Programmation Fonctionnelle en OCaML

Chargée de cours:	Irène Durand,
Chargés de TD/TM	Frédérique Carrère, Irène Durand, Stefka Gueorguieva, Jonas Sénizergues.
Cours, 7 x 1h20	S36 (x2), 37-39, 41, 43, 46
TM en présentiel, 13 séances de 1h20	S36-43, 45-49
Devoir surveillé: S45 (Mercredi 6/11)	
TP noté: S49	
Travail individuel	4h par semaine

https://moodle.u-bordeaux.fr/course/view.php?id=1208

MCC: Session1:0.5EX1+ 0.5CC, Session2:0.5EX2 + 0.5CC
CC: DS 1h30 40%, TP noté 1h30 40%, exos Moodle 20%
Examens 1h30

Art de la programmation

La programmation est un art.
Pour progresser:

- lire du code écrit par des experts
- lire la littérature sur la programmation
- programmer
- maintenir du code écrit par d'autres
- être bien organisé

Objectifs de ce cours

- Acquérir les bases de la programmation fonctionnelle
- Appliquer les principes généraux de programmation
  - Lisibilité du code
  - Réutilisabilité
  - Maintenabilité
  - Efficacité (quand elle ne nuit pas à la lisibilité) $\Rightarrow$ Complexité
  - Test

Paradigmes de programmation

Un paradigme correspond à un style de programmation.

- impératif
- fonctionnel
- objet
- macro

Un langage peut offrir au programmeur plusieurs paradigmes. Un programme peut utiliser de plusieurs paradigmes.

- `Python`: ?
- `Common Lisp`: ?
- `OCaML` ?
- `C`: ?
- `C++`: ?
- `Java`: ?

Paradigmes de programmation: exemples

Paradigmes:

- impératif
- fonctionnel
- objet
- macro

Exemples de langages et leurs paradigmes:

- `Python`: impératif, fonctionnel, objet
- `Common Lisp`: les 4 paradigmes
- `OCaML`: impératif, fonctionnel, objet
- `C`: impératif, macros (basiques)
- `C++`: impératif, objet, macros (basiques)
- `Java`: impératif, objet

Propriétés d'un langage

Un langage de programmation peut-être

- interactif/non interactif
- compilé et/ou interprété
- dynamiquement typé/statiquement typé

OCaML peut-être

- utilisé en mode interactif ou en mode batch.

Paradigme fonctionnel

- La fonction est un objet de base (comme les entiers, flottants, caractères, ..).
  On dit que c'est un objet de première classe; elle peut être:
  - passée en paramètre d'une autre fonction
  - créée et retournée par une fonction
- pas d'effets de bord
  - pas d'affectation
  - on se contente d'appeler des fonctions
  - la structure de contrôle de base est la récursivité.

Paradigme fonctionnel vs paradigme impératif

Le paradigme fonctionnel s'oppose au paradigme impératif
Paradigme impératif:

- un programme a un état: la valeur de l'ensemble de ses variables
- instruction de base est l'affectation
- structure de contrôle de base est la boucle tant-que
- source de nombreuses difficultés et bugs
- produit des programmes plus difficiles à comprendre
- voir bugs célèbres aux conséquences désastreuses

Quand utiliser le paradigme fonctionnel?

- quand il y a des enjeux de sécurité (programmes plus sûrs et même dans certains cas prouvés)
- quand on veut développer rapidement un prototype

L'inconvénient de la programmation fonctionnelle sera principalement la performance; mais ce défaut diminue avec des compilateurs de plus en plus optimisés.

Langage support: `OCaML`

fonctionnel, statiquement typé, interactif mais aussi compilé, impératif (non abordé dans ce cours).
Le typage permet d'éliminer des bugs à la compilation.
Utilisé:

- entreprises (Facebook, Docker, Bloomberg, Jane Street, ...)
- universités (France, USA, Japon, ...)
- ParcoursSup
- en France (CEA, Dassault Systèmes ANSSI)
- à Bordeaux: Shiro Games

Prise de contact avec `OCaML`

développé à l'INRIA `caml.inria.fr`
disponible sur de nombreuses architectures (Linux, Windows, MacOS X, ...)

- Mode **interactif**:
  - Dans un terminal

```
$ ocaml
OCaml version 4.13.2
# 1 + 2;;
- : int = 3
#
```

  - Avec `utop` dans un terminal
  - sous Emacs, modes Tuareg et utop
  - sous VSCode nouveau pluging permettant l'interactivité
- Mode non interactif
  - dans un terminal
  - sous vs-code

Boucle REPL

`OCaML` est un langage interactif. Quand on le lance, on se trouve dans une `REPL`[1], dans laquelle on peut taper des phrases qui sont soit expressions soit des requêtes.
Le système boucle sur les trois opérations suivantes:

- lit (READ) une expression ou une requête (et la met sous une forme interne)
- évalue (EVAL) la forme interne
- affiche (PRINT) le résultat sous forme lisible par l'utilisateur

[1] Read Eval Print Loop

Expressions et requêtes

Une expression a toujours une valeur et toute valeur a un type.
Il existe des types de base comme
`int, float, bool, char, ...`.
Nous verrons dans un prochain chapitre des types composés à l'aide d'opérateurs et comment nommer les types ainsi construits.

Le type d'une expression est le type de sa valeur.

Une requête fait un effet de bord et peut avoir une valeur.

Expressions

Une expression est

- soit un objet de base (nombre, caractère, booléen, fonction, ...),
- soit une expression prédéfinie (`if then else` , `let .. in` `match with` ,
- soit l'application d'une fonction à des arguments qui sont eux-mêmes des expressions[2].

langage interactif $\Rightarrow$ pas de programme principal
n'importe quelle fonction peut être appelée

[2] Noter la définition récursive d'une expression

Application d'une fonction

Notation par défaut: préfixe

la fonction *f* est placée avant ses arguments: *f e1 e2 ....*

Ne **pas** séparer les arguments par une virgule, comme en C.

L'application de fonction est plus prioritaire que les opérateurs usuels.

```
# max 20 12;;
- : int = 20
# max 30 12 * 2;;
- : int = 60
# max 30 (12 * 2);;
- : int = 30
```

Parenthésage

par défaut: `f e1 e2 ... en` équivaut à `(((f e1) e2) ... en)`.

Pour un autre parenthésage, il faut le préciser:
`sqrt (max (cos 3.1415) (sin 3.1415))`.

Opérateurs binaires

opérateurs binaires courants prédéfinis, en particulier opérateurs arithmétiques binaires ⇒ notation infixe: opérateur entre les deux opérandes.

```
# 2 * (1 + 3);;
- : int = 8
```

Infixe ⇒ préfixe
Version préfixe d'un opérateur infixe: le mettre en parenthèse:

```
# 5 + 2;;
- : int = 7
# 5 - 2;;
- : int = 3
# (+) 5 2;;
- : int = 7
# (-) 5 2;;
- : int = 3
```

Nombres et Expressions numériques

types de base: `int` (entiers), `float` (flottants)
Entiers: `3, 4, 1000`
Flottants `2.1, 3.14e-3`

Opérateurs arithmétiques

syntaxe proche du langage mathématique (notation infixe)
À cause du typage, pas de surcharge des opérateurs ⇒
un jeu d'opérateurs pour chaque type:

`int` : `+, -, *, /, mod, abs, succ, pred, ...`

`float` :

`+., -., *., /., **, abs_float, truncate, sqrt, ...`.

```
# 2.1 +. 4.5;;
-: float 6.6
```

```
# 2.1 +. 4.5;;
-: float 6.6
```

Expressions booléennes

- Type booléen `true, false`
- Opérateurs booléens
  - `&&, ||, not`
  - `=` (égalité de valeurs), `<, <=, >, >=`.

```
# 2 * 2 < 3 || 2 = 1 + 1;;
- : bool = true
# not (2 * 2 < 3 || 2 = 1 + 1);;
- : bool = false
```

Commentaires

délimités par les caractères `(*` et `*)`
pas de commentaire de ligne

```
(* ceci est un commentaire *)
```

Expressions conditionnelles

```
(* nombre de solutions d'une équation du 1er degré *)
(* ax + b = 0 *)
if a = 0 then
  if b = 0 then -1
  else 0
else 1
```

Expression `let in`

Pour éviter la duplication d'expressions dans le code, il est conseillé d'utiliser l'expression `let in` qui permet de mémoriser temporairement la valeur d'une ou plusieurs expressions dans des variables temporaires. Ceci permet d'éviter

- la duplication du code
- l'évaluation multiple d'une expression

On peut donner des valeurs à plusieurs variables en parallèle à l'aide du mot-clé `and`.

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

Pour des affectations séquentielles, on utilise le `let in` en cascade.

```
# let x = 2 in
  let y = x * x in y + 1;;
- : int = 5
```

Fonction anonyme (fonction sans nom)

La fonction qui à x associe $3 * x$ est clairement définie et se note en mathématique: $x \mapsto 3 * x$
peut être définie avec une expression utilisant le mot `fun` et la syntaxe: `fun x1 x2 ... -> expression`

paramètres de la fonction: `x1` , `x2` , ...
`expression` : corps de la fonction; évaluation donne, après passage des paramètres, la valeur de retour de la fonction.
la fonction ci-dessus s'écrit

```
fun x -> 3 * x;;
```

Si on entre cette ligne dans l'interpréteur `OCaML` , l'interpréteur répond

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

Il indique que cette expression est une fonction par `<fun>` . En effet, d'habitude, pour une expression, il affiche la valeur de l'expression, mais pas dans le cas d'une fonction.

```
# fun x -> 3 * x;;
- : int -> int = <fun>
```

Il donne aussi son type: `int -> int -> int`. Le nombre de flèches indique le nombre d'arguments de la fonction, ici: 2 (qui sont `x` et `y`). Les types des arguments sont donnés dans l'ordre, et le dernier type, après la dernière flèche, est le type de retour de la fonction.
De même,

```
# fun x y -> float_of_int x +. y;;
- : int -> float -> float = <fun>
# (fun x y -> float_of_int x +. y) 4;;
- : float -> float = <fun>
# (fun x y -> float_of_int x +. y) 4 2.5;;
- : float = 6.5
```

```
# fun x -> 3 * x ;;
- : int -> int = <fun>
# (fun x -> 3 * x) 10;;
- : int = 30
# fun x y -> 3 * x + 5 * y;;
- : int -> int -> int = <fun>
# (fun x y -> 3 * x + 5 * y) 2;;
- : int -> int = <fun>
# (fun x y -> 3 * x + 5 * y) 2 10;;
- : int = 56
```

Conversions

```
float_of_int, int_of_float, int_of_char,
char_of_int,  string_of_int, string_of_bool, ...
```

Requêtes

Les requêtes comme les expressions sont tapées dans la REPL. Elles permettent de faire des opérations non purement fonctionnelles (qui modifient l'état du système).

Requête `let`

- liaison entre une variable et une valeur (donne un nom à une valeur)
- permet de définir des variables globales (indispensable pour enregistrer fonctions et données)
- effet de bord (affecte la valeur à la variable) et retourne la valeur
- ne pas confondre avec l'expression `let ... in`.

Requête `let` : exemples

```
# let x = 3 + 4;;
val x : int = 7
# x;;
- : int 7
# let f = fun x y -> 2 * x + 7;;
val f : int -> 'a -> int = <fun>
# f;;
- : int -> 'a -> int = <fun>
# f 3;;
- : 'a -> int = <fun>
# f 3 4;;
- : int = 13
# x;;
- : int = 7
```

Remarque: certains types peuvent être détectés comme étant arbitraires. Dans ce cas, ces types sont notés `'a`, `'b`, `'c`, etc. et la fonction pourra être utilisée pour n'importe quel type de l'argument correspondant.

Fonction nommée

Puisqu'une fonction anonyme est une expression, on peut la nommer avec la requête `let`.

```
let cube = fun x -> x * x * x
```

La requête `let f = fun x y ... -> corps` s'écrit de fait en utilisant la syntaxe **plus spécifique**:

`let f x y ... = corps`.

```
let cube x = x * x * x
```

```
# let f x y = 2 * x + y;;
val f : int -> int -> int = <fun>
# f 3 4;;
- : int = 10
```

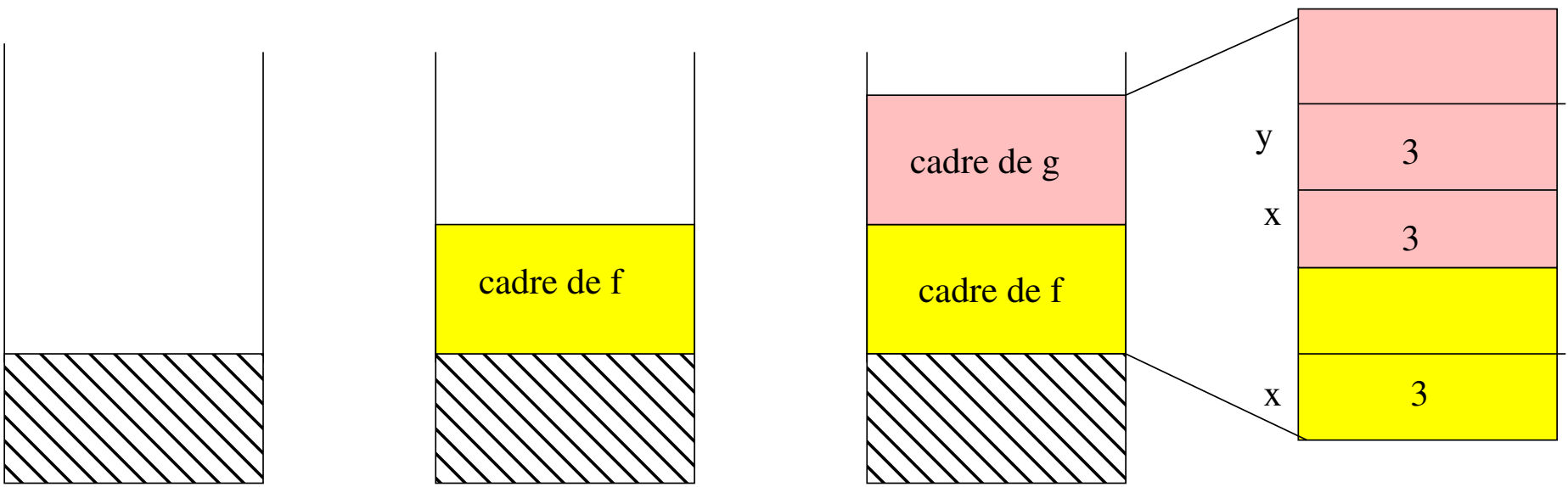
Appel de fonction

Les appels de fonction se font toujours par valeur.
Pour gérer les appels de fonction, le système utilise une *pile*. Lors d'un appel un cadre (frame) est créé et placé en sommet de pile. Il contient en particulier les variables lexicales correspondant aux paramètres de la fonction qui seront initialisées avec les valeurs résultant de l'évaluation des arguments lors du passage de paramètres. Lorsque la fonction retourne, le cadre est dépilé.

Appel de fonction

```
# let g x y = 2 * x + y;;
val g : int -> int -> int = <fun>
# let f x = g x x;;
val f : int -> int = <fun>
# f 3;;
- : int = 9
```

```
# let g x y = 2 * x + y;;
val g : int -> int -> int = <fun>
# let f x = g x x;;
val f : int -> int = <fun>
# f 3;;
- : int = 9
```

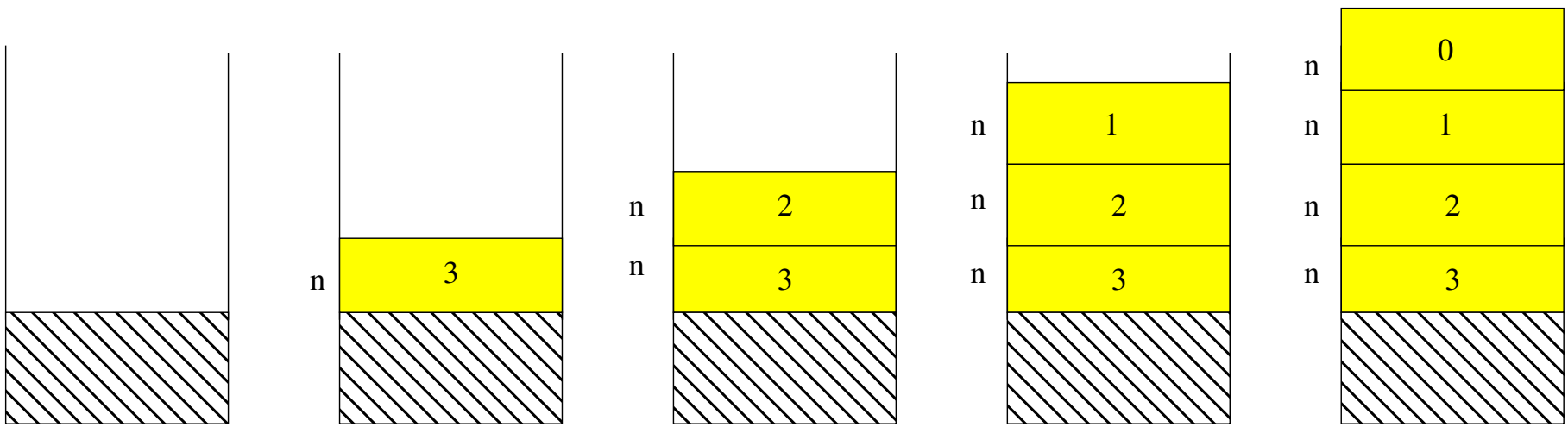


Récursivité

Ce mécanisme permet la récursivité [3]

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

# fact 3;;
- : int = 6
```

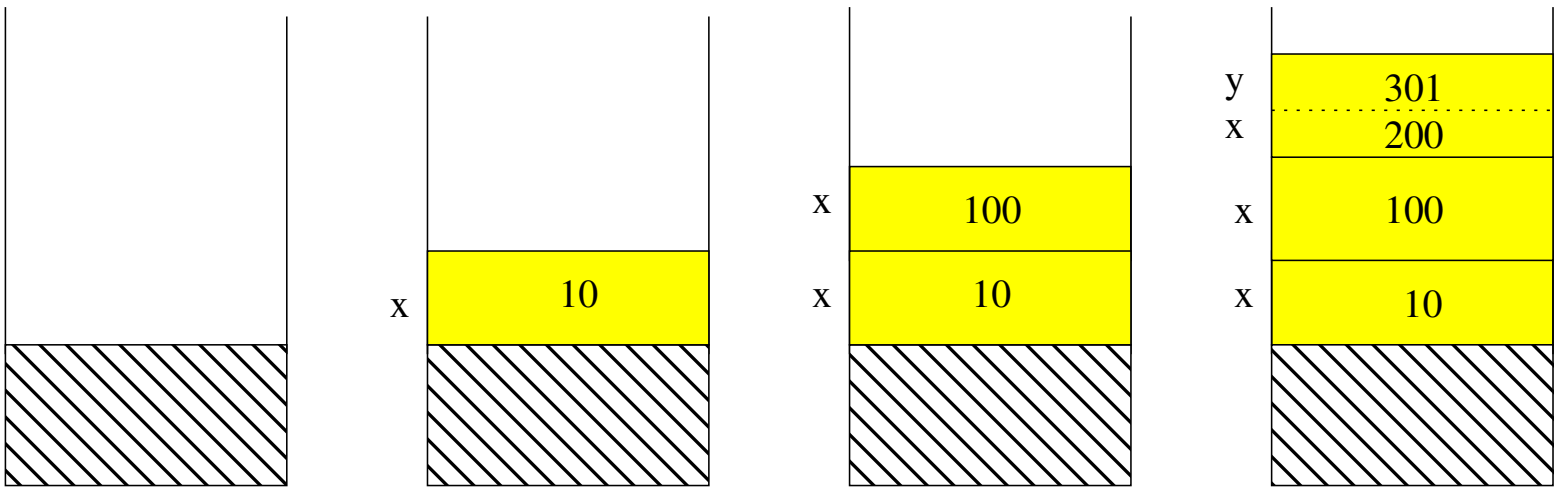


[3] Cobol, anciennes versions de Fortran: pas de récursivité: programmer la pile soi-même

Évaluation d'un `let ... in`

Le mécanisme de pile est aussi utilisé pour l'évaluation d'une expression `let x = ... in`
variable lexicale `x` créée sur la pile le temps de l'exécution du `let`

```
# let x = 10 in
      let x = x * x in
      let x = 2 * x
      and y = 3 * x + 1 in
      (x, y);;
- : int * int = (200, 301)
```



Fonctions récursives

Une fonction *récursive* est appelée dans sa propre définition. Pour écrire une fonction récursive, il est donc nécessaire de *nommer* la fonction, pour pouvoir l'appeler par son nom dans sa définition. En OCaml, la construction `let f ... = ...` ne permet de définir que des fonctions **non** récursives.
À noter que un langage disposant d'une conditionnelle et de fonctions récursives a la puissance des machines de Turing, c'est à dire permet de programmer tout ce qui est programmable.

Requête `let rec`

Pour définir une fonction récursive, il faut explicitement le préciser par la requête `let rec`. Par exemple, la fonction factorielle définie sur les entiers naturels s'écrit de façon naïve:

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

Principe de la récursivité

fonction récursive basée sur ordre bien fondé
ordre bien fondé $<$:

- pas de suite infinie strictement décroissante
- nombre fini d'éléments minimaux

Sous ces hypothèses, on peut

- décrire le calcul sur les éléments minimaux (cas de base ou d'arrêt)
- décrire le calcul des éléments non minimaux en fonction d'éléments plus petits (cas récursifs)

Récursivité (suite)

- En particulier, pour une fonction `f` d'un entier naturel (comme factorielle), $<$ est un ordre bien fondé pour les entiers naturels et il existe un unique élément minimal: 0.
- Il suffit d'indiquer comment calculer `f 0` puis d'indiquer comment calculer `f n` pour `n` $>$ `0` en fonction de `f i` pour `i` $<$ `n`.
- il faut que les appels récursifs se fassent sur des arguments plus petits que ceux passés en paramètres et il faut prévoir les cas d'arrêt (pour tous les éléments minimaux qui se calculent sans appel récursif).

Exemples d'ordres bien fondés

- Entiers naturels munis de $<$. On définit la fonction pour 0 puis pour $n > 0$ en utilisant des appels récursifs sur des valeurs $< n$.
- Couples d'entiers naturels munis de l'ordre lexicographique.

$$(x, y) < (x', y') \text{ ssi } \text{ soit } x < x', \text{ soit } x = x' \text{ et } y < y'$$

On définit la fonction pour $(0, 0)$, puis pour $(x, y) > (0, 0)$ avec des appels sur des valeurs $< (x, y)$.

Récursivité et complexité

Pour évaluer l'efficacité d'une fonction, on évalue le nombre d'opérations élémentaires

- sur une entrée de taille n et dans le pire des cas

Dans le cas d'une fonction récursive, on est souvent amené à résoudre des équations récurrentes.

Récursivité et complexité

Exemple de calcul de 2^x pour $x > 0$.

```
let rec pow2 x =
  if x = 0 then 1
  else 2 * pow2 (x - 1)
```

Comptons le nombre $M(x)$ de multiplications effectuées:

$$M(0)=0$$
$$M(x)=M(x-1)+1$$

On en déduit:

$$M(x) = x$$

et donc que cette fonction est en $O(x)$.
On peut arriver à $O(log(x))$ en utilisant la méthode à la grecque.

Exponentiation à la grecque

On peut faire mieux en utilisant la méthode à la grecque.

```
let rec pow2_log x =
if x = 0 then 1
else let y = pow2_log (x / 2) in
     let p = y * y in
     if x mod 2 = 0 then p
     else p * 2
```

Comptons le nombre $G(x)$ de multiplications effectuées:

$$\begin{aligned} G(0) &= 0 \\ G(x) &<= G(x/2) + 2 \end{aligned}$$

On en déduit qu'il y a $O(log(x))$ multiplications.

Introduction aux types

évaluation d'une expression $\Longrightarrow$ type et valeur

```
# 10 + 4;;
- : int = 14
```

Un type est un ensemble de valeurs.

Exemples:

type Booléen `bool = { true, false }`

type Caractères `char = { 'a', 'A', ...}`

Un certain nombre d'opérateurs (fonctions) sont prédéfinis pour les types prédéfinis.

Les fonctions OCaML sont typées: elles s'appliquent à des arguments ayant chacun un type défini et retournent une valeur d'un type également défini.

Si on utilise une fonction avec au moins un argument n'ayant pas le bon type, l'évaluation s'arrête à la compilation avec une erreur et la fonction n'est pas appelée.

Un type avec un ensemble d'opérateurs s'appliquant sur les valeurs d'un ensemble de types peut être appelé un type abstrait.

Inférence et vérification de types

à la compilation, OCaML infère le type de chaque expression ⇒ détection de certaines erreurs

si échec ⇒ pas d'évaluation

infère le type le plus général possible (variables de type `'a, 'b, ...`)

```
# let f x = x;;
val f : 'a -> 'a = <fun>
# let f x = x, x;;
val f : 'a -> 'a * 'a = <fun>
# let f x y = x, y;;
val f : 'a -> 'b -> 'a * 'b = <fun>
# let f x = x + 1;;
val f : int -> int = <fun>
# let f x = x, x;;
val f : 'a -> 'a * 'a = <fun>
# let f x = x ^ x;;
val f : string -> string = <fun>
# let f x = x ^ (x + 1);;
Error: This expression has type string but an expression
       was expected of type int
```

Opérateurs de types

Les types peuvent être combinés à l'aide d'opérateurs de types pour obtenir de nouveaux types.
Exemples

- type contenant les couples constitués d'un entier et d'un flottant
- type des fonctions des entiers vers les booléens

Opérateur de fonction

L'opérateur de type `->` permet d'obtenir le type d'une fonction d'une variable d'un type vers un autre type

```
# let carre x = x * x;;
  val carre : int -> int = <fun>
# sin;;
- : float -> float = <fun>
```

`carre` fonction prend en paramètre un entier, retourne un entier
`sin` prend en paramètre un flottant, retourne un flottant

Parenthésage

Le parenthésage par défaut d'une séquence d'opérateurs `->` est de droite à gauche `a1 -> a2 -> ... an` est équivalent à `a1 -> (a2 -> (... -> (an-1-> an)...)` contrairement à l'appel de fonction qui est de gauche à droite.

```
# max;;
- : 'a -> 'a -> 'a = <fun>
# max 3;;
- : int -> int = <fun>
# max 3 4;;
- : int = 4
```

`max` est une fonction de deux arguments, `max 3` est une fonction d'un argument entier et fera le max entre cet argument et 3. `max 3 4` est un entier.

- Exemple de la fonction `compose f g`

Opérateur de produit cartésien (couples)

L'opérateur `*` est l'opérateur de produit cartésien. Le produit cartésien de A et de B est l'ensemble:

$$A \times B = \{(a, b) \mid a \in A \text{ et } b \in B\}$$

Il permet de représenter l'ensemble des tuples d'éléments appartement respectivement à un type donné. Exemples: `int * int`, `int * float * int`

```
# 2, 3;;
  - : int * int = (2, 3)
# fst (2, 3)
- : int 2
# snd (2, 3)
- : int 3
# 2, 3.5, 'a';;
- : int * float * char = (2, 3.5, 'a')
```

À noter les accesseurs `fst` et `snd`.

```
# let couple_square x = x, x *x;;
val couple_square : int -> int * int = <fun>
# let pair x = x, x;;
val pair : 'a -> 'a * 'a = <fun>
```

À noter dans le dernier exemple l'apparition de `'a` qui est une variable pouvant représenter un type quelconque. En effet, le corps de la fonction permet d'inférer que le résultat est un couple mais ne permet pas d'obtenir le type de `x`. Nous verrons plus loin cette notion de type paramétré par une variable de type.

Produit cartésien généralisé (tuples)

Le produit cartésien binaire se généralise aux tuples.

$$E_1 \times E_2 \times \cdots \times E_n = \{(e_1, e_2, \cdots, e_n) \mid e_i \in E_i \forall i, 1 \leq i \leq n\}$$

Exemples:

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

Les tuples peuvent être paramètres des fonctions.

```
# let s3 (i, j, f) = i + j + int_of_float f;;
val s3 : int * int * float -> int = <fun>
# s3 (1, 2, 3.);;
- : int = 6
```

Une fonction peut retourner un tuple:

Filtrage

On peut récupérer les valeurs d'un tuple par filtrage:

```
# let tuple = 1, "deux", 3.5;;
val tuple : int * string * float = (1, "deux", 3.5)
# let i, str, f = tuple in i + int_of_float f, str;;
- : int * string = (4, "deux")
```

Requête `type`

requête `type` permet de donner un nom à un type (en général composé)

```
# type point2D = Point of float * float;;
type point2D = Point of float * float
# Point(1.2, 3.4);;
- : point2D = Point (1.2, 3.4)
```

`point2D` est le nom de type choisi. `Point` est le nom de constructeur choisi pour représenter un point. `type`, `of`, `float` sont prédéfinis en OCaML.
Le séparateur `|` correspond à une union (ou somme) de types.

```
# type int_or_infinity = Int of int | Infinity;;
type int_or_infinity = Int of int | Infinity
```

Des valeurs de ce type sont: `Int 5`, `Int (-2)`, `Infinity`.

```
type int_or_infinity = Int of int | Infinity
# let div n d =
  if d = 0 then
    if n = 0 then failwith "undefined form 0/0"
    else Infinity
  else Int(n/d);;
val div : int -> int -> int_or_infinity = <fun>
# div 3 0;;
- : int_or_infinity = Infinity
# div 3 2;;
- : int_or_infinity = Int 1
# div 0 0;;
Exception: Failure "undefined form 0/0".
# type form = Square of float | Circle of float | Rectangle
type form = Square of float | Circle of float | Rectangle o
# Rectangle (10., 3.);;
- : form = Rectangle (10., 3.)
# Square(3.4);;
- : formes = Square 3.4
```

La construction `match`

La construction `match` permet d'inspecter la forme d'une valeur et de récupérer parties de la valeur dans des variables locales.

```
# let perimeter_rect w h = 2. *. (w +. h);;
val perimeter_rect : float -> float -> float = <fun>
# let perimeter_circle r = 2. *. 3.14 *. r;;
val perimeter_circle : float -> float = <fun>
# let perimeter form =
  match form with
    Rectangle(w, h) -> perimeter_rect w h
  | Circle r -> perimeter_circle r
  | Square c -> perimeter_rect c c;;
val perimeter : formes -> float = <fun>
```

Variable anomyme

Le caractère `_` (souligné ou "tiret du 8" sur un clavier AZERTY) correspond à une variable anonyme.
On peut l'utiliser

- à gauche du `=` dans un `let` ou `let ... in`

```
# let x, _, z = 1,2,3 in x, z;;
- : int * int = (1, 3)
```

- dans un motif d'un `match`

```
let est_une_figure carte =
  match carte with
    As _ -> false
  | Numero(_, _) -> false
  | _ -> true
```

- dans un fichier `.ml` pour mémoriser une expression

```
let _ = couleur_carte (Numero(7, Pique))
let _ = couleur_carte (As Coeur)
```

Regroupement des clauses d'un `match`

```
match carte with
  As c -> c
| Roi c -> c
| Dame c -> c
| Valet c -> c
| Numero(_, c) -> c
```

```
match carte with
  As c | Roi c | Dame c | Valet c | Numero(_, c) -> c
```

```
match carte with
  As _ -> false
| Numero(_,_) -> false
| _ -> true
```

```
match carte with
  As _
| Numero(_,_) -> false
| _ -> true
```

Représentation d'un point du plan par un complexe

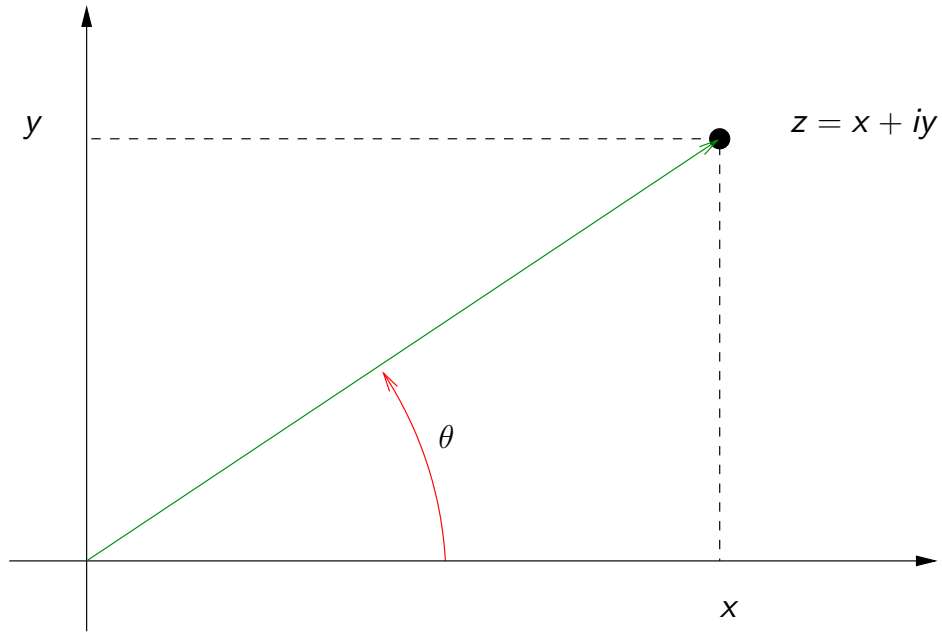


Figure: Plan complexe

plan muni d'un repère orthonormé; au point de coordonnées (x, y), on associe e nombre complexe $z = x + iy$, son affixe.
Pour représenter un point du plan, on utilise le type

```
type mycomplex = C of float * float
```

```
type point = mycomplex
```

Opérations sur les complexes

On écrira en TM les fonctions et variables suivantes:

1. fonction constructeur `make_complex` qui permet de construire un objet de type `mycomplex` fabrique un complexe à partir de ses parties réelle et imaginaire
2. accesseur `realpart` qui permet de récupérer la partie réelle d'un complexe.
3. accesseur `imagpart` qui permet de récupérer la partie imaginaire d'un complexe.
4. variable `c_origin` contenant le point de coordonnées $(0, 0)$.
5. la variable `c_i` ayant pour valeur le complexe $i = (0, 1)$,
6. opérations

   `c_abs, c_sum c1 c2, c_dif c1 c2, c_opp`,

   `c_mul c1 c2, c_abs c, c_sca lambda c, c_exp c`

   qui permettent respectivement de calculer la valeur absolue d'un complexe, la somme, la différence, l'opposé, le produit de deux complexes, la multiplication d'un complexe par un scalaire (float), l'exponentielle complexe.

Transformations dans le plan complexe

Une transformation F du plan transforme tout point P en son image $P' = F(P)$. On peut décrire la transformation F par la fonction f qui appliquée à z l'affixe de P donne z' l'affixe de P'.

$$\begin{array}{rccc} f: & \mathbb{C} & \to & \mathbb{C} \\ & z & \mapsto & z' = f(z) \end{array}$$

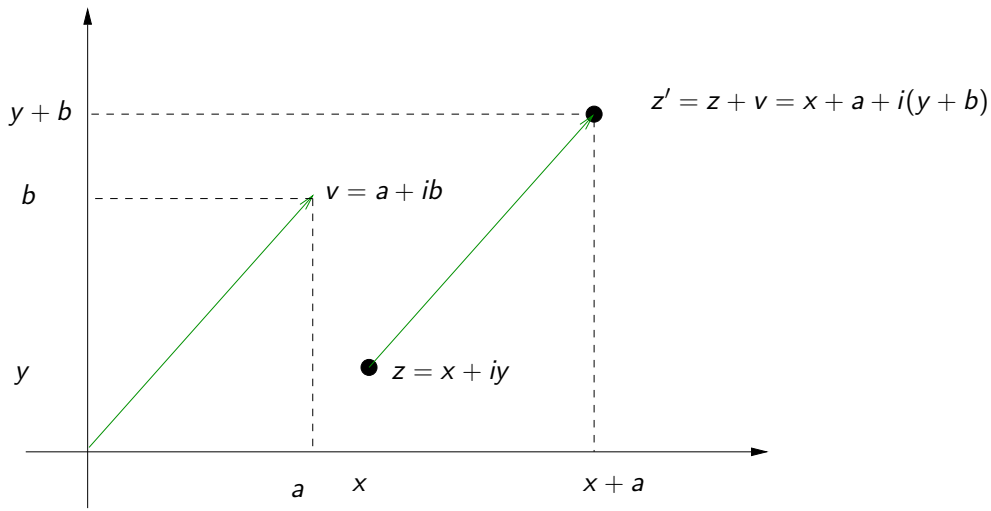


Figure: Translation de vecteur (a, b)

On écrira en TM la fonction `translate c vector`.

Transformations plan complexe (Rotation)

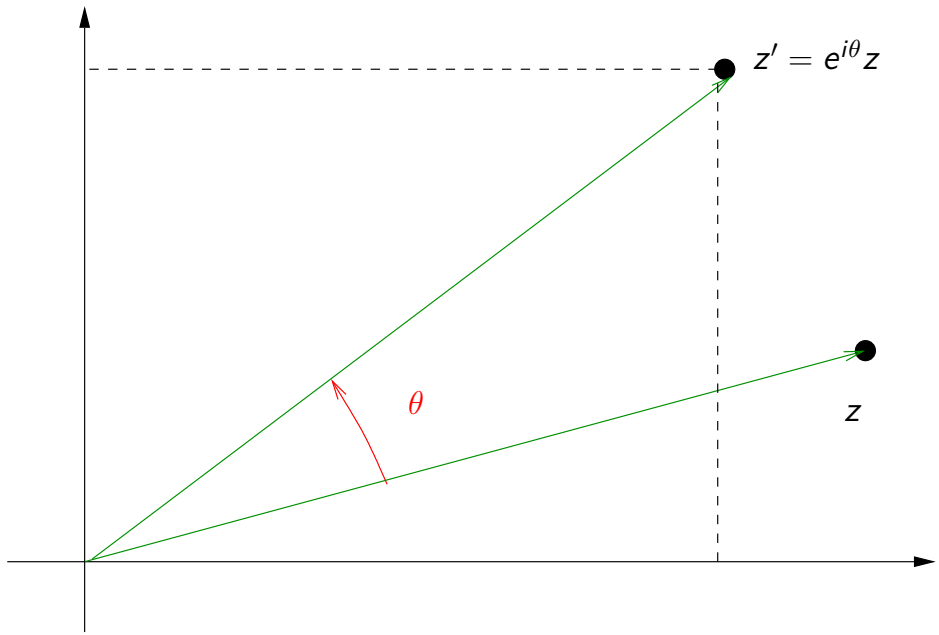


Figure: Rotation d'angle θ

On écrira en TM la fonction `rotate0 c angle`.

Jeux de test

```
let test_fact0 () = (* assert *)
  begin
    assert(fact 0 = 1);
    assert(fact 6 = 720);
  end

let unit_test name condition =
  begin
    print_endline ("Testing " ^ name);
    if condition then print_endline "Passed"
    else begin print_endline "Failed";
                flush stdout;
         end
  end
let test_fact () =
  begin unit_test "fact 0" (fact 0 = 1);
        unit_test "fact 6" (fact 6 = 720);
        unit_test "toto" (fact 6 = 720);
```

Comparaison de langages de programmation

"Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Proto-typing Productivity de 1994, P. Hudak et M. P. Jones
comparaison entre différents langages de
Le logiciel implémenté manipule des zones géométriques
une version simplifiée (mais pas tant que ça) de ce logiciel

Zones du plan représentées par leur fonction caractéristique

On représente une telle zone par sa fonction caractéristique, c'est-à-dire par la fonction booléenne qui prend un point en argument, et retourne `true` si le `point` appartient à la zone et `false` sinon.
Ainsi, la zone représentant le plan tout entier est représentée par la variable `everywhere` suivante:

```
# let everywhere = fun point -> true;;
val everywhere : 'a -> bool = <fun>
```

On peut forcer le type des variables et des paramètres en les faisant suivre de `:` suivi du nom du type.

```
# let everywhere : zone = fun point -> true;;
val everywhere : zone = <fun>
```

Appartenance d'un point à une zone

```
# let c_origin = make_point 0. 0.
# point_in_zone_p c_origin nowhere;;
- : bool = false
# point_in_zone_p c_origin everywhere;;
- : bool = true
```

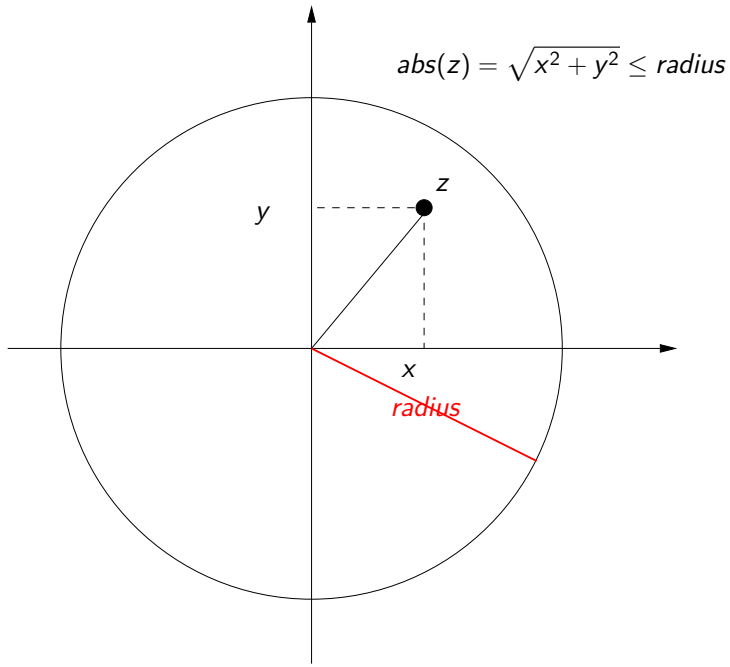
```
# let point_in_zone_p point zone = zone point;;
val point_in_zone_p : 'a -> ('a -> 'b) -> 'b = <fun>
```

En forçant les types:

```
let point_in_zone point (zone:zone) = zone point;;
val point_in_zone : point -> zone -> bool = <fun>
```

Exemple de zone: disque

disque de rayon *radius* centré à l'origine.




```
let make_disk0 radius =
    fun point -> c_abs point <= radius
```

Exemple de transformation: translation

Translation de vecteur $\overrightarrow{u}$ d'affixe z_u.
Le point P' d'affixe z' est dans la nouvelle zone si son antécédent P d'affixe z est dans la zone.

$$z' = z + z_u \Rightarrow z = z' - z_u$$

```
let translate c vector = c_sum c vector
```

Pour qu'un point soit dans la zone translatée, il faut que le point obtenu par la translation inverse soit dans la zone initiale:

```
let translate_zone zone vector =
  fun p -> point_in_zone_p
             (translate p (c_opp vector)) zone
```

Visualisation des zones

À voir en TD machine.

Déclaration d'un type enregistrement

Au lieu d'utiliser des tuples dans lesquels l'ordre des éléments a une importance, on peut utiliser des enregistrements dans lesquels les éléments sont nommés.

```
type <nom_de_type> =
{
  label1 : type1;
  label2 : type2;
  ...
  labeln : typen;
}
```

On peut ainsi redéfinir le type `complex` vu précédemment:

```
type complex = { real : float; imag : float; }
```

Constructeurs et accesseurs

Une valeur du type `nom_de_type` s'écrit:

```
{
  label1 = valeur1;
  label2 = valeur2;
  ...
  labeln = valeurn;
}
```

L'ordre des lignes n'a pas d'importance.

Exemple d'enregistrement

Par exemple, pour le type `complex` :

```
# { real = 1.0; imag = 0. };;
- : complex = {real = 1.; imag = 0.}
# let i = { real = 0.; imag = 1.0 };;
val i : complex = {real = 0.; imag = 1.}
# let c = { imag = 3.; real = 1.0 };;
val c : complex = {real = 1.; imag = 3.}
```

Étant donnée une valeur de type enregistrement, on *accède* à un des champs en suffixant la valeur par le champs voulu. Exemple:

```
# { real = 1.0; imag = 0. }.real;;
- : float = 1.
# i.imag;;
- : float = 1.
```

Types récursifs (ou inductifs)

La récursivité est utilisable dans la définition des types. Ceci permet en particulier de construire des types infinis.
On peut par exemple représenter les listes d'entiers[4].

```
# type intlist = NI | CI of int * intlist;; (* NI: liste d
type intlist = NI | CI of int * intlist
# CI(1, CI(2, CI(3, NI)));;
- : intlist = CI (1, CI (2, CI (3, NI)))
# NI;;
- : intlist = NI
# type 'a truclist = NT | CT of 'a * 'a truclist;;
type 'a truclist = NT | CT of 'a * 'a truclist
# NT;;
- : 'a truclist = NT
# CT('a', CT('b', CT('c', NT)));;
- : char truclist = CT ('a', CT ('b', CT ('c', NT)))
```

[4] Définition en fait inutile, puis qu'il existe un type prédéfini pour les listes que nous verrons page 81

Listes

Comment définir un type liste générique (liste d'éléments appartenant tous à un même type non fixé)?

```
type 'a mylist = Nil | C of 'a * 'a mylist
```

Exemples de fonctions utilisant ce type

- `length`
- `make_list`
- `concat`

Récursivité terminale

Un appel récursif est dit terminal si il est retourné directement par la fonction, c'est-à-dire qu'aucune opération n'est faite avec l'appel récursif mise à part le retour.
Une fonction récursive est dite récursive terminale[5] si tous ses appels récursifs sont terminaux.

La fonction récursive `fact` définie ci-dessous n'est pas récursive terminale car la multiplication par `n` est effectuée entre l'appel récursif `fact (n - 1)` et le retour de la fonction.

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

[5] tail recursive en anglais

Récursivité terminale

fonction pas récursive terminale

$\Rightarrow$ appels empilés (pile d'exécution)

lors de l'appel à `fact 4` seront empilés les appels: `fact 4`, `fact 3`, `fact 2`, `fact 1`, `fact 0`.

Le dernier appel `fact 0` retourne `1`,

`fact 1` retourne `1 * 1 = 1`,

`fact 2` retourne `2 * 1 = 2`,

`fact 3` retourne `3 * 2 = 6`,

`fact 4` retourne `4 * 6 = 24`.

empilement d'appels $\Rightarrow$ débordement de la pile (Stack overflow)

injustifié dans le cas d'un tel calcul qui dans un langage classique se ferait avec une simple boucle.

```
def fact (n):
    p = 1
    for i in range(2, n + 1):
        p *= i
    return p
```

Récursivité terminale

avantage: pas nécessaire d'empiler les appels; l'appel récursif remplace l'appel précédent.
$\Rightarrow$ pas de débordement de pile

- pas toujours possible d'obtenir une fonction récursive terminale
- Souvent, passage d'une fonction non récursive terminale à une fonction récursive terminale par ajout d'un paramètre qui joue le rôle d'accumulateur et dans lequel on calcule la valeur à l'appel et non au retour de l'appel récursif.

- fonction auxiliaire `fact_aux n p` récursive terminale
- `fact` s'écrit en appelant `fact_aux n 1`, `1` étant l'élément neutre pour le produit.

```
let rec fact_aux n p =
  if n = 0 then p
  else fact_aux (n - 1) (n * p)

let fact n = fact_aux n 1
```

`fact_aux 4 1` remplacé par

`fact_aux 3 4` remplacé par

`fact_aux 2 12` remplacé par

`fact_aux 1 24` remplacé par

`fact_aux 0 24` retourne `24`.

`p` (resp. `n`) joue même rôle que `p` (resp. `i`):

```
def fact (n):
    p = 1
    for i in range(n, 0, -1):
        p *= i
    return p
```

fonction auxiliaire à l'intérieur de la fonction principale à l'aide d'un `let rec ... in ...` (sauf si fonction aux utile dans un autre contexte).

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

Exemples: `make_list_rt`, `length_rt`, `reverse_rt`

Type `'a list` prédéfini en `OCaML`

pas nécessaire de définir un type `'a mylist`
(comme vu précédemment)

type prédéfini `'a list` est fourni par le module `List`

listes homogènes (comme dans le cas du type `'a mylist`):
tous les éléments sont d'un même type.

fonctions de ce module seront accessibles avec le préfixe `List.`
(utiliser la complétion pour voir toutes les fonctions du module)

Par exemple, `List.length` (la longueur d'une liste)

Constructeurs et accesseurs pour le type `list`

- constructeur de liste vide est `[]` au lieu de `Nil` pour le type `mylist`.
- `constructeur` permettant de rajouter un élément à une liste est `::` mais contrairement au constructeur `C` du type `mylist`, il s'utilise en notation infixe.

  Ainsi on écrit `e :: l` au lieu de `C(e,l)` dans le type `mylist`.

Les `accesseurs` associés à ce constructeur sont `List.hd` et `List.tl` pour récupérer respectivement la tête (head) `e` et la queue (tail) `l` de la liste `e :: l`.
Ces accesseurs sont peu utilisés du fait que l'on utilisera le plus souvent la construction `match` pour déconstruire une liste.

La fonction de calcul de la longueur d'une liste qui s'écrivait avec le type `mylist`

```
type 'a mylist = Nil | C of 'a * 'a mylist
let rec mylist_length l =
    match l with
      Nil -> 0
    | C(_, t) -> 1 + mylist_length t
```

s'écrit comme suit avec le type prédéfini et la construction `match`

```
let rec list_length l =
    match l with
      [] -> 0
    | _ :: t -> 1 + list_length t
```

ou encore sans utiliser `match`

```
 let rec list_length l =
  if l = [] then 0
  else 1 + list_length (List.tl l)
```

Format externe des listes

La notation `e1 :: e2 :: e3 ... :: en :: []` n'étant pas très agréable à lire, `OCaML` utilise un *format externe* pour écrire et lire des listes: `[e1; e2; ...; en]` est le format sous lequel `OCaML` affiche une liste et un format que l'utilisateur peut utiliser pour entrer une liste.

Exemples:

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# [1.; 2.; 3.];;
- : float list = [1.; 2.; 3.]
# List.hd [1; 2; 3];;
- : int = 1
# List.tl [1; 2; 3];;
- : int list = [2; 3]
# 3 * 3 :: [1; 2; 3];;
- : int list = [9; 1; 2; 3]
# let x = 4;;
val x : int = 4
```

Concaténation de listes

La fonction prédéfinie `List.append l1 l2` retourne une liste constituée des éléments de `l1` suivis des éléments de `l2`.

Exemples:

```
# List.append [1; 2; 3] [4; 5];;
- : int list = [1; 2; 3; 4; 5]
# List.append [1; 2; 3] [];;
- : int list = [1; 2; 3]
# List.append [] [3; 4; 5];;
- : int list = [3; 4; 5]
```

Il existe une version infixe de cette fonction: l'opérateur `@`.

Exemples:

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

Concaténation

à utiliser avec parcimonie.
En effet, sa complexité est en $O(len(l_1))$ car il entraîne une copie de la liste `l1`.
Il peut servir ponctuellement à ajouter[6] un élément `e` en fin d'une liste `l` en utilistant l'expression `l @ [e]`
Par contre l'ajout d'un élément `e` en tête d'une liste `l` se fait en temps constant $O(1)$ grâce à l'expression: `e :: l`.
Le plus souvent, il est préférable de construire une liste à l'envers puis de la retourner en utilisant la fonction `List.rev l` (linéaire par rapport à la longueur de la liste).

[6] en fait, construire une nouvelle liste ayant les mêmes éléments que `l` et `e` à la fin

Exemples de fonctions simples sur les listes

- `List.length`
- `List.mem`
- `List.append`
- `List.rev`
- `List.sort`
- `List.filter`
- `List.map`
- `List.find`, `List.find_all`
- ...

Et il y en a d'autres:

- utiliser `List.` *complétion* pour voir leur nom
- taper leur nom pour voir leur type

type `option` prédéfini

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

type de retour uniforme pour les fonctions ne retournant pas toujours une valeur comme les fonctions de recherche par exemple.

Exemples en direct

```
let rec find_if pred l =
  match l with
    [] -> None
  | e :: t -> if pred e then Some e
              else find_if pred t
```

clause when

```
type pair = P of int * int

let pair_cmp pair =
  let P(x, y) = pair in
  if x > y then 1 else if y > x then -1 else 0

let pair_cmp pair =
  match pair with
    P(x, y) -> if x > y then 1
               else if y > x then -1 else 0

let pair_cmp pair =
  match pair with
    P(x, y) when x > y -> 1
  | P(x, y) when x < y -> -1
  | _ -> 0
```

Filtrage direct `function`

```
let pair_cmp pair =
  match pair with
    P(x, y) when x > y -> 1
  | P(x, y) when x < y -> 1
  | _ -> 0

let pair_cmp = function
    P(x, y) when x > y -> 1
  | P(x, y) when x < y -> 1
  | _ -> 0
```

Comparaison pratique

- utiliser `Sys.time` (appel à la command Unix `time`)
- faire la différence entre les temps de fin et de début de l'excécution

```
utop[30]> Sys.time();;
- : float = 6.536402
utop[31]> let start = Sys.time();;
val start : float = 6.540147
utop[32]> Sys.time() -. start;;
- : float = 0.00526800000000005
```

Exemple de comparaison pratique

```
let time f =
  let start = Sys.time() in
  let _ = f () in
  Sys.time() -. start
```

permet de mesurer le temps d'exécution de la fonction `f` sans argument passée en paramètre.

```
utop[36]> time;;
- : (unit -> 'a) -> float = <fun>
utop[37]> time (fun () -> List.mem 100 (iota 100));;
- : float = 8.00000000111822374e-06
```

Exemples avec trois versions de la fonction qui renverse une liste. (voir `test-efficacite.ml`)

Fonction `List.fold_left`

```
utop[6]> List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

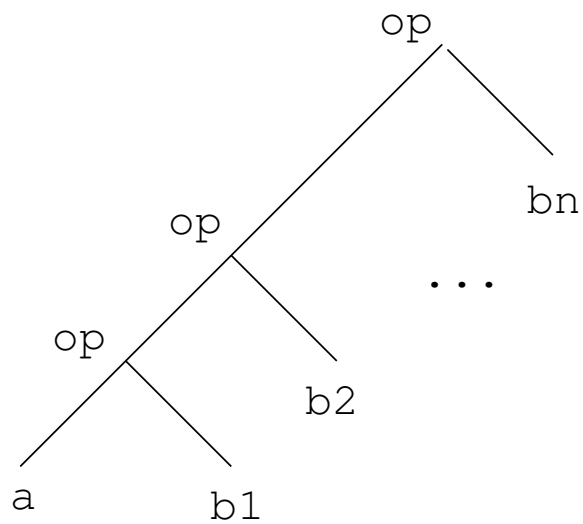
`List.fold_left op a l`

`op` est un opérateur binaire `'a -> 'b -> 'a`

`a` est une valeur de type `'a`

`l` une liste d'éléments de type `'b` : `[b1; b2; ...; bn]`

calcule `op (op ... (op (op a b1) b2) ... ) bn`



Fonction `List.fold_right`

```
utop[7]> List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

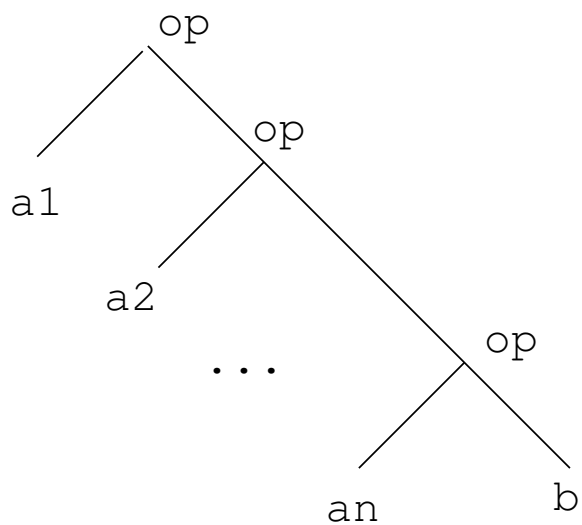
`List.fold_right op l b`

`op` est un opérateur binaire `'a -> 'b -> 'b`

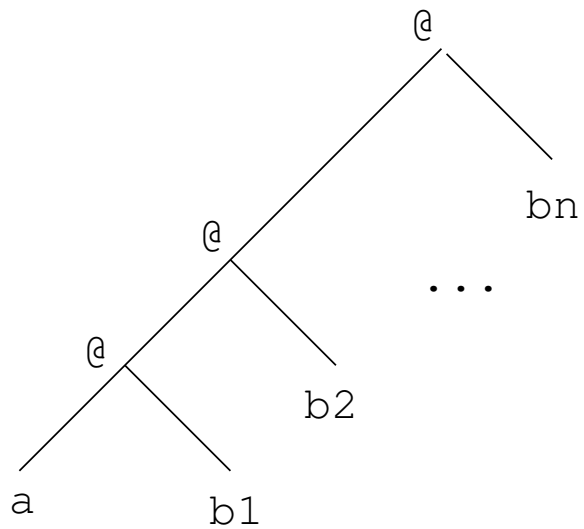
`b` est une valeur de type `'b`

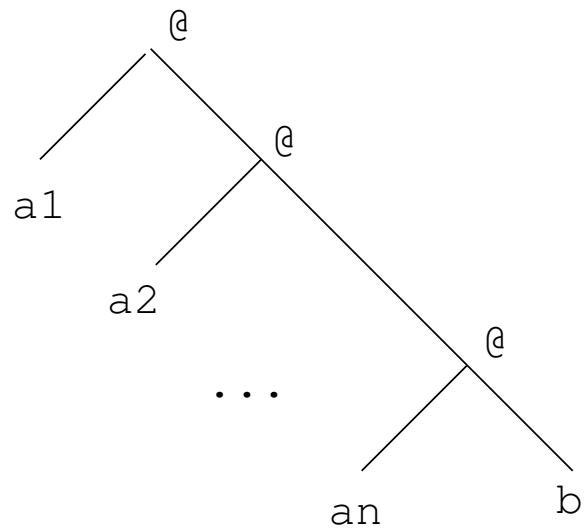
`l` une liste d'éléments de type `'a` : `[a1; a2; ...; an]`

calcule `op a1 (op a2 (... (op an b))...)`



`List.fold_right` vs `List.fold_left`





Comparaison théorique: notion de complexité

estimer théoriquement l'efficacité d'une fonction

- efficacité en temps (nombre d'opérations élémentaires)
- efficacité en espace (espace alloué)
- temps $>=$ espace

Notation $\mathcal{O}$:
La fonction f est dite en $\mathcal{O}(g)$ ssi

$$\exists k \in \mathbb{N}, \exists c > 0, \forall n > k, f(n) \leq cg(n)$$

Exemple: $\mathcal{O}(n \mapsto log_2(n))$
Par abus de notation: $\mathcal{O}(log_2(n))$

Exemples: $\mathcal{O}(log_2(n))$, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, ...

Comparaison théorique: exemple

```
let rec append l1 l2 =
  match l1 with
    [] -> l2
  | h :: t -> h :: append t l2
```

1. Combien d'appels récursifs de `append` ?
2. Combien de fois l'opérateur `::` est-il utilisé?

La récursion se faisant sur `l1`, la complexité dépend uniquement de la longueur de la liste `l1`.
Soit $a(n)$ le nombre d'appels récursifs pour `l1` de longueur n.

$$\begin{cases} a(0)=0 \\ a(n)=1+a(n-1) \end{cases}$$

se résout en $a(n) = n$.
le nombre d'appel à `::` est égal au nombre d'appels récursifs.
La fonction est en $\mathcal{O}(len(l_1))$.

Comparaison théorique: reverse quadratique

```
let rec reverse l =
  match l with
    [] -> []
  | h :: t -> append (reverse t) [h]
```

Soit $c(n)$ le nombre d'utilisation de `::` lors d'un appel à `reverse l` pour une liste `l` de longueur n.

$$\begin{cases} c(0)=0 \\ c(n)=a(n-1)+c(n-1) \end{cases}$$

se résout en $n(n-1)/2$.
La fonction est en $\mathcal{O}(len(l)^2)$.

Comparaison théorique: reverse linéaire

```
let rec rev_append l acc = (* O(len(l)) *)
  match l with
    [] -> acc
    | h :: t -> rev_append t (h :: acc)

let reverse l = rev_append l []
```

On peut montrer que le nombre d'appels récursifs est égal à n si n est la longueur de la liste `l`.
La fonction est en $\mathcal{O}(len(l))$.

Programmation modulaire

La programmation modulaire consiste à découper d'un programme en plusieurs modules.

Un module est un morceau de code définissant des types de données et un ensemble d'opérations sur ces types.

On peut voir ça comme l'implémentation d'un type abstrait.

Les modules ne sont pas propres à OCaML.

Comme en C, on aura une partie interface (`.mli`/`.h`) et une partie implémentation (`.ml`/`.c`).

Intérêt des modules

Les modules permettent de résoudre un certain nombre de problèmes qui se posent en programmation.

- Découpage de la difficulté: une petite entité est plus facile à comprendre, à déboguer (diviser pour régner)
- Réutilisabilité: définir des petites entités réutilisables
- Masquage de l'implémentation: pouvoir changer d'implémentation sans perturber les clients du module
- Contrôle de la visibilité des éléments encapsulés
- Sous-espaces de noms
- Généricité (polymorphisme)

Modules OCaML

Les modules `OCaML` interviennent dans le cadre d'un langage statiquement typé.

Ceci entraîne des contraintes supplémentaires sur la façon de compiler les modules.

La partie implémentation d'un module se fait dans un fichier `.ml`.

La partie visible est l'interface (ou signature) est dans un fichier `.mli` ou est inférée automatiquement.

Définition automatique d'un module

Un fichier `nom.ml` définit automatiquement un module de type `nom`.
Par exemple, le code suivant placé dans le fichier `point.ml` définit un module `Point`. Les noms de module commencent toujours par une majuscule.

```
type point = P of float * float
let p_x point = let P(x, _) = point in x
let p_y point = let P(x, _) = point in y
let p_origin = make_point 0.0 0.0
let p_i = make_point 0.0 (-1.)
let distance x y x' y' = sqrt (x *. x' +. y *. y')
let p_dist p1 p2 = dist (p_x p1) (p_y p1) (p_x p2) (p_y p2)
let p_abs p = p_dist p_origin p
```

Si on charge ce code directement dans la boucle d'interaction, le module n'existe pas puisqu'on n'a pas accès au nom du fichier.
On peut compiler en dehors de l'environnement interactif depuis un terminal:

```
ocamlc -c point.ml
```

Cela crée un fichier d'interface compilée `point.cmi` et le fichier objet `point.cmo`.
Dans un Makefile on peut utiliser la règle

```
%.cmo: %.ml
ocamlc -c $<
```

Le fichier `.cmo` peut être chargé dans la boucle d'interaction grâce à la requête `#load "point.cmo"`.

Accès à un élément d'un module

Par défaut, **tous** les éléments définis dans le module sont accessibles depuis l'extérieur en préfixant par `Nom.element`. Mais on verra par la suite qu'il est possible de cacher une partie de l'implémentation.
Pour ne pas avoir à préfixer (dans un autre module ou dans la boucle d'interaction), utiliser `open Point`.

Définition manuelle d'un module

```
module <Nom> = struct
  ...
end
```

```
module Point =
struct
  type point = P of float * float
  let p_x point = let P(x, _) = point in x
  let p_y point = let P(x, _) = point in y
  let p_origin = make_point 0.0 0.0
  let p_i = make_point 0.0 (-1.)
  let distance x y x' y' = sqrt (x *. x' +. y *. y')
  let p_dist p1 p2 = dist (p_x p1) (p_y p1) (p_x p2) (p_y
  let p_abs p = p_dist p_origin p
end
```

À l'intérieur de `struct end`, on peut mettre `type`, `let`, `module`, `module type`, `exception`, `include`.
Dans ce cours nous ne verrons pas `exception`, `include`.

Tous les éléments du module sont compilés **séquentiellement**. Chaque nouvel élément peut utiliser les liaisons précédentes. Le résultat global est lié à l'identificateur `Nom`.

Si cette déclaration de module est dans un fichier, le module `Point` sera un sous-module du module créé pour le fichier.

Par exemple, si le module est déclaré dans un fichier `plan.ml`, les éléments du module `Point` sont accessibles par `Plan.Point.`.

Signatures

Chaque module a un type appelé signature qui joue le rôle d'interface entre le module et les clients potentiels du module.
Il existe une signature par défaut qui peut être inférée à partir de l'implémentation du module dans laquelle tous les éléments définis par le module et leur type sont visibles.
Pour la voir: `ocamlc -i point.ml`

```
type point = P of float * float
val make_point : float -> float -> point
val p_x : point -> float
val p_y : point -> float
val p_origin : point
val p_i : point
val dist : float -> float -> float -> float -> float
val p_dist : point -> point -> float
val p_abs : point -> float
```

Si le module a été déclaré avec `Module ... struct ... end`

```
module Point :
  sig
    type point = P of float * float
    val make_point : float -> float -> point
    val p_x : point -> float
    val p_y : point -> float
    val p_origin : point
    val p_i : point
    val dist : float -> float -> float -> float -> float
    val p_dist : point -> point -> float
    val p_abs : point -> float
    end
```

Définition de signature

Par défault la signature expose TOUS les types et TOUS les noms.
Si on ne souhaite pas exposer tout, on peut définir la signature d'un module à la main.

- Soit a priori comme spécification d'un module par encore implémenté,
- soit comme une restriction de la signature par défaut pour cacher une partie de l'implémentation.

Ici on cache l'implémentation du type `point` ainsi que certaines fonctions internes au module:

```
sig
type point
val make_point : float -> float -> point
val p_x : point -> float
val p_y : point -> float
val p_origin : point
val p_dist : point -> point -> float
val p_abs : point -> float
end
```

Une signature doit être liée à un nom de signature (par convention en majuscules) à l'aide du mot-clé `module type`

```
module type POINT =
sig
...
end
```

On peut forcer le type de signature du module:

```
module Point : POINT =
struct
...
end
```

Compilation

Pour compiler en dehors de l'environnement interactif, on peut utiliser `ocamlc` qui produit du byte-code pour la machine virtuelle `Zinc` d'OCaML ou `ocamlopt` qui produit du code natif (plus volumineux, plus rapide, pas portable).

```
man ocamlc
```

La commande `ocamlc` ressemble à `cc` ou `gcc`; elle prend en plus en compte le problème des signatures.

Si un fichier `f.mli` existe, le compilateur vérifie que l'implémentation de `f.ml` est conforme à la signature.

Compilation et édition de liens

Cas d'un seul fichier `f.ml`.

`ocamlc f.ml`

produit `a.out`

`ocamlc f.ml -o f.out`

produit `f.out`

Compilation séparée

Chaque fichier `fi.ml` est compilé séparément:

```
ocamlc -c fi.ml
```

produit le fichier objet `fi.cmo` (byte-code) et l'interface compilée `fi.cmi` (sauf si le fichier `fi.mli` existe).

```
ocamlc -o main f1.cmo f2.cmo f3.ml
```

Exemple avec zones

```
ocamlc -c mycomplex.ml
ocamlc -c zones.ml
ocamlfind ocamlc -c -package graphics images.ml
ocamlfind ocamlc -c -package graphics visu_zones.ml
ocamlfind ocamlc -c main.ml
ocamlfind ocamlc -linkpkg -package unix,graphics -o main my
```

Systèmes de compilation automatisés

`make`, `OMake`, `ocamlbuild`, `Oasis`, `Dune`

Foncteurs

Les foncteurs sont les outils de la généricité.
Ce sont des modules paramétrés.