Chapitre 1

option informatique

Les bases de la programmation en Caml

1. Introduction

1.1 Qu'est ce que Caml ?

Caml est un langage de programmation développé par l'INRIA depuis 1985 ; il se range dans la catégorie des langages fonctionnels, mais se prête aussi à la programmation impérative (et orientée objet pour OCaml).
Il existe deux implémentations du langage : OCaml est la version la plus avancée et la plus complète ; comme tout langage moderne elle possède une importante bibliothèque logicielle à même de répondre à tous les besoins des programmeurs qui utilisent ce langage. Caml Light est une version plus légère, destinée à l'enseignement et possédant une bibliothèque logicielle très limitée. Bien que cette version ne soit plus maintenue depuis 2002, c'est la version préconisée pour les concours, c'est donc celle que nous utiliserons.

La programmation impérative repose sur la notion de machine abstraite possédant une mémoire et des instructions modifiant les états de celle-ci grâce à des affectations successives. Cela nécessite pour le programmeur de connaître en permanence l'état de la mémoire que le programme modifie, ce qui peut se révéler une tâche complexe.

La programmation fonctionnelle va mettre en avant la définition et l'évaluation de fonctions pour résoudre un problème, en interdisant toute opération d'affectation. Dans ce mode de programmation, la mémoire est donc gérée automatiquement et de manière transparente pour le programmeur.

La programmation orientée objet s'articule autour des données d'un programme. Ceux-ci sont des objets constitués d'attributs les caractérisant et de méthodes les manipulant (ces dernières pouvant être fonctionnelles ou impératives), l'ensemble des attributs et méthodes constituant une classe.

Figure 1 – Trois grand paradigmes de la programmation moderne.

Vous trouverez sur la page officielle du langage le manuel complet de Caml Light :

`http://caml.inria.fr/pub/docs/manual-caml-light/`

À part pour des besoins très spécifiques, vous n'aurez besoin que du chapitre intitulé *The core library* dans lequel sont expliquées les instructions clés du langage.

1.2 Comment programmer en Caml Light ?

Ce langage n'étant plus maintenu depuis 2002, les outils développés à cette époque commencent à poser des problèmes de compatibilité avec les matériels récents. Heureusement, des bénévoles proposent des solutions pour vous permettre de continuer à coder en Caml Light.

- *The easy way* (pour Windows, Mac OSX ou Linux)

Jean Mouric, un collègue de Rennes, maintient à jour un environnement de développement intégré regroupant un éditeur de texte (aux possibilités malheureusement assez limitées) et un interprète de commande pour une utilisation interactive de Caml-Light.

`http://jean.mouric.pagesperso-orange.fr/`

Installez WinCaml si vous êtes sous Windows, MacCaml si vous êtes sous Mac OSX et JavaCaml si vous êtes sous Linux.

• *The hard way* (pour Linux ou Mac OSX)

Si vous êtes sous Linux ou OSX, que vous utilisez déjà un éditeur de code et que l'installation d'un logiciel en ligne de commande ne vous effraie pas, vous pouvez souhaiter installer uniquement Caml Light pour ensuite l'utiliser en interaction avec votre éditeur de code favori.
Sous Linux, utiliser le script écrit par un autre collègue, Judicaël Courant, que vous trouverez à cette adresse :

`http://judicael.courant.free.fr/2015/02/20/installationcaml.html`

Sous OSX, installer les binaires précompilés que l'on trouve à l'adresse suivante (la version Intel) :

`http://caml.inria.fr/distrib/caml-light-0.80/`

Seul inconvénient sous OSX, la version installée ne contient pas toutes les librairies et en particulier le module graphique, mais a priori vous n'en aurez pas besoin [1].

Une fois l'installation terminée, ouvrez un terminal et vérifiez que vous pouvez ouvrir une session Caml à l'aide de l'instruction **`camllight`**. Il vous reste à lier cette commande à votre éditeur de code (si vous utilisez Emacs, installez le mode Tuareg, qui est un excellent éditeur de code Caml).

• Utilisation interactive

Caml est un langage compilé : on écrit les programmes à l'aide d'un éditeur de textes, et le tout est traité par un compilateur pour créer un exécutable. Il existe cependant une boucle interactive qui permet d'interpréter le langage : l'utilisateur tape des morceaux de programme qui sont traitées instantanément par le système, lequel les compile, les exécute et écrit les résultats à la volée. C'est un mode privilégié pour l'apprentissage du langage.
Lors d'une session interactive, le caractère `#` qui apparait à l'écran est le symbole d'invite du système (le *prompt*). Le texte écrit par l'utilisateur commence après ce caractère et se termine par deux points-virgules consécutifs. Une fois lancé, l'interprète va afficher les types et valeurs des expressions évaluées, avant de réafficher le prompt.
Commençons donc par observer une première session Caml :

```
>       Caml Light version 0.75

# let rec fact = function  (* définition d'une fonction *)
    | 0 -> 1
    | n -> n * fact (n-1) ;;
fact : int -> int = <fun>
# fact 5 ;;                (* application d'une fonction *)
- : int = 120
# fact 2.3 ;;                    (* une erreur de typage *)
Toplevel input:
>fact 2.3 ;;
>     ^^^
This expression has type float, but is used with type int.
#
```

Dans ce premier exemple, nous avons commencé par définir une fonction **`fact`** à l'aide de l'instruction de nommage **`let ... = ...`**. Cette définition est récursive, comme l'indique le mot-clé **`rec`**, et procède par filtrage (nous reviendrons plus loin sur cette notion), ce qui nous permet d'obtenir une définition très proche de la définition mathématique de la fonction factorielle :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

L'interprète nous répond que nous avons défini un nouveau nom (*fact*), qui est de type *int -> int*, et que ce nom a pour valeur une fonction (*<fun>*).
Nous pouvons observer que Caml est pourvu d'une reconnaissante automatique de type : cette fonction ne peut s'appliquer qu'à des entiers, et retournera toujours un entier. Nous en avons l'illustration dans la suite de l'exemple : à la commande suivante, l'interprète nous répond que nous avons simplement calculé une valeur sans la nommer (signalé par le caractère *-*), que cette valeur est de type *int*, et qu'elle vaut *120*. En revanche, appliquer cette fonction à un nombre non entier (ici de type *float*) conduit à un message d'erreur. Caml est en effet un langage fortement typé : les erreurs de typage sont détectées et les conversions implicites de type sont formellement interdites.

1. Il est possible de les installer, mais c'est compliqué. Me demander si nécessaire.

1.3 Typage fort, typage faible

Tout objet défini par un langage de programmation possède un *type* (*int*, *float*, etc) qui caractérise la manière dont il sera représenté en mémoire. Le typage du langage Caml est dit *fort* car il possède les particularités suivantes :

- le typage d'une fonction est réalisé au moment de sa définition ;
- les conversions implicites de type sont formellement interdites.

À l'inverse, le typage du langage Python est faible : une même fonction peut s'appliquer à des objets de type différent. Par exemple, la fonction Python **len** calcule indifféremment la longueur d'une chaîne de caractères ou d'une liste alors qu'en Caml existent deux fonctions distinctes **string_length** et **list_length**.
La conversion implicite de type est aussi possible en Python. Par exemple, nous savons que entiers et flottants sont représentés différemment en mémoire, et qu'en conséquence de quoi les algorithmes d'addition entre entiers ou entre flottants ne sont pas les mêmes. Ainsi, lorsqu'en Python vous additionnez un flottant avec un entier, ce dernier est au préalable converti en flottant avant d'être additionné avec l'algorithme d'addition dédié au type *float*[2] En Caml, cette conversion implicite est impossible, il faut réaliser explicitement la conversion :

```
# 1.0 + 2 ;;
Toplevel input:
> 1.0 + 2 ;;
> ^^^
This expression has type float, but is used with type int.
# 1.0 +. float_of_int 2 ;;
- : float = 3.0
```

On peut observer en outre que Caml n'autorise pas la surcharge d'opérateur : en Python, le symbole + désigne différentes fonctions : l'addition des entiers lorsqu'il est situé entre deux entiers, l'addition des flottants lorsqu'il est situé entre deux flottants (ou entre un entier et un flottant), voire la concaténation lorsqu'il est situé entre deux chaînes de caractères. En Caml, l'addition des entiers se note **+**, l'addition des flottants **+.**, la concaténation des chaînes de caractères **^**, etc.

1.4 Définitions globales et locales

En Python, le nommage diffère suivant qu'on nomme un simple objet (avec =) ou une fonction (avec **def**). Dans un langage fonctionnel, cette différence s'estompe : *les fonctions sont des objets comme les autres*. En Caml, l'instruction **let** attribue un nom à une valeur. Ces définitions ne sont pas modifiables, elles diffèrent donc fondamentalement du mécanisme d'affectation propre à la programmation de style impératif que nous étudierons plus tard. Une fois défini, un nouveau nom est utilisable dans d'autres calculs :

```
# let n = 12 ;;
n : int = 12
# let f = function x -> x * x ;;
f : int -> int = <fun>
# f n ;;
- : int = 144
```

Les définitions précédentes sont *globales* : la liaison qui a été établie entre le nom et sa valeur est permanente. La syntaxe « **let ... = ... in** » permet de définir temporairement un nom pour la seule durée du calcul en question :

```
# let n = 11 in f n ;;
- : int = 121
# n ;;
- : int = 12
```

On peut observer sur l'exemple précédent que même si elles portent le même nom, variables locale et globale restent bien différentiées.
Notons enfin que le mot **and** permet des définitions multiples :

```
# let n = 9 and f = function x -> x * x * x ;;
n : int = 9
f : int -> int = <fun>
```

2. La réalité est même un peu plus complexe, mais là n'est pas le sujet.

Attention, les valeurs ne deviennent visibles qu'après toutes les déclarations simultanées :

```
# let a = 5 and f = function x -> a * x ;;
Toplevel input:
> let a = 5 and f = function x -> a * x ;;
>                                 ^
The value identifier a is unbound.
```

Pour réaliser cette définition il faudrait écrire :

```
# let f = let a = 5 in function x -> a * x ;;
f : int -> int = <fun>
```

1.5 Fonctions

Définir une fonction en Caml est simple et proche de la notation mathématique usuelle : on peut utiliser le constructeur **function** (nous y reviendrons plus loin) mais aussi la syntaxe **let f arg = expr**, où **arg** désigne l'argument de la fonction et **expr** le résultat souhaité. Par exemple, pour définir la fonction $f : x \mapsto x+1$ on écrira :

```
# let f x = x + 1 ;;
f : int -> int = <fun>
# f 2 ;;
- : int = 3
```

On peut noter que l'usage des parenthèses n'est ici pas nécessaire :

$$\boxed{f\ x \text{ est équivalent à } f(x).}$$

En revanche, elles peuvent se révéler indispensables pour des expressions plus complexes :

```
# f 2 * 3 ;;
- : int = 9
# (f 2) * 3 ;;
- : int = 9
# f (2 * 3) ;;
- : int = 7
```

Cet exemple permet de mettre en évidence le fait que les expressions sont associées à gauche :

$$\boxed{f\ x\ y \text{ est équivalente à } (f\ x)\ y.}$$

• Fonctions à plusieurs arguments

Les fonctions possédant plusieurs arguments se définissent à l'aide de la syntaxe **let f args = expr**, où cette fois-ci **args** désigne les différents arguments de la fonction, séparés par un espace. Par exemple, pour définir l'application $g : (x,y) \longmapsto x+y$ on écrira :

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

Attention, il existe ici une ambiguïté : doit-on considérer que g est une fonction à deux variables définies dans $\mathbb{Z}$, ou que g est une fonction à une variable définie dans $\mathbb{Z}\times\mathbb{Z}$? Observons ce que répond l'interprète de commande à cette deuxième solicitation :

```
# let h (x, y) = x + y ;;
h : int * int -> int = <fun>
# h (2, 3) ;;
- : int = 5
```

Le type *int * int* décrit l'ensemble des couples d'entiers, il s'agit donc bien d'une fonction à une variable. Mathématiquement il est facile d'identifier ces deux définitions ; il n'en est pas de même en informatique, et

les langages fonctionnels privilégient la première définition car elle possède un avantage sur la seconde : elle permet aisément de définir des applications partielles. Par exemple, on peut définir la fonction $f : x \mapsto x+1$ à partir de la fonction g en écrivant : `let f x = g 1 x`, ou encore `let f x = (g 1) x` suivant la règle de l'association à gauche. Le langage Caml autorisant la *simplification à droite*, il nous est alors loisible de définir f de la façon suivante :

```
# let f = g 1 ;;
f : int -> int = <fun>
```

Nous n'aurions pas pu procéder à une telle simplification avec la fonction h, car l'écriture `let f x = h (1, x)` ne permet pas de simplification à droite.
La fonction `g` est dite *curryfiée* [3] par opposition à la fonction `h`, non curryfiée.

• Fonctions anonymes

Une fonction est donc un objet typé, qui en tant que tel peut être calculé, passé en argument, retourné en résultat. Pour ce faire, on peut construire une valeur fonctionnelle anonyme en suivant la syntaxe `function x -> expr`. Par exemple, les fonctions `f`, `g` et `h` peuvent aussi être définies de la façon suivante :

```
# let f = function x -> x + 1 ;;
f : int -> int = <fun>
# let g = function x -> function y -> x + y ;;
g : int -> int -> int = <fun>
# let h = function (x, y) -> x + y ;;
h : int * int -> int = <fun>
```

Cette nouvelle définition de `g` a le mérite de mettre en évidence son caractère curryfié, autrement dit la raison pour laquelle `f = g 1` : en effet, `g 1` est équivalent à : `function y -> 1 + y`.
Autre conséquence de la curryfication, on peut observer que le typage est associatif à droite :

int -> int -> int est équivalent à *int -> (int -> int)*.

Remarque. Par essence, l'instruction `function` ne s'utilise que pour des fonctions d'une variable, ce qui conduit à une écriture un peu lourde pour définir de manière anonyme une fonction currifiée telle la fonction `g`. C'est la raison pour laquelle il existe une instruction `fun` qui permet d'alléger l'écriture :

`fun a b c ...` est équivalent à `function a -> function b -> function c -> ...`

Par exemple, la fonction `g` peut aussi être définie par : `let g = fun x y -> x + y`.

2. Les types en Caml

Nous l'avons déjà dit, toute valeur manipulée par Caml possède un *type* reconnu automatiquement par l'interprète de commande sans qu'il soit nécessaire de le déclarer : connaissant les types des valeurs de base et des opérations primitives, le contrôleur de types produit un type pour une phrase en suivant des règles de typage pour les constructions du langage (nous en avons eu un premier aperçu avec la définition des fonctions). Nous allons maintenant passer en revue les principaux types élémentaires ainsi que quelques opérations primitives, avant de décrire les premières règles de construction des types composés.

2.1 Types élémentaires

• Le vide

Le type *unit* est particulièrement simple, puisqu'il ne comporte qu'une seule valeur, notée `()`, et qu'on appelle *void* (c'est l'équivalent de `None` en Python).
Pour comprendre sa raison d'être, imaginons que l'on veuille afficher une chaîne de caractère à l'écran : nous n'avons rien à calculer, seulement modifier l'environnement (c'est à dire réaliser ce qu'on appelle un *effet de bord*). Or un langage fonctionnel ne peut que définir et appliquer des fonctions. Voilà pourquoi la fonction `print_string` est de type *string -> unit* : elle renvoie le résultat *void*, tout en réalisant au passage un effet de bord.

3. Aucun rapport avec la cuisine, cette définition est un hommage au mathématicien Haskell Curry.

```
# print_string "Hello World" ;;
Hello World- : unit = ()
```

Autre exemple, la fonction **print_newline** est de type *unit -> unit* et a pour effet de passer à la ligne. Pour l'utiliser, il ne faut donc pas oublier son argument (qui ne peut qu'être le *void*) :

```
# print_newline () ;;

- : unit = ()
```

• Les entiers

Sur le ordinateurs équipés d'un processeur 64 bits, les éléments de type *int* sont les entiers de l'intervalle $[\![-2^{62}, 2^{62}-1]\!]$, les calculs s'effectuant modulo 2^{63} suivant la technique du complément à deux (voir votre cours d'informatique de tronc commun). Il faut donc veiller à ne pas dépasser ces limites sous peine d'obtenir des résultats surprenants :

```
# let a = 2147483648 in a * a ;;
- : int = -4611686018427387904
# let a = 2147483648 in a * a - 1 ;;
- : int = 4611686018427387903
```

(Vous avez peut-être deviné que $2^{31} = 2\,147\,483\,648$.)
Notez que **max_int** et **min_int** vous donnent les valeurs maximales et minimales de type entier :

```
# max_int ;;
- : int = 4611686018427387903
# min_int ;;
- : int = -4611686018427387904
```

Les opérations usuelles se notent **+**, **-**, ***** (multiplication), **/** (quotient de la division euclidienne) et **mod** (reste de la division euclidienne). Multiplication, quotient et reste ont priorité face à l'addition et la soustraction, dans les autres cas la règle d'association à gauche s'applique, et on dispose des parenthèses pour hiérarchiser les calculs si besoin est :

```
# 1 + 3 mod 2 ;;      (*  1 + (3 mod 2)  *)
- : int = 2
# 2 * 5 / 3 ;;        (*  (2 * 5) / 3    *)
- : int = 3
# 5 / 3 * 2 ;;        (*  (5 / 3) * 2    *)
- : int = 2
```

• Les réels

Les nombres réels sont approximativement représentés par le type *float*. Tout comme en Python, on les définit sous forme décimale (par exemple **3.141592653**) éventuellement accompagnés d'un exposant représentant une puissance de 10 (par exemple **1.789e3**). Les calculs effectués sur les flottants sont bien évidemment approchés. Les opérateurs usuels se notent : **+.**, **-.**, ***.**, **/.** (noter le point qui permet de les différencier des opérateurs sur le type *int*), et ****** (élévation à la puissance). On dispose en outre d'un certain nombre de fonctions mathématiques telles : **sqrt** (racine carrée), **log** (qui désigne le logarithme népérien), **exp**, **sin**, **cos**, **tan**, **asin** (arcsinus), **acos**, **atan**, etc.

```
# let a = 3.141592654 in tan (a /. 4.) ;;
- : float = 1.00000000021
```

Attention à ne pas oublier que Caml est un langage fortement typé : il n'y a pas de conversion implicite de type, et vous ne pouvez donc pas appliquer une fonction du type *float* à une valeur de type *int* :

```
# let a = 3.141592653 in tan (a /. 4) ;;
Toplevel input:
>let a = 3.141592653 in tan (a /. 4) ;;
>                                   ^
This expression has type int, but is used with type float.
```

Le calcul numérique relevant du programme d'informatique de tronc commun nous n'aurons que rarement besoin de manipuler des flottants.

• Caractères et chaînes de caractères

Les caractères sont les éléments du type *char* ; on les note en entourant le caractère par le symbole ` (backcote, ou accent grave). Les éléments de type *string* sont des chaînes de caractères entourées du symbole " (double quote). Nous reviendrons sur la notion de chaîne de caractère lorsque nous aborderons les vecteurs ; pour l'instant, contentons nous de noter quelques fonctions qui pourront nous être utile :
La concaténation des chaînes de caractères est notée par le symbole **^** :

```
# "liberté, " ^ "égalité, " ^ "fraternité" ;;
- : string = "liberté, égalité, fraternité"
```

La fonction prédéfinie **string_length** renvoie la longueur d'une chaîne de caractères ; c'est donc une fonction de type *string -> int*.
La fonction **nth_char** renvoit le n^e caractère d'une chaîne de caractères (attention, dans une chaîne les indices commencent à 0). C'est une fonction de type *string -> int -> char*. Notons que si une chaîne porte un nom **s** alors **s.[k]** retourne le caractère d'indice k de **s** (**s.[k]** est donc équivalent à **nth_char s k**).
La fonction **sub_string** retourne une sous-chaîne partant d'un indice donné et d'une longueur donnée ; c'est une fonction de type *string -> int -> int -> string*.

```
# nth_char "Caml" 1 ;;
- : char = 'a'
# sub_string "informatique" 2 6 ;;
- : string = "format"
```

• Les booléens

Le type *bool* comporte deux valeurs : le vrai (**true**) et le faux (**false**), et dispose des opérations logiques non (**not**) et (**&&**) ou (**||**). Les deux derniers opérateurs fonctionnent suivant le principe de l'évaluation paresseuse : **expr1 && expr2** ne va évaluer la deuxième expression que si la première est vraie, et **expr1 || expr2** ne va évaluer la deuxième expression que si la première est fausse.

```
# not (1 = 2) || (1 / 0 = 1) ;;
- : bool = true
# not (1 = 2) && (1 / 0 = 1) ;;
Uncaught exception: Division_by_zero
```

• Les paires

Il est possible de définir le produit cartésien de deux types : si **x** est un élément de type *'a* et **y** un élément de *'b*, alors **(x, y)** est un élément de type *'a * 'b*.
Par exemple, nous avons vu que la version non currifiée de la fonction $(x, y) \longmapsto x + y$ se définit par :

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

Il est bien entendu possible de faire le produit cartésien d'un nombre plus important de types, mais seuls les paires disposent des fonctions **fst** et **snd** pour récupérer le première et la seconde composante du couple :

```
# fst ("caml", 0.75) ;;
- : string = "caml"
# snd ("caml", 0.75) ;;
- : float = 0.75
```

2.2 Polymorphisme

On a pu constater que Caml est pourvu d'une reconnaissance automatique de type : par exemple, c'est la présence de l'entier `1` et de l'opérateur entier `+` qui permet d'associer à la fonction `function n -> n + 1` le type *int -> int*.
Mais il peut arriver qu'une fonction puisse s'appliquer indifféremment à tous les types ; on dit dans ce cas que cette fonction est *polymorphe*. On utilise alors les symboles *'a*, *'b*, *'c*,... pour désigner des types quelconques. Revenons par exemple sur les fonctions `fst` et `snd` qui agissent sur les paires. Leur définition est évidente, et n'imposent aucune contrainte de typage :

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

Nous avons vu que les opérations algébriques, l'addition par exemple, ne sont pas polymorphes : l'opérateur infixe `+` est de type *int -> int* alors que `+.` est de type *float -> float*. En revanche, les opérations de comparaison telle l'égalité (notée `=`) le sont [4]. On peut utiliser le même opérateur pour comparer entre eux des entiers, des nombres complexes ou des chaînes de caractères (et plus généralement tout objet pour lesquels la comparaison a un sens), car c'est leur représentation machine qui est comparée.

```
# 1 = 1 ;;
- : bool = true
# 1.0 = 2.0 ;;
- : bool = false
# prefix = ;;
- : 'a -> 'a -> bool = <fun>
```

Les autres opérateurs de comparaison polymorphes sont `<>` (la négation de l'égalité), `<`, `>`, `<=` (inférieur ou égal), `>=` (supérieur ou égal).

```
# "anaconda" < "zebre" ;;
- : bool = true
```

(La comparaison des chaînes de caractères utilise l'ordre lexicographique.)

2.3 Types construits

À partir de types existant on peut construire de nouveaux types à l'aide de la syntaxe : `type montype = constr` où `constr` est une construction de type utilisant :

- des noms (analogues à des constantes) ;
- des noms avec paramètre en suivant la syntaxe : *nom* `of` *type* ;
- l'opérateur `|` correspondant au « ou » logique (ce qu'on appelle un type somme) ;
- des types produits (analogue au « et » logique) obtenus en dressant la liste des caractéristiques suivant la syntaxe : { *car1* : *type1* ; ... ; *carN* : *typeN* }.

Par exemple, s'il n'existait déjà, le type *bool* serait défini par : `type bool = true | false`.

On peut donner un exemple un peu plus élaboré de type somme en définissant l'analogue de la droite numérique achevée $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$:

```
# type reel_etendu = Reel of float | Plus_infini | Moins_infini ;;
Type reel_etendu defined.
# Plus_infini ;;
- : reel_etendu = Plus_infini
# Reel 3.14 ;;
- : reel_etendu = Reel 3.14
```

et définir une fonction pour plonger $\mathbb{R}$ dans $\overline{\mathbb{R}}$:

4. Vous risquez malheureusement de souvent confondre l'égalité en Caml qui se note = et l'égalité en Python qui se note ==. Et c'est la même chose avec l'opérateur de différence (<> en Caml et != en Python).

```
# let etendu_of_float x = Reel x ;;
etendu_of_float : float -> reel_etendu = <fun>
# etendu_of_float 3.14 ;;
- : reel_etendu = Reel 3.14
```

La définition d'un type associé aux nombres complexes est un bon exemple de type produit :

```
# type complexe = {Re : float ; Im : float} ;;
Type complexe defined.
```

Pour construire une valeur de ce type, on énumère ses caractéristiques particulières :

```
# let i = {Re = 0. ; Im = 1.} ;;
i : complexe = {Re=0.0; Im=1.0}
# let complexe_of_float x = {Re = x ; Im = 0.} ;;
complexe_of_float : float -> complexe = <fun>
```

Pour accéder à l'une des caractéristiques d'un objet de type produit, il suffit de faire suivre le nom d'un point et du nom de la caractéristique. Définissons par exemple la multiplication des nombres complexes :

```
# let mult x y =
    let a = x.Re and b = x.Im and c = y.Re and d = y.Im in
    {Re = a*.c-.b*.d ; Im = a*.d+.b*.c} ;;
mult : complexe -> complexe -> complexe = <fun>
# mult i i ;;
- : complexe = {Re=-1.0; Im=0.0}
```

• Types récursifs

Il est possible de définir un type récursif, c'est-à-dire intervenant dans sa propre définition. Nous allons à titre d'exemple définir les couleurs par synthèse soustractive, en convenant qu'une couleur est une couleur primaire ou alors un mélange de deux couleurs :

```
# type couleur = Cyan | Magenta | Jaune | Melange of couleur * couleur ;;
Type couleur defined.
```

Cette définition récursive nous permet de définir de nouvelles couleurs :

```
# let rouge = Melange (Magenta, Jaune) ;;
rouge : couleur = Melange (Magenta, Jaune)
# let orange = Melange (Rouge, Jaune) ;;
orange : couleur = Melange (Melange (Magenta, Jaune), Jaune)
```

Écrivons maintenant une fonction qui calcule la composante CMJ d'une couleur :

```
# let rec cmj = function
    | Cyan                  -> (1., 0., 0.)
    | Magenta               -> (0., 1., 0.)
    | Jaune                 -> (0., 0., 1.)
    | Melange (coul1, coul2) -> let (a, b, c) = cmj coul1 and (d, e, f) = cmj coul2
                                in ((a+.d)/.2., (b+.e)/.2., (c+.f)/.2.) ;;
cmj : couleur -> float * float * float = <fun>
# cmj orange ;;
- : float * float * float = 0.0, 0.25, 0.75
```

3. Définition d'une fonction

Nous l'avons déjà dit, une fonction se définit à l'aide de la syntaxe **`let f = function ... -> ...`** et son type est alors de la forme *type1 -> type2*, le typage étant implicite, et polymorphe le cas échéant. Cette syntaxe est en outre équivalente à **`let f ... = ...`**
Rappelons aussi que **`let f = fun x y -> ...`** est la forme raccourcie de :

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

et qu'on peut aussi écrire `let f x y = ...` Son type est alors *type_de_x -> type_de_y -> type_du_résultat*.

3.1 Analyse par cas : le filtrage

Le filtrage est un trait extrêmement puissant de CAML, et à ce titre très fréquemment employé. La syntaxe d'une fonction travaillant par filtrage peut prendre une des formes suivantes :

```
let f = function
  | motif1 -> expr1
  | motif2 -> expr2
  | ..............
  | motifn -> exprn ;;
```

```
let f x = match x with
  | motif1 -> expr1
  | motif2 -> expr2
  | ..............
  | motifn -> exprn ;;
```

On parle de filtrage *implicite* dans le premier cas et de filtrage *explicite* dans le second.
Lors du calcul d'une telle fonction, l'interprète de commande va essayer de faire concorder l'argument avec les motifs successifs qui apparaissent dans la définition. C'est l'expression correspondant au premier motif reconnu qui sera calculée. Bien entendu, il est nécessaire que les différentes expressions du résultat soient du même type.
Lorsque le motif est un nom, il s'accorde avec n'importe quelle valeur, et la reçoit dans l'expression exécutée. Par exemple, la définition de la fonction sinus cardinal sera :

```
# let sinc = function
    | 0. -> 1.
    | x  -> sin (x) /. x ;;
sinc : float -> float = <fun>
```

Le motif `0.` est une constante qui ne concorde qu'avec la valeur `0.` ; le motif `x` est une variable qui concorde avec n'importe quel nombre flottant.

Lorsqu'il n'est pas nécessaire de nommer un motif (ou lorsqu'il a déjà été nommé précédemment, par exemple lors d'un filtrage explicite), on utilise le caractère `_` qui s'accorde avec n'importe quelle valeur. Par exemple, la fonction `sinc` peut aussi s'écrire :

```
# let sinc x = match x with
    | 0. -> 1.
    | _  -> sin (x) /. x ;;
sinc : float -> float = <fun>
```

Attention, l'ordre dans lequel on essaye de faire correspondre un motif et une valeur a de l'importance. Par exemple, la définition ci-dessous est incorrecte, car la valeur `0.` s'accorde avec le premier motif :

```
# let sinc = function
    | x -> sin(x) /. x
    | 0. -> 1. ;;
Toplevel input:
>  | 0. -> 1. ;;
>    ^^
Warning: this matching case is unused.
sinc : float -> float = <fun>

# sinc 0. ;;
- : float = nan.0
```

(Notez que l'interprète de commande indique que le deuxième motif du filtrage est inutile).

Attention à ne pas non plus confondre concordance avec un motif et égalité avec un nom prédéfini. Considérons la fonction suivante, qui détermine si un entier est nul :

```
# let est_nul = function
    | 0 -> true
    | _ -> false ;;
est_nul : int -> bool = <fun>
```

Cette définition est parfaitement correcte, mais essayons de la généraliser en écrivant :

```
# let egal x = function
    | x -> true
    | _ -> false ;;
Toplevel input:
>  | _ -> false ;;
>    ^
Warning: this matching case is unused.
egal : 'a -> 'b -> bool = <fun>
```

Deux indices nous apprennent que cette fonction ne répondra vraisemblablement pas à notre attente : son type, qui devrait être *'a -> 'a -> bool*, et le message de l'interprète indiquant que le second motif ne sera jamais utilisé. En effet, il ne faudra jamais perdre de vue le fait que le filtrage est structurel : on ne peut utiliser dans les motifs que des constructeurs, des constantes et des noms, *à l'exclusion des valeurs calculées*.
En tout état de cause, tout se passe comme si nous avions écrit :

```
let egal x = function y -> true ;;
```

Le filtrage par motif (*pattern matching*) permet de vérifier si l'objet du filtrage possède une structure donnée ; ce n'est pas un test d'égalité avec un nom prédéfini.

• Motif gardé

Pour corriger notre fonction, il est possible d'utiliser un *motif gardé* dont la syntaxe est :

```
| motif when condition ->
```

Dans un tel cas de filtrage, le motif n'est reconnu que si la condition (une expression de type *bool*) est vérifiée. Nous pouvons donc corriger la fonction précédente en :

```
# let egal x = function
    | y when x = y -> true
    | _            -> false ;;
egal : 'a -> 'a -> bool = <fun>
```

Remarque. Cet exemple a pour but de vous faire comprendre la différence entre test d'égalité et concordance avec un motif. Il n'en reste pas moins que la meilleure définition de cette fonction reste sans doute :

```
let egal x y = x = y ;;
```

ou plus simplement encore :

```
let egal = prefix = ;;
```

(l'instruction **prefix** transforme un opérateur infixe en opérateur préfixe.)

Nous venons de voir que l'interprète de commande est capable de repérer des motifs de filtrage superflus. Il est aussi capable de repérer des filtrages incomplets. Observons par exemple cette première tentative pour définir l'addition sur les réels étendus (définis précédemment) :

```
# let addition r s = match (r, s) with
    | (Reel x, Reel y)        -> Reel (x +. y)
    | (Moins_infini, Reel x)  -> Moins_infini
    | (Plus_infini, Reel x)   -> Plus_infini
    | (Reel x, Moins_infini)  -> Moins_infini
    | (Reel x, Plus_infini)   -> Plus_infini ;;
Warning: this matching is not exhaustive.
addition : reel_etendu -> reel_etendu -> reel_etendu = <fun>
```

En effet, nous n'avons pas encore défini l'addition des infinis entre eux :

```
# addition Plus_infini Plus_infini ;;
Uncaught exception: Match_failure ("", 1131, 1373)
```

il nous faut ajouter les règles $(+\infty)+(+\infty)=(+\infty)$ et $(-\infty)+(-\infty)=-\infty$, et lever une exception dans le cas de l'addition entre $+\infty$ et $-\infty$. Faisons une nouvelle tentative :

```
# let addition r s = match (r, s) with
    | (Reel x, Reel y)        -> Reel (x +. y)
    | (Moins_infini, Reel x)  -> Moins_infini
    | (Plus_infini, Reel x)   -> Plus_infini
    | (Reel x, Moins_infini)  -> Moins_infini
    | (Reel x, Plus_infini)   -> Plus_infini
    | (a, a)                  -> a
    | _                       -> failwith "opération non définie" ;;
Toplevel input:
>     | a a                      -> a
>         ^
The variable a is bound several times in this pattern.
```

Nous observons ici une autre contrainte du filtrage : il n'est pas possible d'utiliser dans un motif le même nom à plusieurs reprises. Là encore, l'utilisation d'un motif gardé résout le problème :

```
# let addition r s = match (r, s) with
    | (Reel x, Reel y)        -> Reel (x +. y)
    | (Moins_infini, Reel x)  -> Moins_infini
    | (Plus_infini, Reel x)   -> Plus_infini
    | (Reel x, Moins_infini)  -> Moins_infini
    | (Reel x, Plus_infini)   -> Plus_infini
    | (a, b) when a = b       -> a
    | _                       -> failwith "opération non définie" ;;
addition : reel_etendu -> reel_etendu -> reel_etendu = <fun>
```

Attention, la vérification de l'exhaustivité d'un filtrage dans lequel un motif est gardé suppose que l'expression conditionnelle attachée à celui-ci est fausse ; en conséquence de quoi, ce motif ne sera pas pris en compte. Par exemple, Caml ne peut détecter que le filtrage suivant est exhaustif :

```
# let f = function
    | x when x >= 0 -> x + 1
    | x when x < 0  -> x - 1 ;;
Toplevel input:
Warning: this matching is not exhaustive.
f : int -> int = <fun>
```

En règle générale, laisser des filtrages non exhaustifs dans un programme est considéré comme un laisser-aller de mauvais aloi. Dans la mesure du possible, il est préférable que le dernier motif d'un filtrage soit un motif générique non gardé de manière à assurer l'exhaustivité du filtrage. dans le cas ci-dessus on se contentera donc d'écrire :

```
# let f = function
    | x when x >= 0 -> x + 1
    | x             -> x - 1 ;;
f : int -> int = <fun>
```

3.2 Récursivité

Considérons de nouveau la définition de la fonction factorielle que nous avons donnée en introduction :

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

Le mot clé **rec** indique que nous avons défini un objet *récursif* (ici une fonction), c'est-à-dire un objet dont le nom intervient dans sa définition. Nous approfondirons plus tard cette notion, très importante dans un langage fonctionnel (la majorité des fonctions que nous écrirons seront récursives). Dans l'immédiat, contentons-nous d'observer sa proximité avec nombre de définitions mathématiques, à commencer avec les suites récurrentes.

Notons enfin qu'il est possible de définir deux fonctions mutuellement récursives à l'aide de la syntaxe **`let rec f = ... and g = ...`**
Les deux fonctions suivantes, par exemple, déterminent la parité d'un entier naturel (de manière assez inefficace il faut bien dire) :

```
# let rec est_pair = function
    | 0 -> true
    | n -> est_impair (n-1)
  and est_impair = function
    | 0 -> false
    | n -> est_pair (n-1) ;;
est_pair : int -> bool = <fun>
est_impair : int -> bool = <fun>
```

4. Exercices

4.1 Définitions globales et locales

Exercice 1 Prévoir la réponse de l'interprète de commande après les définitions suivantes :

```
# let a = 2 ;;
a : int = 2
# let f x = a * x ;;
f : int -> int = <fun>
# let a = 3 in f 1 ;;
???
# let a = 3 and f x = a * x ;;
a : int = 3
f : int -> int = <fun>
# f 1 ;;
???
```

Exercice 2 Prévoir la réponse de l'interprète de commande après les deux définitions suivantes :

```
# let a =
     let a = 3 and b = 2 in
        let a = a + b and b = a - b in
           a - b ;;
 ???
# let b = 2 in a - b * b ;;
 ???
```

4.2 Typage

Exercice 3 Donner des expressions CAML qui correspondent aux types suivants :

a) *(int -> int) -> int*

b) *int -> (int -> int)*

c) *int -> int -> int*

d) *int -> (int -> int) -> int*

Exercice 4 Déterminer sans utiliser l'interprète de commandes le type des expressions suivantes :

a) `fun f x y -> f x y;`

b) `fun f g x -> g (f x);`

c) `fun f g x -> (f x) + (g x).`

Exercice 5 Deviner la réponse de l'interprète de commande lorsqu'on saisit l'expression :

```
# let h (f, g) = function x -> f (g x) ;;
???
```

Exercice 6 Les exercices précédents ont été facilités par le choix des lettres **f** et **g** pour désigner implicitement des fonctions et **x** et **y** pour des variables. Oublions maintenant cette aide pour déterminer le type des expressions suivantes :

a) `fun x y z -> (x y) z;`

b) `fun x y z -> x (y z);`

c) `fun x y z -> x y z;`

d) `fun x y z -> x (y z x);`

e) `fun x y z -> (x y) (z x).`

4.3 Définition de fonctions

Exercice 7 Définir une fonction de type *bool -> bool -> bool* représentant l'opérateur logique $\Rightarrow$.

Exercice 8 L'opérateur des *différences finies* Δ associe à toute suite $(u_n)_{n\in\mathbb{N}}$ la suite $(u_{n+1}-u_n)_{n\in\mathbb{N}}$. Écrire une fonction **delta** qui réalise cette transformation. On commencera par en donner son type.

Exercice 9 Définir une fonction **curry** qui transforme une fonction à deux variables non curryfiée en une fonction curryfiée, puis une fonction **uncurry** qui réalise la transformation inverse.

Exercice 10 Un nombre de Hamming est un entier qui ne comporte que des 2, 3 et 5 dans sa décomposition en facteurs premiers. Écrire une fonction **hamming** de type *int -> bool* qui détermine si un entier donné est un nombre de Hamming.

Exercice 11 Définir en Caml les deux fonctions suivantes :

$$f : (p,q) \longmapsto \begin{cases} q & \text{si } p = 0 \\ f(p-1,q)+1 & \text{sinon} \end{cases} \qquad \text{et} \qquad g : (p,q) \longmapsto \begin{cases} q & \text{si } p = 0 \\ g(p-1,q+1) & \text{sinon} \end{cases}$$

Que calculent-elles ?

Exercice 12 Définir en Caml une fonction récursive calculant le produit de deux entiers naturels en suivant le principe de la multiplication égyptienne, qui utilise les formules :

$$p \times q = \begin{cases} \left(\frac{p}{2}\right) \times (2q) & \text{si } p \text{ est pair} \\ \left(\frac{p-1}{2}\right) \times (2q) + q & \text{si } p \text{ est impair} \end{cases}$$

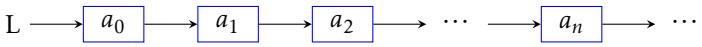
Rédiger en suivant le même principe une fonction qui calcule q^p lorsque p et q sont des entiers.

Chapitre 2

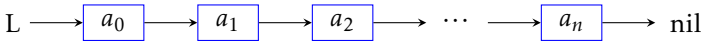
Listes

1. Introduction

En informatique, une *structure de données* est la description d'une structure logique destinée à organiser et à agir sur des données indépendamment de la mise en œuvre effective de ces structures. Nous allons en étudier plusieurs, en commençant par les *listes simplement chainées* (ou plus simplement dans la suite de ce cours, les *listes*).
Une liste est une collection séquentielle et de taille arbitraire de données de même type : chaque élément possède, en plus de la donnée, d'un pointeur vers l'élément suivant de la liste. On peut donc représenter une liste L par la figure suivante :



Néanmoins, un tel schéma est incomplet car taille arbitraire ne signifie pas taille infinie : il est nécessaire qu'une liste se termine. Il faut donc adjoindre à cette description un élément particulier caractérisant la terminaison de la liste, et qu'on appelle le *nil* (abréviation du latin *nihil*, autrement dit, rien).



Ainsi, le type de données abstrait définissant une liste est le suivant :

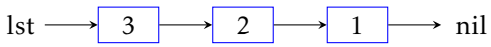
$$\boxed{\text{Liste} = \text{nil} + \text{Élément} \times \text{Liste}}$$

• Mise en œuvre en Caml

Bien que cette structure de donnée soit déjà présente en Caml (et y joue un rôle primordial), nous allons définir un nouveau type qui implémente (si elle n'existait pas déjà) la structure de liste, pour en bien comprendre les avantages et les contraintes. Bien entendu cette définition n'est que provisoire ; une fois cette construction achevée nous nous empresserons de l'oublier pour ne plus utiliser que le type déjà défini en Caml.
On définit donc le type polymorphe et récursif *'a liste* :

```
# type 'a liste = Nil | Cellule of 'a * ('a liste) ;;
Type liste defined.
```

Nous pouvons maintenant définir une première liste d'entiers, par exemple celle représentée par le schéma suivant :

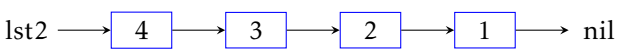



```
# let lst = Cellule (3, Cellule (2, Cellule (1, Nil))) ;;
lst : int liste = Cellule (3, Cellule (2, Cellule (1, Nil)))
```

Pour construire de nouvelles listes à partir de listes déjà créées, il peut être utile de posséder une fonction insérant un nouvel élément en tête de liste :

```
# let construire t q = Cellule (t, q) ;;
construire : 'a -> 'a liste -> 'a liste = <fun>
```

Pour définir la liste représentée ci-dessous, il suffit dès lors d'écrire :



```
# let lst2 = construire 4 lst ;;
lst2 : int liste = Cellule (4, Cellule (3, Cellule (2, Cellule (1, Nil))))
```

Inversement, il faut pouvoir récupérer les éléments d'une liste. Ceci nous amène à définir deux fonctions supplémentaires : une fonction **tete** qui retourne le premier élément d'une liste (si elle existe), et **queue** qui renvoie la liste pointée par la tête de liste :

```
# let tete = function
    | Nil           -> failwith "liste vide"
    | Cellule (t, q) -> t ;;
tete : 'a liste -> 'a = <fun>
# let queue = function
    | Nil           -> failwith "liste vide"
    | Cellule (t, q) -> q ;;
queue : 'a liste -> 'a liste = <fun>
# tete lst2 ;;
- : int = 4
# queue lst2 ;;
- : int liste = Cellule (3, Cellule (2, Cellule (1, Nil)))
```

Nous n'allons pas pousser plus loin la mise en œuvre de ce type puisque le type *'a list* existe d'ores et déjà en Caml, mais on retiendra principalement de cette construction les observations suivantes :

- les listes peuvent grossir dynamiquement, mais les éléments sont toujours rajoutés en tête de liste ;
- on ne peut accéder directement qu'à la tête de la liste.

2. Description des listes en Caml

2.1 Construction d'une liste

Les listes sont délimitées par des crochets `[` et `]`, les éléments (qui doivent être de même type) sont séparés par un point-virgule. Par exemple :

```
# [4; 3; 2; 1] ;;
- : int list = [4; 3; 2; 1]
# ['a'; 'b'; 'c'] ;;
- : char list = ['a'; 'b'; 'c']
# [4; 'a'; 3; 2; 1] ;;
Entrée interactive:
>[4; 'a'; 3; 2; 1] ;;
>^^^^^^^^^^^^^^^^
This expression has type int list, but is used with type char list.
```

L'élément nil est représenté par la liste vide `[]` et a pour type *'a list*.

L'ajout d'un élément en tête de liste est représenté par l'opérateur infixe **::** qu'on appelle *conse* (pour constructeur de liste) :

```
# 5::[4; 3; 2; 1] ;;
- : int list = [5; 4; 3; 2; 1]
# 'a'::'b'::'c'::[] ;;
- : char list = ['a'; 'b'; 'c']
```

À l'inverse, les fonctions **hd** (*head*) et **tl** (*tail*) permettent d'obtenir la tête (le premier élément de la liste) et la queue (la liste privée de son premier élément) d'une liste :

```
# hd [4; 3; 2; 1] ;;
- : int = 4
# tl [4; 3; 2; 1] ;;
- : int list = [3; 2; 1]
```

Notons que ces deux fonctions déclenchent une exception lorsqu'on essaye de les appliquer à la liste vide :

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

2.2 Filtrage et récursivité

Aux caractères utilisables dans un motif évoqués au chapitre précédent viennent s'ajouter les caractères `[ ; ]` et `::`. Par exemple, le motif **`t::q`** est reconnu par toute liste non vide, et dans la suite de l'évaluation **`t`** prendra la valeur de la tête et **`q`** celle de la queue. Il est donc facile de redéfinir à titre d'exemple les fonctions **`hd`** et **`tl`** :

```
# let hd = function                (* fonction prédéfinie en Caml *)
    | []   -> failwith "hd"
    | t::q -> t ;;
hd : 'a list -> 'a = <fun>
# let tl = function                (* fonction prédéfinie en Caml *)
    | []   -> failwith "tl"
    | t::q -> q ;;
tl : 'a list -> 'a list = <fun>
```

On prendra bien note que ces deux fonctions sont de coût constant.

Exercice. Décrire en français courant les listes reconnues par les motifs ci-dessous :

`[x]` `x::[]` `[]::x` `[1; 2; x]` `1::2::[x]` `1::2::x` `x::y::z`

Associé à la récursivité, le filtrage est le mode principal de définition d'une fonction agissant sur les listes. À titre d'exemple, nous allons passer en revue quelques exemples de fonctions agissant sur les liste, la plus-part étant prédéfinies dans le langage.

Calcul de la longueur d'une liste

```
let rec list_length = function     (* fonction prédéfinie en Caml *)
  | []   -> 0
  | t::q -> 1 + list_length q ;;
```

Cette fonction a une complexité en $\Theta(\ell)$, où ℓ désigne la longueur de la liste : en effet, si $C(\ell)$ désigne cette complexité, on dispose de la relation : $C(\ell) = C(\ell-1) + \Theta(1)$ qui conduit par télescopage à $C(\ell) = \Theta(\ell)$.

Obtention du dernier élément d'une liste

```
let rec last = function
  | []   -> failwith "last"
  | [a]  -> a
  | _::q -> last q ;;
```

Cette fonction a une complexité en $\Theta(\ell)$, où ℓ désigne la longueur de la liste.

Test d'appartenance à une liste

```
let rec mem x = function           (* fonction prédéfinie en Caml *)
  | []              -> false
  | t::_ when t = x -> true
  | _::q            -> mem x q ;;
```

Cette fonction a une complexité en $O(\ell)$, où ℓ désigne la longueur de la liste.
Notons que le principe de l'évaluation paresseuse permet d'écrire de manière équivalente :

```
let rec mem x = function
  | []    -> false
  | t::q  -> (t = x) || mem x q ;;
```

Obtention du n^e élément d'une liste

```
let rec nth n = function
  | []              -> raise Not_found
  | t::_ when n = 1 -> t
  | _::q            -> nth (n - 1) q ;;
```

Cette fonction a une complexité en $\Theta(\min(n,\ell))$, où ℓ désigne la longueur de la liste. On voit là une différence fondamentale avec le type *list* du langage Python pour lequel l'accès aux éléments est de coût constant.

Concaténation de deux listes

```
let rec concat lst1 lst2 = match lst1 with
  | []   -> lst2
  | t::q -> t::(concat q lst2) ;;
```

Cette fonction a une complexité en $\Theta(\ell_1)$ où ℓ_1 désigne la longueur de la liste de gauche.

Caml dispose de l'opérateur `@` qui est la forme infixe de cette fonction :

```
# [1; 2; 3] @ [4; 5; 6] ;;
- : int list = [1; 2; 3; 4; 5; 6]
```

Remarque. Le mot-clé **prefix** permet de transformer un opérateur infixe en opérateur préfixe ; ainsi la fonction **concat** que nous venons de définir est équivalente à la fonction **prefix** `@`.

```
# prefix @ [1; 2; 3] [4; 5; 6] ;;
-: int list = [1; 2; 3; 4; 5; 6]
```

3. Fonctionnelles agissant sur les listes

Les fonctions que nous allons étudier maintenant sont un peu plus générales que les précédentes : leur usage au sein d'un code permet d'en simplifier l'écriture et par là même la compréhension.

3.1 Les fonctions map et do_list

Étant données une fonction f de type *'a -> 'b* et une liste $[a_0; \cdots; a_{n-1}]$ de type *'a list*, la fonctionnelle **map** a pour objet de créer la liste $[f(a_0); \cdots; f(a_{n-1})]$. Sa définition est la suivante :

```
let rec map f = function          (* fonction prédéfinie en Caml *)
  | []   -> []
  | t::q -> (f t)::(map f q) ;;
```

Son type est *('a -> 'b) -> 'a list -> 'b list* et sa complexité est en $\Theta(\ell)$, où ℓ désigne la longueur de la liste, si la complexité de la fonction f est constante.

```
# map string_length ["alpha"; "beta"; "gamma"; "delta"] ;;
- : int list = [5; 4; 5; 5]
```

Exercice. Déterminer le type et ce que réalise la fonction :

```
let myst1 l = (map fst l), (map snd l) ;;
```

Étant données une fonction f de type *'a -> unit* et une liste $[a_0; \cdots; a_{n-1}]$ de type *'a list*, la fonctionnelle **do_list** a pour objet d'effectuer la séquence $f(a_0)$; $f(a_1)$; ... ; $f(a_{n-1})$ (ce qui n'a d'intérêt que si f a un effet sur l'environnement). Sa définition est la suivante :

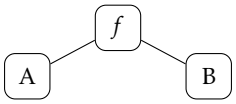
```
let rec do_list f = function      (* fonction prédéfinie en Caml *)
  | []   -> ()
  | t::q -> f t ; do_list f q ;;
```

Son type est *('a -> 'b) -> 'a list -> unit* et sa complexité est en $\Theta(\ell)$, où ℓ désigne la longueur de la liste, si la complexité de la fonction f est constante.

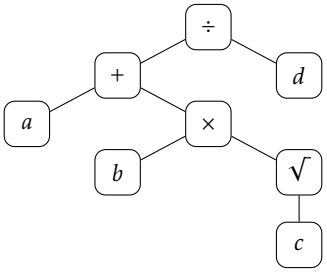
```
# do_list print_string ["alpha"; "beta"; "gamma"; "delta"] ;;
alphabetagammadelta- : unit = ()
```

3.2 Vers une tentative de généralisation : les fonctions list_it et it_list

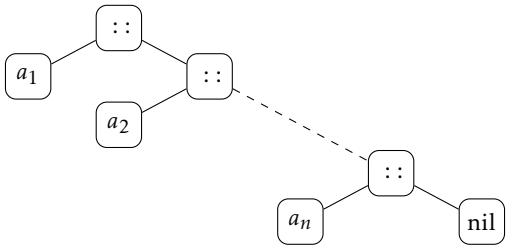
On conviendra aisément que les fonctions écrites jusqu'à présent suivent toutes peu ou prou le même schéma : un filtrage de la liste vide et une fonction à deux arguments avec pour premier la tête de la liste et pour second un appel récursif sur la queue. On peut donc envisager de généraliser cette situation. Pour ce faire, nous allons nous inspirer de la *représentation arborescente* d'une expression : sans rentrer dans les détails (nous y reviendrons au chapitre suivant), disons que les valeurs seront représentées par des feuilles et l'expression $f(a,b)$ par l'arbre :



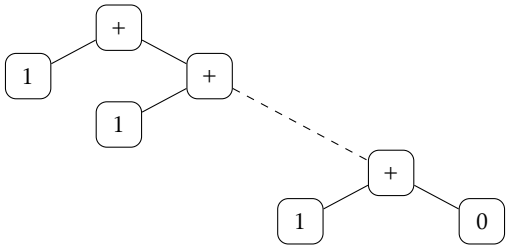
où A et B sont eux-même des arbres représentant respectivement les expressions a et b.
Par exemple, l'expression $\dfrac{a+b\sqrt{c}}{d}$ sera représentée par l'arbre :



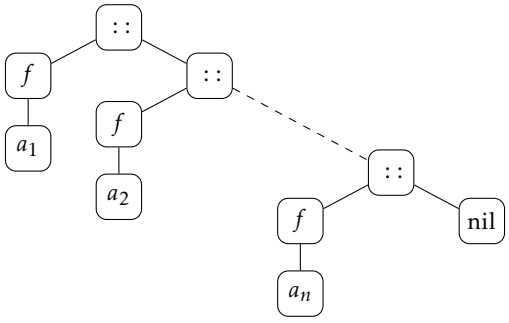
Ainsi, une liste $[a_1;a_2;\cdots;a_n]$ peut être représentée par l'arbre suivant :



On peut alors observer que calculer la longueur de cette liste revient à transformer l'arbre précédent en :

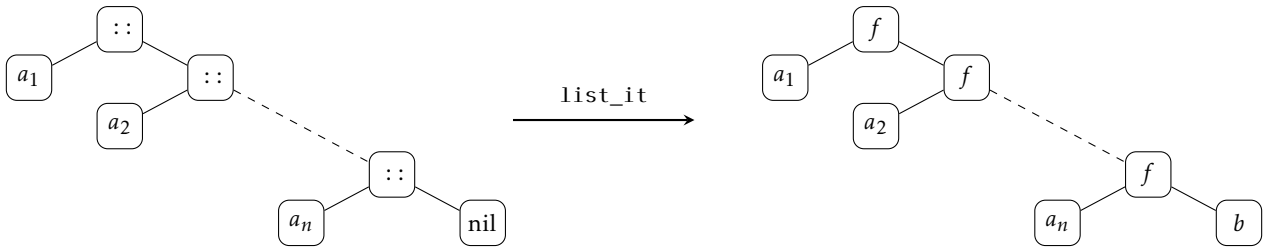


ou encore que lui appliquer la fonction `map f` revient à le transformer en :



• La fonction list_it

Il existe une fonction Caml qui réalise de telles transformations : la fonction **list_it**, dont le type est : *('a –> 'b –> 'b) –> 'a list –> 'b –> 'b*. Elle a pour objet, étant donnés une fonction $f : A \times B \to B$, une liste $[a_1;\cdots;a_n]$ d'éléments de A et un élément $b \in B$, de calculer : $f\Big(a_1, f\Big(a_2, f\Big(a_3 \ldots, f(a_n, b)\Big)\Big)\Big)$, c'est à dire d'effectuer la transformation schématisée par :



On peut alors redéfinir la fonction **list_length** en posant :

```
let list_length lst = list_it (fun a b -> 1 + b) lst 0 ;;
```

et redéfinir la fonction **map** en posant :

```
let map f lst = list_it (fun a b -> (f a)::b) lst [] ;;
```

Exercice. Deviner ce que réalisent les fonctions définies ci-dessous :

a) `let myst2 x lst = list_it (fun a b -> a::b) lst [x] ;;`

b) `let myst3 = list_it (fun a b -> a::b) ;;`

La fonction **list_it** elle-même n'est pas compliquée à définir :

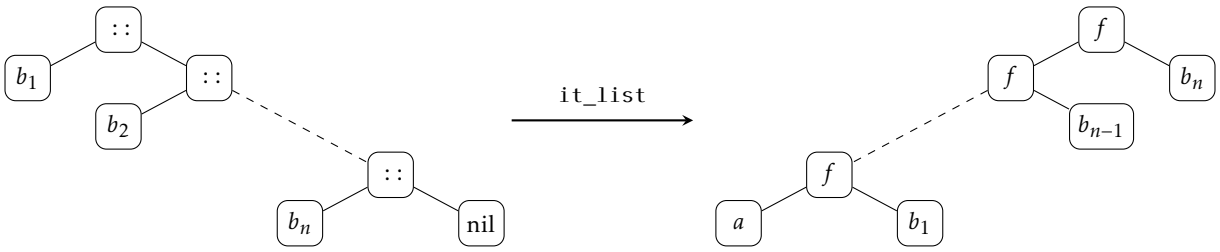
```
let rec list_it f lst b = match lst with        (* fonction prédéfinie en Caml *)
  | []   -> b
  | t::q -> f t (list_it f q b) ;;
```

• La fonction it_list

La fonctionnelle **it_list** a pour type *('a –> 'b –> 'a) –> 'a –> 'b list –> 'a* et a pour objet, étant donnés une fonction $f : A \times B \to A$, un élément $a \in A$ et une liste $[b_1; \cdots ; b_n]$ d'éléments de B, de calculer

$$f\Big(\ldots f\Big(f\Big(f(a, b_1), b_2\Big), b_3\Big)\ldots, b_n\Big),$$

c'est à dire d'effectuer la transformation schématisée par :



On peut par exemple redéfinir la fonction **do_list** de la manière suivante :

```
let do_list f = it_list (fun a b -> f b) () ;;
```

ou définir de nouveau la fonction **list_length** :

```
let list_length = it_list (fun a b -> a + 1) 0 ;;
```

Exercice. Deviner ce que réalisent les fonctions définies ci-dessous :

a) `let myst4 = it_list (prefix ^) "" ;;`

b) `let myst5 l = it_list min (hd l) (tl l) ;;`

La fonction **`it_list`** se définit très simplement de la manière suivante :

```
let rec it_list f a = function                (* fonction prédéfinie en Caml *)
  | []   -> a
  | t::q -> it_list f (f a t) q ;;
```

Remarque. Ces deux fonctions **`list_it`** et **`it_list`** sont des fonctions génériques qui existent dans la plupart des langages fonctionnels (où elles sont en général connues sous le nom de *fold right* et *fold left*). Nous aurons l'occasion de constater que la majorité des fonctions agissant sur les listes suivent un schéma récursif semblable, et qu'en conséquence de quoi peuvent être définies à l'aide de l'une de ces fonctions (voire les deux). L'utilisation de ces fonctions génériques facilite ainsi la preuve de validité des programmes. Elles ne sont pas d'un abord facile mais leur bon usage permet de notablement simplifier certains codes.

4. Exercices

4.1 Parcours d'une liste

Exercice 1 Écrire une fonction qui retourne l'avant-dernier élément d'une liste, s'il existe.

Exercice 2 En procédant par récursivité et filtrage, définir une fonction calculant la somme des éléments d'une liste d'entiers.
En utilisant l'opérateur **`it_list`**, définir une fonction calculant le produit des éléments d'une liste d'entiers.

Exercice 3 Les fonctions suivantes sont prédéfinies en Caml. En procédant par récursivité et filtrage, en donner la définition.

a) **`exists`** est une fonction de type *('a -> bool) -> 'a list -> bool* qui détermine s'il existe un élément de la liste vérifiant une propriété donnée.

b) **`for_all`** est une fonction de type *('a -> bool) -> 'a list -> bool* qui détermine si tous les éléments de la liste vérifient une propriété donnée.

Les définir ensuite à l'aide de l'opérateur **`it_list`**.

Exercice 4 Rédiger une fonction Caml calculant la liste de tous les préfixes d'une liste donnée. Par exemple :

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

Exercice 5 Étant données une fonction de type *'a -> bool* et une liste de type *'a list*, définir une fonction calculant :

a) le n^e élément de la liste vérifiant la propriété (s'il en existe) ;

b) le dernier élément de la liste vérifiant cette propriété.

Exercice 6 L'image *miroir* d'une liste $[a_1;a_2\cdots;a_n]$ est la liste $[a_n;a_{n-1};\cdots;a_1]$ dans laquelle l'ordre des éléments a été inversé.

a) En procédant par récursivité et filtrage, définir une fonction **`rev`** réalisant cette transformation (vous pouvez utiliser l'opérateur de concaténation `@`). Montrer que le coût de cette fonction est quadratique.

b) Définir maintenant la fonction **`rev`** en utilisant l'opérateur **`it_list`**. En évaluer le coût.

c) Rédiger enfin une troisième version de cette fonction, de coût linéaire, et n'utilisant pas l'opérateur **`it_list`**.

Notez que cette fonction est prédéfinie en Caml.

Exercice 7 Écrire une fonction `rotg` qui fait tourner une liste d'un cran vers la gauche. Par exemple, `rotg [1; 2; 3; 4]` renverra la liste `[2; 3; 4; 1]`. Quel est le coût de votre algorithme ?
Mêmes questions pour la fonction `rotd` qui tourne une liste d'un cran vers la droite.

4.2 Insertion et suppression

Exercice 8 Définir une fonction de type *int –> 'a list –> 'a list* qui supprime le n^e élément d'une liste, puis une fonction de type *'a –> int –> 'a list –> 'a list* qui insère un élément dans une liste à la n^e position.

Exercice 9 Déterminer le type et préciser le rôle de la fonction suivante :

```
let rec myst f = function
  | []            -> []
  | t::q when f t -> t::(myst f q)
  | _::q          -> myst f q ;;
```

Exercice 10 On souhaite écrire une fonction `purge` qui, appliquée à une liste, retourne une liste dans laquelle les doublons ont été éliminés (on rappelle que la fonction prédéfinie `mem` détermine si un élément appartient ou pas à une liste).

a) Écrire une première version de `purge` dans laquelle seule la dernière occurrence de chaque doublon sera conservée. Par exemple, `purge [1; 2; 3; 1; 4; 3; 1]` renverra comme résultat `[2; 4; 3; 1]`.

b) Écrire une seconde version de `purge` dans laquelle seule la première occurrence de chaque doublon sera conservée. Cette fois, `purge [1; 2; 3; 1; 4; 3; 1]` renverra le résultat `[1; 2; 3; 4]`.

Exercice 11 On souhaite représenter un ensemble par une liste, chaque élément de l'ensemble ne devant apparaître qu'une seule fois dans la liste, à un emplacement arbitraire.

a) Définir une fonction `intersection` qui calcule l'intersection de deux ensembles. Évaluer son coût en fonction des cardinaux de ces ensembles.

b) Définir de même l'union et la différence symétrique de deux ensembles.

c) Rédiger enfin une fonction `egal` qui détermine si deux ensembles sont égaux.

Évaluer le coût de chacune de ces fonctions.