Premiers programmes en OCaml

Guillaume Rousseau
MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

17 janvier 2025

Introduction

OCaml est un langage de programmation développé dans les années 1980-1990, en France, par des chercheurs et chercheuses de l'INRIA [1]. C'est un langage fonctionnel, présentant des liens forts avec la logique et la théorie de la déduction, ce qui en fait un langage couramment utilisé dans ces domaines de recherche. Il a par exemple été utilisé pour écrire le logiciel **Coq**, un assistant de preuve permettant de fournir des démonstrations formelles de théorèmes, et de montrer la correction de programmes. Coq et OCaml ont été utilisés pour écrire et prouver le compilateur C **CompCert** : ce compilateur génère des programmes à peine plus lents que GCC, mais est garanti sans bug [2].

Le langage donne beaucoup de garanties de sûreté, notamment comparé au C, mais reste relativement performant, il est donc parfois utilisé en industrie pour les systèmes critiques (aéronautique, nucléaire, ...). Facebook utilise également OCaml en interne, et a même développé son propre langage, Reason, mélangeant les notions clés d'OCaml avec des éléments de syntaxe de javascript.

Langage impératif, langage fonctionnel Le C est un langage ***impératif***. Cela signifie qu'un programme C est constitué d'instructions, qui disent à la machine ce qu'elle doit faire. En programmation impérative, on parle d'***état du programme*** (la mémoire, les variables), et les instructions servent à modifier l'état du programme.

OCaml est un langage ***fonctionnel***. La programmation fonctionnelle est un autre paradigme de programmation, dans laquelle un programme est une immense ***expression*** mathématique que l'on évalue. Dans une telle expression, on peut faire intervenir des fonctions, et les fonctions sont des valeurs au même titre que les entiers, les flottants ou les chaînes de caractères.

Notons qu'OCaml est un langage fonctionnel **impur**, car il permet aussi d'utiliser des éléments de programmation impérative. Certains langages, comme Haskell, sont purement fonctionnels !

1. Institut national de recherche en informatique et en automatique
2. Si l'on suppose que Coq est lui-même correct !

1 Premiers programmes

Un programme OCaml peut être compilé comme du C, ou interprété comme du Python.

ocamlc est le compilateur usuel utilisé pour OCaml. Les fichiers OCaml auront pour extension ".ml". Pour compiler un programme "mon_programme.ml", on tape :

```
ocamlc mon_programme.ml -o nom_executable
```

Comme avec gcc, si l'on ne précise pas de nom d'exécutable, celui-ci s'appellera `a.out`.

Pour lancer l'interpréteur OCaml, on peut utiliser la commande ***ocaml***, mais on préfèrera utiliser **utop** qui est un interpréteur plus pratique, avec un historique, de l'auto-complétion, etc... Pour indiquer la fin d'une expression dans l'interpréteur, on utilise `;;`.

Remarque 1. Dans utop, à tout moment, on peut taper CTRL+C pour arrêter la ligne actuelle (en cas de boucle infinie par exemple). On peut aussi taper CTRL+D pour quitter utop.

Question 1. Lancer utop

Question 2. Taper les expressions suivantes (sans oublier `;;` à chaque fois) :

- `5`
- `5.67`
- `true`
- `'h'`
- `"bonjour"`
- `()`

On remarque que lorsqu'on tape une valeur, OCaml détecte tout seul le type, et nous l'affiche. En Ocaml, les types de base sont `int`, `float`, `bool`, `char`, `string` et `unit` :

Comparons avec le C :

- `bool` n'est pas compatible avec les entiers, c'est un type à part
- `string` est un type totalement distinct de `char`.
- `unit` est l'équivalent du type `void`. Il possède un unique élément, noté `()` (et aussi prononcé "unit").

A Opérateurs

Comme tous les langages de programmation, OCaml possède des opérateurs sur les types de base : additions, soustractions, tests d'égalité, d'inégalité, opérations booléennes, etc...

Question 3. Taper les expressions suivantes :

- `3+8`
- `5 = 2+3`
- `9.36 < 1.2`
- `not true`
- `true && false`
- `true || false`
- `"bonjour "^ "tout le monde"`

Question 4. Taper les expressions suivantes :

— `3.65 + 9`
— `3.65 + 2.35`

Que constate-t'on ?

En OCaml, le typage est strict. En particulier, les `int` et les `float` ne peuvent pas être mélangés. Pour additionner des flottants, on doit utiliser l'opérateur `+.` et idem pour la multiplication, la division, la soustraction.

Question 5. Taper les expressions suivantes :

— `2.0 +. 0.5`
— `5. /. 2.`

B Arbre de syntaxe

Un programme OCaml est une expression mathématique, que l'on peut représenter sous une forme graphique appelée "arbre de syntaxe". Voici quelques exemples d'expressions et leurs arbres de syntaxe :

On peut donc définir formellement une expression OCaml comme suit :

Définition 1. Une expression OCaml est :

— Une constante d'un type de base (1, 2, false, true, 3.48, 0.23, ...)
— Un opérateur binaire appliqué sur deux expressions OCaml de types compatibles (`1 + 2`, `(5-3)+ 4`, `(1 = 2)||false`, ...)
— Un opérateur unaire appliqué sur une expression OCaml de type compatible (`-12`, `-(2.32 +. 0.5)`, `not (2 < 5)`)

Cette définition est partielle, on viendra la compléter au fur et à mesure que l'on rencontre de nouveaux éléments de syntaxe d'OCaml.

Une première remarque : cette définition est "récursive" car on définit ce qu'est une expression en fonction des expressions. La terminologie exact est que c'est une définition **inductive** (Cf prochain chapitre).

Notons aussi qu'avec cette définition, il est ambigu de savoir si `1 + 2 + 3` veut dire `(1+2)+3` ou bien `1+(2+3)` : il y a deux arbres de syntaxe possibles. Par associativité de l'addition, on peut choisir arbitrairement l'un ou l'autre. Si l'on considère l'expression `1 + 2 * 3`, il n'y a pas ambiguïté grâce à la précédence des opérateurs : la multiplication est prioritaire sur l'addition, l'expression doit donc être comprise comme `1 + (2 * 3)`.

C Fonctions

Revenons au mélange d'entiers et de flottants. Pour passer de l'un à l'autre, il faut utiliser des fonctions : `float_of_int` et `int_of_float`. En OCaml, **L'application de fonction se fait en écrivant la fonction puis l'argument, séparés d'un espace, sans besoin de parenthèses**.

Question 6. Taper :

- `float_of_int 3`
- `int_of_float 6.3`
- `float_of_int 3 +. 6.21`

On remarque que la dernière expression est comprise comme `(float_of_int 3)+. 6.21` et pas `float_of_int (3 +. 6.21)`. On dit que l'application de fonction est **prioritaire** sur l'addition. L'application de fonction est en réalité prioritaire sur (presque) tout.

En OCaml, **les parenthèses ne servent pas à faire des appels de fonction !** Les parenthèses servent uniquement à **regrouper une expression** pour la rendre "d'un seul tenant". Il est donc inutile d'écrire `float_of_int(5)` car 5 est déjà d'un seul tenant, on peut donc écrire `float_of_int 5` sans parenthèses. En revanche, lorsque l'on écrit `float_of_int (5 + 2)`, les parenthèses sont nécessaires.

Définition 2. On dit qu'une expression est **atomique** si c'est :

- une constante (1, 2, 3.24, true, ...)
- un identifiant (`x`, `y`, ...)
- une expression entre parenthèses (`(2 + 3), ...`

OCaml étant un langage fonctionnel, les **fonctions** sont des valeurs comme les autres :

Question 7. Taper les expressions suivantes :

- `int_of_float`
- `float_of_int`

Pour la dernière expression, l'interpréteur affiche :

```
- : int -> float = <fun>
```

Dans un type, la flèche `->` se lit "flèche" ou "donne". Une fonction de type `A -> B` prend en entrée un élément de type A et calcule un élément de type B. Par exemple, `float_of_int` prend un int et donne un float.

On peut donc étendre notre définition des expressions OCaml :

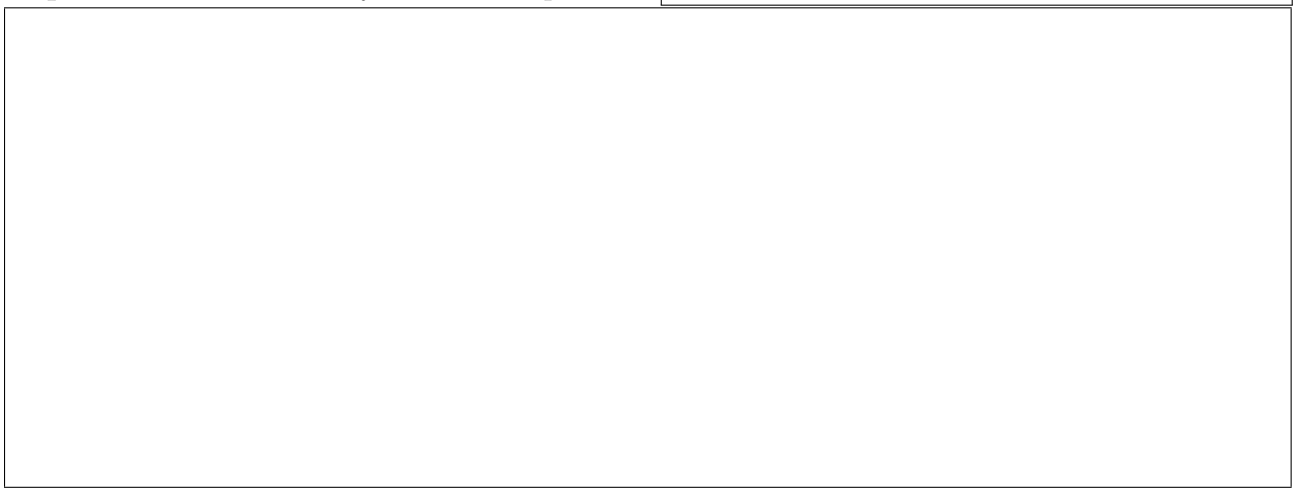
Définition 3. Sont également des expressions OCaml :

- Une expression OCaml entre parenthèses
- Deux expressions OCaml atomiques mises l'une après l'autre.

Le dernier cas correspond à une **application de fonction**.

Dans les arbres de syntaxe, on pourra par exemple représenter les applications de fonction par le mot **App**. On pourra traiter une application comme une sorte d'opérateur binaire.

Exemple 1. Voici l'arbre de syntaxe de l'expression `float_of_int (3 + int_of_float (2.5 *. 2.1))`



En OCaml, on peut créer ses propres fonctions. Par exemple, la fonction $f : x \mapsto x + 1$:

```
1 (fun x -> x + 1)
```

Pour appliquer cette fonction à une autre expression, par exemple $(5 + 3)$, on écrira donc :

```
1 (fun x -> x + 1) (5 + 3)
```

ce qui affiche bien 9. OCaml vérifie toujours que le type de l'argument d'une fonction correspond bien au type attendu. Par exemple, l'expression suivante génèrera une erreur :

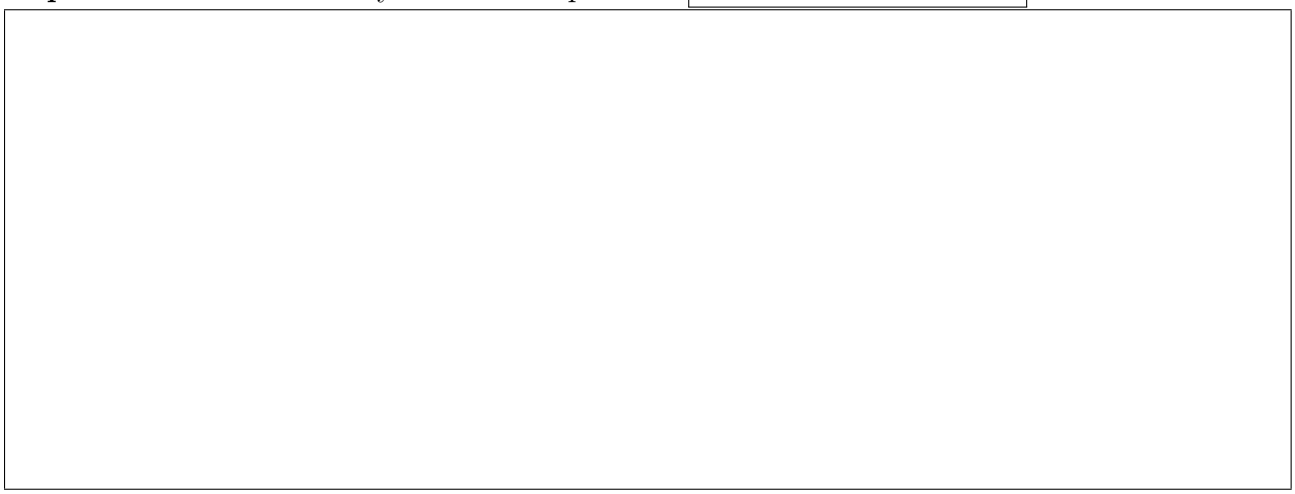
```
1 (fun x -> x + 1) 0.5
```

On peut donc mettre à jour notre définition des expressions :

Définition 4. Si A est une expression, alors `fun x -> A` est également une expression. Une telle expression s'appelle une ***abstraction***.

Dans les arbres de syntaxe, on pourra représenter les abstraction par le mot **Fun**.

Exemple 2. Voici l'arbre de syntaxe de l'expression `(fun x -> x + 1)(5 + 3)`



D Évaluation des expressions, environnements

Dans un langage impératif, on dit que l'on exécute une suite d'instructions. Dans langage fonctionnel comme OCaml, on dit que l'on **évalue** une expression. Évaluer une expression, c'est la transformer en une valeur. Par exemple, lorsque l'on transforme `3 + 2` en 5, on a évalué l'expression

Considérons l'expression `3 + 2`. Cette expression est de type "un opérateur binaire appliqué sur deux expressions". On commence donc par évaluer les deux sous-expressions : `3` et `2`. Ces deux expressions sont des constantes, elles s'évaluent donc en elles-même. On applique alors l'opérateur + sur les deux valeurs obtenus, on obtient 5.

Considérons maintenant l'expression `(3 + 2)* (1 - 4)`. Cette expression est de type "un opérateur binaire appliqué sur deux expressions". On commence donc par évaluer les deux sous-expressions : `(3 + 2)` et `(1 - 4)`. Pour évaluer la première, on fait comme précédemment, on obtient alors 5, et de manière analogue, la deuxième expression s'évalue en -3. Alors, l'expression initiale s'évalue en -15.

On peut donc donner un algorithme simple permettant d'évaluer une expression OCaml :

Algorithme 1 : Évaluation

Entrée(s) : Un arbre de syntaxe OCaml, c'est à dire une expression E
Sortie(s) : La valeur obtenue en évaluant E

```
si E est une constante c alors
    retourner c
si E est de la forme E_1 + E_2 alors
    v1 ← Évaluer E1;
    v2 ← Évaluer E2;
    retourner v1 + v2
si E est de la forme E_1 - E_2 alors
    v1 ← Évaluer E1;
    v2 ← Évaluer E2;
    retourner v1 − v2
// Ainsi de suite
```

Intéressons nous maintenant à l'expression `(fun x -> x+1)(5+3)` Étudions la manière dont OCaml évalue cette expression. Pour commencer, on évalue 5+3 ce qui donne 8. Ensuite, on doit appliquer la fonction à l'argument 8. Pour cela, on stocke quelque part l'information "`x` vaut 8", et on évalue le corps de la fonction : `x + 1`. On doit donc évaluer `x`, et avec l'information que l'on a stocké, on sait que le résultat est 8. Ainsi, on évalue $x + 1$ à 9, et c'est le résultat de l'expression initiale.

Ainsi, lorsque l'on évalue une expression OCaml, on doit stocker des informations sur les différentes variables. L'objet servant à stocker ces informations s'appelle un **environnement** ou un **contexte**.

Exercice 1. Dessiner l'arbre de syntaxe de l'expression `2 * 3 + (2 - 1 / 3)`, et lui appliquer l'algorithme d'évaluation.

Définition 5. Un ***environnement*** est un ensemble d'associations de la forme $x \mapsto v$, où x est un identifiant (i.e. un nom de variable) et v une valeur.

Étant donné E une expression et ρ un environnement, on peut **évaluer** E dans ρ. Modifions notre algorithme d'évaluation pour prendre en compte l'environnement :

Algorithme 2 : `Evaluer`(E, ρ)

Entrée(s) : E un arbre de syntaxe, ρ un environnement
Sortie(s) : La valeur obtenue en évaluant E dans ρ

si E ***est une constante*** c **alors**
  retourner c
si E ***est une variable*** v **alors**
  retourner ***la valeur associée à*** v ***dans*** ρ
si E ***est de la forme*** `(fun x -> E_1) E_2` **alors**
  $v_2 \leftarrow$ évaluer E_2 dans ρ;
  $\rho' \leftarrow \rho$ mis à jour avec `x` $\mapsto v_2$;
  retourner *Evaluer*(E_1, ρ')
si E ***est de la forme*** `E_1 + E_2` **alors**
  $v_1 \leftarrow$ `Evaluer`(E_1, ρ);
  $v_2 \leftarrow$ `Evaluer`(E_2, ρ);
  retourner $v_1 + v_2$
`// Ainsi de suite pour les autres opérateurs binaires...`

Exercice 2. Pour chacune des expressions suivantes, dessiner l'arbre de syntaxe, et appliquer l'algorithme d'évaluation à partir d'un contexte vide.

```
1 (fun a -> a +. 3.) 6.;;
2 (fun x -> x + (fun y -> x*y)(x+1)) ((fun x->6-x) 3);;
```

E Fonctions d'ordre supérieur

Considérons la fonction suivante :

```
1 fun f -> 2 * f 2
```

Si l'on tape l'expression précédente dans utop, l'interpréteur nous dit que le résultat est de type `(int -> int)-> int`. Cette fonction prend donc en paramètre une fonction de type `int -> int`, et renvoie un entier. Par exemple :

```
1 (fun f -> 2 * f 2) (fun x-> x + 1) ;;
```

Question 8. Dessiner l'arbre de l'expression précédente et appliquer l'algorithme d'évaluation.

Question 9. En langage naturel, que fait la fonction `fun f -> 2 * f 2` ?

Définition 6. On dit qu'une fonction est ***d'ordre supérieur*** si elle prend en argument une fonction ou renvoie une fonction.

Exercice 3. Décrire les fonctions suivantes en français.

```
1 fun f -> (fun x -> f (f x)) ;;
2 fun f -> (fun x -> (f (x +. 0.00001) -. f x) /. 0.00001) ;;
3 fun k -> (fun y -> k*y) ;;
```

F Syntaxe “let in”

Il va vite être compliqué de construire des expressions lisibles. Afin de rendre le code plus digeste, on utilise en OCaml une syntaxe qui **ressemble** aux affectations de variable des programmes impératif. Cette syntaxe est construite avec les deux mots-clés `let` et `in`. Par exemple :

```
let x = 2 in
x + 3;;
```

Pour évaluer une expression de la forme `let x = e1 in e2`, on évalue `e1` en une valeur v_1 puis on évalue `e2` dans l'environnement ($\texttt{x} \mapsto v_1$). La preuve que ce n'est pas une affectation de variable est que si l'on essaie d'évaluer x après avoir tapé cette expression, on obtient une erreur car x n'est pas connu. En réalité, x n'existe dans le contexte que lors de l'évaluation de la partie droite du ***let in***.

Remarque 2. `let x = e1 in e2` est un raccourci pour dire `(fun x -> e2)e1;;`. Lorsqu'une syntaxe sert simplement à raccourcir le code, ou à faciliter son utilisation, sans rajouter de fonctionnalité, on dit que c'est du ***sucre syntaxique***.

On peut imbriquer les ***let in*** comme on veut :

Question 10. Taper les expressions suivantes :

```
let x = 3 in
let y = 5 in
x + y;;

let x =
  let t = 3.2 in
  t *. 5.0
in (x < 31.0);;
```

Lorsque l'on écrit `let x = A in B`, on peut voir ça comme rajouter au contexte une valeur pour x, avant de calculer B. Quand on met plusieurs let in à la suite, par exemple

```
let x = 3 in
let y = 5 in
x + y;;
```

on construit le contexte en y ajoutant petit à petit des informations sur les variables.

Dans les programmes OCaml et dans l'interpréteur, il existe un contexte global. Pour rajouter une valeur au contexte global dans l'interpréteur, il suffit de remplacer le “in” par “;;”.

Question 11. Taper le code suivant dans utop :

```
let x = 3;;
let y = 5;;
x + y ;;
let f = fun z -> (x + z);;
f 1;;
let x = 50;;
f 1;;
```

Question 12. Que remarque-t'on avec les 4 dernières lignes ?

Remarque 3. ***let*** signifie “soit” en anglais, donc `let x = ... ;;` se dirait en français “soit x égal à ...”. Cette syntaxe est donc inspirée du vocabulaire mathématique.

Pour les fonctions, il existe un deuxième niveau de sucre syntaxique, qui consiste à éliminer le mot clé `fun` et la flèche.

Question 13. Taper le code suivant :

```
1 let f = fun x -> x * x ;;
2 let g x = x * x;;
```

Regardons la fonction suivante :

```
1 let add x = fun y -> x + y;;
```

cette fonction est de type `int -> (int -> int)`. Autrement dit, elle prend en entrée un entier, et renvoie une fonction. Pour k entier, `add k` est la fonction qui ajoute k à son argument. Donc, on peut écrire :

```
1 let f = add 3;;
2 let x = f 5;;
```

ou même :

```
1 let x = (add 3) 5;;
```

En OCaml, le parenthésage se fait automatiquement à gauche. Autrement dit, les deux expressions suivantes sont équivalentes :

```
1 (add 3) 5;;
2 add 3 5;;
```

Reprenons la fonction `add`. On peut utiliser un nouveau niveau de sucre syntaxique et la définir comme suit :

```
1 let add x y = x + y;;
```

Formellement, cela veut dire que la fonction add prend un argument `x`, et renvoie une fonction qui prend elle même un argument `y`, et qui renvoie $x+y$. Informellement, on dira que `add` prend deux arguments, x et y. Cependant, c'est un ***abus*** de langage !

G Type d'une fonction

Pour une fonction f définie par `let f x1 x2 ... xn = e`, OCaml donne une ***signature*** à f, qui correspond aux types des paramètres et de la valeur calculée. Par exemple :

```
1 let ma_fonction f x y = f (x + y) + f(x - y);;
```

Cette ligne affiche :

```
val ma_fonction : (int -> int) -> int -> int -> int = <fun>
```

On peut lire cette ligne comme :

— `ma_fonction` prend 3 paramètres :

1. Une fonction de type `int -> int`
2. Une valeur de type `int`
3. Une valeur de type `int`

— `ma_fonction` renvoie une valeur de type `int`

H Typage

On a pu remarquer depuis le début du chapitre qu'OCaml peut deviner les types des expressions, grâce à un système appelé **l'inférence de type**. Écrivons le code de la fonction ***identité***, qui renvoie son entrée :

```
let f x = x;;
```

Le type de cette fonction est `'a -> 'a`. Cette notation un peu particulière veut dire : "pour tout type α, on peut donner à la fonction le type $\alpha \to \alpha$". Contrairement au C, en OCaml, une fonction peut prendre et renvoyer plusieurs types totalement différents. Par exemple, la fonction identité peut prendre n'importe quel type et renvoie une valeur du même type que son entrée.

Définition 7. On dit qu'une fonction est **polymorphe** si elle peut prendre en entrée et renvoyer plusieurs types.

On dit qu'OCaml utilise le ***polymorphisme*** de type.

Introduisons une nouvelle syntaxe : `(x1, x2, ..., xn)`. En OCaml, comme en Python, on peut créer des ***tuples***. Regardons le type des tuples :

Question 14. Taper les expressions suivantes et regarder leur type :

```
let p1 = (1, 2, 3) ;;
let p2 = (1, 'a', "toto", 4);;
let diago x = (x, x);;
```

Si $e_1, e_2, \ldots, e_n$ sont des expressions de types respectifs `t1, t2, ..., tn`, alors le tuple $(e_1, e_2, \ldots, e_n)$ est de type `t1 * t2 * ... * tn`. On dit que c'est un type ***produit***.

Pour utiliser un tuple, il est nécessaire de le ***déstructurer***. Cela signifie que l'on va extraire du tuple les différentes composantes. Par exemple :

```
let p1 = (1, 2, 3) ;;
let (x, y, z) = p1;;
```

On a ***déstructuré*** `p1` en utilisant un tuple de variables ayant la même ***structure*** : `(x, y, z)`. Lorsque l'on déstructure ainsi un tuple, ou n'importe quelle autre type (cf plus loin), on parle de ***let déstructurant***.

On peut également déstructurer des tuples directement dans les paramètres d'une fonction. Par exemple, les deux fonctions suivantes sont équivalentes :

```
let echange1 p =
  let (x, y) = p in (y, x)

let echange2 (x, y) = (y, x)
```

Il faut bien noter que les deux fonctions ont la même signature : `('a * 'b)-> ('b * 'a)`. Ces deux fonctions prennent donc un seul argument, de type `('a * 'b)`.

I Récapitulatif

Pour faire des commentaires en OCaml, on les entoure par `(* *)`. Par exemple :

```
(* produit de x et y *)
let mul x y = x * y
```

Revoyons toutes les notions vues jusqu'ici :

— Les types de base : int, float, bool, char, string, unit

— Les opérations sur ces types : comme en C. Le ***non*** booléen se note `not` et les opérateurs flottants nécessitent un point : `+.` au lieu de `+` et idem pour les autres.

Type t	Opérateurs binaires pour t	Opérateurs unaires pour t	Type de l'expression composée
`int`	`+`, `-`, `*`, `/`, `mod`	`-`	`int`
`float`	`+.`, `-.`, `*.`, `/.`, `**` (puissance)	`-.`	`float`
Tous	`>`, `>=`, `=`, `<`, `<=`, `<>` (“différent de”)		`bool`
`bool`	`&&`, `||`	`not`	`bool`
`char`			
`string`	`^` (concaténation)		`string`

— On peut ajouter une variable au contexte globale avec `let x = ... ;;` et ajouter une variable dans un contexte local avec `let x = ... in ...`

— On peut définir des fonctions avec la syntaxe suivante :

```
let f x1 x2 ... xn =
  ...
```

— On applique une fonction avec :

```
f e1 e2 ... ek
```

où `e1, e2, ... ek` sont des expressions ayant des types compatibles avec la signature de la fonction `f`.

— Les parenthèses servent à encadrer les sous-expressions **et pas à appliquer les fonctions**

Voici un exemple de programme mettant en oeuvre ces différentes notions :

```
let x = 3;;
let y = 5;;
let z = 3 * 5 ;;

(* composée de f et g, i.e. la fonction h
   telle que h(x) = f(g(x)) *)
let composition f g =
  fun x -> f (g x);;

(* double de x *)
let double x = 2*x;;

let u = composition double int_of_string ;;
let a = u "12";; (* vaut 24 *)

let quadrupler = composition double double;;
let b = quadrupler 5;; (* vaut 20 *)
```

Quiz Pour chacune des expressions suivantes, prédire son type et sa valeur :

```
(* Q1 *)
let g x = 2*x in
let f x = g x + 1 in
f 5 + f 3;;


(* Q2 *)
let double x = 2*x in
let triple x = 3*x in
double (triple 5);;

(* Q3 *)
let s = "bonjour " in
let saluer x = s ^ x ^ " !!" in
let s = "salut " in
saluer "Jérémy";;

(* Q4 *)
let f x y z = (x z) (y z) in
let g x = let x = x - 1 in x * x in
let h x = fun y -> let x = y in x+1 in
f h g 3;;

(* Q5 *)
let u f (x, y) = f x y in
let g = u (fun x -> (fun y -> x y)) in
g ((fun a -> a+3), 5);;
```

2 Éléments de base d'OCaml

Étudions plusieurs éléments du langage OCaml. Pour chaque nouvel élément, on regardera :

— Un exemple simple ;

— La syntaxe ;

— Le typage, c'est à dire les règles à appliquer pour typer l'expression ;

— La sémantique, c'est à dire la manière dont on évalue la valeur de l'élément.

A If-then else

Comme tous les langages, OCaml dispose d'une construction ***if-then-else*** :

```
(* Renvoie le maximum entre x et y *)
let maximum x y =
  if x < y then y
  else x

(* applique un ET logique sur a et b deux booléens*)
let logic_and a b =
  if a then b else false
```

La syntaxe est la suivante :

```
if E1 then E2 else E3
```

où E1, E2 et E3 sont des expressions.

Pour le typage, l'expression $E1$ doit être du type `bool`, et les expressions $E2$ et $E3$ doivent être du même type T, et alors l'expression totale est de type T également.

Pour la sémantique, voici comment évaluer la valeur de `if E1 then E2 else E3` :

1. Évaluer E1 en une valeur v_1
2. Si v_1 est `true`, évaluer E2
3. Sinon, évaluer E3

Comme les deux branches du if-else doivent avoir le même type, on ne peut pas avoir de if sans else. Une exception : le type `unit` :

```
if true then ();;
if x < y then print_int x;;
```

B Matching

Les mots-clés ***match*** et ***with*** permettent de faire des disjonctions de cas plus puissantes qu'un simple if-else. Voyons quelques exemple :

```
(* Renvoie le nom de x si c'est un chiffre *)
let nom x = match x with
  | 0 -> "Zero"
  | 1 -> "Un"
  ...
  | 9 -> "Neuf"

(* renvoie true si x vaut 0, false sinon *)
let is_zero x = match x with
  | 0 -> true (* si x vaut 0, alors true *)
  | y -> false (* si x vaut n'importe quoi d'autre, alors false *)


(* Première position d'un 0 du triplet t. -1 si t n'a aucun 0 *)
let zero_pos t = match t with
  | (0, y, z) -> 0 (* si la 1ere composante vaut 0 *)
  | (x, 0, z) -> 1 (* si la 2eme composante vaut 0 *)
  | (x, y, 0) -> 2 (* si la 3eme composante vaut 0 *)
  | u -> -1 (* Tous les autres cas *)

(* Renvoie le produit des composantes de p si aucune n'est nulle, sinon la somme *)
let prod_or_sum p = match f 0, f 1 with
  | (0, y, z) -> y + z
  | (x, 0, z) -> x + z
  | (x, y, 0) -> x + y
  | u -> x * y * z
```

La syntaxe précise de cette construction est :

```
match E with
| M1 -> E1
...
| Mn -> En
```

où E, E1, ... En sont des expressions, et où M1, ..., Mn est un **motif**.

Définition 8. Un ***motif*** est (définition temporaire) :

— Soit une constante

— soit une variable

— soit un tuple de motifs

Toutes les variables d'un motif doivent être distinctes.

Un motif représente un **squelette de valeur**. Par exemple, `x` est un motif qui veut dire "n'importe quelle valeur", et `(x, (y, 0), z, (t, u, 5))` est un motif signifiant "un quadruplet, dont :

1. le premier membre est n'importe quoi,
2. le deuxième est un couple dont la deuxième composante est nulle
3. le troisième est n'importe quoi
4. le quatrième est un triplet dont la troisième composante est 5

Ainsi, si l'on compare ce dernier motif avec `(3, (2, 0), 3, (7, 8, 0))`, le motif et la valeur correspondent, et alors `x` prend la valeur 3, `y` prend la valeur 2, etc...

Attention, les termes suivants ne sont pas des motifs :

```
| 1+x (* pas d'opérateur autorisé *)
|(x, x) (* pas de variable en double *)
```

Le typage d'un match with est comme suit : E peut être de n'importe quel type, du moment que le type respecte les motifs M1, ... Mn. Les E1, ..., En doivent être d'un même type T, et alors l'expression totale est de type T également. Pour chaque couple Mk, Ek, il faut que les types des variables de Mk soient cohérents dans Ek.

La sémantique de cette construction est comme suit. Pour évaluer l'expression :

```
match E with
| M1 -> E1
...
| Mn -> En
```

il faut :

1. Évaluer E en une valeur v
2. Comparer v avec les motifs M1, ..., Mn, dans l'ordre
3. Lorsqu'un motif Mk correspond, on évalue l'expression Ek correspondante, en ayant ajouté au contexte les variables du motif Mk en les faisant correspondre à v.

Exemple 3. Voyons en détail comme est évaluée l'expression suivante :

```
match (1, (2, 1+2), 3+1) with
| (0, _, _) -> 0
| (x, (y, 0), z) -> (x - y) * z
| (x, (y, _), z) -> (x + y) * z
```

La variable `_`, appelée "underscore", joue un rôle particulier : elle peut être présente plusieurs fois dans un motif, mais ne figurera pas dans le contexte. On appelle ce motif particulier un ***joker***, ou ***wildcard*** en anglais, il sert donc à ignorer ou à jeter à la poubelle des parties de la valeur matchée. Par exemple :

```
(* Calcule x*y *)
let mult x y = match x, y with
| 0, _ -> 0
| _, 0 -> 0
| _ -> x*y
```

Variables et motifs Un motif est un squelette, il décrit donc une **forme** mais ne contient pas de valeur. Par exemple :

```
let equal x y =
  match x with
  | y -> true
  | _ -> false
```

La fonction ci-dessus n'est pas correcte. En effet, le `y` dans le premier motif n'a aucun lien avec le `y` en paramètre. On pourrait remplacer le `y` du motif par n'importe quel identifiant, et même par un underscore :

```
let equal x y =
  match x with
  | _ -> true
  | _ -> false
```

Il est alors clair que cette fonction renvoie **toujours** true !

A retenir : on ne peut **pas** utiliser une variable préexistante dans un motif pour comparer la valeur du match à la valeur de cette variable.

Matching incomplet Lorsqu'un match with ne couvre pas tous les cas possibles du type concerné, on dit qu'il est incomplet, ou non-exhaustif. Par exemple :

```
let f x = match x with
| 0 -> 1
| 2 -> 3
```

Si l'on essaie d'évaluer cette expression, OCaml va raler, et dire que le matching n'est pas exhaustif. Il donnera même un exemple de valeur qui n'est pas matchée ! On doit ***toujours*** couvrir tous les cas dans un match with. Cependant, il se peut qu'un cas ne soit pas sensé arriver car on l'empêche dans le code. Dans ce cas, on peut utiliser la fonction `failwith`, qui affiche un message d'erreur et arrête le programme. Par exemple :

```
let parite x = x mod 2
let est_pair y = match parite y with
| 0 -> true
| 1 -> false
| _ -> failwith "Ne doit pas arriver"
```

Il ne faut **jamais** laisser un matching incomplet.

On peut aussi utiliser les motifs avec les let-in. On parle alors de ***let déstructurant***. On peut aussi utiliser le joker dans ce contexte :

```
let p = (2, "bla", true)
let a, b, c = p

let p = "important", "a jeter", "aussi a jeter"

let x, _, _ = p
```

C Fonctions récursives

En OCaml, on n'utilise pas (pour l'instant) de boucle for ou de boucle while. A la place, on utilise des ***fonctions récursives***, c'est à dire des fonctions qui s'appellent elles-même. La récursivité est au coeur de la programmation fonctionnelle.

En OCaml, il faudra trouver des formules permettant de définir tous les objets que l'on manipule ***récursivement***. Par exemple, écrivons une fonction qui calcule a^b pour $a, b \in \mathbb{N}$.

La définition de la puissance est :

$$\forall a \in \mathbb{N}, \forall b \in \mathbb{N}, a^b = \prod_{i=1}^{b} a$$

Cependant, on peut remarquer la chose suivante pour $a, b \in \mathbb{N}$:

— Si $b = 0$, $a^b = 1$

— Sinon, $a^b = a^{b-1} \times a$

Ces relations permettent de ***caractériser*** ou même de **définir** récursivement ce que signifie a^b. On veut donc écrire :

```
1 let puissance a b =
2   if b = 0 then 1
3   else a * puissance a (b-1)
```

mais cela n'est pas accepté. En effet, au moment où l'on écrit le corps de la fonction puissance, l'identifiant `puissance` ne fait pas partie de l'énvironnement. En OCaml, pour autoriser une fonction à utiliser son propre nom, i.e. pour préciser que l'on écrit une fonction récursive, on utilise le mot clé ***let rec*** :

```
1 let rec puissance a b =
2   if b = 0 then 1
3   else a * puissance a (b-1)
```

Le typage et la sémantique du let rec sont les mêmes que pour le ***let*** classique. Cependant, lorsque l'on évalue l'application d'une fonction récursive, la fonction elle-même sera dans le contexte.

Exercice 4. Créez un fichier "exp.ml". Vous pourrez l'exécuter dans l'interpréteur en y tapant :

```
1 #use "exp.ml";;
```

Question 1. Recopiez la fonction puissance. Exécutez le code dans l'interpréteur puis testez la fonction sur quelques exemples.

Question 2. Trouver une formule de récurrence pour la factorielle, et en déduire une fonction OCaml `factorielle: int -> int`.

Question 3. Trouver une formule de récurrence permettant d'écrire l'exponentiation rapide sous forme récursive :

— Si $b = 0$, `exp a b` = ...

— Sinon :

— Si b est pair, `exp a b` = ... (faire apparaître un terme de la forme `exp X Y`)

— Sinon, `exp a b` = ... (faire apparaître un terme de la forme `exp X Y`)

Question 4. Implémenter l'exponentiation rapide, ré-exécuter le fichier "exp.ml" puis tester la fonction.

D Listes

Un nouveau type : les listes. En OCaml, les listes sont définies récursivement, comme suit :

— La liste vide, notée `[]`, est une liste

— Si $E1$ est une expression de type `'a` et $E2$ une expression de type `'a list`, alors `E1::E2` est une liste, contenant $E1$ suivi des éléments de $E2$. On dit que $E1$ est la ***tête*** de liste, et que $E2$ est la ***queue*** de liste.

Attention, la tête d'une liste est un élément, mais la queue d'une liste est une liste. On peut donc voir les listes OCaml comme des piles : on n'a accès qu'à l'élément au début d'une liste.

Exemple 4. Voyons des exemples de listes :

```
1  let l_vide = []
2  let l_simple = 3::6::8::[] (* comme 3 :: (6 :: (8 :: [])) *)
3
4  (* Renvoie une liste de taille n contenant
5     exclusivement des 1 *)
6  let rec ones n = match n with
7    | 0 -> []
8    | _ -> 1:: ones (n-1)
9
10 let cinq_uns = ones 5
```

On remarque qu'OCaml affiche les listes sous la forme `[v1; v2; ... ; vn]`. On peut également utiliser cette syntaxe pour définir des listes :

```
1  let l_a = [1;2;3;4;5;6]
2  let l_b = 1::2::3::4::5::6::[] ;
```

La première version est du sucre syntaxique, c'est la deuxième forme qui est plus fidèle à la forme qu'ont les listes.

Une liste doit contenir uniquement des expressions d'un même type T, et dans ce cas, la liste sera de type `T list`. Par exemple, `["blabla"; "bli"; "toto"]` est de type `string list`, mais `[1; 2.1; 0]` contient des entiers et des flottants, et n'est donc pas une expression bien typée.

Les listes forment un **type polymorphique**. Par exemple, la liste vide est typée `'a list` par OCaml. Cela signifie que c'est une liste de type `T list`, pour tout type T. Ainsi, lorsque l'on écrira des fonctions générales sur les listes, on pourra les appliquer sur des listes de n'importe quel type !

Les listes viennent également agrandir la liste des ***motifs*** :

— `[]` est un motif

— si `M` et `L` sont des motifs, alors `M::L` est un motif

Par exemple : `x::y::q` est un motif signifiant "deux éléments puis une liste". Ce motif sera compatible avec toutes les listes de taille au moins 2, et permettra de récupérer les deux premiers éléments, et la liste des éléments suivants. `(x, y):: q` est un motif signifiant "une liste de couples, d'au moins un élément". Il permettra de matcher par exemple `[("bla", 5); ("titi", 129)]`, et alors `x` vaudra `"bla"` et `y` vaudra `5`.

Utilisons ces motifs pour créer quelques fonctions :

```
(* Renvoie la somme des éléments de l: int list *)
let rec somme_liste l = match l with
  | [] -> 0
  | x::q -> x + somme_liste q

(* Renvoie true si la liste est de taille pair, false sinon *)
let rec taille_paire l = match l with
  | [] -> true
  | x::[] -> false
  | x::y::q -> taille_paire q (* enlever deux éléments conserve la parité *)
```

Tri par sélection Écrivons le tri par sélection en OCaml. On aura besoin d'une fonction de sélection qui extrait le max d'une liste : codons une fonction `selection: 'a list -> ('a * 'a list)` qui étant donné une liste L renvoie le couple (x, L') avec x le max de L, et L' la liste L où une occurence de x a été supprimée.

Pour coder cette fonction, on doit réfléchir inductivement : si j'ai une liste Q et que j'ai obtenu son max m et la liste Q' des autres éléments, comment faire pour obtenir le max de $x :: Q$? Il suffit de regarder le maximum entre x et m : c'est le maximum de $x :: Q$, et l'autre peut être remis au début de la liste. Cette remarque nous donne une définition inductive de la fonction :

```
(* Couple (a, t) avec a le maximum de l et t la
   liste l privée d'une occurence de a *)
let rec selection l = match l with
  | [] -> failwith "empty list"
  | x::[] -> x, []
  | x::q -> let (m, q') = selection q in
            if x > m then (x, m::q')
            else (m, x::q')
```

A partir de cette fonction, on peut construire une fonction récursive de tri par sélection :

```
let rec select_sort l = match l with
  | [] -> []
  | x::q -> let (m, q) = selection l in m::(select_sort q)
```

E Alias de type

En OCaml, on peut définir ses propres types. Par exemple, comme en C, on peut redéfinir un type pré-existant en lui donnant un autre nom :

```
type nombre = int
type vecteur3d = float * float * float

(* fonctions |N -> |B *)
type filtre_entier = int -> bool
```

Bien qu'OCaml devine tout seul le type des expressions que l'on écrit, il ne devine pas toujours par lui même les types définis ainsi :

```
let x = (0., 0., 0.) (* affiche float*float*float *)
let y = 5 (* affiche nombre plutôt que int *)
```

Cependant, on peut spécifier le type des variables avec " :" comme suit :

```
let x: int = 5

let f (x:int) : int = x + 1 (* prend en entrée un int et renvoie un int *)

let f: (int -> int) = fun x -> x - 1

let p: vecteur3d = (2., 3.2, -8.54)

let equal_5: filtre_entier = fun x -> x = 5

let produit_scalaire (a:vecteur3d) (b:vecteur3d) : float =
  let xa, ya, za = a in
  let xb, yb, zb = b in
  xa *. xb +. ya *. yb +. za *. zb


(* Ici, le type vecteur3d serait inféré automatiquement àcause du type
   de produit_scalaire *)
let norme_carree (a:vecteur3d) : float = produit_scalaire a a
```

F Type somme

Le type produit correspond au produit cartésien d'ensemble. Le type somme correspond plutôt à l'union disjointe.

Un type somme permet de représenter des catégories d'objets ayant plusieurs "cas". Un tel type est constitué de ***constructeurs***, qui sont les différents mots clés que l'on utilise pour construire des objets de ce type. Chaque constructeur correspond à un "cas" différent.

Exemple 5. On veut implémenter un type pour les fournitures scolaires. On veut pouvoir représenter :

- Les stylos BIC, qui peuvent être de couleurs différentes
- Les règles, qui peuvent être de tailles différentes et peuvent être centrées ou pas (i.e. 0 est au centre ou au bord)
- Les gommes

Créez un fichier "fourniture.ml". Vous pourrez l'exécuter dans l'interpréteur en tapant :

```
#use "fourniture.ml";;
```

On définit le type somme avec :

```
type fourniture =
    | Stylo of string (* couleur *)
  | Regle of int * bool (* taille en cm, centrée ou non *)
  | Gomme
```

Ce qui se lit : “Il y a trois types de fournitures : Les stylos, qui sont paramétrés par une chaîne de caractères, les règles, paramétrées par un entier et un booléen, et les gommes”. Les commentaires précisent à quoi servent les paramètres.

On peut ensuite créer des éléments de ce type :

```
let x = Stylo "rouge"
let r1 = Regle (30, true)
let r2 = Regle (20, false)
let g = Gomme
```

Notons qu’OCaml peut automatiquement tester l’égalité entre deux objets d’un même type :

```
let x = Stylo "rouge"
let y = Stylo "rouge";;
x = y;; (* Vaut true *)
```

La manière principale de manipuler les types définis ainsi est le ***match with***. Par exemple, on suppose que le prix des fournitures est comme suit :

— Les gommes coûtent 1,50€ ;

— Les stylos bleus coûtent 1,20€, les autres coûtent 1€ ;

— Une règle de l centimètres coûte $1 + \frac{l}{15}$ euros.

Voici comment on implémenterait une fonction calculant le prix d’une fourniture en OCaml :

```
(* prix de fourn en euros *)
let prix (fourn: fourniture) : float =
  match fourn with
  | Gomme -> 1.5
  | Stylo "bleu" -> 1.2
  | Stylo _ -> 1.0
  | Regle(longueur, _) -> 1.0 +. float_of_int longueur /. 15.0
```

On peut représenter une trousse comme une liste de fournitures. Écrivons une fonction qui calcule le nombre de gommes dans une trousse :

```
type trousse = fourniture list;;

let rec nombre_gommes (t:trousse) : int =
  match t with
  | [] -> 0
  | Gomme::q -> 1 + nombre_gommes q
  | _::q -> nombre_gommes q
```

Exercice 5. On suppose qu'une trousse seule coûte 5€. Écrire une fonction calculant le prix total d'une trousse, en comptant tout ce qu'elle contient :

```
let rec prix_trousse (t: trousse) : float = ...
```

La syntaxe pour les type somme est donc ;

```
type nom_type =
     Constructeur1 of type1 (* ou Constructeur1 *)
   | Constructeur2 of type2 (* ou Constructeur2 *)
      ...
   | ConstructeurN of typeN (* ou ConstructeurN *)
```

On rajoute également à la définition des ***motifs*** :

— Si `Cons` est un constructeur d'un type somme, et `M` est un motif, alors `Cons M` est un motif.

Rien n'empêche un type de s'auto-référencer, on dit alors que c'est un type ***inductif***, ou récursif. Par exemple, si l'on veut créer un type représentant les expressions arithmétiques :

```
type expr =
  | Constante of int
  | Plus of expr * expr
  | Fois of expr * expr
```

Autrement dit : une expression est soit une constante entière, soit la somme de deux expressions soit le produit de deux expressions. L'expression $(1 + 2)$ s'écrira :

```
Plus (1, 2)
```

L'expression $(1 + 2) * (3 + 7 * 6)$ s'écrira :

```
let my_expr =
  Fois (
    Plus(
      Constante 1,
      Constante 2
    ),
    Plus(
      Constante 3,
      Fois (
        Constante 7,
        Constante 6
      )
    )
  )
```

Pour manipuler des types inductifs, il faudra généralement utiliser des fonctions récursives. Sur l'exemple précédent, écrivons une fonction qui évalue une expression arithmétique :

```
let rec eval e = match e with
  | Constante n -> n
  | Plus (e1, e2) -> eval e1 + eval e2
  | Fois (e1, e2) -> eval e1 * eval e2
```

(Remarquez que cette fonction ressemble de près à l'algorithme d'évaluation des expressions OCaml décrit plus tôt dans le chapitre. En fait, à ce stade, vous pourriez créer un type pour les expressions OCaml, un type pour les environnement, et recoder OCaml en OCaml !)

Tentons de recoder le type des listes OCaml. On remarque que les listes sont polymorphes, autrement dit elles peuvent s'adapter à des types différents. On voudrait écrire :

```
type liste =
  | Vide (* liste vide *)
  | Cons of 'a * liste (* Constructeur: Cons (x, q) sera x suivi de q *);;
```

Cependant, OCaml affiche : `Error: The type variable 'a is unbound in this type declaration.`

Pour définir ce type, on doit le ***paramétrer*** :

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

La syntaxe générale des types paramétrés est :

```
type ('a1, 'a2, ..., 'an) nom_type =
  | Constructeur1 of type1
   ...
  | Constructeurk of typek
;;
```

On peut ensuite manipuler ce type normalement, et OCaml gèrera le polymorphisme :

```
let ma_liste_1 = Cons (5, Cons(6, Vide)) ;; (* [5, 6] *)
let ma_liste_2 = Cons ("bla", Cons("blo", Cons ("bli", Vide))) ;; (* ["bla", "blo", "bli"] *
    )

let tete l = match l with
  | Vide -> failwith "liste vide"
  | Cons (x, q) -> x
;;
```

Exercice 6. Écrire une fonction `vraie_liste: 'a liste -> 'a list` qui transforme une liste de notre type liste maison en une liste standard OCaml :

3 Récursivité

A Correction d'une fonction récursive

Lorsque l'on écrit du code OCaml, on n'utilise pas de boucles. Donc, il nous faut un nouvel outil pour prouver la correction des fonctions. Considérons un exemple simple : la multiplication :

```
1 (* Renvoie x fois y, pour x, y entiers positifs *)
2 let rec mult x y =
3   if x = 0 then 0
4   else y + mult (x-1) y
```

Cette fonction est correcte si elle renvoie bien ce qui est indiqué dans sa documentation, donc ici si elle renvoie bien le produit de ses deux arguments. Remarquons que le commentaire de la fonction précise une précondition nécessaire à la correction : x et y doivent être positifs.

Pour étudier cette expression, on introduit la fonction mathématique correspondante :

$$\begin{array}{rcccl} & & \mathbb{N} & \longrightarrow & \mathbb{N} \\ \textbf{mult} & : & x, n & \mapsto & \begin{cases} 0 & \text{si } n = 0 \\ x + \textbf{mult}(x, n-1) & \text{sinon} \end{cases} \end{array}$$

Cette fonction est ***définie par récurrence***. Commençons par montrer qu'elle est bien définie, c'est à dire que la fonction OCaml correspondante termine. Pour cela, il faut remarquer qu'un appel à la fonction **mult** cause un appel récursif, sur une entrée strictement plus petite. Comme les entrées sont dans N (par précondition), et comme il n'existe pas de suite infinie strictement décroissante dans N, la fonction s'arrête bien.

Montrer la correction de la fonction, c'est montrer la propriété suivante :

$$\forall x \in \mathbb{N}, \forall n \in \mathbb{N}, \textbf{mult}(x, n) = x \times n$$

Pour cela, on procède par récurrence. Plus précisément, posons $x \in \mathbb{N}$, et montrons par récurrence sur $\mathbb{N}$ la propriété suivante :

$$\forall n \in \mathbb{N}, P(n) : \textbf{mult}(x, n) = x \times n$$

— Pour $n = 0$, $\textbf{mult}(x, 0) = 0$: $P(0)$ est vraie.

— Soit $n \in \mathbb{N}^*$, supposons $P(n-1)$. Alors, $\textbf{mult}(x, n-1)$ vaut $x \times (n-1)$. Or, $n > 0$ donc $\textbf{mult}(x, n) = x + \textbf{mult}(x, n-1) = x + x \times (n-1)$ par hypothèse de récurrence. Donc, $\textbf{mult}(x, n) = x \times n$: $P(n)$ est vraie

Voyons un exemple plus complexe, en reprenant le tri par insertion :

```
1  (* Renvoie une copie de l, où x a été inséré dans l'ordre.
2     l doit être croissante *)
3  let rec insert x l = match l with
4    | [] -> [x]
5    | y::q -> if x < y then x :: l
6              else         y :: insert x q ;;
7
8  (* Renvoie une copie triée de l *)
9  let rec insert_sort l = match l with
10   | [] - []
11   | x::q -> let q = insert_sort q in insert x q ;;
```

On considère les fonctions mathématiques correspondantes. On ne décrit pas pour l'instant l'ensemble mathématique précis où vivent les listes, on y viendra au chapitre suivant.

$$\textbf{insert} \quad : \quad x, L \quad \mapsto \quad \begin{cases} [x] & \text{si } L \text{ est vide} \\ x :: y :: Q & \text{si } L \text{ de la forme } y :: Q \text{ et } x < y \\ y :: (\textbf{insert}(x, Q) & \text{si } L \text{ de la forme } y :: Q \text{ et } x \geq y \end{cases}$$

$$\textbf{tri\_insert} \quad : \quad L \quad \mapsto \quad \begin{cases} [] & \text{si } L \text{ est vide} \\ (\textbf{insert}(x, \textbf{tri\_insert}(Q)) & \text{si } L \text{ de la forme } x :: Q \end{cases}$$

Ces fonctions sont définies par récurrence sur la taille de L. Au prochain chapitre, on dira qu'elles sont définies par ***induction*** sur la structure des listes.

Commençons par la terminaison d'insert. On pose $f : L \mapsto |L|$. Soit L une liste, notons $L_0, \ldots, L_k, \ldots$ les listes sur lesquelles on appelle récursivement la fonction **insert** lors de l'évaluation de $\textbf{insert}(x, L)$. On remarque que la suite $(f(L_k))_{k\in\mathbb{N}}$ est strictement décroissante et à valeurs dans $\mathbb{N}$. Elle est donc finie : la fonction **insert** termine.

La terminaison de la fonction de tri se prouve de manière analogue. Montrons maintenant formellement la correction des deux fonctions. Au chapitre suivant, nous verrons une généralisation du principe de récurrence qui permettra d'écrire de manière beaucoup plus élégante et courte les preuves de ce style.

— Pour **insert** : On raisonne par récurrence sur la taille de la liste L en entrée. On pose x un élément à insérer. Montrons la propriété suivante par récurrence sur $n \in \mathbb{N}$:

$\forall n \in \mathbb{N}, \forall L$ liste de taille n, si L est triée, alors $\textbf{insert}(x, L)$ est contient les éléments de L ainsi que x, et est triée.

- Si $|L| = 0$ alors $L = []$, et $[x]$ contient bien les éléments de L (personne) ainsi que x, et est triée.
- Soit L une liste avec $|L| > 0$. On écrit $L = y :: Q$. Si $x < y$ alors $x :: y :: Q = x :: L$ est bien une copie de L où x a été ajouté, triée. Sinon, par hypothèse de récurrence, $\textbf{insert}(x, Q)$ est une copie triée de Q où x a été ajouté, et comme L est triée, $y \leq \min Q$ et $y \leq x$. Donc, $y :: \textbf{insert}(x, Q)$ est triée, et contient bien x ainsi que les éléments de L, à savoir y et les éléments de Q

— Pour **insert_sort** : **Exercice** : Montrer par récurrence la propriété suivante :

$\forall n \in \mathbb{N}, \forall L$ liste de taille n, $\textbf{insert\_sort}(L)$ est triée et contient les éléments de L.

B Récursivité terminale

Rappelons le principe de la pile d'appel en C : lorsque l'on appelle une fonction, on ***empile*** sur la pile une zone mémoire pour stocker les variables de la fonction. On doit également stocker l'adresse de retour, c'est à dire l'instruction à laquelle aller à la fin de la fonction.

En OCaml, l'évaluation d'une expression va générer beaucoup d'appels de fonctions récursifs. Regardons un morceau de code de l'algorithme permettant d'évaluer les expressions OCaml :

Algorithme 3 : Évaluation OCaml

Entrée(s) : E une expression OCaml, C un contexte
Sortie(s) : La valeur correspondant à E évaluée dans le contexte C

si E ***est de la forme*** `E1 + E2` **alors**
  $v_1 \leftarrow$ **Evaluer**(E_1, C);
  $v_2 \leftarrow$ **Evaluer**(E_2, C);
  retourner $v_1 + v_2$
// ...
si E ***est de la forme*** `(fun x -> E2) E1` ***ou*** `let x = E1 in E2` **alors**
  $v_1 \leftarrow$ **Evaluer**(E_1, C);
  $C' \leftarrow C + [x \mapsto v_1]$ `// Contexte additionnel`
  $v_2 \leftarrow$ **Evaluer**(E_2, C');
  retourner v_2

Donc, sur la machine, le programme qui évalue les expressions OCaml doit faire beaucoup d'appels de fonction, et risque donc remplir la pile d'appel. Considérons les deux fonction suivantes :

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

(* Renvoie la factorielle de n multipliée par accu *)
let rec factorielle_bis accu n =
  if n = 0 then accu
  else factorielle_bis (accu*n) (n-1) ;;

(* On remarque que factorielle_bis 1 n renverra la factorielle de n *)
```

Pour évaluer `factorielle 2`, on doit :

1. Calculer factorielle 2 :
   (a) Caluler factorielle 1
      i. Caluler factorielle 0
      ii. On renvoie donc 1
   (b) On obtient 1, on multiplie par 1, on renvoie donc 1
2. On obtient 1, on multiplie par 2, on renvoie donc 2

Ainsi, on doit plonger au fond des appels récursifs, puis remonter à la surface en appliquant les opérations nécessaires. On doit donc ***empiler*** les appels de fonction les un sur les autres, puis les ***dépiler***. Donc, si l'on tente d'évaluer `factorielle 10000000`, on aura une erreur de dépassement de pile.

En revanche, pour évaluer `factorielle_bis 1 2` :

1. Calculer factorielle 1 2
2. On doit donc renvoyer factorielle_bis 2 1
   (a) Calculer factorielle 2 1
   (b) On doit donc renvoyer factorielle_bis 2 0
      i. Calculer factorielle_bis 2 0
      ii. On renvoie donc 2

Donc, il n'y a pas de phase de "remontée à la surface". On n'a donc pas besoin d'empiler les appels : chaque appel successif va pouvoir ***remplacer*** l'appel précédent. Les différentes stack frames peuvent donc se remplacer les unes les autres, au lieu de s'empiler. Si l'on évalue `factorielle_bis 10000000`, aucun problème de pile !

Définition 9. On appelle fonction ***récursive terminale*** (en anglais : ***tail-recursive***) toute fonction récursive ne nécessitant aucun traitement à la remontée d'une valeur, autre que le retour de cette valeur.

Proposition 1. Les fonctions récursives terminales peuvent être compilées / interprétées de façon à garder une taille de pile d'appel constante.

Remarquons que pour `factorielle_bis`, on n'écrit pas directement la fonction factorielle, mais une fonction ***plus générale***, que l'on applique avec un bon paramètre ensuite. La programmation récursive terminale nécessite souvent d'utiliser une méthode par ***accumulateur*** qui consiste à stocker le résultat temporaire dans les arguments, dans un accumulateur.

Exemple 6. Prenons la fonction d'exponentiation rapide et tentons d'en trouver une version récursive terminale.

```
1 let rec fast_exp a b =
2   match (b, b mod 2) with
3   | (0, _) -> 1
4   | (_, 0) ->    fast_exp (a*a) (b/2)
5   | _      ->  a * fast_exp (a*a) (b/2);;
```

Remarquons que cette fonction n'est pas récursive terminale à cause du dernier cas.

Pour déterminer ce que l'accumulateur va stocker, et ce que la fonction généralisée va faire, il peut être utile de réfléchir à l'algorithme impératif et à l'invariant de boucle qui permet de prouver sa correction. Pour l'exponentiation rapide, on rappelle :

```
1  int fast_exp(int a, int b){
2    int res = 1, A = a, B = b;
3    while (B>0){ // invariant: a puissance b = (A puissance B) * res
4      if (B%2 == 1){
5        res = A*res;
6      }
7      a = A*A;
8      b = B/2;
9    }
10   return res;
11 }
```

On avait l'invariant suivant : $res \times A^B = a^b$, avec $A = a, B = b$ et $res = 1$ au départ. Pour $a, b, r \in \mathbb{N}$, notons $f(a, b, r) = a^b \times r$. On a $a^b = f(a, b, 1)$, $f(a, 0, r) = r$ et l'invariant de boucle précédent nous dit que :

$$\begin{array}{rcl} f(a,b,r) & = & f(a^2, \frac{b}{2}, r) \text{ si } b \text{ pair} \\ f(a,b,r) & = & f(a^2, \frac{b-1}{2}, ar) \text{ si } b \text{ impair} \end{array}$$

On va donc rendre la fonction terminale ainsi en lui faisant calculer non pas a^b pour a, b donnés, mais $a^b \times r$ pour a, b, r donnés. On pourra ainsi faire des appels récursifs où les valeurs de a, b, r changent comme dans la boucle du code C :

```
1  (* renvoie accu fois (a puissance b) *)
2  let rec fast_exp_aux a b accu =
3    match (b, b mod 2) with
4    | (0, _) -> accu
5    | (_, 0) -> fast_exp_aux (a*a) (b/2) accu
6    | _      -> fast_exp_aux (a*a) (b/2) (a*accu) ;;
7
8  let fast_exp a b = fast_exp_aux a b 1;;
```

La variable `accu` du code OCaml va jouer le rôle de `res` dans le code C et grossir en stockant de plus en plus d'information, pour être renvoyée à la fin. On appelle un tel paramètre de fonction un **accumulateur**. Il n'est pas toujours plus facile de repasser par le point de vue impératif pour mettre au point une version récursive terminale d'une fonction OCaml, c'est souvent plus simple de réfléchir à comment introduire un accumulateur

Exercice 7.

Écrivons quelques fonctions récursives sur les listes, en cherchant systématiquement une version récursive terminale.

Question 1. Définir une fonction `longueur` permettant de calculer la longueur d'une liste

Question 2. Définir une fonction `reverse` permettant de renverser une liste

Question 3. La fonction `map: ('a -> 'b)-> 'a list -> 'b list` qui applique une fonction à tous les éléments d'une liste

Exercice 8. On considère le type suivant représentant des expressions arithmétiques :

```
type expr =
  | Int of int
  | Plus of expr * expr ;;
```

Question 15. Définir une fonction `eval: expr -> int` qui évalue une expression. Essayer de la rendre récursive terminale. Que remarque-t'on ?

C Arbre d'appel

En OCaml, on peut ***tracer*** une fonction pour observer ses valeurs d'entrée et de sortie au cours des appels récursifs générés par une exécution. Par exemple :

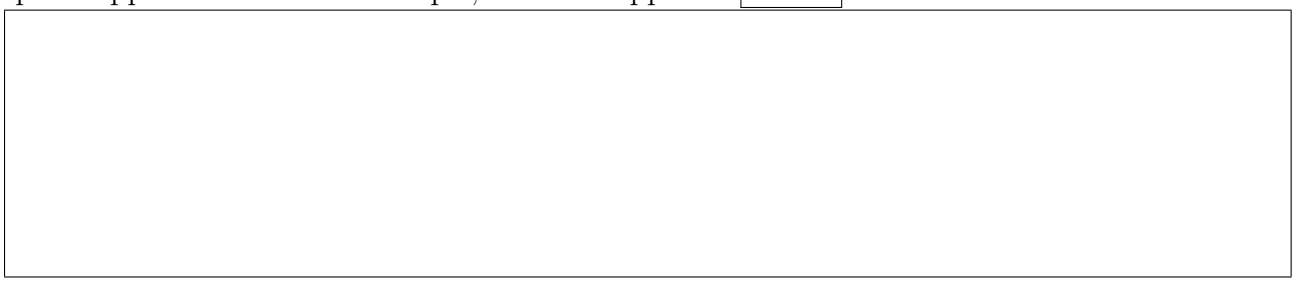
```
let rec fact n =
  if n <= 1 then 1
  else n * fact (n-1)
;;
#trace fact ;;
fact 4;;
```

On verra à l'écran :

```
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
```

`f <-- x` signifie que l'on appelle f sur x, et `f --> x` signifie qu'un appel s'est terminé et a renvoyé x.

Un ***arbre d'appel*** permet de représenter schématiquement une telle exécution. On suit la trace de la fonction, en descendant dans l'arbre lors d'un nouvel appel, et en remontant lorsqu'un appel termine. Par exemple, l'arbre d'appel de `fact 4` est :



Vocabulaire : Les cercles représentant des appels sont nommés ***noeuds*** de l'arbre. Le premier noeud, correspondant à l'appel initial, est appelé ***racine***.

L'arbre d'appel permet d'estimer la complexité d'une fonction : Si chaque appel demande un nombre constant d'opérations hors appels récursifs, la fonction aura une complexité proportionnelle au nombre de noeuds de l'arbre d'appel.

Dans l'exemple précédent, l'arbre d'appel de `fact n` a n noeuds pour $n \in \mathbb{N}^*$, et chaque appel demande un temps constant excepté l'appel récursif : la complexité est en $\mathcal{O}(n)$.

Exercice 9. On considère la suite suivante :

$$\begin{aligned} u_0 &= 2 \\ \forall n \in \mathbb{N}, u_{n+1} &= |1 - 3u_n + 2u_n^2| \end{aligned}$$

On propose la fonction suivante permettant de calculer la suite u :

```
let rec u n =
  assert n >= 0;
  match n with
  | 0 -> 2
  | _ -> if 1 - 3 * u n + 2 * u n * u n > 0 then
        1 - 3 * u n + 2 * u n * u n
      else
        -(1 - 3 * u n + 2 * u n * u n)
;;
```

Dessiner les quelques premiers étages de l'arbre d'appel de `u 5`. Estimer le nombre de noeuds dans l'arbre d'appel de `u n` pour $n \in \mathbb{N}$, et en déduire la complexité de cette fonction. Proposer une fonction plus rapide.

D Complexité

La méthode la plus générale pour analyser la complexité d'une fonction récursive est de trouver une relation de récurrence sur la complexité, et de résoudre cette relation.

Exemple 7. Pour la fonction `fact`, notons C_n le coût de l'appel `fact n`. Pour $n > 1$ on a $C_n = C_{n-1} + \mathcal{O}(1)$. Donc, à partir d'un certain rang n_0, pour une certaine constante $A > 0$, $C_n \leq C_{n-1} + A$. On a donc :

$$\begin{aligned} C_n &\leq C_{n-1} + A \\ &\leq C_{n-1} + C_{n-2} + 2A \\ &\leq C_{n_0} + (n - n_0)A \\ &\leq C_{n_0} + An \end{aligned}$$

on retrouve donc à nouveau que la fonction est en $\mathcal{O}(n)$.

Exercice 10. De même, trouver une relation de récurrence pour la fonction `u` de l'exemple précédent, et retrouver la complexité asymptotique.

On considère l'exponentiation rapide, que l'on peut écrire en OCaml comme suit :

```
(* calcule a puissance n *)
let rec fast_exp (a : int) (n : int) : int = match n with
  | 0 -> 1
  | _ -> if n mod 2 = 0 then fast_exp (a*a) (n/2)
      else              a * fast_exp (a*a) (n/2)
```

La complexité de cette fonction par rapport à n vérifie la relation de récurrence suivante :

$$C_n = C_{\lfloor \frac{n}{2} \rfloor} + \mathcal{O}(1)$$

Étudions pour simplifier la relation suivante :

$$\begin{aligned} C_0 &= 1 \\ C_n &= C_{\lfloor \frac{n}{2} \rfloor} + 1 \end{aligned}$$

qui ne changera pas le comportement asymptotique. Pour $n = 2^p$ une puissance de 2, on a :

$$C_n = C_{2^p} = C_{2^{p-1}} + 1 = C_{2^{p-2}} + 1 + 1 = \cdots = C_{2^{p-p}} + p = C_1 + p = C_0 + p + 1 = p + 2$$

Donc, $C_{2^p} = p + 2$. Admettons que la suite C_n est croissante. Alors, pour n quelconque, on peut encadrer n comme suit :

$$2^{\lfloor \log_2(n) \rfloor} \leq n < 2^{\lfloor \log_2(n) \rfloor + 1}$$

Par croissance, on a donc $\lfloor \log_2(n) \rfloor + 2 = C_{2^{\lfloor \log_2(n) \rfloor}} \leq C_n \leq C_{2^{\lfloor \log_2(n) \rfloor + 1}} = \lfloor \log_2(n) \rfloor + 3$

Ainsi, $C_n = \Theta(\log n)$: on retrouve la complexité logarithmique de l'exponentiation rapide impérative.

Exercice 11. On propose le code suivant OCaml pour le tri fusion :

```
(* renvoie deux listes l1 l2 de même taille à1 près,
   contenant les éléments de l *)
let rec separer (l: 'a list) : 'a list * 'a list =
  match l with
  | [] -> [], []
  | [x] -> [x], []
  | x::y::q -> let l1, l2 = separer q in x::l1, y::l2

(* Fusionne l1 et l2 deux listes supposées triées par ordre croissant *)
let rec fusion (l1: 'a list) (l2: 'a list) : 'a list =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | x1::q1, x2::q2 -> if x1 < x2 then x1 :: fusion q1 l2
                  else x2 :: fusion l1 q2

let rec tri_fusion (l : 'a list) : 'a list =
  match l with
  | [] | [x] -> l
  | _ -> let l1, l2 = separer l in fusion (tri_fusion l1) (tri_fusion l2)
```

Question 1. Montrer la correction des trois fonctions

Question 2. Donner la complexité des fonctions `separer` et `fusion` en fonction de la taille de leurs arguments.

Question 3. En notant C_n le nombre d'opérations de `tri_fusion` sur une liste de taille n, justifier que C_n vérifie la relation de récurrence suivante :

$$C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + \Theta(n)$$

Question 4. Étudier la relation suivante simplifiée :

$$C_n = 2C_{\frac{n}{2}} + n$$

en calculant d'abord C_{2^p} pour p entier. On admettra que $(C_n)_{n \in \mathbb{N}}$ est croissante. En conclure la complexité du tri fusion.

Question 5. Retrouver cette complexité en étudiant l'arbre d'appel de `tri_fusion`.