

$$\cfrac{1}{1 + \cfrac{1}{1 + \ddots}}$$

Cette fraction continue converge vers $1/\phi$ où ϕ est le nombre d'or : $\phi = \left(1 + \sqrt{5}\right)/2$.

Supposons que `N` et `D` soient des fonctions de 1 argument (l'indice i) renvoyant les termes N_i et D_i d'une certaine fraction continue (`N` et `D` de type `int -> float`). La fonction `approx_fc` qui suit est telle que `(approx_fc (N,D) k)` calcule l'approximation de rang k de la fraction continue. En voici une version récursive :

```
let approx_fc (N,D) k =
   let rec aux i =
      if (i > k)
         then 0.0
         else div_float (N i)
                        (add_float (D i) (aux (succ i)))
   in aux 1 ;;
```

Voici une version itérative (récursive terminale) :

```
let approx_fc (N,D) k =
   let rec aux = fun
      | 0 r -> r
      | i r -> aux (pred i)
                   (div_float (N i) (add_float (D i) r))
   in aux 0.0 k ;;
```

Une fraction continue pour π fut découverte en 1658 par le mathématicien anglais lord Brouncker :

$$(4/\pi) - 1 = 1/(2 + 3^2/(2 + 5^2/(2 + 7^2/(2 + 9^2/(2 + \ldots$$

Vous pouvez la tester, mais cette formule converge lentement.

Les fractions continues sont aussi utilisées pour exprimer les fonctions. Dans ce cas, N_i et D_i peuvent être des constantes ou eux-mêmes des fonctions d'une ou plusieurs variables. Par exemple, une

fraction continue pour $\arctan x$ fut publiée en 1770 par le mathématicien allemand J. H. Lambert :

$$\arctan x = \cfrac{x}{1 + \cfrac{(1x)^2}{3 + \cfrac{(2x)^2}{5 + \cfrac{(3x)^2}{7 + \ddots}}}}$$

La fonction `atan_fc` est telle que `(atan_fc x k)` calcule une approximation d'ordre `k` de la fonction arctangente.

```
let atan_fc x k =
   let N i = if (i = 1)
             then x
             else (carré (mult_float x (float_of_int (i - 1))))
   and D i = float_of_int((2 * i) - 1)
   in approx_fc (N,D) k ;;
```

Voici un test :

```
atan_fc 10.0 100 ;;
- : float = 1.47112766825
atan 10.0 ;;
- : float = 1.4711276743
```

Les shémas de calcul utilisés pour les fractions continues peuvent être généralisés. Comparer :

$$(N_1 / (D_1 + N_2 / (D_2 + \cdots + N_k / (D_k + 0) \cdots)))$$
$$(N_1 + D_1 \times (N_2 + D_2 \times \cdots \times (N_k + D_k \times 1) \cdots))$$

et (en généralisant) :

$$(T_1 \quad \Xi_1 \quad (T_2 \quad \Xi_2 \quad \cdots (T_k \quad \Xi_k \quad R)))$$

où Ξ_i est une suite d'opérateurs, T_i une suite de termes, et R un terme. La fonction `accumulation_OTRR` (Opérateur/Terme/Reste/Rang) est telle que l'évaluation de

```
(accumulation_OTRR op t r k)
```

calcule la valeur d'une telle expression avec `(op i)` $= \Xi_i$, `(t i)` $= T_i$, et `r` $= R$.

```
let accumule_OTRR opérateur terme reste k =
   let rec aux i =
      if (i = k)
         then (opérateur i) (terme i) reste
         else (opérateur i) (terme i) (aux (succ i))
   in aux 1 ;;
```

Cette fonction peut être utilisée pour calculer une approximation au rang k de l'application (de $\mathbb{R}^+$ vers $\mathbb{R}^+$) définie par

$$f(x) = \sqrt{x + \sqrt{x + \sqrt{x + \sqrt{x + \ldots}}}}$$

```
let racines_emboitées x k =
   let oper = fun i a b -> sqrt(add_float a b)
   and term = fun i -> x
   in accumule_OTRR oper term 1.0 k ;;
```

Vous pouvez vérifier que $f(1) = \phi$:

```
let nombre_d'or = (1.0 +. sqrt(5.0)) /. 2.0 ;;
nombre_d'or : float = 1.61803398875
let aussi_nombre_d'or = racines_emboitées 1.0 25 ;;
aussi_nombre_d'or : float = 1.61803398875
```

La série entière du sinus, à savoir

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots + (-1)^k \frac{x^{2k+1}}{(2k+1)!} + \cdots$$

(convergente pour tout $x \in \mathbb{C}$), peut se mettre sous la forme (schéma de calcul de Horner) :

$$\sin x = x \times \left(1 - \frac{x^2}{2 \times 3} \times \left(1 - \frac{x^2}{4 \times 5} \times \left(1 - \frac{x^2}{6 \times 7} \times \cdots\right)\right)\right)$$

On pourrait donc utiliser `accumule_OTRR` pour calculer des approximations de $\sin x$. Nous allons faire plus général en nous intéressant à la série hypergéométrique.
On appelle série hypergéométrique de Gauss la (les) série(s) définie par

$$\left(\begin{matrix} a_1, a_2, \ldots, a_N \\ b_1, b_2, \ldots, b_M \end{matrix} \middle| \, z \right) = \sum_{k \geq 0} t_k$$

où

$$t_0 = 1 \quad ; \quad t_{k+1} = \frac{P(k)}{Q(k)} \, \frac{z}{(k+1)} \, t_k = \tau_{k+1} \, t_k$$

et où P et Q sont deux polynômes définis par

$$\begin{aligned} P(X) &= (X + a_1)(X + a_2) \ldots (X + a_N) \\ Q(X) &= (X + b_1)(X + b_2) \ldots (X + b_M) \end{aligned}$$

En utilisant le schéma de calcul de Horner, on a :

$$\left(\begin{matrix} a_1, a_2, \ldots, a_N \\ b_1, b_2, \ldots, b_M \end{matrix} \middle| \, z \right) = 1 + \tau_1 \times (1 + \tau_2 \times (1 + \tau_3 \times (1 + \tau_4 \times \cdots)))$$

On utilisera donc `accumule_OTRR` pour calculer des approximations de

$$\left(\begin{matrix} a_1, a_2, \ldots, a_N \\ b_1, b_2, \ldots, b_M \end{matrix} \middle| \, z \right)$$

cela nous donne :

```
let hypergéométrique les_a les_b z n =

   let liste_vers_polynôme liste x =
      let rec aux = fun
         | [] r -> r
         | (h::t) r -> aux t (mult_float r (add_float x h))
      in aux liste 1.0
   in

   let P = (liste_vers_polynôme les_a) and Q = (liste_vers_polynôme les_b)
   in

   let ops i   = if (impair i)
                 then add_float
```

```
                 else mult_float
   and terms i = if (impair i)
                 then 1.0
                 else let k = (i / 2)
                      in (div_float (mult_float z
                                                 (P (float_of_int (k - 1))))
                                    (mult_float (float_of_int k)
                                                 (Q (float_of_int (k - 1)))))
   in accumule_OTRR ops terms 1.0 (2 * n) ;;
```

Les séries hypergéométriques ont été intensivement étudiées (par Gauss en particulier); entre autres résultats, vous pouvez montrer et vérifier (avec CAML) que $e^z = \left(\,\middle|\, z\right)$ (convergente pour tout $z \in \mathbb{C}$) et que $\ln(1+z) = z \left({1,1 \atop 2}\middle| -z\right)$ (convergente pour $|z| \leq 1$). On dispose de résultats plus remarquables, dont celui dû à Pfaff (1797) :

$$\frac{1}{(1-z)^a} \left({a,b \atop c}\middle| \frac{-z}{1-z}\right) = \left({a, c-b \atop c}\middle| z\right)$$

7 structures algébriques et types caml

7.1 la fonction puissance (¶)

Il est facile d'écrire une fonction `puissance` qui calcule, étant donné un nombre x et un entier n, le nombre x^n; cela se fait tout naturellement de manière récursive :

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

On peut reprocher à cette fonction de réclamer en argument toujours le même x non modifié; on doit surtout lui reprocher d'être récursive sous forme non terminale; voici mieux :

```
let puissance x n =
   let rec aux = fun
      | 0 r -> r
      | k r -> aux (k - 1) (x * r)
   in aux n 1 ;;
```

Pour ceux que la récursivité chagrine toujours (ils ont tort), il est maintenant aisé de donner une version purement itérative (pouark) de puissance :

```
let puissance x n =
   let r = ref 1
   in for k = 1 to n do
         r := x * !r
      done;
      !r ;;
```

7.2 l'exponentiation "indienne" (¶)

Commençons par donner un exemple de récursivité multiple (¶) :

```
let rec pair = function
   | 0 -> true
   | 1 -> false
   | n -> impair (n - 1)
```

```
and impair = function
   | 0 -> false
   | 1 -> true
   | n -> pair (n - 1) ;;
```

Mais il nettement plus efficace et plus simple d'écrire :

```
let pair   n = (n mod 2 = 0)
and impair n = (n mod 2 = 1) ;;
```

Les Indiens (les habitants des Indes, pas les Sioux) proposent alors cette façon de calculer x^n :

```
let rec puissance x = function
   | 0 -> 1
   | 1 -> x
   | n -> if (pair n)
          then puissance (x * x) (n / 2)
          else x * (puissance (x * x) (n / 2)) ;;
```

7.3 digression : la fonction puissance dans les monoïdes

Un peu à titre de divertissement, nous allons généraliser la fonction puissance de manière à ce qu'elle opère dans n'importe quel monoïde. Rappellons qu'un monoïde est un triplet $(E, \times, 1)$ où E est un ensemble, $\times$ une opération binaire (ou loi de composition interne) associative sur E ayant un élément neutre (noté) 1. On commence par déclarer le type monoïde :

```
type 'a monoïde = {Mon_Op : 'a -> 'a -> 'a; Mon_Un : unit -> 'a} ;;
```

Un élément de ce type sera (représentera) une structure de monoïde sur les éléments de type `a`. Nous pouvons maintenant re-définir la fonction puissance en passant un argument supplémentaire qui sera la structure de monoïde (`s`) dans laquelle on désire opérer :

```
let puissance s x n =
   let rec aux = fun
      | 0 r -> r
      | k r -> aux (k - 1) (s.Mon_Op x r)
   in aux n (s.Mon_Un()) ;;
```

Le type de `puissance` est

```
puissance : 'a monoïde -> 'a -> int -> 'a
```

Un exemple pas trop verbeux :

```
let string_monoïde = {Mon_Op = (prefix ^); Mon_Un = (function () -> "")} ;;
string_monoïde : string monoïde = {Mon_Op ; Mon_Un }
puissance string_monoïde "bla" 3 ;;
- : string = "blablabla"
```

7.4 digression : structure de groupe

Nous venons de nous intéresser aux structures de monoïde; faisons de même pour les structures de groupe : un élément du type `'a groupe` sera (représentera) une structure de groupe sur les éléments de type `a`.

```
type 'a groupe = {Grp_Op : 'a -> 'a -> 'a;
                  Grp_Sym : 'a -> 'a;
                  Grp_Un : unit -> 'a} ;;
```

On sait, qu'étant donné deux groupes G_1 et G_2, il y a une manière canonique de définir une structure de groupe sur le produit cartésien $G_1 \times G_2$; cela est facile à programmer :

```
let produit_groupe g1 g2 =
   {Grp_Op  = (fun (x1,x2) (y1,y2) -> (g1.Grp_Op x1 y1, g2.Grp_Op x2 y2)) ;
    Grp_Sym = (function (x,y) -> (g1.Grp_Sym x, g2.Grp_Sym y)) ;
    Grp_Un  = (function () -> (g1.Grp_Un(), g2.Grp_Un()))
    } ;;
```

On sait aussi, qu'étant donné un ensemble E et un groupe G, l'ensemble des applications de E dans G peut être canoniquement muni d'une structure de groupe; cela nous donne la fonction suivante :

```
let exp_groupe g =
   {Grp_Op  = (fun f1 f2 x -> g.Grp_Op (f1 x) (f2 x)) ;
    Grp_Sym = (fun f x -> g.Grp_Sym (f x)) ;
    Grp_Un  = (fun () x -> g.Grp_Un())
    } ;;
```

Voici enfin les types de ces deux fonctions tels qu'ils sont calculés par CAML :

```
produit_groupe : 'a groupe -> 'b groupe -> ('a * 'b) groupe
exp_groupe     : 'a groupe -> ('b -> 'a) groupe
```

cela est tout à fait satisfaisant. Nous laissons le lecteur poursuivre dans cette voie s'il le désire.

7.5 ordre lexicographique (¶)

7.5.1 première version

Dans ce qui suit, les mots sur le type `'a` sont représentés par le type `'a list`. On passe en argument à `ordre_lexicographique` :

- `eq` qui est l'égalité sur le type `'a`;
- `infeq` qui est le inférieur-ou-égal sur le type `'a`;
- les deux mots (représentés par les listes) `m1` et `m2`.

```
let ordre_lexicographique eq infeq m1 m2 =
   let rec lexrec = function
         ([], _)           -> true
      |  (_::_, [])        -> false
      |  (a1::r1,a2::r2)   -> if (eq a1 a2)
                                 then lexrec(r1, r2)
                                 else infeq a1 a2
   in lexrec (m1,m2);;

ordre_lexicographique :
 ('a -> 'b -> bool) ->
    ('a -> 'b -> bool) ->
       'a list -> 'b list -> bool
```

7.5.2 généralisation

On peut généraliser cela en définissant le type `'a ordre` :

```
type 'a ordre = {Eq : 'a -> 'a -> bool; Infeq : 'a -> 'a -> bool};;
```

Puis (`R` est un ordre)

```
let étendre_aux_mots R =
   let eq = R.Eq
   and infeq = R.Infeq
   in let rec eq_mots = fun
           []       []       -> true
         | (h1::t1) (h2::t2) -> (eq h1 h2) & (eq_mots t1 t2)
         | _        _        -> false
      and infeq_mots = fun
           []       _        -> true
         | (_::_)   []       -> false
         | (h1::t1) (h2::t2) -> if eq h1 h2
                                   then infeq_mots t1 t2
                                   else infeq h1 h2
   in {Eq = eq_mots ; Infeq = infeq_mots} ;;

étendre_aux_mots : 'a ordre -> 'a list ordre
```

On peut alors commencer par définir une valeur de type `int ordre` (ce sera `naturel`) :

```
let naturel =
   let (int_eq : int -> int -> bool) = (prefix = )
   and (int_infeq : int -> int -> bool) = (prefix <=)
   in { Eq = int_eq ; Infeq = int_infeq } ;;

naturel : int ordre = {Eq=<fun>; Infeq=<fun>}
```

Puis, notant l'ensemble des mots sur A par A^*, on définit ici successivement l'ordre lexicographique sur $\mathbb{N}^*$, puis sur $\mathbb{N}^{**}$, puis sur $\mathbb{N}^{***}$:

```
(* on peut classer les mots *)
let naturel_star = étendre_aux_mots naturel ;;
naturel_star : int list ordre = {Eq=<fun>; Infeq=<fun>}

(* on peut classer les phrases *)
let naturel_star_star = étendre_aux_mots naturel_star ;;
naturel_star_star : int list list ordre = {Eq=<fun>; Infeq=<fun>}

(* on peut classer les livres *)
let naturel_star_star_star = étendre_aux_mots naturel_star_star ;;
naturel_star_star_star : int list list list ordre = {Eq=<fun>; Infeq=<fun>}

(* on pourrait classer les bibliothèques, cela devient borgésien *)
```

8 encapsulation de plusieurs types : un exemple

Les listes, les vecteurs et les chaînes de caractères partagent un certain nombre de caractéristiques que l'on peut résumer en disant (vaguement) que ce sont des séquences finies (et indexées par les entiers naturels) d'objets de même type. On s'attache ici à préciser cette notion.

8.1 approche fonctionnelle

On peut penser modéliser/formaliser cette notion de séquence finie en faisant du type `'a séquence` une abréviation du type `int -> 'a`. Le problème est, qu'alors, nous sommes incapables de repérer

la fin de la séquence si cette dernière est bien une séquence finie. C'est pourquoi nous définissons le type des séquences de la manière suivante :

```
type 'a séquence == int -> ('a option) ;;
```

On a utilisé le type `'a option` qui est pré-défini en CAML de la manière suivante :

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

Si s est une séquence et si n est le plus petit entier naturel tel que $(s\ n) =$ `None` nous saurons alors que la séquence s est finie et qu'elle ne comporte que les termes $s_0, \ldots, s_{n-1}$; bien sûr, si un tel entier n n'existe pas c'est que la séquence s est infinie (notre implémentation des séquences permettrait aussi de gérer ce genre d'objets).

Voici la séquence vide :

```
let (séq_vide : 'a séquence) =
   function k -> None ;;
```

et une fonction qui teste si une séquence est vide :

```
let (séq_est_vide : 'a séquence -> bool) =
   function s ->
      match (s 0) with
         | None -> true
         | _    -> false ;;
```

Remarquer que l'on indique systématiquement le type de la fonction; ceci est tout à fait contraire à l'usage en CAML qui dispose d'un "synthétiseur de types". Cette pratique est motivée par le fait que les séquences (telles qu'on les a définies) ne sont pas vraiment un type, mais seulement une abréviation de type. Si on omet l'indication de typage, CAML donne à l'objet `séq_vide` le type `int -> 'a option`.

Ceci dit, on programme sur les séquences à peu près comme sur les listes :

```
let (séq_tête : ('a séquence) -> 'a) =
   function s ->
      let s0 = (s 0)
      in match s0 with
            | None -> failwith "séquence vide"
            | (Some x) -> x ;;

let (séq_queue : ('a séquence) -> ('a séquence)) =
   fun s k -> s (succ k) ;;

let (séq_ajoute : 'a -> ('a séquence) -> ('a séquence)) =
   fun
      | h t 0 -> Some h
      | h t k -> t (pred k) ;;
```

Voici quelques fonctions classiques qui sont d'habitude définies sur les listes :

```
let (séq_longueur : 'a séquence -> int) =
   let rec séq_longueur_aux n s =
      if séq_est_vide s
      then n
      else séq_longueur_aux (succ n) (séq_queue s)
   in (function s -> séq_longueur_aux 0 s) ;;
```

```
let (séq_concatène : 'a séquence -> 'a séquence -> 'a séquence) =
   fun s1 s2 ->
      let rec append s =
         if séq_est_vide s
         then s2
         else séq_ajoute (séq_tête s) (append (séq_queue s))
      in append s1 ;;

let rec (séq_map : ('a -> 'b) -> 'a séquence -> 'b séquence) =
   fun f s ->
      if séq_est_vide s
      then séq_vide
      else séq_ajoute (f (séq_tête s)) (séq_map f (séq_queue s)) ;;
```

Il va sans dire que toute les fonctions que l'on peut écrire sur les séquences s'appliqueront alors sur les listes, les vecteurs, les chaînes de caractères, etc; il suffit de disposer des fonctions qui réalisent le passage liste vers séquence, vecteur vers séquence, etc.

Donnons, par exemple, les fonctions qui réalisent le passage (les "coercions") liste vers séquence et vecteur vers séquence :

```
let rec list_nth k = function
   |  []  -> failwith "list_nth"
   | h::t -> if k=0 then h else list_nth (k - 1) t ;;

let (list_vers_séquence : 'a list -> 'a séquence) =
   function l ->
      let n = list_length l
      in function k ->
            if (k >= 0) && (k < n)
            then Some (list_nth k l)
            else None ;;

let (vect_vers_séquence : 'a vect -> 'a séquence) =
   function v ->
      let n = vect_length v
      in function k ->
            if (k >= 0) && (k < n)
            then Some v.(k)
            else None ;;
```

Les fonctions inverses s'écriraient tout aussi facilement.

8.2 une autre approche

On peut penser gérer les séquences comme nous l'avons fait pour les structures de groupe et de monoïde; cela nous donnerait quelque chose du genre

```
type ('a , 'b) Sequence =
   {Seq_Head : 'b -> 'a;
    Seq_Tail : 'b -> 'b;
    Seq_Cons : 'a -> 'b -> 'b;
    Seq_Len  : 'b -> int;
    Seq_Void : unit -> 'b;
    } ;;
```

Le champ Seq_Len (la longueur) permet de tester la vacuité d'une séquence. On définirait alors la structure de séquence sur les listes par :

```
let list_as_seq =
   {Seq_Head = hd;
    Seq_Tail = tl;
    Seq_Cons = (fun h t -> h::t) ;
    Seq_Len  = list_length;
    Seq_Void = (function () -> []);
    } ;;
```

Voici, par exemple, la fonctionnelle map programmée dans ce style; sds est une structure de séquence, f est une opération unaire, et s un objet du type sur lequel est définie la structure de séquence sds.

```
let rec map_sequence sds f s =
   if sds.Seq_Len s = 0
   then sds.Seq_Void()
   else sds.Seq_Cons (f (sds.Seq_Head s)) (map_sequence sds f (sds.Seq_Tail s)) ;;
```

Par exemple :

```
map_sequence list_as_seq (function x -> x * x) [1;2;3;4] ;;
- : int list = [1; 4; 9; 16]
```

8.3 dernière approche

Voici, pour terminer, une dernière manière de voir les choses. On peut simplement penser rassembler à l'aide d'un type somme les types prédéfinis de CAML qui possèdent les propriétés des séquences finies :

```
type 'a séquence =
   | Liste of 'a list
   | Vecteur of 'a vect
   | Chaîne of string ;;
```

Cette idée tombe, dans le cas qui nous occupe, sous le coup de deux critiques (sévères, bien sûr) :

(1) programmer, par exemple, une fonction de concaténation qui opère génériquement sur les séquences devient très lourd (essayez...)

(2) tout est à refaire si l'on définit un nouveau type possédant les propriétés des séquences finies.

9 utilisation des “effets de bord”

9.1 fonctions à “variables rémanentes”

Voici un premier exemple :

```
let pas f n sn =
   let r_n = ref n and r_sn = ref sn
   in function
         | false -> (!r_n, !r_sn)
         | true  -> begin
                       r_n := !r_n + 1;
                       r_sn := f (!r_n) (!r_sn);
                       (!r_n, !r_sn)
```

```
                    end ;;

pas : (int -> 'a -> 'a) -> int -> 'a -> bool -> int * 'a
```

Si nous définissons `foo` par :

```
let foo = pas (fun x y -> x* y) 1 1 ;;
```

Alors...

```
foo true ;;
- : int * int = 2, 2
foo true ;;
- : int * int = 3, 6
foo true ;;
- : int * int = 4, 24
foo true ;;
- : int * int = 5, 120
foo false ;;
- : int * int = 5, 120
```

Un autre exemple. On pourrait appeler “accumulateur” une fonction qui, appelée avec un argument numérique, accumule celui-ci dans une somme et renvoie cette somme. Voici une fonction qui généralise cette notion :

```
let fabriquer_accumulateur opérateur valeur_initiale =
   let accumulateur = ref valeur_initiale
   in function x -> accumulateur := opérateur (!accumulateur) x ;
                    !accumulateur ;;

fabriquer_accumulateur : ('a -> 'b -> 'a) -> 'a -> 'b -> 'a
```

9.2 un exemple amusant : une fonction qui s'use

Voici une fonction sinus qui s'use si l'on s'en sert :

```
let sinusite =
   let c = ref 0
   in function x -> (c := !c + 1; div_float (sin x) (float_of_int !c)) ;;

sinusite : float -> float
```

Et voici une fonction qui sert à fabriquer des fonctions qui s'usent (cela n'est guère utile...) :

```
let usure f =
   let c = ref 0
   in function x -> (c := !c + 1; div_float (f x) (float_of_int !c)) ;;

usure : ('a -> float) -> 'a -> float
```

9.3 un autre exemple (moins amusant) : gestion de comptes bancaires

```
let compte_bancaire =
   let bilan = ref 0
   in function n -> if (!bilan + n) < 0
                       then failwith "provision insuffisante !"
                       else bilan := !bilan + n ;
                    !bilan ;;
```

```
let fabriquer_compte_bancaire bilan_initial =
   let bilan = ref bilan_initial
   in function n -> if (!bilan + n) < 0
                     then failwith "provision insuffisante !"
                     else bilan := !bilan + n ;
                    !bilan ;;
```

Pouvez-vous modifier `compte_bancaire` (et `fabriquer_compte_bancaire`) de manière à ce qu'un mot de passe soit demandé et qu'une fonction `appeler_les_flics` (qui pourra se contenter d'afficher "`22 !`" à l'écran) soit invoquée en cas de trois tentatives successives (et infructueuses) d'accéder au compte?

9.4 un dernier exemple : comptage d'appels

Le but du jeu est de garder mémoire du nombre d'appels (non récursifs) à une fonction donnée. On commence par définir un type (`comptage`) qui sera celui des fonctions "suivies". La significations des champs est la suivante :

- `Fun` pour appeler la fonction;
- `Reset` pour remettre le compteur d'appels à zéro;
- `Get` pour consulter le compteur.

```
type ('a,'b) comptage =
   {Fun : 'a -> 'b ; Reset : unit -> unit ; Get : unit -> int} ;;
```

Puis, on définit la fonction `suivre` :

```
let suivre f =
   let compteur = ref 0
   in {Reset = (function () -> begin compteur := 0; () end);
       Fun   = (function x  -> begin compteur := !compteur + 1; (f x) end);
       Get   = (function () -> !compteur)} ;;

suivre : ('a -> 'b) -> ('a, 'b) comptage
```

Vous pouvez utiliser tout cela pour étudier le coût (en nombre d'appels à la comparaison) de `tri_fusion` (entre autres).

10 itérations bornée et non bornée

10.1 itération bornée

10.1.1 trois manières de voir les choses (¶)

On peut voir l'itération bornée de trois manières différentes (au moins); on montrera que ces trois points de vue sont équivalents.

- Schéma de récursion primitive (`PR`). Étant données deux fonctions g et h, une fonction f est dite définie par schéma de récursion primitive si

$$\begin{aligned} f(0,x) &= g(x) \\ f(n+1,x) &= h(f(n,x),n,x) \end{aligned}$$

  Si les fonctions g et h sont définies sur tous leurs arguments (i.e. les calculs s'achèvent toujours (i.e. en un temps fini !)), alors il est clair que f possède aussi cette propriété.

- Itération bornée $\mathcal{F}$ d'une fonction f (IB). Étant donnée une fonction f terminant toujours, si on définit une nouvelle fonction par

$$(n, x) \mapsto (\mathcal{F}_n\, f)\, x = \left| \begin{array}{ll} x & \text{si } n = 0 \\ (\mathcal{F}_{n-1}\, f)(f\, x) & \text{si } n \neq 0 \end{array} \right.$$

alors cette fonction termine toujours elle aussi.

- La boucle "for" (BF). On donnera une fonction encapsulant un usage typique de la boucle "for".

Donnons maintenant les fonctions CAML correspondantes

- PR : le schéma de récursion primitive :

```
let PR g h =
   let rec f = fun
      | 0 x -> g x
      | n x -> h (f (pred n) x) (pred n) x
   in f ;;
```

En voici une version récursive terminale :

```
let PR_terminal g h n x =
   let rec aux = fun
      | 0 r -> r
      | n r -> aux (pred n) (h r (pred n) x)
   in aux n (g x) ;;
```

Le type de ces deux fonctions est :

```
PR          : ('a -> 'b) -> ('b -> int -> 'a -> 'b) -> int -> 'a -> 'b
PR_terminal : ('a -> 'b) -> ('b -> int -> 'a -> 'b) -> int -> 'a -> 'b
```

- IB : la fonction d'itération bornée :

```
let rec IB = fun
   | 0 f -> Id
   | n f -> compose (IB (n - 1) f) f ;;
```

Il n'est pas difficile d'en donner une version récursive terminale ! :

```
let rec IB_terminale = fun
   | 0 f x -> x
   | n f x -> IB_terminale (pred n) f (f x) ;;
```

Le type de ces deux fonctions est :

```
IB            : int -> ('a -> 'a) -> 'a -> 'a
IB_terminale  : int -> ('a -> 'a) -> 'a -> 'a
```

- BF : la boucle for dans son usage typique :

```
let BF n1 n2 action r =
   begin
      for i = n1 to n2 do action (i, r) done;
      !r;
   end ;;
```

Le typage donne :

```
BF : int -> int -> (int * 'a ref -> 'b) -> 'a ref -> 'a
```

10.1.2 ces trois points de vue sont équivalents

Si les fonctions qui suivent ne le démontre pas, elles le montrent.

Le schéma `IB` est exprimable par le schéma `PR` :

```
let IB_par_PR = PR (fun x -> Id)
                   (fun r n x -> compose r x) ;;

   IB_par_PR : int -> ('a -> 'a) -> 'a -> 'a
   IB        : int -> ('a -> 'a) -> 'a -> 'a
```

Le schéma `PR` est exprimable par le schéma `IB` :

```
let PR_par_IB g h n x =
   snd (IB n
          (function (n, r) -> (n + 1, h r n x))
          (0, (g x))
       ) ;;

   PR_par_IB : ('a -> 'b) -> ('b -> int -> 'a -> 'b) -> int -> 'a -> 'b
   PR        : ('a -> 'b) -> ('b -> int -> 'a -> 'b) -> int -> 'a -> 'b
```

Le schéma `IB` est exprimable par une boucle `for` :

```
let IB_par_for n f x =
   let r = ref x
   in begin
        for v = 1 to n do r := f x done;
        !r;
      end ;;

   IB_par_for : int -> ('a -> 'a) -> 'a -> 'a
   IB         : int -> ('a -> 'a) -> 'a -> 'a
```

Le schéma `PR` est exprimable par une boucle `for` :

```
let PR_par_for g h n x =
   let ref_res = ref (g x)
   in begin
        for i=1 to n do ref_res := h (!ref_res) (i-1) x done;
        !ref_res;
      end ;;

   PR_par_for : ('a -> 'b) -> ('b -> int -> 'a -> 'b) -> int -> 'a -> 'b
   PR         : ('a -> 'b) -> ('b -> int -> 'a -> 'b) -> int -> 'a -> 'b
```

La boucle for dans son usage typique (`BF`) est exprimable par le schéma `IB` :

```
let BF_par_IB n1 n2 action r =
   let aux (i, r) = (i + 1, begin action (i, r) ; r end)
   in !(snd( (IB (n2 - n1 + 1) aux) (n1, r) )) ;;

   BF_par_IB : int -> int -> (int * 'a ref -> 'b) -> 'a ref -> 'a
   BF        : int -> int -> (int * 'a ref -> 'b) -> 'a ref -> 'a
```

10.2 itération non bornée

L'itération non bornée, par rapport à l'itération bornée, fait perdre la propriété de terminaison sur tous les arguments. C'est ce qui fait toute la puissance (théorique et pratique) de ce schéma.

10.2.1 deux manières de voir les choses (¶)

- Itération non bornée $\mathcal{W}$ d'une fonction f (`INB`). Ce schéma consiste à itérer la fonction f tant que $(p\,x)$ reste vrai :

$$x \mapsto (\mathcal{W}\ p\,f)\,x = \left|\begin{array}{ll} x & \text{si } (p\,x) = \mathsf{false} \\ (\mathcal{W}\ p\,f)(f\,x) & \text{si } (p\,x) = \mathsf{true} \end{array}\right.$$

  Ce schéma se code sans difficulté en CAML :

```
let rec INB p f x =
   if (p x)
   then INB p f (f x)
   else x ;;
```

  Voici le type de `INB` :

```
INB : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a
```

- La boucle "while" (`BW`). Voici une fonction encapsulant un usage typique du "while" :

```
let BW test action r =
   begin
      while (test r) do (action r) done ;
      r ; (* ou !r, si l'on veut marquer que r peut être une référence *)
   end ;;
```

  Son type est :

```
BW : ('a -> bool) -> ('a -> 'b) -> 'a -> 'a
```

10.2.2 ces deux points de vue sont équivalents

```
let BW_par_INB test action r =
   let p x = test x
   and f x = (action x ; x)
   in INB p f r ;;

   BW_par_INB : ('a -> bool) -> ('a -> 'b) -> 'a -> 'a
   BW         : ('a -> bool) -> ('a -> 'b) -> 'a -> 'a

let INB_par_BW p f x =
   let test r = (p !r)
   and action r = (r := f !r)
   in !(BW test action (ref x)) ;;

   INB_par_BW : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a
   INB        : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a
```

10.2.3 le "while" est plus "puissant" que le "for" (¶)

Les deux fonctions suivantes montrent que l'itération bornée est exprimable par l'itération non bornée :

```
let IB_par_while n f x =
   let (rx, rn) = (ref x, ref n)
   in begin
         while (!rn <> 0) do (rx := f !rx ; rn := !rn - 1) done;
         !rx
      end ;;

   IB_par_while : int -> ('a -> 'a) -> 'a -> 'a
   IB           : int -> ('a -> 'a) -> 'a -> 'a

let IB_par_BW n f x =
   let test r = (!(snd r) <> 0)
   and action r = ( fst r := f !(fst r) ; snd r := !(snd r) - 1 )
   in !(fst(BW test action (ref x, ref n))) ;;

   IB_par_BW : int -> ('a -> 'a) -> 'a -> 'a
   IB        : int -> ('a -> 'a) -> 'a -> 'a
```

Le "while" est donc d'une "puissance" supérieure ou égale à celle du "for". En fait, on montre (en théorie de la calculabilité) que le "while" est strictement plus "puissant" que le "for".

10.3 passage à la récursivité terminale (quand cela est possible)

Pour la factorielle :

```
let fact_rt n =
   let rec aux = fun
      | 0 r -> r
      | k r -> aux (k - 1) (k * r)
   in aux n 1;;

let fact_par_IB n = snd(
   IB n
      (function (n, fn) -> (n + 1, fn * (n + 1)))
      (0,1)
      ) ;;
```

Pour la suite de Fibonacci :

```
let rec fib = function (* version non récursive terminale *)
   | 0 -> 1
   | 1 -> 1
   | n -> fib(n - 1) + fib(n - 2) ;;

let fib_rt n =
   let rec aux = fun
      | 0 r1 r2 -> r1
      | 1 r1 r2 -> r2
      | n r1 r2 -> aux (n - 1) r2 (r1 + r2)
   in aux n 1 1 ;;

let fib_par_IB n = snd(
   IB n
      (function (x, y) -> (x + y, x))
      (1, 1)
      ) ;;
```

Remarques importantes.

- Si l'on dispose d'une définition de f par schéma PR, il est toujours possible de l'exprimer sous forme d'un schéma IB. L'idée générale est contenue dans le code de la fonction `PR_par_IB`.
- Une fois écrite la version `IB` de f il est trivial d'en donner une version récursive terminale : comparer les versions `fib_par_IB` et `fib_rt` de la suite de Fibonacci.
- Cette version récursive terminale se transforme facilement en une boucle : comparer `fib_rt` et

  ```
  let fib_par_for n =
     let r1 = ref 1 and r2 = ref 1 and tampon = ref 0
     in for i = 2 to n do
           tampon := !r1;
           r1 := !r2;
           r2 := !tampon + !r2
        done;
        !r2 ;;
  ```

- À mon sens, les versions récursives sont toujours plus claires que les versions itératives et (c'est lié) il est beaucoup plus facile de raisonner sur les versions récursives que sur les versions itératives : comparer `fib_par_for` et `fib_rt`...

11 structure de liste

11.1 définition du type liste (¶)

Le type `'a list` est pré-défini en CAML mais pourrait être re-défini :

```
type 'a liste = Vide | Cellule of 'a * ('a liste) ;;
```

Voici les fonctions fondamentales qui opèrent sur ce type :

```
let tête = function
   | Vide -> failwith "tête"
   | Cellule(t,q) -> t

and queue = function
   | Vide -> failwith "queue"
   | Cellule(t,q) -> q

and construire t q = Cellule(t,q) ;;
```

- `Vide` correspond à `[]`
- `tête` correspond à `hd`
- `queue` correspond à `tl`
- `construire` correspond à `(fun h t -> (h::t))`

11.2 itérateurs sur les listes

Commençons par remarquer que les entiers pourrait être définis en CAML de la manière suivante :

```
type nat =
   | Zéro
   | Succ of nat ;;
```

Ce type fait intervenir un constructeur 0-aire (`Zéro`) et un constructeur unaire (`Succ`). Le type des listes fait intervenir un constructeur 0-aire (`[]`) et un constructeur binaire (`::`, sous forme infixe). Le schéma de calcul "récursion primitive" a été défini en utilisant le type `int`; il pourrait être défini en utilisant le type `nat` :

```
let PR_nat g1 g3 =
   let rec f = fun
      | Zéro     x -> g1 x
      | (Succ n) x -> g3 n x (f n x)
   in f ;;
```

Le i dans les g i indique le nombre d'argument. En essayant de généraliser ce schéma au type liste on obtient par exemple :

```
let PR_list g1 g2 g4 =
   let rec f = fun
      |   []   x -> g1 x
      | (h::t) x -> g4 h t x (f t (g2 h x))
   in f ;;
```

L'intérêt de ce schéma tient au fait suivant : si les fonctions g i terminent pour tout argument, alors la fonction `(PR_list g1 g2 g4)` (i.e. la fonction `f`) possède la même propriété; pour le voir, il suffit de remarquer d'une part que le couple `((h::t), x)` est toujours lexicographiquement supérieur au couple `(t,y)` et d'autre part que la terminaison est assurée sur les couples `([],x)`.

On peut faire de même avec le schéma d'itération bornée en remplaçant les entiers par les `nat`.

```
let rec IB_nat = fun
   | f Zéro     x -> x
   | f (Succ n) x -> IB_nat f n (f x) ;;
```

Ce qui est récursif terminal. Essayons de trouver un analogue sur les listes. La fonction qui suit est aussi sous forme récursive terminale :

```
let rec IB_list = fun
   | f   []   x -> x
   | f (h::t) x -> IB_list f t (f x h) ;;

IB_nat  : ('a -> 'a) -> nat -> 'a -> 'a
IB_list : ('a -> 'b -> 'a) -> 'b list -> 'a -> 'a
```

Remarques.

- CAML propose la fonctionnelle `it_list` dont le comportement est le suivant

  `it_list f x [a_1 ; ... ; a_N]`

  renvoie

  `(f (...(f (f x a_1) a_2)...) a_N)`

  son type est

```
it_list   : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

La librairie définit cette fonction par :

```
let it_list f = it_list_f
   where rec it_list_f a = function
      | [] -> a
      | b::l -> it_list_f (f a b) l ;;
```

ce qui peut aussi s'écrire

```
let it_list f =
   let rec calcule = fun
      | x   []   -> x
      | x (h::t) -> calcule (f x h) t
   in calcule ;;
```

ou encore

```
let rec it_list = fun
   | f x   []   -> x
   | f x (h::t) -> it_list f (f x h) t ;;
```

Cette fonction n'est autre, à une interversion d'argument près, que `IB_list`.

- CAML propose aussi la fonctionnelle `list_it` qui peut s'écrire (sous forme récursive NON terminale)

```
let rec list_it = fun
   | f   []   x -> x
   | f (h::t) x -> f h (list_it f t x) ;;
```

et dont le type est

```
list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Son comportement est le suivant.

```
list_it f [a_1; ... ; a_N] x = (f a_1 (f a_2 (...(f a_N x)...)))
```

- Remarquer que

```
it_list (fun t h -> h::t) [] [1;2;3;4;5] ;;
- : int list = [5; 4; 3; 2; 1]
```

On peut définir la fonction qui renverse une liste par :

```
let reverse = it_list (fun t h -> h::t) [] ;;
```

- Les deux fonctions `list_it` et `it_list` sont définissables l'une par l'autre. En effet, si l'on définit `permute` par

```
let permute f x y = f y x ;;
```

alors, la fonction `item_list` (qui suit) est égale à la fonction `it_list` et la fonction `list_item` (qui suit) est égale à la fonction `list_it` :

```
let item_list f x l = list_it (permute f) (reverse l) x
and list_item f l x = it_list (permute f) x (reverse l) ;;
```

- On accordera une légère préférence à `it_list` car celle-ci est sous forme récursive terminale (c'est virtuellement une boucle).

11.3 quelques fonctions sur les listes (¶)

11.3.1 longueur d'une liste

```
let rec longueur = function
   |  []  -> 0
   | _::t -> 1 + (longueur t) ;;
```

Voici une version récursive terminale :

```
let longueur_rt l =
   let rec aux = fun
      |   []   r -> r
      | (_::t) r -> aux t (succ r)
   in aux l 0 ;;
```

On peut aussi utiliser nos itérateurs pour trouver la longueur :

```
let longueur_par_list_it l =
   list_it (fun x n -> succ n) l 0 ;;
```

Ou encore :

```
let longueur_par_list_it l =
   list_it (fun _ n -> succ n) l 0 ;;
```

Mais il vaut mieux utiliser `it_list` qui est sous forme récursive terminale; cette version est tout aussi efficace qu'une version itérative :

```
let longueur_par_it_list l =
   it_list (fun n x -> succ n) 0 l ;;
```

Ou encore :

```
let longueur_par_it_list = it_list (fun n _ -> succ n) 0 ;;
```

On trouvera facilement une version utilisant le schéma PR.

11.3.2 calcul du minimum

La fonction suivante calcule sans trop de peine le minimum de ses deux arguments :

```
let minimum2 infeq a b =
   if (infeq a b)
   then a
   else b ;;
```

On peut alors calculer le minimum d'une liste (non vide) :

```
let rec minimum infeq = function
   | [a]  -> a
   | h::t -> minimum2 infeq h (minimum infeq t) ;;
```

La version qui suit est itérative; pardon : récursive terminale...

```
let minimum infeq l =
   let rec recherche_minimum = fun
      | [] r -> r
      | (h::t) r -> recherche_minimum t (minimum2 infeq h r)
   in recherche_minimum (tl l) (hd l) ;;
```

Pourquoi ne pas utiliser `it_list` ?

```
let minimum infeq l = it_list (minimum2 infeq) (hd l) l;;
```

11.3.3 n-ième élément d'une liste

Sans commentaires :

```
let rec list_nth = fun
   | _   []   -> failwith "list_nth"
   | 0 (h::t) -> h
   | n (h::t) -> list_nth (pred n) t ;;
```

11.3.4 concaténation de deux listes

Remarquer que la récursion ne s'effectue que sur la première liste :

```
let concatène l1 l2 =
   let rec concat = function
      |  []  -> l2
      | h::t -> h::(concat t)
   in concat l1 ;;
```

On peut écrire aussi :

```
let concatène l1 l2 = list_it (fun a x -> a::x) l1 l2 ;;
```

En revanche, attention à la fonction suivante :

```
let concatène_à_l'envers l1 l2 = it_list (fun x a -> a::x) l2 l1 ;;
```

Il faut signaler que `concatène` est disponible sous forme infixe en CAML :

```
[1;2] @ [2;3;4] ;;
- : int list = [1; 2; 2; 3; 4]
```

Sachez enfin que `concatène` se dit souvent `append`.

11.3.5 image miroir

La version suivante est récursive terminale : c'est virtuellement une boucle.

```
let image_miroir l =
   let rec aux = fun
      |   []   r -> r
      | (h::t) r -> aux t (h::r)
   in aux l [] ;;
```

La complexité de `image_miroir` est donc un $O(N)$ où N est la longueur de la liste. Mais pourquoi ne pas écrire (c'est tout aussi itératif) :

```
let image_miroir = it_list (fun x a -> a::x) [] ;;
```

En revanche, cette version est à éviter :

```
let rec miroir_gentil_miroir = function
   |  []  -> []
   | h::t -> concatène (miroir_gentil_miroir t) [h] ;;
```

Sauriez-vous montrer qu'elle est en $O(N^2)$? Quant à celle-ci il faut l'oublier :

```
let rec sept_ans_de_malheur = function
   | [ ] -> [ ]
   | [a] -> [a]
   |  l  -> (hd (sept_ans_de_malheur (tl l)))
             ::
            (sept_ans_de_malheur ((hd l)
                                  ::
                                 (sept_ans_de_malheur
                                    (tl (sept_ans_de_malheur (tl l)))))) ;;
```

11.3.6 insertion d'un élément

La fonction `insère` insère (en poussant les autres) le nouvel élément `a` à la `i`-ème place dans la liste `l`; les indices commencent à 0 :

```
let insère i a l =
   let rec aux = fun
      | _ []     -> failwith "insère"
      | 0 l      -> a::l
      | k (h::t) -> h::(aux (pred k) t)
   in aux i l ;;
```

Exemple :

```
insère 2 0 [0;1;2;3;4];;
- : int list = [0; 1; 0; 2; 3; 4]
```

On peut aussi se proposer d'insérer un élément dans une liste déjà triée (en préservant l'ordre, bien sûr) :

```
let insère_dans_liste_triée infeq x l =
   let rec aux = function
      | [] -> [x]
      | h::t as l -> if (infeq x h) then (x::l) else (h::(aux t))
   in aux l ;;
```

11.3.7 le tri par insertion sur les listes

Le tri par insertion sur les listes peut donc s'écrire :

```
let rec tri_insertion infeq = function
   |  []  -> []
   | h::t -> insère_dans_liste_triée infeq h (tri_insertion infeq t) ;;
```

11.3.8 suppression d'un élément

Cette version supprime *toutes les occurences* de `x` dans la liste `l` :

```
let supprime x l =
   let rec aux = function
      |  []  -> []
      | h::t -> if (h = x) then (aux t) else (h::(aux t))
   in aux l ;;
```

Voici `supprime` à l'œuvre :

```
supprime 3 [1;2;3;1;2;3] ;;
- : int list = [1; 2; 1; 2]
```

11.3.9 le tri par sélection sur les listes

Aucune prétention d'efficacité; on le donne uniquement pour éclairer (j'espère) l'algoritme.

On commence par modifier `supprime` de manière à ce qu'elle ne s'en prenne qu'à la première occurence recontrée :

```
let supprime_un_seul x l =
   let rec aux = function
      |  []  -> []
      | h::t -> if (h = x) then t else (h::(aux t))
   in aux l ;;
```

Le tri par sélection sur les listes peut alors s'écrire de la façon suivante :

```
let tri_sélection infeq l =
   let rec trier = fun
      | [] r -> r
      | x  r -> let m = (minimum infeq x)
                in trier (supprime_un_seul m x) (r @ [m])
   in trier l [] ;;
```

Mais il vaut mieux écrire :

```
let tri_sélection infeq l =
   let supeq x y = infeq y x
   in let rec trier = fun
         | [] r -> r
         | x  r -> let m = (minimum supeq x)
                   in trier (supprime_un_seul m x) (m::r)
      in trier l [] ;;
```

Quant au tri-bulle sur les listes mieux vaut n'y pas penser...

11.3.10 le tri rapide sur les listes

Sur les listes, mieux vaut utiliser le tri-fusion; sur les vecteurs, mieux vaut utiliser le tri rapide. La version du "tri rapide" sur les listes que l'on va donner ici n'a de rapide que le nom : on fait appel à la concaténation, ce qui est coûteux en temps et en espace mémoire. En revanche, la fonction `quicksort` (quick sort = tri rapide) présentée ici donne une idée très claire de cet algorithme. Il est conseillé de lire les exemples d'abord.

La fonction suivante partitionne une liste selon un prédicat :

```
let partition prédicat liste =
   let rec aux = fun
      |   []   r1 r2 -> (r1, r2)
      | (h::t) r1 r2 -> if (prédicat h)
                        then aux t r1 (h::r2)
                        else aux t (h::r1) r2
   in aux liste [] [] ;;
```

Son type :

```
partition : ('a -> bool) -> 'a list -> 'a list * 'a list
```

Et voici "quicksort" :

```
let quicksort infeq =
   let rec trier = function
      | []   -> []
      | [a]  -> [a]
      | h::t -> let (l1, l2) = partition (infeq h) t
                in (trier l1) @ [h] @ (trier l2)
   in trier ;;
```

et son type :

```
quicksort : ('a -> 'a -> bool) -> 'a list -> 'a list
```

Exemples :

```
let a = random_list 7 7 ;; (* fonction définie dans le paragraphe sur les tris *)
a : int list = [1; 7; 6; 3; 6; 3; 2]

let pair n = (n mod 2 = 0) ;;
pair : int -> bool

partition pair a ;;
- : int list * int list = [3; 3; 7; 1], [2; 6; 6]

let infeq = prefix <= ;;
infeq : 'a -> 'a -> bool

partition (infeq 3) a ;;
- : int list * int list = [2; 1], [3; 6; 3; 6; 7]

quicksort infeq a ;;
- : int list = [1; 2; 3; 3; 6; 6; 7]
```

11.3.11 quelques versions (purement) itératives

On ne donnera de version purement itérative que pour la longueur et le maximum; vous pouvez vous "divertir" à en écrire d'autres.

```
let longueur_itérative l =
   let x = ref l and i = ref 0
   in while !x <> [] do
         i := !i + 1; x := tl !x
      done;
      !i ;;

let maximum_itératif l =
   let x = ref l and m = ref 0
   in while !x <> [] do
         if (hd !x) > !m then m := hd !x; x := tl !x
      done;
      !m ;;
```

12 procédures de tri (¶)

12.1 quelques utilitaires

Quelques manipulations élémentaires sur les références :

```
let incr_puis_val r = ( r := !r + 1 ; !r )
and val_puis_incr r = ( r := !r + 1 ; !r - 1)
and décr_puis_val r = ( r := !r - 1 ; !r )
and val_puis_décr r = ( r := !r - 1 ; !r + 1) ;;
```

La fonction `indice_max` donne l'indice maximum (n) d'un vecteur : c'est sa longueur (N) diminuée de 1 ($n = N - 1$) car les indices commencent à 0.

```
let indice_max v = (vect_length v) - 1 ;;
```

La fonction suivante permet l'échange, dans le vecteur v du contenue des "cases" numéro i et numéro j.

```
let échange v i j =
   let vi = v.(i)
   in v.(i) <- v.(j) ; v.(j) <- vi ;;
```

La fonction `random_vect` permet de fabriquer un vecteur de taille `len` ne contenant que des entiers compris entre 0 (compris) et `max` compris; ces entiers sont répartis au hasard.

```
let random_vect len max =
   let v = make_vect len 0
   in for i = 0 to (len - 1) do
         v.(i) <- random__int (max + 1)
      done;
      v ;;
```

La même chose, mais sur les listes.

```
let random_list len max =
   let rec aux = fun
      | 0 r -> r
      | i r -> aux (i - 1) ((random__int (max + 1)) :: r)
   in aux len [] ;;
```

12.2 le tri bulle (bubble sort)

```
let parcourir infeq v =
   for i = 1 to (indice_max v) do
      if not (infeq v.(i - 1) v.(i))
      then échange v (i - 1) i
   done ;;

let trié infeq v =
   let résultat = ref true
   in for i = 1 to (indice_max v) do
         résultat := !résultat && (infeq v.(i - 1) v.(i))
      done;
      !résultat ;;
```

La version qui suit est légèrement plus efficace.

```
let trié infeq v =
   let rec regarde = fun
      | _ false -> false
      | i true -> if i = indice_max v
                  then true
                  else regarde (i + 1) (infeq v.(i) v.(i + 1))
   in regarde 0 true ;;
```

```
let tri_bulle infeq v =
   while not (trié infeq v) do (parcourir infeq v) done ;;
```

La version qui suit est légèrement plus efficace.

```
let tri_bulle infeq v =
   for i = (indice_max v) downto 0 do
      for j = 1 to i do
         if not (infeq v.(j - 1) v.(j))
         then échange v (j - 1) j
      done
   done ;;
```

12.3 le tri par sélection (selection sort)

```
let indice_du_plus_petit infeq v i =
   let n = indice_max v
   and m = ref i
   in for j = (i + 1) to n do
         if infeq v.(j) v.(!m) then m := j
      done;
      !m ;;

let tri_sélection_v1 infeq v =
   let n = indice_max v
   in for i = 0 to (n - 1) do
         échange v i (indice_du_plus_petit infeq v i)
      done ;;
```

12.4 le tri par insertion (insertion sort)

```
let insère infeq v i =
   let j = ref i and x = v.(i)
   in while (!j > 0)  && not(infeq v.(!j - 1) x) do
         v.(!j) <- v.(!j - 1);
         j := !j - 1;
      done;
      v.(!j) <- x ;;

let tri_insertion infeq v =
   for i = 1 to (indice_max v) do insère infeq v i done ;;
```

12.5 le tri rapide (quick sort), ou tri de Hoare, ou tri par segmentation

Il est inventé par Hoare en 1960.

```
let partition infeq v i j =
   let pivot = v.(i) and k = ref i
   in for l = i + 1 to j do
         if infeq v.(l) pivot
         then échange v l (incr_puis_val k)
      done;
      échange v i !k;
      !k ;;
```

```
let rec le_noyau_du_tri_rapide infeq v i j =
   if (i < j)
   then let k = partition infeq v i j
        in begin
             le_noyau_du_tri_rapide infeq v i (k - 1);
             le_noyau_du_tri_rapide infeq v (k + 1) j;
           end ;;

let tri_rapide infeq v =
   le_noyau_du_tri_rapide infeq v 0 (indice_max v) ;;
```

12.6 le tri fusion (merge sort)

Les tris précédents opéraient (par effets de bord) naturellement sur des vecteurs; le tri fusion opère naturellement sur les listes.

```
let tri_fusion infeq liste =

   let rec découpe_en_deux = function
      | []  -> [],[]
      | [a] -> [a],[]
      | a :: b :: q -> let l1,l2 = découpe_en_deux q
                       in (a :: l1),(b :: l2)

   and fusionne l1 l2 = match l1,l2 with
      | [],_ -> l2
      | _,[] -> l1
      | a :: q1 , b :: q2 -> if (infeq a b)
                             then a :: (fusionne q1 l2)
                             else b :: (fusionne l1 q2)

   in let rec le_noyau_du_tri_fusion = function
         | []  -> []
         | [a] -> [a]
         |  l  -> let l1,l2 = découpe_en_deux l
                  in fusionne (le_noyau_du_tri_fusion l1)
                              (le_noyau_du_tri_fusion l2)

      in le_noyau_du_tri_fusion liste ;;
```

13 deux utilisations des listes (¶)

Les listes peuvent commodément représenter certains êtres mathématiques; on traitera ici deux exemples : polynômes et matrices.

13.1 polynômes creux

Un polynôme du genre $1 + 2X^2 + X^{100}$ est dit un "polynôme creux" : son degré n est élevé mais pour la majorité des entiers k tels que $0 \leq k \leq n$ le coefficient de X^k est nul. Représenter un tel polynôme par un tableau de longueur $n + 1$ serait un gaspillage de ressource mémoire; on préfère sacrifier l'avantage de l'accès en temps constant aux coefficients a_k du polynôme $\sum_k a_k X^k$ que procurent les vecteurs et représenter le polynôme par une liste.

On représente donc un polynôme comme une liste *ordonnée* de monômes non nuls et on représente un monôme $a_k X^k$ sous la forme du couple (k, a_k). Un polynôme creux à coefficients réels sera donc stocké comme une liste de couples d'un entier naturel et d'un nombre réel.

- le polynôme nul est stocké comme la liste vide;
- le polynôme $1 + 2X^2 + X^{100}$ est stocké comme $[(0, 1.0); (2, 2.0); (100, 1.0)]$.

Cela nous conduit aux (abréviations de) types CAML suivants :

```
type 'a monôme == int * 'a ;;
type 'a polynôme == ('a monôme) list ;;
```

Voici la procédure d'addition des polynômes; elle ne soulève aucune difficulté mais on prendra garde au fait qu'elle traite et doit fournir des polynômes ordonnés suivant les puissances croissantes.

```
let rec add_p p1 p2 =
   match (p1, p2) with
      | [], _ -> p2
      | _, [] -> p1
      | ((d1,a1)::r1), ((d2,a2)::r2) ->
            if d1 < d2
               then (d1, a1)::(add_p r1 p2)
            else if d2 < d1
               then (d2, a2)::(add_p p1 r2)
            else if (add_float a1 a2) <> 0.0
               then (d1, add_float a1 a2)::(add_p r1 r2)
            else add_p r1 r2 ;;
```

Pour définir le produit de deux polynômes, on commence par définir le produit d'un polynôme par un monôme : $aX^n \sum_k b_k X^k = \sum_k ab_k X^{n+k}$.

```
let rec prod_mp (n, a) p = (* produit d'un monôme par un polynôme *)
   if a = 0.0
   then []
   else match p with
      | [] -> []
      | (k1, b1)::r -> (n + k1, mult_float a b1)::(prod_mp (n, a) r) ;;
```

Le produit est alors immédiat :

```
let rec prod_p p1 p2 =
   match p1 with
      | [] -> []
      | m1::r1 -> add_p (prod_mp m1 p2) (prod_p r1 p2) ;;
```

13.2 matrices creuses

Une matrice est dite creuse si la majorité de ses coefficients sont nuls. On utilisera une représentation des matrices creuses du même type que celle utilisée pour les polynômes creux : la matrice $a_{i,j}$ sera représentée par la liste $[\ldots; ((i,j), a_{i,j}); \ldots]$, *les éléments de cette liste étant ordonnée suivant l'ordre lexicographique sur les (i,j) : on devra prendre garde à ce que les fonctions que l'on écrira sur les matrices creuses conservent cette propriété.*
Rappelons la définition de l'ordre lexicographique strict (on le notera $\prec$ par la suite) :

```
let ols (i1,j1) (i2,j2) =
       (i1 < i2) || ((i1 = i2) && (j1 < j2)) ;;
```

En fait, la comparaison CAML est générique et '<' correspond, sur les couples, à l'ordre lexicographique strict ($\prec$) : le travail est déjà fait. L'addition des matrices creuses étant similaire à l'addition des polynômes creux, il suffit quasiment de recopier.

```
let rec add_mat m1 m2 =
   match (m1,m2) with
      | [] , _ -> m2
      | _ , [] -> m1
      | ((d1, a1)::r1) , ((d2, a2)::r2) ->
            if d1 < d2
               then (d1, a1)::(add_mat r1 m2)
            else if d2 < d1
               then (d2, a2)::(add_mat m1 r2)
            else if (add_float a1 a2) <> 0.0
               then (d1, add_float a1 a2)::(add_mat r1 r2)
            else add_mat r1 r2 ;;
```

En ce qui concerne le produit des matrices creuses, on commence par définir le produit d'une matrice B par une matrice élémentaire du type $aE_{i,j}$ où $E_{i,j}$ est la matrice qui a des zéros partout, sauf à l'intersection de la i-ième ligne et de la j-ième colonne où figure un 1.

On rappelle que

$$E_{i,j}E_{k,l} = \delta_j^k E_{i,l}$$

avec $\delta_j^k = 1$ si $j = k$ et $\delta_j^k = 0$ sinon.

On en déduit que si $B = bE_{k,l} + B_1$ alors

$$aE_{i,j}B = ab\delta_j^k E_{i,l} + aE_{i,j}B_1$$

Le lemme suivant nous garantit que l'ordre lexicographique ($\prec$) sur les indices (i, j) correspondant à des termes non nuls est conservé. J'emprunte ce paragraphe à D. Monasse dans [8].

Lemme. L'ordre lexicographique ($\prec$) sur les matrices $E_{i,j}$ est compatible avec la multiplication à gauche et à droite.

Démonstration. Bien entendu notre affirmation est un peu abusive, à moins de prendre par convention que les deux affirmations $A \prec 0$ et $0 \prec A$ sont vraies. Ceci signifie simplement que si $(i, j) \prec (i', j')$ et si les produits sont tous deux non nuls, $E_{i,j}E_{k,l} \prec E_{i',j'}E_{k,l}$ et de même $E_{k,l}E_{i,j} \prec E_{k,l}E_{i',j'}$.

Examinons le premier cas. Le fait que les deux produits soient non nuls nécessite $j = j' = k$. Comme $(i, j) \prec (i', j')$, c'est donc que $i < i'$ et donc $(i, l) \prec (i', l)$ soit $E_{i,j}E_{k,l} = E_{i,l} \prec E_{i',l} = E_{i',j'}E_{k,l}$.

Dans le second cas, le fait que les deux produits soient non nuls nécessite $i = i' = l$. Comme $(i, j) \prec (i', j')$, c'est donc que $j < j'$ et donc $(k, j) \prec (k, j')$ soit $E_{k,l}E_{i,j} = E_{k,j} \prec E_{k,j'} = E_{k,l}E_{i',j'}$.

□

Tout cela conduit à la fonction CAML suivante :

```
let rec prod_mat_elem ((i, j), a) = function
   | [] -> []
   | ((k, l), b)::B1 ->
         if (j = k)
         then let c = a *. b
              in if (c <> 0.0)
                 then ((i, l), c)::(prod_mat_elem ((i, j), a) B1)
                 else prod_mat_elem ((i, j), a) B1
         else prod_mat_elem ((i, j), a) B1 ;;
```

On peut alors écrire une fonction de produit de matrices creuses comme pour les polynômes creux. Le lemme précédent nous garantit encore une fois que l'ordre lexicographique sur les indices des termes non nuls est conservé.

```
let rec prod_mat = fun
   | _ [] -> []
   | [] _ -> []
   | (a1::r1) m2 -> add_mat (prod_mat_elem a1 m2) (prod_mat r1 m2) ;;
```

14 structure de pile (¶)

On donnera une implémentation des piles calquée sur celle que propose la librairie CAML standard : cette implémentation en vaut d'autres; elle utilise le type `'a list`.

14.1 les fonctions indispensables

- `type 'a stack`. C'est le type des piles contenant des éléments de type `'a`.
- `exception Empty_Stack`. Cette exception est déclenchée quand la fonction `pop` est appliquée à une pile vide.
- `new_stack : unit -> 'a stack`. L'appel `new_stack()` renvoie une nouvelle pile, initialement vide.
- `push : 'a stack -> 'a -> unit`. L'appel `(push s x)` ajoute l'élément `x` au sommet de la pile `s`; `()` de type `unit` est renvoyée : cette fonction opère par effet de bord.
- `pop : 'a stack -> 'a`. L'appel `(pop s)` renvoie l'élément au sommet de la pile `s`; cet élément est enlevé de la pile `s` (effet de bord); si la pile `s` est vide, `(pop s)` déclenche l'exception `Empty_Stack`.

14.2 le "code" de ces fonctions

On définit le type des piles d'objets de type `'a`. On a décidé d'être anglophone à cause des incontournables fonctions `push` et `pop`. Ainsi, '`Content`' désigne-t-il le contenu d'une pile :

```
type 'a stack = { mutable Content : 'a list } ;;
```

L'exception pile-vide :

```
exception Empty_Stack ;;
```

Le test "la pile est-elle vide ?" :

```
let is_empty_stack s =
   match s.Content with
      | [] -> true
      | _  -> false ;;
```

Fabrique une nouvelle pile (indispensable !) :

```
let new_stack () = { Content = [] } ;;
```

Les fonctions qui sont l'essence des piles : `push` et `pop`.

```
let push s x =
   s.Content <- (x::s.Content) ;;

let pop s =
   match s.Content with
      |  []  -> raise Empty_Stack
      | h::t -> begin
                  s.Content <- t;
                  h
                end ;;
```

On doit programmer sur les piles uniquement avec les fonctions données plus haut : c'est la raison d'être du "type abstrait" pile.

La fonction suivante écrase une pile `s` :

```
let clear_stack s =
   while not(is_empty_stack s) do (pop s) done ;;

clear_stack : 'a stack -> unit
```

Mais, si l'on est en train d'écrire une librairie sur les piles, on peut profiter du détail de l'implémentation :

```
let clear_stack s = s.Content <- [] ;;
```

Mêmes remarques en ce qui concerne la fonction `scan_stack` qui consulte non destructivement le sommet de la pile `s` :

```
let scan_stack s =
   let sommet = pop s
   in (push s sommet; sommet) ;;

scan_stack : 'a stack -> 'a

let scan_stack s = hd(s.Content) ;;
```

15 manipulation de termes

Les fonctions qui vont suivre font appel aux deux fonctions CAML suivantes

```
let append x y = x @ y ;;
let compose f g x = f (g x) ;;
```

15.1 déclarations de type (¶)

Les termes que nous avons en vue de traiter sont des termes du genre $F(x, G(x, y), F(w, z, t))$: y apparaissent des opérateurs (ici F et G) et des variables (ici x, y, w, z, t). Les opérateurs ont des *arités* : l'arité d'un opérateur est simplement le nombre d'arguments qu'il attend (ici F est d'arité 3 et G est d'arité 2). Les constantes (distinguées des variables) seront conçues commme des opérateurs d'arité 0.

Nous excluons totalement de notre étude des termes où apparaît la distinction entre variables libres et variables liées, comme $x \mapsto G(x, y)$ (c'est vraiment trop difficile pour le moment).

Informatiquement, les variables seront de type `'a` et les opérateurs de types `'b`; ceci nous laisse la liberté de représenter, par exemple, les variables par des entiers et les opérateurs par des chaînes de caractères.

Cela nous conduit aux déclarations suivantes :

```
type ('a,'b) terme =
   | Variable of 'a
   | Terme of 'b * (('a,'b)terme list)

and 'b graduation == ('b * int) list

and ('a,'b) substitution == ('a * ('a, 'b) terme) list

and ('a,'c) valuation == ('a * 'c) list
```

```
and ('b,'c) sémantique == ('b * ('c list -> 'c)) list

and ('a, 'b) atome =
   | Var of 'a
   | Op of 'b

and ('a, 'b) expression == ('a, 'b) atome list ;;
```

Les deux derniers types (`atome` et `expression`) nous servirons pour construire et manipuler les expressions postfixées correspondant aux termes.

15.2 fonctions d'accès (¶)

```
let graduation_op =
   fun graduation opérateur -> assoc opérateur graduation ;;

let substitution_var s v =
   match v
   with (Variable n) -> try (assoc n s) with _ -> v ;;

let valuation_var v x = (assoc x v) ;;

let sémantique_op s op = (assoc op s) ;;
```

15.3 ce terme est-il valide ? (¶)

```
let (légitimer_avec : 'b graduation -> ('a, 'b) terme -> bool) =
   function graduation ->
      let rec légitimer = function
         | Variable a -> true
         | Terme(op,l) -> ((graduation_op graduation op) = list_length l)
                          &&
                          (for_all légitimer l)
      in légitimer ;;
```

15.4 substitutions

```
let (appliquer_substitution : ('a,'b) substitution
                              -> ('a, 'b) terme
                              -> ('a, 'b) terme) =
   function s ->
      let rec app_subst = function
         | Terme(op,l) -> Terme(op, map app_subst l)
         | v -> substitution_var s v
      in app_subst ;;
```

15.5 évaluation (¶)

```
let (évaluer_terme : ('b,'c) sémantique
                   -> ('a,'c) valuation
                   -> ('a,'b) terme
                   -> 'c) =
   fun s v ->
```

```
      let rec val = function
         | Variable n -> valuation_var v n
         | Terme(op, l) -> (sémantique_op s op) (map val l)
      in val ;;
```

15.6 une fonctionnelle générale d'itération sur les termes

```
let traverse =
   fun f g v e ->
      let rec trav = function
         | Variable n -> v n
         | Terme(opérateur, opérandes) -> f opérateur
                                            (list_it (compose g trav)
                                                     opérandes
                                                     e)
      in trav ;;
```

15.7 des termes aux expressions postfixées (¶)

```
let (terme_vers_expression : ('a, 'b) terme -> ('a, 'b) expression) =
   let v n = [Var n]
   and f n x = append x [Op n]
   and g = append
   and e = []
   in traverse f g v e ;;
```

15.8 évaluation des expressions postfixées à l'aide d'une pile (¶)

```
let (popn : 'a stack -> int -> 'a list) =
   fun p n ->
      let rec aux = fun
         | 0 r -> r
         | k r -> aux (k - 1) ((pop p)::r)
      in aux n [] ;;

let (évaluer_expression : ('b,'c) sémantique    (* ce sera S *)
                          -> 'b graduation      (* ce sera G *)
                          -> ('a,'c) valuation  (* ce sera V *)
                          -> ('a,'b) expression (* ce sera E *)
                          -> 'c                 (* le type du résultat *)
                          ) =
   let la_pile = new_stack()
   in fun S G V E ->
   let rec évaluer = function
      | []         -> ()
      | (Var n)::t -> (push la_pile (valuation_var V n) ; évaluer t)
      | (Op n)::t  -> let a = graduation_op G n
                      in let f = sémantique_op S n
                      in let arg = popn la_pile a
                      in (push la_pile (f arg) ; évaluer t)
   in begin
        évaluer E;
        pop la_pile;
      end ;;
```

16 analyse lexicale et analyse syntaxique : une introduction

Nous avons pour but, dans la prochaine section, de décrire en CAML un mini-langage de programmation assez proche de CAML lui-même. Ce langage aura une syntaxe plus simple que celle de CAML; il faudra quand même en faire l'analyse lexicale et syntaxique avant d'en évaluer les expressions.

L'objet de cette section est de vous montrer ce que sont analyse lexicale et analyse syntaxique ainsi que de vous présenter des objets CAML (les "streams") utiles dans ce contexte.

En anglais, analyseur lexical se dit "lexical analyzer" ou "lexer" ou "scanner"; on utilise souvent en français les néologismes "lexeur" ou "scanneur". En anglais, analyseur syntaxique se dit le plus souvent "parser"; on utilise souvent en français le néologisme "parseur". Ne vous laissez pas impressionner par ce jargon, on verra tout à l'heure ce qu'il recouvre.

Pour écrire cette section, je me suis inspiré du cours de Michel Mauny ([1], chapitre 10) et de la lettre de CAML ([7], numéro 0). Vous y trouverez des exemples moins triviaux qu'ici, d'autres renseignements et d'autres renvois bibliographiques. Je vous engage aussi à regarder l'excellent [6], qui développe un autre point de vue. Enfin, les "incontournables" sur le sujet de l'analyse lexicale/syntaxique sont les livres de Haho, Hopcroft et Ullman chez Addison-Wesley.

16.1 les flots caml

Oublions temporairement notre mini-langage pour nous concentrer sur un type d'objet CAML non encore abordé : les "streams", ce que l'on traduira par "flots" (ou "flux", si vous préférez). Les streams sont agréables à utiliser pour l'écriture des analyseurs lexicaux et des analyseurs syntaxiques.

16.1.1 le type "stream" des flots caml

Les streams appartiennent à un type CAML prédéfini dont la représentation reste cachée à l'utilisateur (à moins qu'il aille voir dans les sources). Bien sûr et heureusement, il est possible de construire des streams "à la main" ou en utilisant quelques fonctions prédéfinies.

Le type stream est un type paramétré : on peut construire et manipuler des streams d'entiers, de caractères, etc. La syntaxe des streams ressemble un peu à celle des listes. Voici le flot vide :

```
[< >] ;;
- : '_a stream = <abstr>
```

Les flots non vides ont des éléments qui sont écrits précédés d'un quote "'" :

```
[< '1 ; '2 ; '3 >] ;;
- : int stream = <abstr>

[< '1 ; 2 ; '3 >] ;;
Toplevel input:
>[< '1 ; 2 ; '3 >] ;;
>        ^
This expression has type int,
but is used with type int stream.
```

Les éléments qui ne sont pas précédés par un quote sont compris comme des sous-flots (substreams) qui sont destinés à être expansés et concaténés au flot englobant :

```
[< '0 ; [< '1 ; '2 >] ; '3 >] ;;
- : int stream = <abstr>
```

De manière plus générale, `[< 'x ; f ; 'y >]` représente un flot constitué d'un premier élément `x`, d'un sous-flot `f`, et d'un élément `y`. Si le flot est de type `'a stream`, alors `x` et `y` sont du type `'a`, et, bien sûr, `f` du type `'a stream`. L'erreur la plus fréquente est sans doute d'oublier le quote devant les éléments du flot.

Signalons enfin que, lorsque l'on utilise les flots en analyse lexicale ou syntaxique, ils sont le plus souvent créés à l'aide des deux fonctions `stream_of_string`, qui fabrique un flot à partir d'une chaîne de caractères, et `stream_of_channel`, qui associe un flot à un canal d'entrée.

```
stream_of_string  : string -> char stream
stream_of_channel : in_channel -> char stream
```

16.1.2 les flots sont paresseux

Les flots sont soumis à "l'évaluation paresseuse" (lazy evaluation) : ils ne sont construits que progressivement et lorsque nécessaire (pour une consultation de leurs éléments par exemple). Cela est très utile quand un flot est associé à un canal d'entrée comme le clavier. En effet, si les flots n'étaient pas paresseux, un programme évaluant '`stream_of_channel std_in`' (`std_in` pour standard input, c'est en général le clavier, mais pas toujours) devrait lire tout ce qui se présente à l'entrée standard jusqu'à lecture du caractère `end_of_file` (fin de fichier) avant de passer la main au reste du programme : toute utilisation interactive des entrées-sorties serait alors impossible.

De plus, et c'est important et nouveau, l'évaluation paresseuse des flots permet de manipuler des flots infinies. Voici, par exemple, le flot des entiers naturels :

```
let les_entiers =
   let rec les_entiers_à_partir_de n =
      [< 'n ; les_entiers_à_partir_de (n + 1) >]
   in les_entiers_à_partir_de 0 ;;

les_entiers : int stream = <abstr>
```

Les flots se comportent donc un peu comme des listes infinies; on va voir, toutefois, que les flots ne sont pas vraiment assimilables à des listes, mêmes infinies.

16.1.3 regarder un flot, c'est le tuer (presque)

Les flots n'ont pas finis de vous surprendre (et peut-être de vous dérouter) car l'accès à un flot est destructif : si on lit le premier élément d'un flot, on le retire du flot. Les flots sont donc des structures de données mutables.

Voici une fonction qui renvoie la tête d'un flot et un exemple d'utilisation sur le flot infini des entiers (précédemment défini) :

```
let next = function
   | [< 'x >] -> x
   | [< >]    -> raise (Failure "flot vide") ;;

next : 'a stream -> 'a

next les_entiers ;;
- : int = 0
next les_entiers ;;
- : int = 1
next les_entiers ;;
- : int = 2
```

Un motif de filtrage comme `[< 'x >]` filtre un segment initial du flot; un tel motif peut se lire “le flot dont le premier élément est filtré par `x`”. On constate de plus que, si le filtrage réussit, le flot est physiquement modifié pour ne plus contenir ce qui a été filtré.

Signalons que le motif `[< >]` filtre n'importe quel flot non destructivement, car tout flot commence par du vide; ce n'est (dans la fonction `next`) qu'intervenant après le motif `[< 'x >]` que le motif `[< >]` ne filtre que le flot vide. Là encore, les flots diffère des listes.

Il n'y a pas, dans le filtrage d'un flot, de notion de 'tail', comme pour les listes. Si le filtrage habituel sur les listes s'écrit

```
match liste with
    | h::t -> ...
    |  []  -> ...
```

le filtrage habituel sur les flots est plutôt du genre

```
match flot with
    | [< 'x >] -> ...
    | [< 'y >] -> ...
    | [< >]    -> ...
```

Lors d'une définition récursive, au lieu d'appliquer récursivement la fonction sur le tail `t` *comme pour les listes, on l'appliquera au flot lui-même, puisque le filtrage en aura supprimé la tête.*

Voici, par exemple, une fonction qui transforme un flot (non infini) en liste.

```
let rec list_of_stream s =
   match s with
      | [< 'h >] -> h :: (list_of_stream s)
      | [< >]    -> [] ;;

list_of_stream : 'a stream -> 'a list

list_of_stream [< '1 ; '2 ; '3 >] ;;
- : int list = [1; 2; 3]
```

Étant destructif, le filtrage se passe donc de façon tout à fait particulière. Le plus surprenant est sans doute que le filtrage ne choisit le motif filtrant qu'à l'aide du seul premier élément de chaque motif. Ainsi, par exemple, CAML signalera que, dans le cas suivant, le deuxième motif n'a aucune chance d'être utilisé :

```
let bidon flot = match flot with
    | [< '1 ; '2 ; s >] -> s
    | [< '1 ; '3 ; s >] -> s
    | [< '2 ; s >] -> s
    | [< >] -> failwith "non prévu" ;;

Toplevel input:
>    | [< '1 ; '3 ; s >] -> s
>           ^
Warning: this matching case is unused.

bidon : int stream -> int stream
```

Remarquons que le premier cas de filtrage aurait pu tout aussi bien s'écrire `[< '1 ; '2 >] -> flot` puisque les deux premiers éléments auront alors été détruits du flot. Le dernier cas du filtrage filtre un flot vide, mais aussi n'importe quel flot ne commençant ni par `[< '1 ; '2 >]`, ni par `[< '2 >]`; en effet, tout flot commence par du vide.

Dans le cas où le filtrage est aiguillé sur un des motifs, il ne peut changer d'avis ; si la suite du motif ne passe pas, on déclenche l'exception `Parse_error`, et ainsi on obtiendra en poursuivant l'exemple précédent :

```
bidon [< '1 ; '4 >] ;;
Uncaught exception: Parse_error

bidon [< '1 ; '3 >] ;;
Uncaught exception: Parse_error
```

En revanche, si aucun (début de) motif ne filtre le (début du) flot, c'est l'exception `Parse_failure` qui est déclenchée. Ainsi :

```
let autre_bidon flot = match flot with
   | [< '1 >] -> "oui"
   | [< '2 >] -> "non" ;;

autre_bidon : int stream -> string

autre_bidon [< '2 >] ;;
- : string = "non"

autre_bidon [< '5 >] ;;
Uncaught exception: Parse_failure
```

16.1.4 motifs de filtrage "fonctionnels"

Les filtres pour les flots ne sont pas restreints à n'utiliser que des motifs quotés; il y a beaucoup plus rusé et on va voir tout de suite que cette nouvelle façon de filtrer un flot est précieuse.

Voilà de quoi il s'agit. En ce qui concerne les flots, une fonction peut apparaître dans un motif de filtrage; cette fonction sera appelée sur le flot argument lui-même et le résultat de cet appel sera filtré par un motif qui suit le nom de la fonction. De cette manière, si `f` est une fonction du type `'a stream -> 'b`, et si `s` est du type `'a stream`, le motif `[< f x >]` filtre tout flot de type `'a stream`, et crée une nouvelle liaison, qui à la variable `x` lie la valeur (de type `'b`) de l'appel de la fonction `f` sur le flot `s`. Un exemple rendra, sans nul doute, les choses plus claires. Voici une fonction comptant le nombre de `1` apparaissant dans un flot (non infini) d'entiers.

```
let rec nombre_de_un = function
   | [< '1 ; nombre_de_un x >] -> 1 + x
   | [< '_ ; nombre_de_un x >] -> x
   | [< >] -> 0 ;;

nombre_de_un : int stream -> int

nombre_de_un [< '1 ; '2 ; '1 ; '2 ; '1 ; '2 >] ;;
- : int = 3
```

16.1.5 construction du flot infini des nombres premiers par le crible d'Ératosthène

Pour finir cette présentation, voici un bel exemple qui utilise la notion d'évaluation paresseuse pour manipuler des flots infinis. À quelques variantes près, cet exemple est dû à Laurent Chéno, dans *la lettre de* CAML*, numéro 0.* Le début est classique :

```
exception Flot_Vide ;;

let nouvel_élément_du_flot = function
```

```
   | [< 'x >] -> x
   | [< >]    -> raise Flot_Vide ;;

let rec les_entiers_à_partir_de n = [< 'n ; les_entiers_à_partir_de (n + 1) >] ;;

nouvel_élément_du_flot  : 'a stream -> 'a
les_entiers_à_partir_de : int -> int stream
```

Voici une fonction qui ne laisse subsister du flot `s` que les éléments qui satisfont au prédicat `f`.

```
let rec filtre_flot f s = match s with
   | [< 'x >] -> if (f x)
                 then [< 'x ; filtre_flot f s >]
                 else [< filtre_flot f s >]
   | [< >]    -> [< >] ;;

filtre_flot : ('a -> bool) -> 'a stream -> 'a stream
```

La construction du crible, modulo l'emploi d'un prédicat de divisibilité, est alors facile :

```
let non_divisible_par a x = (x mod a) <> 0 ;; (* a ne divise pas x *)

let rec crible_d'Ératosthène s = match s with
   | [< 'n >] -> [< 'n ;
                    crible_d'Ératosthène (filtre_flot (non_divisible_par n) s) >]
   | [< >]    -> [< >] ;;

non_divisible_par    : int -> int -> bool
crible_d'Ératosthène : int stream -> int stream
```

Et voici une fonction qui renvoie, à chaque appel, le nouvel élément de la suite des nombres premiers :

```
let nouveau_premier =
   let e = les_entiers_à_partir_de 2
   in let p = crible_d'Ératosthène e
   in function () -> nouvel_élément_du_flot p ;;

nouveau_premier : unit -> int
```

Testons.

```
nouveau_premier ();;
- : int = 2
nouveau_premier ();;
- : int = 3
nouveau_premier ();;
- : int = 5
nouveau_premier ();;
- : int = 7
nouveau_premier ();;
- : int = 11
nouveau_premier ();;
- : int = 13

...etc...

nouveau_premier ();;
- : int = 151
```

Terminons cet exemple en signalant que la fonction suivante renvoie toujours le premier nombre premier (qui est 2).

```
let pas_nouveau () =
   nouvel_élément_du_flot (crible_d'Ératosthène (les_entiers_à_partir_de 2)) ;;
```

Comprenez-vous pourquoi ?

16.2 analyse lexicale et syntaxique

16.2.1 mise en place formelle

Qu'est-ce qu'un langage ? Pour pouvoir parler de langage, on se donne tout d'abord un alphabet A qui est un ensemble fini non vide de "caractères". Un langage est alors une partie (finie ou non) de l'ensemble A^* des mots finis sur cet alphabet. Un langage est donc un élément de 2^{A^*}, ensemble des parties de A^*.

Remarquons tout de suite que, l'alphabet A étant un ensemble fini non vide, l'ensemble A^* est lui dénombrable (il a la cardinalité de $\mathbb{N}$); quant à l'ensemble de tous les langages sur A, c'est à dire 2^{A^*}, il est donc définitivement non-dénombrable (sa cardinalité, strictement supérieure à celle de $\mathbb{N}$, est celle de $\mathbb{R}$).

Qu'est-ce, maintenant, qu'une grammaire ? Disons honnêtement que la définition mathématique d'une grammaire existe (au même titre que celle des espaces vectoriels — tout le monde s'entendant à peu près sur la définition), mais que nous ne donnerons pas ici cette définition. On peut au moins dire qu'une grammaire est une chaîne (finie) de caractères (issus d'un autre alphabet fini B, mais on a souvent $A \cap B \neq \emptyset$) qui décrit (d'une certaine manière) un certain langage... Par exemple, les expressions régulières sont un certain type de grammaire; mais il y a d'autres types de grammaires. Donons deux exemples d'expressions régulières. L'expression régulière `(a|b)*` décrit le langage formé des mots constitués d'une suite fini éventuellement vide (c'est le sens du `*`) de `a` ou (c'est le sens du `|`) de `b`. L'expression régulière `(a|b|c)*(x|y)` décrit le langage formé des mots constitués d'une suite fini éventuellement vide (`*`) de `a` ou (`|`) de `b` ou (`|`) de `c` et se terminant par un `x` ou (`|`) par un `y`. Les expressions régulières sont au programme de la Spé. ainsi que leur compilation sous forme d'automates finis.

Notons au passage que, concrètement, une grammaire se doit d'obéir elle-même à une autre grammaire; sans cela, elle court grand risque de ne pas être comprise. Eh oui...

Fixons un alphabet $A = B$ suffisamment vaste. La cardinalité de l'ensemble de toutes les grammaires possibles est dénombrable (une grammaire est un élément de $A^* = B^*$). La cardinalité de l'ensemble de tous les langages possibles est strictement supérieure au dénombrable. On en conclut qu'il y a des langages pour lesquels aucune grammaire ne saurait exister; ce sont des langages indescriptibles (ou indicibles, voire ineffables); et il y en a beaucoup; et c'est même la majorité (car $\mathbb{R}$ privé de $\mathbb{N}$ a même cardinalité que $\mathbb{R}$).

Dans la série des remarques de base, ou presque, disons aussi qu'une grammaire donnée ne décrit qu'un langage (c'est le but recherché !) mais qu'à un langage donné peut correspondre plusieurs grammaires, c'est à dire plusieurs manières de le décrire. En restant dans le cadre des expressions régulières, les deux grammaires `a*a` et `aa*` décrivent le même langage (formé des suites non vides de `a`).

Que faire d'une grammaire (à part lui souhaiter sa fête) ? L'intérêt minimal d'une grammaire `G` décrivant un langage L n'existe que si l'on sait écrire une fonction `reconnaître` qui, étant donnés `G` et une chaîne `s` de caractères (i.e. un mot) quelconque sur A, est capable (en un temps fini) de dire si oui ou non le mot `s` est un élément du langage L.

```
reconnaître : grammaire -> string -> bool
```

En informatique, on verra que l'on demande nettement plus à `reconnaître` que d'éternuer un simple booléen.

Quels sont les types fréquents de grammaires que l'on rencontre dans la Nature (i.e. dans la littérature informatique/mathématique) ? Le type le plus simple de grammaire consiste en l'énumération des mots du langage; mais le langage doit alors être fini. Viennent ensuite les expressions régulières (introduites par Kleene), puis les grammaires context-free (introduites par Chomsky), puis les grammaires context-sensitive; à chaque fois la classe des langages descriptibles augmente strictement. Seules les expressions régulières sont au programme de la Spé.

On arrêtera là cette mise en place formelle; le lecteur doit être convaincu qu'il ne s'agit que d'une mise en place; tenter d'épuiser le sujet couvrirait plusieurs volumes comme celui-ci.

16.2.2 distinction du niveau lexical et du niveau syntaxique

Considérons une phrase (un élément) d'un langage réel (le français, en l'occurence) : "le chat boit du lait; ensuite, il fera sa toilette.". On a intérêt, si l'on veut manipuler les choses commodément, à distinguer deux niveaux.

Le niveau (1) est le niveau lexical de la phrase. Cette phrase est constituée du caractère 'l', puis de 'e', puis de ' ', etc. À faire cette remarque, on constate qu'on a intérêt à distinguer plusieurs sortes de caractères : les caractères "normaux" comme 'a'; les séparateurs de mots comme le blanc ' '; les caractères de ponctuations comme la virgule, le point-virgule et le point. Au terme de l'analyse lexicale on devra avoir reconnu les mots (éventuellement classés par genre : verbe, article, etc.) et les signes de ponctuation.

Le niveau (2) est le niveau syntaxique de la phrase, en général plus difficile à décrire. Notre phrase exemple est composée de deux phrases séparées par une ponctuation (le point-virgule); la sous-phrase 1 est "le chat boit du lait"; elle se compose d'un groupe nominal ("le chat"), d'un verbe ("boit"), d'un groupe nominal complément d'objet direct du verbe ("du lait").

Pour des raisons assez évidentes, on ne va pas persévérer dans la description d'un langage comme le français; on va s'interesser plutôt aux langages informatiques.

Les programmes (informatiques) ne sont que des chaînes de caractères. La phase d'analyse lexicale consistera à diviser cette chaîne en "identificateurs", "nombres entiers", "commentaires", "mots-clefs", etc. Ces unités lexicales sont appelées tokens en anglais et lexèmes en français (mais les anglo-saxons utilisent parfois eux aussi "lexemes"). Voici quelques exemples de spécifications lexicales informelles :

Toute séquence de blancs, tabulations, sauts de ligne est équivalente à un seul blanc.

Un commentaire commence par '`(*`' *et se termine par* '`*)`'.

Un identificateur est une séquence de lettres et de digits commençant par une lettre; une variable est un identificateur qui n'est pas un mot-clef.

La description du niveau lexical ne requiert en géneral que des grammaires assez frustes : pour les langages informatiques, les expressions régulières suffisent amplement.

La phase d'analyse syntaxique consiste ensuite à organiser la séquence des lexèmes en structures hiérarchiques comme "expressions arithmétiques", "conditionnelle if-then-else", "définition", etc.

16.2.3 analyse lexicale et syntaxique d'un langage simple

Nous allons essayer de manipuler informatiquement un langage simple (sous-langage d'un langage informatique réel) qui sera celui (noté EAR) des Expressions ARithmétiques (entières). Ce langage est construit sur l'alphabet formé des 17 caractères suivants : `0`, `1`, ..., `9`, `+`, `-`, `*`, `/`, `)` et `(` ainsi que le blanc.

Donnons tout de suite, et sans formalisme, une idée de ce que nous voulons. Par exemple, "`((3 * 7) - 3)`" sera un élément de EAR, tandis que "`88))**`" ne sera pas un élément de EAR. La chaîne "`(3 * 7) - 3`" ne sera pas, non plus, un élément de EAR car on a omis la paire de parenthèses englobante. Dans EAR, les 4 "opérateurs" (`+`, `-`, `*`, `/`) sont tous binaires et toujours parenthésés; les blancs ne sont pas significatifs; le langage ne concerne que des constantes entières.

Décrivons informellement et rapidement les étapes d'analyse lexicale et syntaxique du langage EAR; nous verrons ensuite comment les réaliser.

Notre point de départ sera une chaîne de caractère CAML sensée être un élément de EAR. C'est le cas, par exemple, de cette chaîne (plutôt laide) :

```
let exemple_1 =  " (( 21 +  12) *8 )  " ;;
```

Nous transformeront cette chaîne en flot à l'aide de `stream_of_string` pour obtenir un flot de caractère (`char stream`). Voici la liste associée à ce flot :

```
list_of_stream (stream_of_string exemple_1) ;;
- : char list =
 [` `; `(`; `(`; ` `; `2`; `1`; ` `; `+`; ` `; ` `; `1`; `2`; `)`; ` `; `*`;
  `8`; ` `; `)`; ` `; ` `]
```

Il est clair qu'il y a beaucoup de "bruit de fond", provenant des blancs inutiles et mal placés.

Partant du flot des caractères (`char stream`), l'analyse lexicale doit produire le flot des lexèmes (tokens), c'est à dire des unités lexicales du langage. Pour le langage EAR les lexèmes seront de seulement de deux types : les entiers et les mots-clefs (keywords). Les mots-clefs de EAR seront `+`, `-`, `*`, `/`, `)` et `(`. Les blancs seront supprimés, car non-significatifs. Voici la liste associée à ce flot de lexèmes (`token stream`); c'est déjà plus clair :

```
[Kwd "("; Kwd "("; Int 21; Kwd "+"; Int 12; Kwd ")"; Kwd "*"; Int 8; Kwd ")"]
```

Puis, partant du flot des lexèmes (`token stream`), l'analyse syntaxique doit produire l'arbre correspondant à l'expression de départ. Au terme de l'analyse syntaxique on est donc passé de la syntaxe concrète (la chaîne de caractère) à la syntaxe abstraite (l'arbre que cette chaîne représente). Bien sûr, cet arbre doit être d'un type CAML qui correspond à la sémantique (le sens, par opposition à la syntaxe) du langage que l'on manipule.

Voici une grammaire (au niveau syntaxique, et non au niveau lexical) de nos expressions arithmétiques :

```
expr := entier
      | (expr + expr)
      | (expr - expr)
      | (expr * expr)
      | (expr / expr)
```

Et voici le type CAML des arbres que l'on associera aux expressions arithmétiques :

```
type terme =
   | Entier of int
   | Plus  of terme*terme (* pour le '+' *)
   | Minus of terme*terme (* pour le '-' *)
   | Star  of terme*terme (* pour le '*' *)
   | Slash of terme*terme (* pour le '/' *) ;;
```

Attention, ici le lien grammaire-type est direct, mais ce n'est pas toujours le cas.

Le résultat de l'analyse syntaxique du flot de lexèmes issu de l'analyse lexicale de `exemple_1` devra être du type `terme`. Voici le résultat, il est sans surprise :

```
Star (Plus (Entier 21, Entier 12), Entier 8)
```

Se pose ensuite le problème d'évaluer correctement et rapidement un tel arbre; on sait (pour ce genre d'expression) pouvoir le faire efficacement à l'aide d'une pile manipulant l'expression postfixée issue de l'arbre. Ce n'est pas (ici) notre propos. Pour nous, maintenant que l'on sait ce que vouloir, se pose le problème de réaliser effectivement l'analyse lexicale, puis l'analyse syntaxique de EAR.

16.2.4 analyse lexicale de EAR

Voici une partie (c'est la fonction `eat_integer`) d'un lexeur pour EAR. Cette fonction lit un flot constitué des caractères de `0` à `9` et renvoie l'entier CAML correspondant. Attention, cette fonction n'est qu'une partie du lexeur que l'on devrait écrire.

```
let digit_of_char c = (int_of_char c) - (int_of_char `0`) ;;

let rec eat_int accu flot = match flot with
   | [< '(`0`..`9`) as x >] -> eat_int (10*accu + (digit_of_char x)) flot
   | [< >] -> accu ;;

let eat_integer = eat_int 0 ;;

digit_of_char : char -> int
eat_int       : int -> char stream -> int
eat_integer   : char stream -> int

eat_integer [< '2 ; '7 ; '2 >] ;;
Toplevel input:
>eat_integer [< '2 ; '7 ; '2 >] ;;
>            ^^^^^^^^^^^^^^^^^^
This expression has type int stream,
but is used with type char stream.
(* oui, pardon..., je reprends : *)

eat_integer [< '`2` ; '`7` ; '`2` >] ;;
- : int = 272
```

Plutôt que d'écrire complètement un lexeur pour EAR (ce qui fort ennuyeux), je préfère vous présenter la librairie CAML `genlex` qui est capable *d'engendrer* très simplement des analyseurs lexicaux du type de celui utilisé pour CAML lui-même. Heureusement, la création du lexeur est paramétrable par la donnée des mots-clefs du langage que l'on manipule.

Cette librairie contient la définition d'un type `token`, définition qui est la suivante :

```
type token = Int of int
           | Float of float
           | Char of char
           | String of string
           | Ident of string
           | Kwd of string ;;
```

Les différents lexèmes sont donc :

- `Int` et `Float` pour les entiers et les flottants (écrits avec les conventions lexicales de CAML);
- `Char` pour les constantes du type caractère (entourés de backquotes, mêmes conventions lexicales que CAML);
- `String` pour les constantes du type chaîne de caractères (entourées de guillemets, mêmes conventions lexicales que CAML);
- `Ident` pour les identificateurs (mêmes conventions lexicales que CAML);
- `Kwd` pour les mots-clefs (keywords) introduits par l'utilisateur (voir ci-dessous).

On dispose de la fonction

```
make_lexer : string list -> (char stream -> token stream)
```

qui construit le lexeur du langage de l'utilisateur. Le premier argument (de type `string list`) est simplement la liste des mots-clefs. Lors de l'analyse lexicale, un identificateur sera retourné en tant que mot-clef `Kwd` si il fait partie de cette liste, et en tant qu'identificateur `Ident` s'il n'en fait pas partie. Tout simplement.

Résolus que nous sommes à utiliser la librairie `genlex`, nous sommes prêts à définir un lexeur pour EAR :

```
#open "genlex" ;; (* il faut taper le #, cette incantation ouvre la librairie *)

let lexer = make_lexer ["+" ; "-" ; "*" ; "/" ; ")" ; "("] ;;

let lex_string s = lexer (stream_of_string s) ;;

let list_of_lex_string s = list_of_stream (lexer (stream_of_string s)) ;;

lexer              : char stream -> token stream
lex_string         : string -> token stream
list_of_lex_string : string -> token list
```

Un exemple s'impose :

```
let exemple_1 =  " (( 21 +  12) *8 )  " ;;
exemple_1 : string = " (( 21 +  12) *8 )  "

list_of_lex_string exemple_1 ;;
- : token list =
 [Kwd "("; Kwd "("; Int 21; Kwd "+"; Int 12; Kwd ")"; Kwd "*"; Int 8; Kwd ")"]
```

Et voilà. Maintenant, il reste à faire l'analyse syntaxique. À propos, on ne dit pas *incantation* pour des phrases CAML du genre '`#open "genlex"`' mais plutôt *directive*; le diligent lecteur (contexte-sensitif et contexte-intentionnel) saura choisir.

16.2.5 analyse syntaxique de EAR

Voici le texte complet du parseur de EAR. C'est un jeu d'enfant, mais la syntaxe (vraiment très simple) de notre langage EAR y est pour beaucoup.

```
type terme =
   | Entier of int
   | Plus   of terme*terme
   | Minus  of terme*terme
   | Star   of terme*terme
   | Slash  of terme*terme ;;

let rec parse = function
   | [< '(Int n) >]
          -> Entier n
   | [< '(Kwd "(") ; parse x ; '(Kwd opérateur) ; parse y ; '(Kwd ")") >]
          -> match opérateur with
                | "+" -> Plus  (x, y)
                | "-" -> Minus (x, y)
                | "*" -> Star  (x, y)
                | "/" -> Slash (x, y) ;;
```

```
let parse_lex_string s = parse (lexer (stream_of_string s)) ;;

parse            : token stream -> terme
parse_lex_string : string -> terme
```

Un exemple ?

```
parse_lex_string exemple_1 ;;
- : terme = Star (Plus (Entier 21, Entier 12), Entier 8)
```

Un autre :

```
parse_lex_string "(((6 * 3) / 2) + 5)" ;;
- : terme = Plus (Slash (Star (Entier 6, Entier 3), Entier 2), Entier 5)
```

17 un mini-langage fonctionnel

Le but du jeu, dans cette section, est de donner le texte CAML de l'interprète d'un langage de programmation proche de CAML lui-même. Contrairement à ce que le lecteur pourrait penser de prime abord, il n'y a pas forcément pétition de principe à donner une description de CAML en CAML (les informaticiens parlent assez pompeusement d'une description "méta-circulaire"). En effet, si cette description est de suffisamment "bas niveau", elle pourrait être traduite facilement (par quelqu'un ayant l'habitude) dans un autre langage que CAML (langage C, assembleur, etc.).

Disons quand même que nous allons limiter nos ambitions : on ne décrira pas le langage CAML lui-même, mais un Mini-Langage fonctionnel (on l'appellera $\mathcal{ML}$! pas MLF) possédant les traits essentiels de CAML mais (1) la syntaxe de ce langage sera différente; (2) le nombre de constructions du langage sera nettement moindre qu'en CAML; (3) enfin, nous ne parlerons pas de synthèse de type.

En ce qui concerne le point (1) nous utiliserons une syntaxe plus simple que celle de CAML; outre l'agrément de la simplicité on pourra distinguer facilement le "langage-objet" ($\mathcal{ML}$) du "langage de l'observateur" (CAML).

En ce qui concerne le point (2) nous verrons que ce n'est pas un problème, en ce sens que tous les algorithmes exprimables en CAML pourraient être codés en $\mathcal{ML}$ (aux prix de "quelques" contorsions).

En ce qui concerne le point (3) la différence est essentielle : des algorithmes (ayant un sens) codables en $\mathcal{ML}$ ne le seront plus en CAML car refusés par son synthétiseur de type. (On en verra un exemple.)

17.1 présentation du mini-langage ($\mathcal{ML}$)

Notre petit $\mathcal{ML}$ va sembler très démuni; en effet, les termes (`type term`) du langage se limitent, pour l'utilisateur, à cinq sortes :

- Les variables.
- L'application d'un terme `a` (plutôt destiné à être une "fonction") à un autre terme ("argument" du premier) `b`. Ceci est noté `(! a b)`.
- "L'abstraction". Le terme est classique dans ce contexte; il s'agit "d'abstraire" un terme `t` par rapport à une variable `v` : en d'autres termes, on considère la fonction qui à `v` associe `t` (pouvant ou non contenir `v`). Ceci est noté `(? v t)`. L'analogue CAML serait '`function v -> t`'.
- La conditionnelle `if` dont la syntaxe est `(if t a b)`; l'analogue CAML serait '`if t then a else b`'. On verra que l'on pourrait (presque) se passer de la conditionnelle.
- Enfin, merveille, on peut enrichir $\mathcal{ML}$ de définitions; elles sont introduites par `(= v t)` où `v` est une variable et `t` un terme. C'est l'analogue de '`let v = t ;;`'.

On voit que l'on ne dispose ni des booléens, ni des couples, ni même des entiers : toutes ces notions sont définissables, et seront définies, en $\mathcal{ML}$; il en est de même de la récusivité.

17.2 les types caml correspondants aux termes $\mathcal{ML}$

Voici donc la "syntaxe abstraite" de $\mathcal{ML}$:

```
type term =
   | Variable of string
   | Fonction of string * term
   | Application of term * term
   | Definition of string * term
   | Conditionnal of term * term * term
   | Closure of string * term * environnement
and environnement == (string * term) list ;;
```

Le constructeur `Closure` n'est pas accessible à l'utilisateur (aucune syntaxe concrète ne correspond à cette clause de la syntaxe abstraite); il en est de même pour le type `environnement`. Ces constructions seront manipulées de manière interne par l'évaluateur $\mathcal{ML}$.

17.3 analyse lexicale de $\mathcal{ML}$

Il n'y a ni difficultés ni surprises car nous utiliserons la librairie `genlex`. Outre les parenthès ouvrantes et fermantes, nos mots-clefs se limitent à '`=`' (qui introduit les définitions); '`?`' (qui introduit les abstractions (les fonctions)); '`!`' (qui introduit les applications (d'un terme à un autres)); '`if`' (qui introduit les formes conditionnelles if-then-else).

```
#open "genlex" ;;

let lexer = make_lexer ["=" ; "?" ; "!" ; "if" ; ")" ; "("] ;;

let lex_string s = list_of_stream (lexer (stream_of_string s)) ;;

lexer      : char stream -> token stream
lex_string : string -> token list
```

Un essai rapide :

```
lex_string "(? x x)" ;;
- : token list = [Kwd "("; Kwd "?"; Ident "x"; Ident "x"; Kwd ")"]
```

17.4 analyse syntaxique de $\mathcal{ML}$

Ni difficultés ni surprises là non plus : on utilise la technique des flots CAML éprouvées quand on parlait de l'analyseur syntaxique pour "EAR". On est, bien sûr, grandement aidé par la simplicité de la syntaxe choisie pour $\mathcal{ML}$: toute les termes sont parenthésés et le mot-clef de tête annonce tout de suite la couleur. Le langage SCHEME utilise le même genre de syntaxe.

```
let rec parser flux = match flux with
   | [< '(Ident s) >]
      -> (Variable s)
   | [< '(Kwd "(") ; '(Kwd k) ; sous_flux >]
      -> match k with
           | "!"  -> (match sous_flux with
                       [< parser a1 ; parser a2 ; '(Kwd ")") >]
                       -> Application(a1, a2))
           | "?"  -> (match sous_flux with
                       [< '(Ident s) ; parser t ; '(Kwd ")") >]
                       -> Fonction(s, t))
```

```
          | "="  -> (match sous_flux with
                      [< '(Ident s) ; parser t ; '(Kwd ")") >]
                      -> Definition(s, t))
          | "if" -> (match sous_flux with
                      [< parser t ; parser a ; parser b ; '(Kwd ")") >]
                      -> Conditionnal(t, a, b))
          | _    -> failwith "parser" ;;

let parse_lex_string s = parser (lexer (stream_of_string s)) ;;

parser           : token stream -> term
parse_lex_string : string -> term
```

Un exemple; on écrira ainsi la définition $\mathcal{ML}$ de la fonction identité :

```
parse_lex_string "(= Id (? x x))" ;;
- : term = Definition ("Id", Fonction ("x", Variable "x"))
```

17.5 l'interprète de $\mathcal{ML}$

Avant d'aborder l'évaluation de $\mathcal{ML}$, notons qu'il est dépourvu de la forme 'let-in' qui permet l'introduction de liaisons locales; ce n'est pas grave car, en CAML, toute expression du genre

```
let var = val in expression
```

est remplaçable par

```
(function var -> expression) val
```

Notons également que notre langage autorise des définitions à n'importe quel niveau; c'est comme si il était possible d'écrire en CAML

```
let a = 13 in let f x = a * x ;;
```

Ce qui s'écrit plutôt (en CAML)

```
let f =
   let a = 13
   in (function x -> a * x) ;;
```

Toute évaluation $\mathcal{ML}$ s'effectue dans un environnement et les définitions $\mathcal{ML}$ seront engrangées dans l'environnement global; la fonction `reset` permet (sous CAML) de remettre cet environnement "à zéro".

On aura compris qu'un environnement est une liste de liaisons, c'est à dire une liste de couples variable-valeur ou plus précisément ici une liste de couples du type `string * term`.

La fonction `find` permet de retrouver, dans un environnement, la valeur d'une variable. La fonction `extend` permet d'adjoindre à un environnement une nouvelle liaison, c'est à dire un nouveau couple variable-valeur.

```
let (find : environnement -> string -> term) =
   fun env var -> try assoc var env
                  with Not_found -> (Variable var) ;;

let (extend : environnement -> string -> term -> environnement) =
   fun env var val -> (var, val) :: env ;;

let (global_environnement : environnement ref) = ref [] ;;

let reset () = global_environnement := [] ;;
```

On notera que, si une variable n'a pas de valeur (ou plutôt, n'a été l'objet d'aucune liaison dans un environnement donné), sa valeur est elle-même. Il est important de voir que cette convention est la seule acceptable dans un langage qui est dépourvu de tout type de base (booléen, entiers, etc.).

Voici enfin la fonction `eval` qui évalue les expressions $\mathcal{ML}$:

```
let rec (eval : environnement -> term -> term) = function env -> function
   | Variable(v)
      -> find env v
   | Fonction(v, t)
      -> Closure(v, t, env)
   | Closure(_, _, _) as e
      -> e
   | Application(a1, a2)
      -> let a'1 = eval env a1
         and a'2 = eval env a2
         in begin
               match a'1 with
                  | Closure(v, t, env') -> eval (extend env' v a'2) t
                  | _                   -> Application(a'1, a'2)
            end
   | Definition(v, t)
      -> let t' = eval env t
         in begin
               global_environnement := extend !global_environnement v t' ;
               t' ;
            end
   | Conditionnal(t, a, b)
      -> let t' = eval env t
         in match t' with
               | (Variable "true")  -> eval env a
               | (Variable "false") -> eval env b
               | _                  -> Conditionnal(t', a, b) ;;

eval : environnement -> term -> term
```

On notera la structure récursive de `eval` et le passage de l'environnement en argument. Quelques commentaires :

- Pas de surprise en ce qui concerne les variables.
- Si l'on doit évaluer une forme du genre `(? x (! (! f x) a))` dans un environnement `env` donné, il s'agit de garder mémoire de `env` qui donne une valeur aux variables libres de l'expression fonctionnelle : ici les variables libres sont `f` et `a`. Donc, pour une expression `(? x E)`, on construira une "clôture" constituée du triplet `x`, `E` et `env`. Ce mécanisme correspond à une compilation (*très* minimaliste) des formes fonctionnelles.
- Bien sûr, la valeur d'une clôture est cette clôture elle-même.
- Nous allons exploiter les clôtures pour évaluer les applications (d'un terme à un autre). Soit donc à évaluer (dans `env`) une expression `(! a1 a2)`. On commence par évaluer dans `env` les termes `a1` et `a2` pour obtenir respectivement `a'1` et `a'2`. Ceci fait, on doit distinguer deux cas : soit `a'1` est une valeur fonctionnelle, soit ce n'en est pas une. Dans le dernier cas, il n'y a pas grand chose à faire et l'on renvoie `(! a'1 a'2)`; CAML déclencherait une exception. Dans le premier cas, `a'1` est obligatoirement une clôture (car `a'1` est le résultat de l'évaluation de `a1`); soit `Closure(v, t, env')` cette clôture. Il s'agit alors d'évaluer `t` dans l'environnement `env'` (et non `env`) auquel on aura adjoint la liaison qui à la variable `v` donne la valeur `a'2`. Et le tour est joué.

- L'évaluation des formes du genre définition n'appelle pas de commentaires. Remarquer quand même que l'on renvoie la valeur de la nouvelle liaison : ce choix est parfaitement arbitraire.
- L'évaluation de la forme if-then-else appelle deux commentaires. Tout d'abord, $\mathcal{ML}$ utilise de manière interne deux variables réservées qui sont `true` et `false`. En second lieu, et c'est ce qui fait tout l'intérêt de la forme if-then-else, la clause 'else' (`b`) n'est pas évaluée si le test (`t`) vaut `true` et, réciproquement, la clause 'then' (`a`) n'est pas évaluée si le test (`t`) vaut `false`.

17.6 amélioration des clôtures et donc de `eval`

On aura remarqué que l'évaluation d'une forme fonctionnelle `(? x E)` dans un environnement `env` se résume à la construction d'une clôture où la totalité de l'environnement `env` est gardé en mémoire. C'est franchement du gaspillage de ressources mémoire car il s'agit de ne mémoriser que les valeurs des variables libres de `(? x E)`. Je propose donc l'amélioration suivante.

```
let rec (free_var : term -> string list) = function
   | Variable v            -> [v]
   | Application(a1, a2)   -> union (free_var a1) (free_var a2)
   | Fonction(v, t)        -> subtract (free_var t) [v]
   | Conditionnal(t, a, b) -> union (free_var t) (union (free_var a) (free_var b))
   | Definition(v, t)      -> union [v] (free_var t)
   | Closure(_, _, _)      -> [] ;;

let (remain : environnement -> string list -> environnement) =
   function env ->
      let rec aux = function
         |  []  -> []
         | h::t -> extend (aux t) h (find env h)
      in aux ;;

let (make_closure : string -> term -> environnement -> term) =
   fun v t env ->
      let env' = remain env (subtract (free_var t) [v])
      in Closure(v, t, env') ;;
```

La fonction `free_var` calcule la liste des variables libres d'un terme. La fonction `remain` reçoit un premier argument `env` qui est un environnement et un second qui est une liste de variable v_i; elle fabrique alors un nouvel environnement qui ne laisse subsister de `env` que les liaisons concernant les variables v_i. Enfin, on l'aura deviné, `make_closure` fabrique des clôtures moins goinfres en ressources mémoire que dans la section précédente. On en déduit une nouvelle fonction `eval` qui ne diffère de la précédente que par le traitement de la clause `Fonction` :

```
let rec (eval : environnement -> term -> term) =
   function env ->
      function
         | Variable(v)
           -> find env v
         | Fonction(v, t)
          -> make_closure v t env
         | ...etc.
```

17.7 pretty-print de $\mathcal{ML}$

Rien de bien passionnant; cette fonction est donnée pour mémoire; on choisit d'être laconique quand il s'agit d'imprimer une valeur du type `Closure` : en effet, il faudrait imprimer l'environnement, ce qui semble rédhibitoire, vu leur tendance à l'embonpoint dans notre évalateur.

```
let rec print_term = function
   | Variable(s) -> print_string s
   | Application(a, b)
      -> print_string "(! " ;
         print_term a ;
         print_string " " ;
         print_term b ;
         print_string ")"
   | Fonction(v, t)
      -> print_string "(? " ;
         print_string v ;
         print_string " " ;
         print_term t ;
         print_string ")"
   | Conditionnal(t, a, b)
      -> print_string "(if " ;
         print_term t ;
         print_string " " ;
         print_term a ;
         print_string " " ;
         print_term b ;
         print_string ")" ;
   | Closure(_, _, _)
      -> print_string "<closure>"
   | Definition(v, t)
      -> print_string "(= " ;
         print_string v ;
         print_string " " ;
         print_term t ;
         print_string ")" ;;
```

On a écrit une fonction `print_term_verbose` qui imprime les environnements, mais on ne donnera pas son texte ici. (Il ne présente pas d'intérêt.)

17.8 une boucle read-eval-print pour $\mathcal{ML}$

Un interprète (ou interpréteur) se présente en général sous la forme d'une boucle infinie qui, perpétuellement, lit un terme `t`, l'évalue, et imprime sa valeur `t'`. Par "lecture" on doit bien sûr entendre l'analyse lexicale, suivie de l'analyse syntaxique d'une chaîne de caractères `s` (syntaxe concrète) qui représente le terme `t` lui-même (syntaxe abstraite).

Voici une fonction read-eval-print :

```
let REP flag s =
   let imprime =
      if flag
      then print_term_verbose
      else print_term
   in begin
         print_string "ML value : " ;
         imprime (eval !global_environnement
                       (parser (lexer (stream_of_string s)))) ;
         print_newline() ;
      end ;;
```

Et voici un "toplevel" minimaliste pour $\mathcal{ML}$ (i.e. une boucle read-eval-print) :

```
let toplevel flag =
   try while true do
      let entrée = read_line ()
      in begin
            flush stdout ;
            REP flag entrée ;
            print_newline() ;
         end
      done
   with _ -> () ;;

REP      : bool -> string -> unit
toplevel : bool -> unit
```

Il n'y a plus, maintenant, qu'à essayer d'utiliser $\mathcal{ML}$.

17.9 utilisation de $\mathcal{ML}$

17.9.1 utilisation de $\mathcal{ML}$: les booléens

Notre langage est muni d'une forme conditionnelle simple (if-then-else) mais il est dépourvu de booléens. Comment faire?

Commençons par remarquer que des valeurs booléennes purement fonctionnelles pouraient être écrite en CAML de la manière suivante :

```
let vrai x y = x and faux x y = y ;;
```

L'idée sous-jacente à cette définition est que les booléens *sont* alors des conditionnelles : au lieu d'écrire '`if bool then A else B`', on écrit simplement '`(bool A B)`'. Attention, ne pas oublier qu'en CAML, `(a b c)` est compris comme `((a b) c)`. En voici la preuve :

```
vrai "c'est VRAI" "c'est FAUX" ;;
- : string = "c'est VRAI"

faux "c'est VRAI" "c'est FAUX" ;;
- : string = "c'est FAUX"
```

Ayant cette remarque en tête, on écrit facilement le `non`, le `et` et le `ou`.

```
let non p = p faux vrai ;;

let ou a b = a vrai b and et a b = a b faux ;;
```

Il va suffire de transcrire les définitions CAML ci-dessus, pour obtenir ci-dessous les fonctions $\mathcal{ML}$ correspondantes.

```
begin reset(); toplevel false end ;;

(* maintenant, un ML tout frais nous écoute; parlons-lui *)

(= vrai (? x (? y x)))
ML value : <closure>

(= faux (? x (? y y)))
ML value : <closure>
```

```
(= non (? p (! (! p faux) vrai)))
ML value : <closure>

(= ou (? a (? b (! (! a vrai) b))))
ML value : <closure>

(= et (? a (? b (! (! a b) faux))))
ML value : <closure>
```

Si nous voulons faire un essai, il s'avère un peu décourageant :

```
(! (! et vrai) faux)
ML value : <closure>
```

"`<closure>`" dit-il; et c'est tout. C'est pourquoi nous définissons la fonction $\mathcal{ML}$ suivante qui convertit les booléens "fonctionnels" `vrai` et `faux` en simples variables $\mathcal{ML}$ `true` et `false` (qui seront réservées à ce seul usage). J'insiste sur le fait que `true` et `false` sont représentés de manière interne par '`Variable "true"`' et '`Variable "false"`'; cela est utilisé dans l'évaluateur (fonction `eval`). Bien sûr, ces variables $\mathcal{ML}$ `true` et `false` n'ont rien à voir avec les booléens de CAML.

```
(= convert_bool (? b (! (! b true) false)))
ML value : <closure>
```

On peut reprendre nos essais :

```
(! convert_bool (! (! et vrai) faux))
ML value : false

(! convert_bool (! (! ou vrai) faux))
ML value : true
```

17.9.2 utilisation de $\mathcal{ML}$: les couples

Définir les couples de valeurs, c'est, au minimum, créer une fonction `pair` et deux fonctions `fst` et `snd` telles que si `c` = `(pair a b)`, alors `(fst c)` = `a` et `(snd c)` = `b`. On vérifie facilement que les fonctions CAML qui suivent remplissent ce contrat.

```
let pair a b f = f a b
and fst c = c vrai
and snd c = c faux ;;
```

Il va suffire de transcrire en $\mathcal{ML}$.

```
(= pair (? a (? b (? f (! (! f a) b)))))

(= fst (? c (! c vrai)))

(= snd (? c (! c faux)))
```

Essayons :

```
(= c (! (! pair a) b))
ML value : <closure>

(! fst c)
ML value : a

(! snd c)
ML value : b
```

17.9.3 utilisation de $\mathcal{ML}$: les entiers

Notre Mini-Langage souffre d'un manque cruel : il est dépourvu d'entiers; cela semble sans espoir (le mathématicien allemand J.W.R. Dedekind disait “Dieu a créé les entiers, l'Homme a fait le reste”). On va voir néanmoins que les entiers peuvent être définis et manipulés en $\mathcal{ML}$: on exploitera une idée du mathématicien américain A. Church.

L'idée est de représenter l'entier n ($n \in \mathbb{N}$) par la fonctionnelle (c'est l'entier de Church correspondant) qui, à une fonction f, associe son itérée n-ième (au sens de la composition); naturelement, à l'entier 0 (zéro) est associé la fonctionnelle qui, à une fonction f, associe l'identité ($f^0(x) = x$). Voici les définitions CAML correspondantes.

```
let int_of_church n = n (function x -> x + 1) 0 ;;

let rec church_of_int = fun
   | 0 f x -> x
   | n f x -> f ((church_of_int (n - 1) f) x) ;;

int_of_church (church_of_int 13) ;;
- : int = 13
```

Dans ce qui suit, le test d'égalité à zéro (`est_zéro`) mérite un peu d'attention; je conseille de l'essayer à la main pour les entiers de Church 0, 1, 2; on aura vite fait de comprendre. Les fonctions successeur (`succ`), addition (`add`), et multiplication (`mult`), sont aisées; on n'a pas écrit l'exponentiation, mais elle est de la même veine. La fonction prédécesseur est assez jolie; en voici l'idée. On commence par définir la fonction $(a, b) \mapsto (a+1, a)$: c'est la fonction baptisée `succ_pair`. Une fois que l'on dispose de cette fonction, il suffit de l'itérer n fois sur $(0, 0)$ pour obtenir $(n, n-1)$. On dispose alors du prédécesseur de n en deuxième position du couple. Enfin, dès que l'on dispose de la fonction prédécesseur, la soustraction (`sub`) est facile à définir.

```
let zéro f x = x ;; (* zéro est aussi (church_of_int 0) *)

let succ n f x = f (n f x) ;;

let est_zéro n = n (vrai faux) vrai ;;

let add n p f x = (n f) (p f x) ;;

let mult n p f x = (n (p f)) x ;;

let succ_pair c = pair (succ (fst c)) (fst c) ;; (* 'c' est un couple *)

let pred n = snd (n succ_pair (pair zéro zéro)) ;;

let sub n p f x = (p pred) n f x ;;
```

Transcrivons en $\mathcal{ML}$.

```
(= zero (? f (? x x)))

(= is_zero (? n (! (! n (! vrai faux)) vrai)))

(= succ (? n (? f (? x (! f (! (! n f) x))))))

(= add (? n (? p (? f (? x (! (! n f) (! (! p f) x)))))))

(= mult (? n (? p (? f (? x (! (! n (! p f)) x))))))
```

```
(= succ_pair (? c (! (! pair (! succ (! fst c))) (! fst c))))

(= pred (? n (! snd (! (! n succ_pair) (! (! pair zero) zero)))))

(= sub (? n (? p (? f (? x (! (! (! (! p pred) n) f) x))))))
```

On laissera le lecteur faire ses essais (moi, j'suis sûr qu'ça marche).

17.9.4 utilisation de $\mathcal{ML}$: la récursivité

$\mathcal{ML}$ ne possède ni boucle (ni `for`, ni `while`), ni possibilité de définition récursive; là encore, cela semble sans espoir; là encore, la déréliction sera de courte durée : la Providence va se manifester d'éclatante manière sous l'apparence du combinateur `Z` de Turing.

On prendra l'exemple de la factorielle et on nommera `E[f,n]` l'expression $\mathcal{ML}$ suivante.

```
(if (! convert_bool (! is_zero n))
    (! succ zero)
    (! (! mult n) (! f (! pred n))))
```

On aura reconnu le "corps" de la fonction factorielle. Si nous essayons de définir la factorielle par `(= f (? n E[f,n]))`, cette tentative est vouée à l'échec car la récursivité n'est pas reconnue; c'est comme si nous écrivions '`let f = ...`' au lieu de '`let rec f = ...`' en CAML.

On appelle combinateur de point fixe `Y` une fonctionnelle $\mathcal{ML}$ telle que l'évaluation d'une quelconque expression de la forme `(! Y t)` entraîne l'évaluation de `(! t (! Y t))`. Le lecteur courageux peut vérifier que l'on peut prendre `Z`, dont la définition suit, comme combinateur de point fixe.

```
REP false "(= Z
              (? f (! (? x (! f (? y (! (! x x) y))))
                      (? x (! f (? y (! (! x x) y))))
                   )))" ;;
ML value : <closure>
- : unit = ()
```

Son analogue CAML devrait s'écrire

```
let Z = function f ->
   (function x -> (f (function y -> ((x x) y))))
   (function x -> (f (function y -> ((x x) y))))
   ;;
Toplevel input:
>   (function x -> (f (function y -> ((x x) y))))
>                                      ^
This expression has type 'a -> 'b,
but is used with type 'a.
```

Mais il n'est bien sûr pas typable. Donnons alors un exemple d'utilisation de `Z` en SCHEME (qui est un bon analogue de CAML, mais sans système de types) :

```
(define Z
  (lambda (f)
    (
     (lambda (x) (f (lambda (z) ((x x) z))))
     (lambda (x) (f (lambda (z) ((x x) z))))
    )
    ))
```

Compte tenue de la propriété de Z, il suffit alors d'écrire, pour encoder la récursivité :

```
(define factorielle
  (Z
    (lambda (f)
      (lambda (n)
        (if (= 0 n)
          1
          (* n (f (- n 1))))))))

: (factorielle 26)
403291461126605635584000000
```

En $\mathcal{ML}$, nous écrirons :

```
REP false "(= fact
              (! Z
                (? f (? n
                       (if (! convert_bool (! is_zero n))
                           (! succ zero)
                           (! (! mult n) (! f (! pred n)))
                       )))))" ;;
ML value : <closure>
- : unit = ()
```

Un test s'impose; on calculera 4! = 24; ce sera `essai`; puis, nous appliquerons 24 fois `pred` à `essai` pour tester l'égalité du résultat à `zero`. (Oui, $\mathcal{ML}$ devient intelligent, mais ses moyens de communication (ses entrées-sorties) restent indigents.)

```
REP false "(= essai (! fact (! succ (! succ (! succ (! succ zero))))))" ;;
ML value : <closure>
- : unit = ()

REP false "(! convert_bool (! is_zero
(! pred (! pred (! pred (! pred (! pred (! pred
(! pred (! pred (! pred (! pred (! pred (! pred
(! pred (! pred (! pred (! pred (! pred (! pred
(! pred (! pred (! pred (! pred (! pred (! pred essai)
))))))))))))))))))))))))" ;;
ML value : true
- : unit = ()
```

ÇA MARCHE !, ou, mieux dit, ça fonctionne. (Mais ça se traîne complétement et ce n'est pas une surprise; les vrais entiers sont quand même une bonne chose...)

Partie III
MISCELLANÉES

18 les versions 0.73 et 0.74 de caml

Les différences essentielles (au niveau d'utilisation de ce poly) ont été signalées au fil du texte. Je reproduis ici *verbatim* le fichier `CHANGES` de la version 0.74.

```
* Typing: when typing a sequence (e1;e2), warn if e1 does not have
  type "unit".

* Standard library:
  - module string: added index_char, rindex_char, index_char_from,
    rindex_char_from.
  - module vect: added init_vect.
  - module format: added a printf facility to control the pretty-printer.
  - module float: more floating-point functions (all ANSI-C functions).
  - The iterators do_list, do_vect et al have more restricted types, e.g.
    ('a -> unit) -> 'a list -> unit instead of ('a -> 'b) -> 'a list -> unit.
  - module sys: new function sys__time to measure elapsed time.

* MS Windows port: fixed bugs in functions over graphics__image.

* libnum: square root rewritten. Many bug fixes to support 64 bits
  architectures. Lot of functions rewritten or added. More tests
  added.
  Module nat: now performs sanity checks.
  New module fnat: the same functionality as module nat, without
  sanity checks.
  The toplevel "camlnum" now comes with printers installed for the
  num types.
  Assembly code for pentium processors is now supported (on average
  this implementation is approximately 3 times faster than the pure C
  implementation) (set the variable BIGNUM_ARCH to pentium in the
  Makefile of the contrib directory).
```

Voici le fichier `KNOWN-BUGS` de la version 0.74.

```
The following problems have not been fixed at the time of this release:

1- Stream concatenation using [< ... >] does not always preserve the
   sharing among streams, and sometimes duplicate stream subcomponents.
   For instance, if you define s' = [< '1; s >] and then read alternatively
   from s' and from s, a given element of s can be read twice.
   The problem occurs only if s is in tail position inside s'.
   To guarantee proper sharing, move s in non-tail position, e.g.
   take s' = [< '1; s; [<>] >].
```

19 bibliographie

1. Functional programming using CAML LIGHT; par Michel Mauny. Il s'agit d'un cours enseigné par un des concepteurs du langage; ce document (en format `postscript`) fait partie de la distribution standard de CAML (disponible sur le serveur `ftp` de l'INRIA). J'aime beaucoup ce texte; lecture hautement recommandée donc, mais en anglais.

2. The CAML LIGHT system, release 0.7..., documentation and user's manual; par Xavier Leroy; ce document (en format `postscript`) fait partie de la distribution standard de CAML (disponible sur le serveur `ftp` de l'INRIA). Le titre laisse supposer que ce document est écrit en anglais : c'est le cas.

3. Structure and interpretation of computer programs; par H. Abelson, G. J. Sussman with J. Sussman; publié par MIT Press. L'ouvrage a été traduit en français sous le titre "structure et interprétation des programmes informatiques" et publié chez InterÉditions. Il s'agit d'un cours remarquable destiné au premier cycle du célèbre MIT; ils utilisent un autre langage fonctionnel que CAML : SCHEME. J'insiste : *c'est un livre remarquable.*

4. Tant que nous sommes à SCHEME (qui est un très beau langage, très proche de CAML), mentionnons la référence : the "Revised^4 Report on the algorithmic language SCHEME", connue sous le nom de R4RS. Ce document (en format `postscript`) est disponible sur plusieurs serveurs `ftp` dont celui de l'INRIA.

5. Le langage CAML; par P. Weiss et X. Leroy; publié chez InterÉditions. Deux concepteurs du langage ont fait l'effort d'écrire un livre (1) agréable, (2) nourrissant, et (3) en français sur CAML (version 0.6) ! À pratiquer, donc. Le manuel de référence (en français) de la version 0.6 est également disponible chez le même éditeur.

6. Approche fonctionnelle de la programmation; par G. Cousineau et M. Mauny (deux concepteurs du langage); publié chez Édiscience. *Un très beau livre qui ne fait pas du tout double emploi avec les précédents.*

7. La lettre de CAML numéro 0,1,2,3,4,5,6,7,..., publiée par Laurent Chéno. Je laisse Laurent Chéno présenter la lettre : *"Voici donc le numéro 0 de La lettre de* CAML. *Mon intention est de lancer un petit bulletin de liaison entre utilisateurs de* CAML LIGHT, *en pensant tout particulièrement aux collègues des classes prépas chargés de l'enseignement de l'option informatique. [...] J'y verrais bien des exemples de programmation, des algorithmes, des analyses d'algorithme, des remarques sur nos implémentations, des utilitaires... [...] Je suis réfractaire à toutes les bibles : le programme officiel de nos classes ne saurait être autre chose qu'un guide (un prétexte, même) pour ce bulletin; nous nous évaderons sans scrupule."* La présentation des lettres de CAML est superbe; le contenu parfois difficile au niveau sup.; elles sont disponibles sur les serveurs de l'INRIA.

8. Cours complet pour la sup. MPSI, option informatique; par D. Monasse; publié par Vuibert. Le cours d'informatique pour la deuxième année existe également, par le même auteur, chez le même éditeur. C'est le premier cours complet publié pour l'option informatique.