Programmation Fonctionnelle
L3 informatique Université de Bordeaux
Collège Science & Technologie
Année 2024-2025

Rappel : les documents concernant cette unité d'enseignement sont disponibles sous https://moodle.u-bordeaux.fr/course/view.php?id=1208. Accès anonyme pour utilisateurs non inscrits avec mot de passe `oK@ml!`

Table des matières

Chapitre 1

Introduction

La programmation est un art qui nécessite beaucoup d'expérience. Pour apprendre à programmer, il faut lire beaucoup de code écrit par des experts, lire la littérature sur la programmation, programmer, maintenir du code écrit par d'autres personnes, apprendre à être bien organisé.

L'objectif de cette UE est d'acquérir les bases de la *programmation fonctionnelle*. Dans ce cadre nous essaierons d'appliquer les principes généraux de programmation suivants :

- Lisibilité du code
- Maintenabilité
- Réutilisabilité
- Efficacité (quand elle ne nuit pas à la lisibilité) ($\Longrightarrow$ Complexité)
- Test

1.1 Paradigmes de programmation

Voir la page Wikipédia. [1]

- impératif
- fonctionnel
- objet
- macro

Exercice 1.1 Citer des langages comportant ces paradigmes.

Un langage de programmation peut-être

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

Exercice 1.2 Citer des langages ayant ces caractéristiques.

Paradigme fonctionnel

En programmation fonctionnelle, la fonction est un object de base : elle peut être passée en paramètre, retournée par une fonction ; on dit que c'est un objet de *première classe*. Il n'y a pas d'effets de bord (donc pas d'affectation) et les seules structures de contrôle sont le `si-alors-sinon` et la *récursivité*. Un programme n'a pas d'état ; les fonctions ne font que retourner des valeurs.

[1] Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme. Par exemple, en programmation orientée objet, les développeurs peuvent considérer le programme comme une collection d'objets en interaction, tandis qu'en programmation fonctionnelle un programme peut être vu comme une suite d'évaluations de fonctions sans états. Lors de la programmation d'ordinateurs ou de systèmes multi-processeurs, la programmation orientée processus permet aux développeurs de voir les applications comme des ensembles de processus agissant sur des structures de données localement partagées.

De la même manière que des courants différents du génie logiciel préconisent des méthodes différentes, des langages de programmation différents plaident pour des « paradigmes de programmation » différents. Certains langages sont conçus pour supporter un paradigme, en particulier (Smalltalk et Java, qui supportent la programmation orientée objet, tandis que Haskell supporte la programmation fonctionnelle) alors que d'autres supportent des paradigmes multiples (à l'image de C++, Common Lisp, OCaml, Oz, Python, Ruby, Scala ou Scheme).

De nombreux paradigmes de programmation sont aussi célèbres pour les techniques qu'ils prohibent que pour celles qu'ils permettent. La programmation fonctionnelle pure, par exemple, interdit l'usage d'effets de bord ; la programmation structurée interdit l'usage du goto. En partie pour cette raison, les nouveaux paradigmes sont souvent considérés comme doctrinaires ou abusivement rigides par les développeurs habitués aux styles déjà existants. Cependant, le fait d'éviter certaines techniques peut permettre de rendre plus aisée la démonstration de théorèmes sur la correction d'un programme — ou simplement la compréhension de son fonctionnement — sans limiter la généralité du langage de programmation. De plus, il est possible d'écrire un programme en adoptant la programmation orientée objet même si le langage, par exemple le langage C, ne supporte pas ce paradigme.

La relation entre les paradigmes de programmation et les langages de programmation peut être complexe, car un langage de programmation peut supporter des paradigmes multiples. Pour citer un exemple, C++ est conçu pour supporter des éléments de programmation procédurale, de programmation orientée objet et de programmation générique. Cependant, concepteurs et développeurs décident de la méthode d'élaboration d'un programme en utilisant ces éléments de paradigmes. Il est possible d'écrire un programme purement procédural en C++, comme il est possible d'en écrire un purement orienté objet, ou encore qui relève des deux paradigmes.

Paradigme impératif

La programmation fonctionnelle s'oppose au paradigme *impératif* dans lequel le programme a un *état* (la valeur de l'ensemble de ses variables) et l'instruction d'affectation modifie l'état du programme. L'instruction de base est l'affectation et la structure de contrôle de base est la boucle `tant-que`.

Ce paradigme est source de nombreuses difficultés et de bugs (Wikipedia) et produit des programmes plus difficiles à comprendre.

Bugs logiciels aux conséquences désastreuses

Aéronautique

1962 Perte d'itinéraire de la sonde **Mariner 1** (NASA) au lancement.
2 causes dont erreur de transcription d'une équation

1996 Auto-destruction d'**Ariane 5** (1er vol, 36 secondes après le décollage).
Cause : Conversion flottant 64 bits vers entier 16 bits

2004 Blocage du robot **Mars Rover Spirit**.
Cause : trop de fichiers ouverts en mémoire flash.

Médecine

85-87 5 morts par irradiations massives dues à la machine **Therac-25**.
Cause : Conflit d'accès aux ressources entre 2 logiciels

Télécommunications

1990 Crash à grande échelle du réseau **AT & T**, effet domino.
Cause : Toute unité défaillante alertait ses voisines, mais la réception du message d'alerte causait une panne du récepteur.

Énergie

2003 Panne d'électricité aux USA & Canada, **General Electric**.
Cause : à nouveau mauvaise gestion d'accès concurrents aux ressources dans un programme de surveillance.

Informatique

1994 Bug du Penthium FDVI Intel sur opérations flottantes.
cause : Algorithme de division erroné (trouvé par Th Nicely).

06-08 Clés générées par **OpenSSL** et données cryptées non sûres, impactant les applications l'utilisant (comme ssh).

78-95 Faille du protocole d'authentification Needham-Schroeder. Protocole très simple (3 messages échangés).
Problème : Attaque *man in the middle* détectée par G. Lowe. Utilisé 17 ans avec cette faille.

09-15 Faille de sécurité dans le système Linux. Le bug de Grub. Pendant 6 ans, les versions du système Linux ont présenté une très belle faille de sécurité.
Cause : Erreur de programmation (en C) de la fonction qui saisit le nom de l'utilisateur.

http://hmarco.org/bugs/CVE-2015-8370-Grub2-authenticationbypass.html

Pourquoi et quand utiliser le paradigme fonctionnel

Les bugs dans les programmes peuvent avoir des conséquences dramatiques (humaines, environnementales, économiques, ...).

La programmation fonctionnelle permet de développer plus rapidement des programmes plus sûrs et même dans certains cas **prouvés**.

On utilise la programmation fonctionnelle quand il y a des enjeux de sécurité ou pour développer rapidement un prototype.

L'inconvénient d'un langage purement fonctionnel étant la performance, on utilisera un langage impératif lorsqu'il y a des contraintes de performances mais pas de sécurité.

1.2 Langage support : OCaML

Le langage support est `OCaML`.
Il est fonctionnel, statiquement typé, interactif mais aussi compilé, impératif (non abordé dans ce cours)
Le typage permet d'éliminer une partie des bugs à la compilation.

Où est utilisé ocaml

- Dans les entreprises comme *Facebook, Docker, Bloomberg, Jane Street*
- Dans les universités (France, US, Japon).
- En France, CEA, Dassault Systèmes ANSSI.
- À Bordeaux : Shiro Games.

Prise de contact

Le langage `OCaml` http://caml.inria.fr a été développé à l'INRIA (Institut National de Recherche en Informatique et en Automatique) et est disponible sur de nombreuses architectures (Linux, Windows,Mac OS X etc.).

Mode intéractif

- `ocaml utop` dans un terminal
- Sous `emacs`, modes `tuareg` et `utop`.
  - Rajouter

    ```
    (add-hook 'tuareg-mode-hook 'utop-minor-mode)
    ```

    dans votre `.emacs`
  - `C-c C-b` pour compiler tout le buffer, `C-x C-e` pour compiler l'expression courante
  - historique dans la REPL
- Sous `vs-code`, plugin adapté

```
utop[1]> print_string "Hello\n";;
Hello
- : unit = ()
utop[2]> 1 + 2;;
- : int = 3
utop[3]>
```

Mode non intéractif

Dans le mode non intéractif, on perd la possibilité de tester librement expression de manière interactive.

```
$ cat hello.ml
print_string "Hello\n"
$ ocamlc -o hello hello.ml
```

```
$ ./hello
Hello
```

On peut fournir des arguments au programme et les récupérer dans le code par un mécanisme similaire à celui du langage C : `Sys.argv`.

```
$ cat argv.ml
Printf.printf "argc = %d\n" (Array.length Sys.argv);
Printf.printf "argv[0] = %s\n" Sys.argv.(0);
Printf.printf "argv[1] = %s\n" Sys.argv.(1);
Printf.printf "argv[2] = %s\n" Sys.argv.(2);
$ ocamlc -o argv argv.ml
$ argv 1 2
argc = 3
argv[0] = ./argv
argv[1] = 1
argv[2] = 2
```

1.3 Organisation de l'UE

6-7 CM de 1h20 + 12 TM de 1h20 + 1 TP noté de 1h20.

Évaluation

Contrôle continu : 1 DS (1h30) et 1 TP noté (1h30)

Examen final : 1h30

Moodle : en continu

Session1 : 0.5 CC + 0.5 EX1

Session2 : 0.5 EX2 + 0.5 max(EX2, CC)

```
a finir
```

Équipe pédagogique

- Alexandre Blanché
- Frédérique Carrère
- Irène Durand (responsable de l'UE)
- Stefka Gueorguieva
- Victor Nador
- Vincent Penelle

Tutorat

```
à compléter
```

Comment réussir ?

- Être actif en CI et TM
- Chercher les exercices par soi-même
- 2h à 4h de travail personnel par semaine
- Utiliser le forum de Moodle
- En cas de grosses difficultés, s'adresser au tutorat

Chapitre 2

Premiers pas

2.1 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

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 et un type.

[1] Read Eval Print Loop

2.2 Expressions

Voir Chapitre 1 du polycopié Marché-Treinen (sur Moodle).

Une *expression* est

- soit un objet de base (nombre, caractère, booléen, …),
- 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].

Dans un langage interactif, la notion de programme principal n'a pas de sens puisqu'à tout moment n'importe quelle fonction peut être appelée et donc jouer le rôle de programme principal. Rien n'empêche d'appeler une fonction `main` pour la distinguer des autres mais il n'y a aucune obligation à cela.

Application d'une fonction

La notation par défaut est la notation *préfixe* dans laquelle la fonction f est placée **avant** ses arguments :

f e1 e2 ….

On ne doit **pas** séparer les arguments par une virgule, comme en C. Enfin, l'application de fonction est plus prioritaire que les opérateurs usuels.

Exemple :

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

Le **parenthésage par défaut** est : `(((f e1) e2) ...)` .

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

Certains opérateurs binaires courants prédéfinis, en particulier les opérateurs arithmétiques binaires, utilisent la notation *infixe* : l'opérateur se trouve **entre** les deux opérandes.

Exemple :

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

Pour obtenir la version préfixe d'un opérateur infixe, il suffit de le mettre entre parenthèses :

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

[2] Noter la définition récursive d'une expression

Nombres et Expressions numériques

Les types de base sont `int` pour les entiers et `float` pour les nombres flottants. Exemples :
Entiers : `3, 4, 1000`
Flottants `2.1, 3.14e-3`.

Opérateurs arithmétiques

La syntaxe est proche du langage mathématique (notation infixe). À cause du typage, il n'y a **pas de surcharge** des opérateurs et il existe donc un jeu d'opérateurs pour les entiers

`int` : `+, -, *, %, mod, abs, succ, pred, ...`
et un jeu d'opérateurs pour les flottants

`float` : `+., -., *., /., **, abs_float, truncate, sqrt, ...`.
Exemples :

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

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

Exercice 2.1 Pour chacune des expressions suivantes, indiquer si elle est correcte ; si c'est le cas donner sa valeur et son type. **Note.** La fonction `int_of_float` convertit un `float` en `int` (en supprimant sa partie décimale), et la fonction `float_of_int` convertit un `int` en `float`.

1. `12 + 30`
2. `12.0 + 30`
3. `12.0 + 30.0`
4. `int_of_float 12.5 + 30`
5. `float_of_int 12 +. 30.0`

Expressions booléennes

Type booléen

`true, false`

Opérateurs booléens

- `&&, ||, not`
- `=` (égalité de valeurs), `<, <=, >, >=`.
  Exemples :

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

Exercice 2.2 Donner la valeur de chacune des expressions booléenes suivantes :

1. `3 = 4 || 4 = 4`
2. `3 = 4 && 4 = 4`
3. `not (3 = 4) && 4 = 4`
4. `not (3 = 4 || 4 = 4)`

Exercice 2.3 Pour chacune des expressions suivantes, indiquer si elle est correcte ; si c'est le cas donner sa valeur et son type.

1. `12 + 30 = 10 + 32`
2. `12.0 + 30.0 = 10 + 32`
3. `12.0 +. 30.0 = float_of_int (10 + 32)`
4. `12.0 +. 30. = 42. && 3 + 4 = 4 + 4`
5. `12.0 +. 30. = 42. && 3 + 4 != 4 + 4`
6. `12.0 +. 30. = 42. || 3 + 4 = 4 + 4`
7. `12.0 +. 30. = 42. && not (3 + 4 = 4 + 4)`
8. `int_of_string "12" + int_of_string "30"`
9. `int_of_string "12" + int_of_string "30" = 42`

Commentaires

Les commentaires sont délimités par les caractères `(*` et `*)`.
Il n'existe pas de commentaire de ligne.

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

Expressions conditionnelles

```
(* nombre de solutions d'une équation du premier 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
```

Exercice 2.4 Pour chacune des expressions suivantes, indiquer si elle est correcte ; si c'est le cas, donner sa valeur et son type.

1. `let x = 12.0 in x +. 30.0`
2. `let x = 12.0 in if x +. 30.0 = 42.0 then 42 else 0`
3. `let x = 12.0 in if x +. 30.0 = 42.0 then "42" else 0`

Exercice 2.5 Pour chacune des deux expressions mathématiques suivantes dépendant d'un nombre flottant f, écrire une expression `OCaML` permettant de la calculer en appelant une seule fois la racine carrée et le log. Pour cela, factoriser les calculs de $\ln f$, $\sqrt{f}$ et $\ln sqrtf$ en utilisant des expressions `let in` .

$$(\sqrt{f} + \ln f) * (\sqrt{f} - 2 \ln f)$$

$$(\sqrt{f} + \ln \sqrt{f}) * (\sqrt{f} - 2 \ln \sqrt{f})$$

Conversions

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

Exercice 2.6 Tester les fonctions de conversion `float_of_int`, `int_of_float`, `int_of_char`, `char_of_int` , `string_of_int`, `string_of_bool`, `...` .

2.3 Fonctions anonymes

Une fonction *anonyme* est une fonction sans nom. Par exemple, la fonction qui à x associe $3 * x$ est clairement définie et se note en mathématique :

$$x \mapsto 3 * x$$

Une fonction *anonyme* peut être définie avec une expression utilisant le mot `fun` et la syntaxe suivante : `fun x1 x2 ... -> expression`.

Les paramètres de la fonction sont `x1`, `x2`, … et `expression` correspond au corps de la fonction ; son évaluation donne, après passage des paramètres, la *valeur de retour* de la fonction. Par exemple, la fonction d'addition peut être représentée par l'expression suivante :

```
fun x y -> x + y;;
```

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

```
- : int -> 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. 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. Dans l'exemple de l'addition, ces 3 types sont `int`.

De même, la fonction suivante :

```
fun x y -> float_of_int x +. y
```

a comme type :

```
int -> float -> float
```

En effet, on a appliqué la fonction `float_of_int` au paramètre `x`. L'interpréteur peut donc en déduire que `x` est un entier, de type `int`. De même, `y` est ajouté au `float` `float_of_int(x)` avec l'opérateur `+.` ce qui permet de déduire que `y` doit lui-même être un `float`. En résumé, `x` est un `int`, `y` est un `float` et la valeur de retour est aussi un `float`. Exemples :

```
# 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
```

Exercice 2.7 Donner l'expression d'une fonction anonyme qui double son argument. Donner un exemple d'appel de cette fonction.

2.4 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`

La requête `let` permet d'effectuer une liaison entre une variable et une valeur (en d'autres termes, de donner un nom à une valeur). Elle permet donc de définir des variables globales ce qui est indispensable pour enregistrer des fonctions et des données. Grâce à la requête `let`, on peut mémoriser une valeur dans une variable globale. Cette requête fait un effet de bord (affecte la valeur à la variable) et retourne la valeur. 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 par OCaml. 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.

Exercice 2.8 Indiquer, parmi les phrases suivantes `OCaML`, lesquelles sont :

- une expression ; Dans ce cas, donner son type et sa valeur.
- une requête let (liaison variable valeur). Dans ce cas, donner le nom, le type et la valeur de la variable.
- une phrase incorrecte à cause d'une erreur de syntaxe. Dans ce cas, expliquer pourquoi la phrase est syntaxiquement incorrecte.
- une phrase incorrecte à cause d'une erreur de type. Dans ce cas, expliquer l'erreur de type.

1. `let y = let x = 12.0 in x +. 30.0`
2. `let y = let x = 12.0 in if x +. 30.0 then 42 else 0`
3. `let x = 3 in let y = 4 in x + y`
4. `let z = let x = 3 in let y = 4 in x + y`
5. `let x = 3 in let y = 4`

2.5 Exercices

Exercice 2.9 Écrire une expression fonction d'une variable `i` qui retourne une chaîne de caractères : `"positive"` si `i` est strictement positif, `"negative"` si `i` est strictement négatif, `"nul"` sinon.

Exercice 2.10 Mêmes questions que pour l'exercice 2.8.

1. `fun x -> x`
2. `(fun x -> x) 5`
3. `fun x -> x + 1`
4. `let f x y = x - y in f (f 1 1) 1`
5. `let compose = fun f g -> fun x -> f (g x)`
6. `let compose f g = fun x -> f (g x)`
7. `let compose f g x = f (g x)`
8. `let compose = fun f g x -> f (g x)`
9.
```
let mystere =
  let square = fun x -> x * x in
  let compose = fun f g -> fun x -> f (g x) in
  compose square square
```

Exercice 2.11 Écrire une fonction qui prend en paramètres 3 `float` : `a`, `b` et `c`, et qui renvoie le nombre de solutions de l'équation $ax^2 + bx + c = 0$. Lorsque le nombre de solutions est infini (par exemple quand $a = b = c = 0$, la fonction renverra -1). Pour alléger le code, on pourra écrire une fonction `discriminant` prenant les trois paramètres `a`, `b`, `c` qui calcule le discriminant de l'équation.

Chapitre 3

Premières fonctions, récursion

3.1 Fonctions nommées

Définition de fonction

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

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

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

```
let add x y = x + y
```

Exemples :

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

Exercice 3.1 1. Écrire une fonction `test` qui prend en arguments trois entiers `x`, `y`, `z` et retourne `true` si `z` est la somme de `x` et `y`, et `false` sinon.

2. Utiliser la fonction `test` pour les valeurs 1, 2, 3, puis 2, 3, 4 des arguments `x`, `y`, `z`.

Exercice 3.2 Écrire une fonction `valeur_p4` qui prend en argument un entier x et retourne $3x^4 + 7x - 1$.

Exercice 3.3 Écrire une fonction `polynome3 a b c` qui prend en arguments trois entiers `a, b, c` et retourne la fonction polynôme $x \mapsto ax^2 + bx + c$. Donner des exemples d'appels de la fonction retournée par `polynome3`

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.

Ce mécanisme permet la récursivité [1].

Le même mécanisme est utilisé quand on évalue une expression type `let x = ... in` : la variable lexicale `x` est créée sur la pile le temps de l'exécution du `let`.

Dans un langage ne disposant pas de la récursion, la seule solution pour utiliser la récursion est de programmer soi-même la pile des appels.

[1] Certains langages de programmation comme Cobol ou les anciennes versions de Fortran n'autorisent pas la récursivité.

3.2 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 par

$$\begin{cases} 0! = 1 \\ n! = n(n-1)! \quad \texttt{pour } n > 0 \end{cases}$$

s'écrit de façon naïve :

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

Principe de la récursivité

Une fonction récursive est basée sur un ordre *bien fondé*. Dans un ordre bien fondé $<$, il n'y a pas de suite infinie strictement décroissante et il y a un nombre fini d'éléments minimaux.

Sous ces hypothèses, on peut

- décrire le calcul sur les éléments minimaux
- décrire le calcul des éléments non minimaux en fonction d'éléments plus petits.

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.

Exercice 3.4 Soit la fonction `fact` définie comme suit :

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

Dessiner la pile des appels pour `fact 5`.

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`.

Dans une fonction récursive, 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 récursifs).

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 $< (x, y)$.

Exercices

.

Exercice 3.5 Soient n, p deux entiers. On rappelle que $\text{pgcd}(n, 0) = n$ et que $\text{pgcd}(n, p) = \text{pgcd}(p, n \mod p)$. Écrire une fonction `pgcd` calculant le pgcd de deux entiers.

Exercice 3.6 La fonction d'Ackermann est un exemple de fonction à croissance **très** rapide. Elle est définie par

$$ack(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ ack(m - 1, 1) & \text{si } m > 0 \text{ et } n = 0 \\ ack(m - 1, ack(m, n - 1)) & \text{sinon} \end{cases}$$

Écrire un programme `ack` calculant cette fonction. **Attention à ne pas la tester sur de trop grands entiers.**

Exercice 3.7 La suite de Fibonacci est définie par $u_0 = u_1 = 1$ et pour tout $n \geqslant 2$, $u_n = u_{n-1} + u_{n-2}$.

1. Écrire une fonction `fib` calculant le n-ème terme de cette suite.
2. Combien de sommes sont utilisées lors de l'appel `fib n` ?
3. En utilisant une fonction auxiliaire qui calcule le couple (u_n, u_{n+1}), améliorer `fib` pour que l'appel `fib n` n'utilise qu'un nombre linéaire de sommes.

Exercice 3.8 Supposément posé par les recruteurs d'**Amazon**.

- https://youtu.be/5o-kdjv7FD0
- Une personne monte un escalier à n marches.
- Elle monte à chaque pas soit une, soit deux, soit trois marches.
- De combien de façons peut-elle monter l'escalier ?

Écrire une fonction `stairs` qui prend en argument un entier n et calcule le nombre de façons de gravir un escalier de n marches sachant qu'on peut choisir de monter une, deux ou trois marches à chaque pas.

Exercice 3.9 1. Écrire une fonction `power` qui prend en argument deux entiers b et n avec $b \neq 0$, et qui calcule b^n.

2. Combien de multiplications sont effectuées lors de l'appel `power b n` ?
3. Peut-on en utiliser moins ?

Exercice 3.10 Écrire une fonction `iterate` qui prend en argument une fonction f et un entier k et qui renvoie la fonction $x \mapsto \underbrace{f(f(\cdots f}_{k \text{ fois}}(x)\cdots)$. Quel est le type de cette fonction ?

Exercice 3.11 1. Écrire la fonction `multiplicateur` qui prend en paramètre un entier `n` et retourne la fonction de type `int -> int` qui à un entier `i` associe `2 * i` .

$$\begin{array}{rccc} multiplicateur : & \mathbb{N} & \to & \mathbb{N} \mapsto \mathbb{N} \\ & n & \mapsto & i \mapsto n * i \end{array}$$

Exemples :

```
utop[2]> multiplicateur;;
- : int -> int -> int = <fun>
utop[3]> multiplicateur 10;;
- : int -> int = <fun>
utop[4]> multiplicateur 10 2;;
- : int = 20
utop[5]> multiplicateur 100 3;;
- : int = 300
```

Exercice 3.12 On considère la suite récurrente d'ordre 2 suivante :

$$\begin{cases} u_0 &= 0 \\ u_1 &= 3 \\ u_n &= -u_{n-1} + 2u_{n-2}, \forall n \geqslant 2 \end{cases}$$

1. Écrire une fonction récursive `seq_aux` de type `int -> int * int` qui à tout entier naturel n associe le couple (u_n, u_{n+1}).
2. En déduire une fonction `seq` qui à tout entier naturel n associe u_n.
3. Quel est le type de `seq` ?
4. Quelle est la complexité de `seq` en fonction de son paramètre n ?
5. Rechercher sur internet (ou dans le cours d'algo des arbres) comment on peut obtenir une complexité logarithmique pour ce format de suite (à la Fibonacci) en utilisant des matrices et la multiplication "à la Grecque" pour multiplier les matrices.

Exemples :

```
# seq_aux;;
- : int -> int * int = <fun>
# seq_aux 0;;
- : int * int = (0, 3)
# seq_aux 1;;
- : int * int = (3, -3)
# seq_aux 2;;
- : int * int = (-3, 9)
# seq 3;;
- : int = 9
```

Exercice 3.13 Soit f une fonction de $\mathbb{R}$ dans $\mathbb{R}$.

Si f est dérivable, la fonction dérivée de f, f' peut être définie par

$$\begin{array}{rccc} f' : & \mathbb{R} & \to & \mathbb{R} \\ & x & \mapsto & \lim_{h \to 0} \tau(f, h, x) \end{array}$$

où

$$\tau(f, h, x) = \frac{f(x+h) - f(x-h)}{2h}$$

En prenant un h petit (proche de 0), on obtient la fonction f'_h, dérivée approchée de f, définie par

$$\begin{array}{rccc} f'_h: & \mathbb{R} & \to & \mathbb{R} \\ & x & \mapsto & \tau(f,h,x) \end{array}$$

Écrire la fonction `derivee` qui prend en paramètre f, h et retourne f'_h. Exemples :

```
# epsilon;;
- : float = 1e-06
# derivee;;
- : (float -> float) -> float -> float -> float = <fun>
# derivee (fun x -> 4. *. x +. 3.) epsilon 10.;;
- : float = 3.99999999700639819
# derivee (fun x -> x *. x *. x +. 5.) epsilon 2.;;
- : float = 11.9999999999009788
```

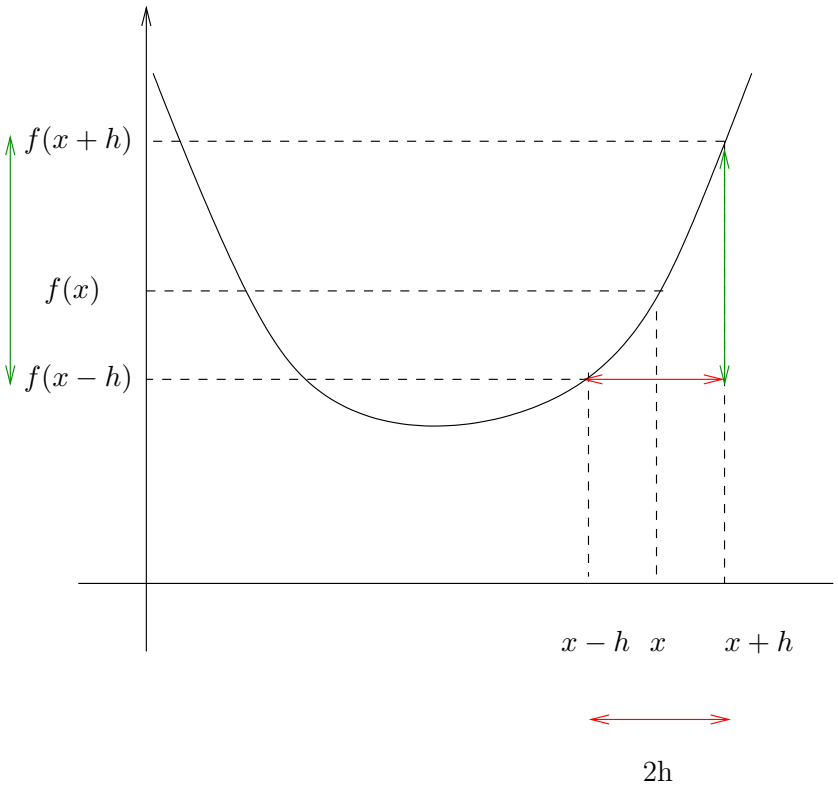
On s'intéresse maintenant à la dérivée $n$ème $f^{(n)}$ d'une fonction f qui peut être définie par

$$\begin{cases} f^{(0)} = f \\ f^{(n)} = (f')^{(n-1)} \end{cases}$$

Écrire la fonction `derivee_n` qui prend en paramètre un entier n, f et h et retourne la fonction dérivée $n$ème (approchée) de f.

Exemples :

```
# derivee_n;;
- : int -> (float -> float) -> float -> float -> float = <fun>
# derivee_n 2 (fun x -> x *. x *. x +. 5.) epsilon 2.;;
- : float = 12.0015108961979422
```



Chapitre 4

Introduction aux types

Nous avons déjà remarqué que le résultat de l'évaluation d'une expression produit non seulement sa valeur mais aussi son type.

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

Un *type* est un ensemble de valeurs.

Par exemple, le type Booléen `bool` contient les deux valeurs `true` et `false`, le type `char` les caractères `'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*.

4.1 Inférence et vérification de types

À la compilation, OCaML infère le type de chaque expression. Certaines erreurs sont ainsi détectées à la compilation quand le type attendu pour un paramètre d'une fonction n'est pas le bon. Si le système échoue à inférer le type de l'expression, l'expression n'est pas évaluée.

Pour chaque expression, OCaML, infère (calcule) le type le plus général possible. Quand il ne peut inférer un type particulier, il utilise une (ou plusieurs) variable de type `'a, 'b, ...`.

Exemples :

```
# 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
```

4.2 Polymorphisme

`OCaML` infère le type le plus général possible.

Exercice 4.1 Quel est le type de la fonction `fun x y -> (x, y)` ?

Exercice 4.2 Quel est le type de la fonction `compose` vue précédemment ?

```
let compose f g = fun x -> f (g x)
```

4.3 Opérateurs de types

Les types peuvent être combinés à l'aide d'opérateurs de types pour obtenir de nouveaux types. Par exemples, le type contenant les couples constitués d'un entier et d'un flottant. le 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 (possiblement le même).

Exemples :

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

La fonction `carre` est une fonction qui prend en paramètre un entier qui retourne un entier tandis que la fonction `sin` prend en paramètre un flottant et retourne un flottant.

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.

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')
# 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 les accesseurs `fst` et `snd`.

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.

Exercice 4.3 On reprend la suite de Fibonacci définie par $u_0 = 1, u_1 = 1$ et pour tout $n \geqslant 2$, $u_n = u_{n-1} + u_{n-2}$. En utilisant une fonction auxiliaire qui calcule le couple (u_n, u_{n+1}), améliorer `fib` pour que l'appel `fib n` n'utilise qu'un nombre linéaire de sommes.

Produit cartésien généralisé (tuples)

Le produit cartésien binaire se généralise aux tuples.

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

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)
```

Exercice 4.4 Quel est le type et la valeur des expressions suivantes ?

```
'x', 2.1, (true, 0)
3 + 2, false || 2 = 3, "bonjour"
let p = 1, 2 in snd p, fst p
```

Exercice 4.5 Pour chacun des types suivants, donner une expression ayant ce type ainsi que la valeur de l'expression.

```
int * bool * string
(int * bool) * string
int * (bool * string)
```

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 :

```
# let next (u, v) = v, (u + v);; (* cf Fibonacci *)
val next : int * int -> int * int = <fun>
# next (1,1);;
- : int * int = (1, 2)
# next(next (1,1));;
- : int * int = (2, 3)
# next(next(next (1,1)));;
- : int * int = (3, 5)
```

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")
```

4.4 Requête `type`

La requête `type` permet de donner un nom à un type (en général composé à l'aide d'opérateurs de types).

```
# 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`.

```
# 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 of float * float;;
type form = Square of float | Circle of float | Rectangle of float * float
# Rectangle (10., 3.);;
- : form = Rectangle (10., 3.)
# Square(3.4);;
- : formes = Square 3.4
```

4.5 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. Elle est composée du mot-clé `match` suivi de la valeur à inspecter suivie du mot-clé `with` puis d'une suite de *clauses* chacune de la forme `motif -> expression`. La valeur retournée par l'expression `match` est la valeur de l'expression de la première clause dont le motif correspond à la valeur inspectée.

```
# 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 en particulier

- 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`

Les clauses ayant la même expression peuvent être regroupées. Les motifs correspondants son séparées par des `|`.

```
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
```

Exercice 4.6 Soient les types `couleur` et `carte` définis comme suit :

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

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

- Écrire un accesseur `couleur_carte carte` de type `carte -> couleur` qui retourne la couleur d'une carte.
- Écrire un prédicat `est_de_couleur carte couleur` de type `carte -> couleur -> bool` qui retourne `true` si `carte` est de couleur `couleur`. On utilisera l'accesseur `couleur_carte`.
- Écrire un prédicat `est_une_figure carte` de type `carte -> bool` qui retourne `true` si `carte` est une figure, `false` sinon.

Exercice 4.7

1. Définir un type `carburant` ayant trois constructeurs `Diesel`, `Essence` ou `Electrique`.
2. Un *véhicule* est caractérisé par son carburant et son nombre de roues. Définir un type `vehicule` répondant à ces critères.
3. Écrire le constructeur `make_vehicule` de type `carburant -> int -> vehicule`.
4. Écrire les accesseurs `carburant_of` de type `vehicule -> carburant` et `nb_wheels_of` de type `vehicule -> int` qui retournent respectivement le carburant et le nombre de roues d'un véhicule.
5. Lors des pics de pollution, les véhicules diesel à 4 roues au moins sont interdits. Écrire une fonction `can_run : vehicule -> bool` qui teste si un véhicule est autorisé.
6. Pour rouler 100km, un véhicule électrique consomme environ 10kWh, un véhicule diesel consomme environ 6L de carburant, et un véhicule essence consomme environ 8L. Sachant qu'1kWh coûte 0.25 EUR et qu'un litre de carburant coûte 1.5 EUR, écrire une fonction `consommation : vehicule -> int -> float` telle que `consommation v n` renvoie le coût d'utilisation du véhicule `v` sur `n` kilomètres.

Exercice 4.8 Rappels :

$$\begin{array}{ll} \frac{dc}{dx} & = 0, \\ \frac{dx}{dx} & = 1, \\ \frac{d(u+v)}{dx} & = \frac{du}{dx} + \frac{dv}{dx}, \\ \frac{d(uv)}{dx} & = u\frac{dv}{dx} + v\frac{du}{dx}, \\ \frac{d(exp(u))}{dx} & = \frac{du}{dx} exp(u). \end{array}$$

On représente des expressions arithmétiques utilisant les opérations $+, -, \times, exp$ par le type suivant :

```
type expr =
          Var of string
        | Number of float
        | Plus of expr * expr
        | Minus of expr * expr
        | Mult of expr * expr
        | Exp of expr
```

Ainsi, une variable x est représentée par l'expression OCaML `Var "x"` et $3x^2 + 2x + 1$ par

```
Plus (Mult (Number 3., Mult (Var "x", Var "x")),
  Plus (Mult (Number 2., Var "x"), Number 1.))
```

1. Définir deux variables `vx` et `vy` contenant respectivement des variables x et y.
2. Définir la variable `e1` contenant l'expression $2x + 1$.
3. Définir la variable `e2` contenant l'expression $3x^2 + 2x + 1$.
4. Implémenter une fonction `derivee var expr` par une traduction directe des cinq règles ci-dessus, i.e. par un simple filtrage avec cinq cas distincts.
   Exemple d'utilisation :

```
utop[83]> vx;;
- : expr = Var "x"
utop[84]> e1;;
- : expr = Plus (Mult (Number 2., Var "x"), Number 1.)
utop[85]> derivee vx e1;;
- : expr =
Plus (Plus (Mult (Number 2., Number 1.), Mult (Number 0., Var "x")),
 Number 0.)
utop[86]> e2;;
- : expr =
Plus (Mult (Number 3., Mult (Var "x", Var "x")),
 Plus (Mult (Number 2., Var "x"), Number 1.))
utop[87]> derivee vx e2;;
- : expr =
Plus
 (Plus
   (Mult (Number 3.,
     Plus (Mult (Var "x", Number 1.), Mult (Number 1., Var "x"))),
    Mult (Number 0., Mult (Var "x", Var "x"))),
 Plus (Plus (Mult (Number 2., Number 1.), Mult (Number 0., Var "x")),
  Number 0.))
```

5. Écrire la fonction `derivee_n var expr n` qui retourne une expression correspondant à la dérivée n-ème de `expr`.
   Exemples :

```
utop[91]> derivee_n vx e2 2;;
- : expr =
Plus
 (Plus
```

```
    (Plus
      (Mult (Number 3.,
        Plus (Plus (Mult (Var "x", Number 0.), Mult (Number 1., Number 1.)),
         Plus (Mult (Number 1., Number 1.), Mult (Number 0., Var "x")))),
       Mult (Number 0.,
        Plus (Mult (Var "x", Number 1.), Mult (Number 1., Var "x")))),
     Plus
      (Mult (Number 0.,
        Plus (Mult (Var "x", Number 1.), Mult (Number 1., Var "x"))),
       Mult (Number 0., Mult (Var "x", Var "x")))),
   Plus
    (Plus (Plus (Mult (Number 2., Number 0.), Mult (Number 0., Number 1.)),
      Plus (Mult (Number 0., Number 1.), Mult (Number 0., Var "x"))),
     Number 0.))
    utop[92]>
```

6. Les expressions auraient bien besoin d'être simplifiées. Proposer des pistes pour simplifier les expressions.

Chapitre 5

Application : Plan complexe

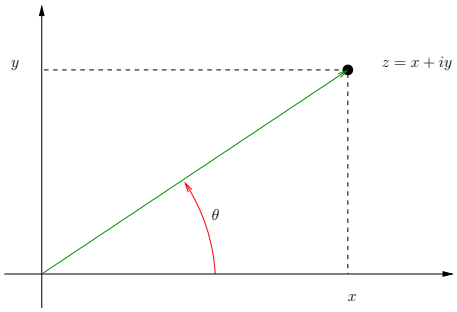


FIG. 5.1 : Plan complexe

5.1 Représentation d'un point du plan par un complexe

Exercice 5.1 Dans un plan muni d'un repère orthonormé, on associe au point P de coordonnées (x, y) son affixe, le nombre complexe $z = x + iy$.
Pour représenter un point du plan, on utilise le type

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

Tous ce qui suit doit être écrit dans un fichier nommé `mycomplex.ml`.

1. Définir le type `mycomplex`.
2. Écrire la fonction constructeur `make_complex` qui fabrique un complexe à partir de sa partie réelle et de sa partie imaginaire
3. Écrire l'accesseur `realpart` qui permet de récupérer la partie réelle d'un complexe.
4. Écrire l'accesseur `imagpart` qui permet de récupérer la partie imaginaire d'un complexe.
5. En utilisant `make_complex`, définir
   - la variable `c_origin` ayant pour valeur le complexe $(0., 0.)$ (point origine du plan complexe),
   - la variable `c_i` ayant pour valeur le complexe $i = (0., 1.)$,
   - la variable `p_12` ayant pour valeur le complexe $(1., 2.)$.
   - Implémenter les opérations
     `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.
     Remarque : il faut que le code des opérations reste valable si on change le type `mycomplex` pour un enregistrement[a] par exemple ; il faut donc utiliser la fonction `make_complex` et ne pas utiliser le constructeur `C`.

[a] Voir le chapitre "Enregistrements"

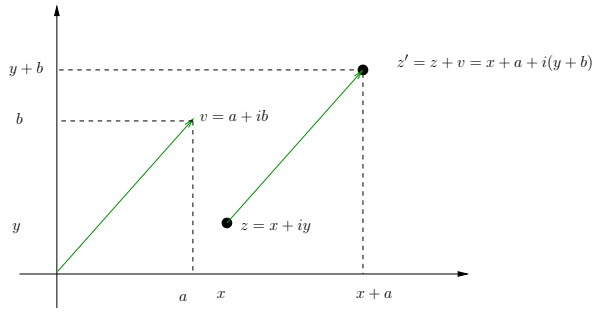


FIG. 5.2 : Translation de vecteur (a, b)

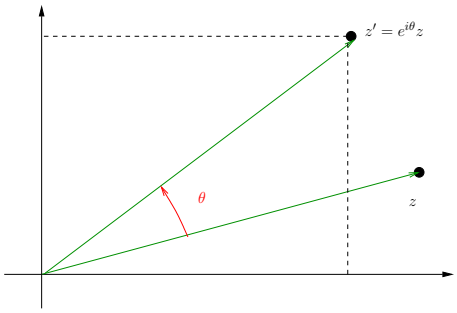


FIG. 5.3 : Rotation d'angle θ

Exercice 5.2 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'.

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

Soit O le point origine du plan, c'est-à-dire le point d'affixe $(0., 0.)$.

1. Soit la translation de vecteur $\overrightarrow{u}$ d'affixe z_u. Donner l'affixe z' du transformé d'un point d'affixe z par cette translation.
2. Donner l'affixe z' du transformé d'un point d'affixe z par la rotation de centre O et d'angle θ.
3. Donner l'affixe z' du transformé d'un point d'affixe z par la rotation de centre C d'affixe z_c et d'angle θ,

Exercice 5.3 1. Écrire la fonction `translate c vector` qui donne l'affixe du point obtenu par translation de vecteur `vector:mycomplex` du point d'affixe `c`.

2. Écrire la fonction `rotate0 c angle` qui donne l'affixe du point obtenu par rotation d'angle `angle:float` du point d'affixe `c`.
3. Écrire la fonction `rotate c angle center` qui donne l'affixe du point obtenu par rotation d'angle `angle:float` du point d'affixe `c` autour du point d'affixe `center`.

Exercice 5.4 Tester vos fonctions avec le fichier `test_mycomplex.ml`.

Chapitre 6

Application : Zones du plan

6.1 Zones représentées par leur fonction caractéristique

Exercice 6.1 [a]

Dans leur article “Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.368.1058&rep=rep1&type=pdf de 1994, Paul Hudak et Mark P. Jones rapportent une expérience rare de comparaison entre différents langages de programmation. Lelogiciel implémenté manipule des zones géométriques. Nous allons étudier une version simplifiée (mais pas tant que ça) de ce logiciel.

Une zone géométrique est un ensemble de points du plan. 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 correspondant qu plan tout entier est représentée par la variable `everywhere` suivante :

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

1. Éditer le fichier `zones.ml`.
2. Comprendre le code de la fonction `point_in_zone_p`.
3. Comprendre le code de la fonction `everywhere`.
4. Lancer `utop` et visualiser la zone `everywhere`.
5. Définir la zone vide dans la variable `nowhere`.

   Exemples :

```
# 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>
```

Pour forcer les types :

```
type zone = mycomplex -> bool

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

[a] Pour passer à la partie `zones`, il est nécessaire que le fichier `mycomplex.ml` soit correct.

6.2 Manipulation de zones

Exercice 6.2 1. L'appel `translate_zone zone vector` retourne la zone translatée par le vecteur `vector` (représenté comme un point). Comprendre le code donné pour cette fonction.

2. Écrire une fonction `make_rectangle width height` qui retourne la zone rectangulaire définie par les points $(0,0), (width, 0), (width, height), (0, height)$, c'est-à-dire la fonction qui prend un point en argument et retourne `true` si cet argument est dans le rectangle (ou sur sa bordure), et `false` sinon.
3. L'appel `zone_intersection zone1 zone2` retourne la zone correspondant aux points se trouvant à la fois dans les zones `zone1` et `zone2`. Comprendre le code donné pour cette fonction.
4. Écrire une fonction `zone_union zone1 zone2` qui retourne l'union des zones `zone1` et `zone2` (c'est-à-dire l'ensemble des points qui appartiennent à l'une ou l'autre des zones).
5. Écrire une fonction `zone_complement zone1` qui retourne la zone correspondant aux points ne se trouvant pas dans la zone `zone1`.
6. Écrire une fonction `zone_difference zone1 zone2` qui retourne la zone correspondant aux points se trouvant dans la zone `zone1` mais pas dans la zone `zone2`.

Exercice 6.3 1. Comprendre le code de la fonction `make_disk0 radius` qui retourne un disque de rayon `radius` centré au point $(0,0)$.

2. Dans un repère orthonormé, dessiner la zone définie par l'expression suivante.
   `zone_union (make_rectangle 2. 4.) (make_disk0 3.)`

```
(* Given a zone, move it by a vector indicated as a point passed as the argument. *)
let translate_zone zone vector =
  fun p -> point_in_zone_p (c_dif p vector) zone;;
```

Exercice 6.4 1. S'inspirer de la fonction `translate_zone` pour rajouter une fonction `scale_zone0`. Cette nouvelle fonction aura deux paramètres, le premier est la zone à transformer, et le deuxième est le point (complexe) (λ_1, λ_2). La fonction applique à la zone la transformation

$$(x, y) \mapsto (\lambda_1 x, \lambda_2 y).$$

2. Écrire une fonction `scale_zone zone coeff point` qui effectue la mise à l'échelle par rapport au point `point` et non pas par rapport à l'origine.

Exercice 6.5 Écrire une fonction `make_disk radius point` qui retourne un disque de rayon `radius` centré au point `point`.

Exercice 6.6

```
let test () = fun _ ->
  begin
  let c = make_disk0 1. in
  let c1 = translate_zone c (make_point 1. 0.) in
```

```
  assert (point_in_zone_p (make_point 0.0 0.5) c);
  assert (not (point_in_zone_p (make_point 1.0 0.5) c));
  assert (point_in_zone_p (make_point 0.5 0.) (zone_intersection c c1));
  end

let _ = test ()
```

En vous inspirant du jeu de tests ci-dessus, écrire un jeu de test pour toutes les fonctions.

Exercice 6.7 1. Regarder le contenu du fichier `visu-zones.ml`.

2. Utiliser la fonction `view_zone zone` pour produire une image PNG et visualiser une partie finie d'une zone.
3. Pour adapter la taille de la zone visualisée, utiliser `view_zone_size zone size`.

Exercice 6.8 1. Écrire une fonction `rotate_zone0 zone angle` qui effectue une rotation d'angle `angle` autour de l'origine.

2. Écrire une fonction `rotate_zone zone angle point` qui effectue une rotation d'angle `angle` autour du point `point`.

Chapitre 7

Enregistrements

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**.

7.1 Déclaration d'un type enregistrement

```
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; }
```

7.2 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.
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.
```

Exercice 7.1 Réécrire les opérations sur les complexes le type suivant.

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

Exercice 7.2 1. Écrire un type `date` de type enregistrement, contenant trois champs entiers : `day`, `month`, `year`.

2. Définir une variable `today` contenant la date d'aujourd'hui.

3. Écrire un prédicat `date_infeg` qui s'applique à deux dates `date1` et `date2` et qui retourne `true` si `date1` précède `date2`.

Chapitre 8

Types inductifs (récursifs)

La récursivité est utilisable dans la définition des types. Ceci permet en particulier de construire des types infinis. On parle alors de types *inductifs*.

On peut par exemple représenter les listes d'entiers[1].

```
# type intlist = NI | CI of int * intlist;; (* NI: liste d'entiers vide *)
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)))
```

[1] Définition en fait inutile, puis qu'il existe un type prédéfini pour les listes que nous verrons au chapitre 10

8.1 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
```

Exercice 8.1 1. Définir un type `'a mylist` permettant de représenter les listes d'éléments de type `'a`.

2. Écrire la fonction `mylist_length` qui prend en argument une liste de type `mylist` et qui retourne le nombre d'éléments de la liste. Exemple :

```
# len Nil;;
- : int = 0
# len (C(0, C(1, C(3, C(4, C(5, Nil))))));;
- : int = 5
```

3. Écrire la fonction `interval_list n p` de type `int -> int -> int mylist` qui retourne la liste ordonnée des entiers contenus dans l'intervalle $[n, p]$. Exemple :

```
# interval_list 4 1;;
- : int mylist = Nil
# interval_list 2 5;;
- : int mylist = C (2, C (3, C (4, C (5, Nil))))
```

4. Écrire la fonction `map`, qui prend en argument

   - une fonction de type `'a -> 'b`, et
   - une liste de type `'a mylist`

   et telle que `map f l` renvoie la liste de type `'b mylist` obtenue en appliquant `f` à chaque élément de `l`. Exemple :

```
# map (fun x -> x+10) (C(0, C(1, C(3, C(4, C(5,Nil))))));;
- : int mylist = C (10, C (11, C (13, C (14, C (15, Nil)))))
```

5. Écrire une fonction `filter pred l` de type `('a -> bool) -> 'a mylist -> 'a mylist` qui retourne la listes des éléments de `l` qui vérifient le prédicat `pred`. Exemple :

```
# filter (fun x -> x mod 2 = 0) (C(0, C(1, C(3, C(4, C(5, Nil))))));;
- : int mylist = C (0, C (4, Nil))
```

8.2 Entiers de Peano

Exercice 8.2 1. Les entiers de Peano sont une représentation des entiers naturels. Ils sont construits à partir de 0 en appliquant la fonction successeur. Par exemple, 1 est le successeur de 0, et 2 est le successeur du successeur de 0.
Pour un élément $m \in \mathbb{N}$ on peut définir les suites $(m+n)_{n\in\mathbb{N}}$ et $(m \times n)_{n\in\mathbb{N}}$ comme il suit :

$$(m+n)_{n\in\mathbb{N}} = \left\{ \begin{array}{ll} & m+0=m \\ \forall n \in \mathbb{N} & m+S(n)=S(m+n) \end{array} \right\} \tag{8.1}$$

$$(m\times n)_{n\in\mathbb{N}} = \left\{ \begin{array}{ll} & m\times 0=0 \\ \forall n \in \mathbb{N} & m\times S(n)=(m\times n)+m \end{array} \right\} \tag{8.2}$$

a. Proposer un type OCaml appelé `peano` pour représenter les entiers de cette façon. Le type aura deux constructeurs : un pour représenter 0, l'autre pour représenter le successeur d'un entier.

b. Écrire la fonction d'addition sur ces entiers.

c. Écrire la fonction de multiplication sur ces entiers.

d. Écrire les fonctions de conversion `peano_of_int` et `int_of_peano`.

Chapitre 9

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*[1] 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)
```

Quand une fonction n'est pas récursive terminale, à l'exécution les appels à la fonction doivent être empilés sur la pile d'exécution. Ainsi, lors de l'appel à `fact 4` seront empilés les appels : `fact 4`, `fact 3`, `fact 2`, `fact 1`, `fact 0`.
Le dernier appel retournera `1`,
puis l'appel `fact 1` se terminera avec la multiplication par `1` et retournera `1`,
puis l'appel `fact 2` se terminera par la multiplication par `2` et retournera `2`,
puis l'appel `fact 3` se terminera par la multiplication par `3` et retournera `6`,
puis l'appel `fact 4` se terminera par la multiplication par `4` et retournera `24`.

Cet empilement d'appels sera la cause de débordement de la pile (Stack overflow) injustifiés dans le cas d'un tel calcul qui dans un langage classique se ferait avec une simple boucle. Exemple en `Python`.

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

L'avantage d'une fonction récursive terminale est qu'il n'est pas nécessaire d'empiler les appels puisque qu'il n'y a rien à faire avec la valeur de retour de l'appel, et donc au lieu d'être empilé, l'appel récursif viendra juste remplacer l'appel précédent. Ainsi, il n'y aura pas de débordement de pile dû aux appels récursifs.

Remarque : il n'est pas toujours possible d'écrire une fonction récursive terminale.

Souvent, le passage d'une fonction non récursive terminale à une fonction récursive terminale se fait 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.

Pour la fonction `fact`, on utilise un paramètre supplémentaire `p` dans lequel on va accumuler le produit. Dans un premier temps, on peut écrire une fonction auxiliaire `fact_aux n p` récursive terminale. La fonction `fact` s'écrit en appelant `fact_aux n 1`, `1` étant l'élément neutre pour le produit.

[1] *tail recursive* en anglais

```
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
```

Dans la pile, l'appel à `fact_aux 4 1` sera remplacé par l'appel à
`fact_aux 3 4` qui sera remplacé par l'appel à
`fact_aux 2 12` qui sera remplacé par l'appel à
`fact_aux 1 24` qui sera remplacé par l'appel à
`fact_aux 0 24` qui retournera `24`.

Remarquer que le paramètre `p`, joue le même rôle que la variable `p` et `n` joue le rôle de `i` dans le code `Python` suivant :

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

En `OCaML`, on définira habituellement les fonctions auxiliaires à l'intérieur de la fonction principale à l'aide d'un `let rec ... in ...` (sauf si la fonction auxiliaire peut être 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
```

Exercice 9.1 Écrire une version récursive terminale de la fonction factorielle.

Exercice 9.2 Écrire une version récursive terminale de la fonction `sum` de type `int -> int -> int` qui retourne la somme de entiers de l'intervalle `[n, p]`.

Exercice 9.3 Écrire une version récursive terminale de la fonction `mylist_length` vue au chapitre sur les types récursifs.

Exercice 9.4 Écrire une version récursive terminale de la fonction `interval_list` vue au chapitre sur les types récursifs.

Remarque : L'utilisation d'un accumulateur fonctionne pour les fonctions présentant un seul appel récursif. Pour les fonctions présentant plusieurs appels récursifs, comme c'est souvent le cas par exemple pour les fonctions travaillant sur les arbres, une technique classique consiste à utiliser des accumulateurs comportant plus d'information, comme des fonctions (ce style de programmation est appelé *programmation par continuations*).

Note importante. Lorsque vous écrivez des fonctions auxiliaires, il est important de savoir ce qu'elles doivent faire, non seulement pour les valeurs de l'accumulateur qui vous intéressent (ici par exemple, quand l'accumulateur est la liste vide `Nil`), mais de façon générale.

Chapitre 10

Listes

10.1 Type 'a list prédéfini en OCaML

Il n'est pas nécessaire de définir un type `'a mylist` (comme fait au chapitre précédent) puisque qu'il existe le type prédéfini `'a list` en OCaML. Ce type est fourni par le module `List`. Les fonctions de ce module seront accessible avec le préfixe `List`. Par exemple `List.length` pour la longueur d'une liste.[1] Ces listes sont comme dans le cas du type `'a mylist` des listes homogènes c'est-à-dire dont tous les éléments sont d'un même type.

Constructeurs et accesseurs pour le type mylist

Le constructeur de liste vide est `[]` au lieu de `Nil` pour le type `mylist`. Le constructeur permettant de rajouter un élément à une liste est `::` mais contrairement au constructeur `C` dy 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] Utiliser la complétion de `List.` pour voir les fonctions disponibles dans ce module.

```
# [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
# x :: x * x :: [1; 2; 3];;
- : int list = [4; 16; 1; 2; 3]
```

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]
```

Cet opérateur est à utiliser avec parcimonie. En effet, sa complexité est en $O(len(l1))$ car il entraîne une copie de la liste `l1`.

Il peut servir ponctuellement à ajouter[2] 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`.

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

10.2 Exemples de fonctions sur les listes

Exercice 10.1 1. Réécrire avec le type prédéfini `list` de `OCaML`, les fonctions `interval_list`, `map`, `list_length` et `filter` écrites précédemment avec le type `'a mylist` .

2. Écrire les fonctions de conversion entre les listes de type `'a mylist` et les listes prédéfinies en `OCaML` (`list_of_mylist`, `mylist_of_list`).

Exercice 10.2 1. Écrire une fonction `replicate x k` qui construit la liste composée de `k` répétitions de l'élément `x` .

2. Quel est le type de votre fonction ?

3. Écrire une version récursive terminale de cette fonction.

10.3 Génération de listes

Exercice 10.3 1. Écrire une fonction `reverse l` qui prend en paramètre une liste `l` quelconque et retourne une liste constituée des éléments de `l` en sens inverse.

2. Indiquer si la fonction écrite est récursive terminale. Donner sa complexité.
3. Donner une version linéaire et récursive terminale.

Exercice 10.4 1. Écrire une fonction `iota_r` qui prend en argument un entier `n` et renvoie la liste `[n; n-1; ...; 1]`.

2. Écrire une fonction `iota` qui prend en argument un entier `n` et renvoie la liste `[1; ...; n-1; n]`.
3. Donner les complexités de vos fonctions.
4. Donner des versions linéaires et récursives terminales de ces fonctions.
5. Tester les fonctions avec des valeurs de `n` de plus en plus grandes.
6. Expliquer pourquoi la solution suivante est mauvaise :

```
let rec bad_iota n =
  if n = 0
  then []
  else (bad_iota (n - 1)) @ [n]
```

10.4 Autres exercices sur les listes

Exercice 10.5 1. Écrire une fonction `member` qui teste si son premier argument `x` appartient à la liste donnée par son second argument `l`. Cette fonction renvoie donc soit `true`, soit `false`.

2. Quel est le type de cette fonction ?

3. La fonction demandée existe dans la bibliothèque standard `OCaML` et s'appelle `List.mem`. Comparer votre code à celui de la fonction `List.mem`.

Exercice 10.6 1. Écrire une fonction `count x l` qui compte le nombre de fois que son premier argument `x` appartient à la liste donnée par son second argument `l`. Cette fonction renvoie donc un entier.

2. Quel est le type de cette fonction ? Comparer ce type avec celui de la fonction `member`.

3. Réécrire la fonction `member` en utilisant `count`. On appellera cette nouvelle fonction `member'`.

Exercice 10.7 Un moyen de tester la différence d'efficacité entre deux fonctions est d'utiliser la commande unix `time`, qui renvoie le temps CPU, en secondes, utilisé par le programme depuis le début de son exécution. `OCaml` permet en fait de le faire dans le programme lui-même : il suffit d'enregistrer ce temps avant et après l'évaluation d'un bloc de code et de faire une soustraction pour connaître le temps CPU pris par ce bloc.

Le code suivant permet de le faire pour les fonctions `member` et `List.mem`.

```
let l = iota 100000

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

# time (fun () -> member 99999 l);;
- : float = 3.0000000001972893e-06
# time (fun () -> List.mem 99999 l);;
- : float = 2.99999999997524469e-06
```

Tester la différence d'efficacité des fonctions `member` et `member'`.

Exercice 10.8 1. Utiliser la fonction `List.mem` pour écrire une fonction `list_subset` qui prend en argument deux listes `l` et `l'`, et qui teste si tout élément de `l` apparaît dans `l'`.

2. Tester la fonction `list_subset` sur plusieurs exemples bien choisis.

Exercice 10.9 Une liste `l` est une *permutation* d'une liste `l'` si elle est composée des mêmes éléments, chacun apparaissant dans les deux listes avec le même nombre d'occurrences, mais éventuellement dans un autre ordre. Par exemple, `[1;2;3;3]` est une permutation de `[3;2;3;1]`, mais pas de `[1;2;3]`, car 3 apparaît un nombre de fois différent dans les 2 listes.

1. Utiliser la fonction `count` vue précédemment pour écrire une fonction qui prend en argument deux

listes `l` et `l'`, et qui teste si `l` est une permutation de `l'`.

2. Tester la fonction sur plusieurs exemples bien choisis.

Une liste `l` est un *préfixe* d'une liste `l'` s'il existe une liste `l''` telle que `l' = l @ l''`.

Exercice 10.10 Une liste `l` est un *prefixe* d'une liste `l'` s'il existe une liste `l''` telle que `l'= l @ l''`.

1. Écrire une fonction qui prend en arguments deux listes `l` et `l'` et qui teste si `l` est un préfixe de `l'`.
2. Tester !

Exercice 10.11 1. Écrire une fonction récursive `squares` qui prend un argument une liste d'entiers et qui renvoie la liste des carrés de ces entiers.

2. Réécrire la fonction `map` en récursif terminal.
3. Réécrire la fonction `squares` en utilisant la fonction `map`.

Exercice 10.12 1. Écrire une fonction `sum` qui prend un argument une liste d'entiers et qui calcule la somme de ses éléments.

2. Écrire une fonction `prod` qui prend un argument une liste d'entiers et qui calcule le produit de ses éléments.
3. Tester la dernière fonction sur la liste `0 :: (iota 10000)`.
4. Améliorer la fonction pour le cas où la liste contient un élément nul.

Exercice 10.13 — Fonction break. Soit la fonction `break` suivante :

```
let rec break p l =
  match l with
  | [] -> ([], [])
  | a :: r -> let (l1, l2) = break p r
    in if p a then (a :: l1, l2) else (l1, a :: l2)
```

1. Quel est le type de la fonction `break` ?
2. Quelle est la valeur, en général, de `break p l` ?
3. Écrivez une version récursive terminale `breakfast` de la fonction `break`. Si vous utilisez une fonction auxiliaire, expliquez ce qu'elle calcule en fonction de ses arguments.

Chapitre 11

Application : album photo

Exercice 11.1 — Album photo. On veut manipuler une liste de descriptions de photos. Chaque description comporte une année de prise de vue et une liste de sujets. On définit donc les types suivants :

```
type subject= Selfie | Monument | Mirror_of_Water | Fashion | People| My_plate_at_the_restaurant
type photo = Photo of int * (subject list)
type album = photo list
```

Un exemple d'album est le suivant :

```
let my_album =
  [Photo(2016, [Selfie; Mirror_of_Water]);
   Photo(2014, [Selfie; People]);
   Photo(2014, [Selfie; Monument; Fashion]); Photo(2012, [My_plate_at_the_restaurant; People])]
```

1. Écrire les accesseurs `photo_year photo` , `photo_subjects photo` qui retournent respectivement l'année et la liste des sujets de `photo` .
   `photo_year` est de type `photo -> int` et `photo_subjects` est de type `photo -> subject list` .
2. Écrire un prédicat `has_subject subject photo` qui retourne `true` si `subject` appartient à la liste des sujets de `photo` .
3. Écrire une fonction `select_by_subject` de type `subject -> album -> album` telle que `select_by_subject subject album` retourne la liste des photos de l'album `album` dont ***au moins un des sujets*** est `subject` .
   Ainsi, `select_by_subject Selfie my_album` retourne :
   ```
   [Photo (2016, [Selfie; Mirror_of_Water]); Photo (2014, [Selfie; People]);
    Photo (2014, [Selfie; Monument; Fashion])]
   ```
4. Écrire une fonction `select_by_year` de type `(int -> bool) -> album -> album` telle que l'appel `select_by_year p album` retourne la liste des photos de l'album `album` dont l' année satisfait la fonction booléenne `p` .
   Par exemple, `select_by_year (fun x -> x >= 2014) my_album` retourne :
   ```
   [Photo (2016, [Selfie; Mirror_of_Water]); Photo (2014, [Selfie; People]);
    Photo (2014, [Selfie; Monument; Fashion])]
   ```

Exercice 11.2 On définit maintenant le type `criteria` suivant.

```
type criteria =
```

```
  Subject of subject
| Date  of (int -> bool)
| Or    of criteria * criteria
| And    of criteria * criteria
| Not   of criteria
```

1. Écrire le critère spécifiant « le sujet contient `Selfie` mais pas `People` et la photo a été prise en 2014 ou après ».
2. Écrire une fonction `satisfies` de type `criteria -> photo -> bool` qui teste si une photo satisfait un critère.
3. Écrire une fonction `select` de type `criteria -> album -> album` telle que `select criteria album` renvoie la liste des photos de l'album `album` satisfaisant le critère `criteria`.

Chapitre 12

Listes (suite)

12.1 Fonctions `fold`

Exercice 12.1 Les fonctions `sum`, `prod` et `op_prod` vues précédemment suivent le même schéma de récursion. On peut écrire deux fonctions pour décrire de tels schémas :

- Si `l` est la liste `[b1; ...; bn]`, `left_fold op a l` vaut `op (...(op (op a b1) b2)...) bn`.
- Si `l` est la liste `[a1; ...; an]`, `right_fold op l b` vaut `op a1 (op a2 (...(op an b)...))`.

Cet exercice demande de comprendre et programmer ces fonctions. Elles seront ensuite utilisées pour reprogrammer des fonctions déjà vues.

1. Pour comprendre ces schémas, calculer à la main
   - `left_fold op a l` et
   - `right_fold op l b`

   pour `op` la fonction `fun x y -> x + y`, `l` la liste `[1;2;3]`, et `a` et `b` valant `0`.
2. Écrire la fonction `right_fold`.
3. Écrire la fonction `left_fold`.

   Note : Ces fonctions existent dans la bibliothèque standard sous les noms `List.fold_right` et `List.fold_left`.

Exercice 12.2 En utilisant les fonctions `List.fold_left` et/ou `List.fold_right`, écrire ou réécrire les fonctions suivantes.

1. Une fonction `length l` qui calcule la longueur de la liste `l`.
2. Une fonction `reverse l` qui calcule la liste renversée de `l`.
3. Une fonction `maximum l` qui calcule le maximum de la liste d'entiers `l`.
4. Une fonction `filter p l`, où `p` est un prédicat s'appliquant aux éléments de `l`, qui calcule la liste des éléments de `l` qui satisfont le prédicat `p`.
5. Une fonction `remove_duplicates l` qui ne garde qu'une occurrence de chaque élément de `l`.
6. La fonction `append` qui concatène deux listes.
7. La fonction `map f l` écrite dans la feuille 1.

Chapitre 13

Recherche

13.1 Type `option`

le type `'a option` prédéfini par `OCaml`, que l'on peut réécrire comme :

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

Par exemple, une valeur du type `int option` est soit `None`, soit `Some x` où `x` est un entier. L'idée est que ce type permet de représenter des valeurs optionnelles. Par exemple, `Some 3` représente l'entier 3 et `None` ne représente aucune valeur.

Ce type permet d'obtenir un type de retour uniforme pour les fonctions ne retournant pas toujours une valeur comme les fonctions de recherche par exemple.

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

13.2 Recherche dans une liste

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

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

13.3 Représentation de dictionnaires

Un dictionnaire est une structure de donnée permettant de d'associer des valeurs à des clés. Chaque clé présente dans le dictionnaire a une valeur associée. La structure de donnée doit permettre :

- de consulter la valeur associée à une clé,
- d'ajouter une entrée, en associant une valeur à une nouvelle clé,
- de modifier une valeur déjà associée à une clé existante,
- de supprimer une entrée, c'est-à-dire de supprimer une clé et sa valeur associée.

Par exemple, dans un dictionnaire linguistique, les clés sont les mots et les valeurs sont les définitions. Dans la structure de données « dictionnaire », on peut bien sûr avoir des clés qui sont de type différent du type `string`, comme par exemple des clés entières.

Dans ce TD, on va écrire plusieurs implantations de cette structure de donnée et des fonctions permettant de la manipuler.

On se servira du type `'a option` prédéfini par OCaml, que l'on peut ré-écrire comme :

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

Par exemple, une valeur du type `int option` est soit `None`, soit `Some x` où `x` est un entier. L'idée est que ce type permet de représenter des valeurs optionnelles. Par exemple, `Some 3` représente l'entier 3, et `None` ne représente aucune valeur. Cela sera utile, par exemple, pour la fonction permettant de consulter la valeur associée à une clé dans un dictionnaire : si la clé n'existe pas dans le dictionnaire, la fonction retournera `None`.

Exercice 13.1 La première implantation à réaliser représente un dictionnaire par une fonction, de type `'k -> 'v option`, où `'k` est le type des clés et `'v` le type des valeurs.

1. Définir un type `('k,'v) dict` permettant de représenter un dictionnaire par une fonction de type `'k -> 'v option`.
2. Définir le dictionnaire vide `empty_dict`, de type `('k,'v) dict`.
3. Écrire une fonction `find` de type `'k -> ('k,'v) dict -> 'v option` telle que `find key dict` calcule la valeur à la clé `key` dans le dictionnaire `dict` si cette clé existe, et `None` sinon.
4. Écrire une fonction `add_or_update` de type `'k -> 'v -> ('k,'v) dict -> ('k,'v) dict`, telle que `add_or_update key val dict` renvoie le dictionnaire `dict` dans lequel on a associé la clé `key` à la valeur `val`. Si la clé `key` n'était pas dans le dictionnaire, l'appel `add_or_update key val dict` ajoute cette clé. Sinon, elle la nouvelle valeur associée à la clé `key` déjà présente est `val` dans le dictionnaire renvoyé. Autrement dit, on considère que l'adjonction d'une donnée à une clé déjà présente met à jour l'ancienne donnée (en l'écrasant).
5. Écrire une fonction `remove` de type `'k -> ('k,'v) dict -> ('k,'v) dict` telle que `remove k dict` renvoie le dictionnaire `dict` dans lequel on a enlevé la l'entrée correspondant à la clé `key`, si cette clé est présente dans `dict` (et renvoie `dict` s'il ne contient pas la clé `key`).

Exercice 13.2 La seconde implantation demandée représente un dictionnaire comme une liste de couples (clé, valeur). Dans cette version, lorsqu'on veut mettre à jour une valeur associée à une clé déjà présente dans le dictionnaire, on la masquera simplement par un nouvel ajout. Par exemple, la liste suivante représente un dictionnaire à 3 entrées :

```
[(1,"Lundi"); (2,"Mardi"); (3,"Jeudi")]
```

Si on veut remplacer la valeur associée à la clé `3` par `"Mercredi"`, on obtiendra la liste suivante :

```
[(3,"Mercredi"); (1,"Lundi"); (2,"Mardi"); (3,"Jeudi")]
```

L'entrée `(3, "Jeudi")` de cette liste est masquée par l'entrée `(3, "Mercredi")` qui arrive plus tôt dans la liste (parce qu'elle a été insérée plus récemment). La valeur associée à l'entrée 3 est donc `"Mercredi"`.

Reprendre les questions 1 à 5 de l'exercice précédent avec cette nouvelle structure de données.

Vous devez d'abord redéfinir le type `dict`. Une fois ce type défini, la constante `empty_dict` et les fonctions doivent avoir le **même type** que dans l'exercice précédent. Vous avez le choix, pour la question 1 (définition du type) :

- soit de faire porter aux secondes composantes des couples des valeurs de type `'v`,
- soit de leur faire porter des valeurs de type `'v option`.

Dans le premier cas, la fonction `remove` ne peut pas être implantée, on la définira ainsi :

```
let remove key dict = raise (Failure "Unimplemented")
```

Cette écriture signifie que la fonction provoque une exception (les exceptions seront abordées ultérieurement). Dans le second cas (utilisation du type `'v option`), pour simuler la suppression de l'élément ayant la clé `key`, on il suffira d'insérer `(key, None)`.

Exercice 13.3 Reprendre les questions 1 à 5 de l'exercice 1 en utilisant comme structure de données des listes de couples `(clé,valeur)` **sans répétition de clé** : une clé se trouvera 0 ou 1 fois dans le dictionnaire.

Ainsi, la suppression dans le dictionnaire `[(1,"Lundi"); (2,"Mardi"); (3,"Jeudi")]` de la clé `3` renverra le dictionnaire `[(1,"Lundi"); (2,"Mardi")]`.

Ensuite, l'ajout de `(3, "Mercredi")` au dictionnaire `[(1,"Lundi"); (2,"Mardi")]` renverra le dictionnaire `[(3,"Mercredi"); (1,"Lundi"); (2,"Mardi")]`.

Exercice 13.4 On a vu dans l'exercice 1 qu'on peut utiliser des fonctions pour représenter des dictionnaires. Dans cet exercice, on utilise des fonctions pour représenter des listes.

1. Par quelle fonction pourrait-on représenter la liste `["Hello"; "World!"]` ? La liste `[0;2;4;6;8]` ?
2. En utilisant le type `'a option`, proposer un type `'a fun_list` pour représenter une liste d'éléments par une fonction.
3. Quelle est la condition sur une fonction du type écrit pour qu'elle représente effectivement une liste finie d'éléments ? La représentation devra permettre de calculer la longueur de la liste représentée par une fonction. Dans les questions suivantes, sauf dans les deux dernières, on suppose cette condition réalisée.
4. Écrire une fonction `fun_nil : 'a fun_list` qui représente la liste vide.
5. Écrire une fonction `fun_is_nil : 'a fun_list -> bool` qui teste si son argument représente la liste vide.
6. Écrire une fonction `fun_nth : int -> 'a fun_list -> 'a option` telle que `fun_nth n l` renvoie le n-ième élément de la liste `l`.
7. Écrire une fonction `fun_cons : 'a -> 'a fun_list -> 'a fun_list` telle que `fun_cons x l` renvoie la liste construite en ajoutant l'élément `x` en tête de la liste `l`.
8. Écrire une fonction `fun_tail : 'a fun_list -> 'a fun_list` telle quel `fun_tail l` renvoie la liste représentée par `l` privée de son 1er élément.
9. Écrire une fonction `fun_length : 'a fun_list -> int` qui calcule la longueur de la liste représentée

par son premier argument.

10. Écrire une fonction `fun_map : ('a -> 'b) -> 'a fun_list -> 'b fun_list` telle que `fun_map f l` renvoie la liste dont les éléments sont ceux de `l` auxquel on a appliqué la fonction `f`.
11. Écrire une fonction `list_of_fun_list : 'a fun_list -> 'a list` telle que `list_of_fun_list l` calcule la liste OCaml représentée par la liste fonctionnelle `l`.
12. Écrire une fonction `fun_list_of_list : 'a list -> 'a fun_list` réalisant la conversion inverse.
13. Le type que vous avez défini permet aussi de représenter des listes infinies. Écrire la représentation de la liste infinie des carrés d'entiers.
14. Écrire une fonction `truncate : 'a fun_list -> int -> 'a fun_list` telle que `truncate l n` renvoie la liste `l` dans laquelle les éléments aux positions supérieures ou égales à `n` ont été suprimés.

Exercice 13.5 1. Évaluer la complexité de recherche d'un élément dans un dictionnaire implanté avec la structure de données de l'exercice 4.3.

Pour réduire le temps de recherche d'un élément, on organise les dictionnaires dans des arbres. Un tel arbre aura le type

```
type ('k,'v) abr = Empty | Node of ('k,'v) abr * 'k * 'v * ('k,'v) abr
```

Pour accélérer la recherche dans un tel arbre, on range les entrées (clé,valeur) en respectant la règle suivante : en tout nœud interne ayant une clé k, les nœuds internes du sous-arbre gauche ont une clé strictement inférieure à k, et les nœuds internes du sous-arbre droit ont une clé strictement supérieure à k. Toutes les opérations produisant des arbres devront respecter cette règle, et l'opération de recherche d'un élément pourra l'utiliser.

Cela suppose donc qu'on dispose d'un ordre total sur l'ensemble des clés. Comme on ne sait pas à l'avance quel sera le type des clés, cet ordre total sera représenté par une fonction de comparaison de clés, qui retournera une valeur indiquant si la première clé est inférieure, égale ou supérieure à la seconde. On définit donc le type

```
type comparison = Lt | Eq | Gt
```

où `Lt` signifie inférieur (lower than), `Eq` signifie égal, et `Gt` signifie plus grand que (greater than).

2. Définir un type `type ('k,'v) dict` utilisant le type type `('k,'v) abr`.
3. Définir une **fonction** `empty_dict: ('k -> 'k -> comparison) -> ('k,'v) dict` qui construit un dictionnaire vide incorporant une fonction de comparaison donnée en premier argument.
4. Écrire une fonction `find : 'k -> ('k,'v) dict -> 'v option` telle que `find key dict` renvoie `None` si la clé `key` n'est pas trouvée dans le dictionnaire, et la valeur appariée à la clé `key` si elle est présente dans le dictionnaire.
5. Écrire une fonction `add : 'k -> 'v -> ('k, 'v) dict -> ('k, 'v) dict` telle que `add key value dict` renvoie le dictionnaire obtenu à partir de `dict` dans lequel la clé `key` a été ajoutée si elle n'était pas présente, et dans lequel sa valeur associée est `value` (qu'elle soit présente ou non dans `dict`).
6. ($\star$) Écrire une fonction `remove : 'k -> ('k,'v) dict -> ('k,'v) dict` telle que `remove key dict` renvoie le dictionnaire `dict` dans lequel la clé `key` a été supprimée. Si la clé n'existe pas, on terminera la fonction en lançant une exception `failwith "Should not happen"`. Cette question est plus difficile : réfléchir aux différents cas de figure, selon que les fils du nœud à supprimer sont 2 feuilles, une feuille et un nœud interne, ou deux nœuds internes (cas le plus compliqué).

Chapitre 14

Efficacité

14.1 Rappels

Exponentielle en base a

On appelle fonction exponentielle réelle, toute fonction de $\mathbb{R}$ dans $\mathbb{R}$, non identiquement nulle et continue en au moins un point, transformant une somme en produit, c'est-à-dire vérifiant l'équation fonctionnelle

$$\forall u, v \in \mathbb{R} f(u+v) = f(u) \times f(v).$$

Une telle fonction f est continue et strictement positive et pour tout réel $a > 0$, l'unique f telle que $f(1) = a$ est appelée exponentielle de base a et se note exp_a. $exp_a(x)$ se note a^x.

Exponentielle en base e

En mathématiques, la fonction exponentielle est la fonction notée exp qui est sa propre dérivée et qui prend la valeur 1 en 0. Elle est utilisée pour modéliser des phénomènes dans lesquels une différence constante sur la variable conduit à un rapport constant sur les images. Ces phénomènes sont en croissance dite « exponentielle ».

On note e la valeur de cette fonction en 1. Ce nombre e qui vaut approximativement $2,71828$ s'appelle la base de la fonction exponentielle et permet une autre notation de la fonction exponentielle :

$$\forall x, exp(x) = e^x$$

La fonction exponentielle est le cas particulier des fonctions de type appelées exponentielles de base a tel que $f(0) = 1$.

On peut la déterminer comme limite de suite ou à l'aide d'une série entière.

$$exp(x) = \Sigma_{n=0}^{+\infty} \frac{x^n}{n!}$$

C'est la bijection réciproque de la fonction logarithme népérien.

Logarithmes (Wikipédia)

En mathématiques, le logarithme de base b d'un nombre réel strictement positif est la puissance à laquelle il faut élever la base b pour obtenir ce nombre. Par exemple, le logarithme de 1000 en base 10 est 3, car 1000 = 10×10×10 = 103. Le logarithme de x en base b est noté logb(x). Ainsi log10(1000) = 3.

Tout logarithme transforme un produit en somme :

$$\log_b(x \cdot y) = \log_b x + \log_b y$$

un quotient en différence :

$$\log_b \left(\frac{x}{y} \right) = \log_b x - \log_b y$$

une puissance en produit :

$$\log_b(x^p) = p \log_b x.$$

Propriétés

$$log_a(n) = log_b(n)/log_b(a)$$

`rajouter quelques formules?`

14.2 Complexité

Notion de complexité

Dans cette partie, on s'intéresse à l'écriture de fonctions **efficaces**. L'efficacité d'une fonction est souvent mesurée en termes de temps et d'espace nécessaires pour évaluer la fonction sur ses entrées. Bien sûr, ce temps et cet espace dépendent de ses entrées, et en particulier de leur taille : l'évaluation d'une fonction prendra en général plus de temps (et d'espace) sur une entrée de grande taille que sur une entrée de petite taille.

Les deux quantités qu'il est utile d'estimer sont les suivantes :

- l'efficacité en temps d'une fonction. On peut estimer cette quantité sur des entrées de taille donnée en comptant le nombre d'opérations élémentaires utilisées lors de l'appel de la fonction, dans le pire des cas sur toutes ces entrées. Par exemple, pour une fonction qui construit une liste, compter le nombre de fois où l'opérateur `::` est utilisé donne une estimation du temps nécessaire à l'évaluation de la fonction sur son entrée.
- l'efficacité en espace, qui mesure la place mémoire nécessaire à l'exécution de la fonction. Cette estimation est particulièrement importante pour les fonctions récursives, dans laquelle la gestion de la mémoire n'est pas explicitée par le programmeur. En particulier, les fonctions qui ne sont pas *récursives terminales* utilisent un espace sur la pile d'exécution qui est souvent très limitatif. Écrire une version récursive terminale ne diminue pas nécessairement l'espace mémoire nécessaire à l'évaluation de la fonction, mais permet qu'il soit alloué, au moins en partie, ailleurs que sur cette pile (qui est souvent très limitée).

Notation $\mathcal{O}$

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

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

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

Exercice 14.1 1. On considère la fonction suivante qui concatène deux listes :

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

Combien d'appels récursifs l'appel `append l1 l2` provoque-t-il ? Donnez votre réponse en fonction de la longueur des listes `l1` et `l2`.

2. Combien de fois l'opérateur `::` est-il utilisé sur l'appel `append l1 l2` ?

3. On utilise la fonction `append` pour écrire la fonction `reverse` suivante, qui renverse une liste :

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

On appelle $c(n)$ le nombre d'utilisations de l'opérateur `::` sur l'appel `reverse l`, où n est la taille de la liste `l`. Remarquer que ce nombre dépend seulement de la longueur de la liste `l` (et pas des valeurs des éléments de la liste). En utilisant la question précédente et la définition récursive de `reverse`,

écrire une récurrence satisfaite par $c(n)$.

4. Déduire de la question précédente la valeur de $c(n)$.

Exercice 14.2 Dans cet exercice, on veut écrire une fonction `reverse_efficace` telle que `reverse_efficace l` utilise n fois l'opérateur `::`, où n est la longueur de la liste `l`.

1. Que calcule la fonction suivante ? Justifiez votre réponse.

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

2. En utilisant la fonction `rev_append` de la question 1, écrivez une fonction `reverse_efficace` telle que `reverse_efficace [a1;...;an]` calcule la liste `[an;...;a1]`.
3. Montrez que `reverse_efficace l` effectue bien n utilisations de l'opérateur `::`, où n est la longueur de `l`.

Exercice 14.3 1. Définir un type `e_b_c` (pour **e** pour entier, **b** pour booléen, **c** pour chaîne) permettant de représenter soit un entier, soit un booléen, soit une chaîne de caractères.

2. Écrire une fonction `somme` qui prend en argument une liste d'éléments de type `e_b_c` et calcule la somme de tous les entiers de cette liste (les autres éléments de la liste seront ignorés).
3. Écrire une version récursive terminale de la fonction `somme` précédente.
4. Écrire une fonction `filtre_int` qui prend en argument une liste d'éléments de type `e_b_c` et calcule la liste de tous les entiers de cette liste (les autres éléments de la liste seront ignorés).
5. Écrire une version récursive terminale de la fonction `filtre_int` précédente.
6. Écrire une fonction `concat` qui prend en argument une liste d'éléments de type `e_b_c` et calcule la concaténation de toutes les chaînes de cette liste.
7. Écrire une version récursive terminale de la fonction `concat` précédente.

Exercice 14.4 On veut écrire une version récursive terminale de la fonction `power`. On considère la fonction $g(a, x, n) = ax^n$.

1. Écrire une équation liant $g(a, x, n)$ et $g(a', x', n-1)$, pour $n > 0$ et a', x' bien choisis.
2. En déduire une fonction `power` récursive terminale.
3. Reprendre les 2 questions précédentes en
   - écrivant une récurrence liant $g(a, x, n)$ et $g(a', x', n/2)$,
   - écrivant une fonction `power'` récursive terminale et plus efficace que la fonction `power` ci-dessus.
4. Combien d'appels récursifs sont-ils effectués dans le premier cas, en fonction de l'exposant n ? Dans le second ?
5. Écrire une version générique `power_gen mult one x n` de cette fonction, de type

   `('a -> 'a -> 'a) -> 'a -> 'a -> int -> 'a`,

   où `mult` est une fonction de multiplication, et qui calcule la puissance `n`-ème de `x` (pour cette multiplication) multipliée par `one`.

Exercice 14.5 La suite de Fibonacci est définie par $F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n > 1$.

1. Écrire une fonction récursive naïve `fib1: int -> int` telle que `fib1 n` calcule F_n.
2. Montrer que le nombre d'additions effectuées par `fib1 n` est $2^{\Theta(n)}$.
3. Écrire une version `fib2 n`
   - récursive terminale,
   - et effectuant seulement $\Theta(n)$ additions.

   Aide : considérer la *suite de Fibonacci généralisée* définie par $G_0 = a$, $G_1 = b$ et $G_n = G_{n-1} + G_{n-2}$ pour $n > 1$, où a et b sont deux entiers naturels quelconques.
4. Soit F la matrice suivante :
$$F = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}.$$
   Montrer que pour tout $n > 0$, on a :
$$F^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$
5. Utiliser la fonction `power_gen` de l'exercice précédent pour écrire une fonction `fib3` telle que `fib3 n` calcule F_n en effectuant $\Theta(\log n)$ opérations arithmétiques (addition ou multiplication d'entiers).

Exercice 14.6 1. Écrire une fonction `decomp10` qui a un entier `n` associe la liste des chiffres de sa décomposition en base 10, chiffre de poids faible en tête. Par exemple, `decomp10 239847` doit retourner `[7; 4; 8; 9; 3; 2]`.

2. Écrire la fonction réciproque, qui prend en argument une liste de chiffres de 0 à 9, et qui retourne l'entier codé par cette liste (chiffre de poids faible en tête).
3. Reprendre les questions 1 et 2 en remplaçant chiffre de poids faible par chiffre de poids fort.
4. Que faut-il changer aux questions précédentes pour traiter les mêmes questions en base 2 ?

Chapitre 15

Tris

15.1 Tri rapide

Exercice 15.1 Réviser le principe du *tri rapide*.

Exercice 15.2 Écrire une fonction `partition` de type `'a -> 'a list -> 'a list * 'a list` qui prend en paramètre un pivot p, une liste l et qui retourne un couple de listes (l_1, l_2) telles que l_1 contient les éléments de l qui sont $< p$ et l_2 les autres. Exemples :

```
# l;;
- : int list = [1; 5; 3; 6; 8; 3; 9; 10; 2]
# partition  5 l;;
- : int list * int list = ([2; 3; 3; 1], [10; 9; 8; 6; 5])
# partition  3 l;;
- : int list * int list = ([2; 1], [10; 9; 3; 8; 6; 3; 5])
```

Exercice 15.3 Écrire la fonction `quick_sort` de type `'a list -> 'a list` qui prend en paramètre une liste l et qui retourne la liste triée selon le prédicat $<$ des éléments de l en suivant l'algorithme récursif du tri rapide. Comme pivot, on utilisera le premier élément de la liste. Exemples :

```
# quick_sort [];;
- : 'a list = []
# quick_sort  [1; 5; 3; 6; 8; 3; 9; 10; 2];;
- : int list = [1; 2; 3; 3; 5; 6; 8; 9; 10]
```

15.2 Tri fusion

Exercice 15.4 Réviser le principe du *tri fusion*.

Exercice 15.5 Écrire une fonction `split` de type `'a list -> 'a list * 'a list` qui prend en paramètre une liste l et qui retourne un couple de liste (l_1, l_2) résultant du découpage de la liste l en deux sous-listes dont la différence des longueurs est $\leqslant 1$. Exemples :

```
# split [1; 5; 3; 6; 8; 3; 9; 10; 2];;
- : int list * int list = ([1; 5; 3; 6], [8; 3; 9; 10; 2])
# split [1; 5; 3; 6; 8; 3; 9; 10; 2; 7];;
- : int list * int list = ([1; 5; 3; 6; 8], [3; 9; 10; 2; 7])
```

Exercice 15.6 Écrire la fonction `merge` de type `'a list -> 'a list -> 'a list` qui prend en paramètre deux listes triées l_1 et l_2 et qui retourne la liste triée éléments de l_1 et l_2 obtenue par fusion des deux listes. Exemples :

```
# merge [1;3;5;9] [4;6;7];;
- : int list = [1; 3; 4; 5; 6; 7; 9]
```

Exercice 15.7 En utilisant les fonctions `split` et `merge`, écrire la fonction `merge_sort` de type `'a list -> 'a list` qui prend en paramètre une liste l et qui retourne la liste triée des éléments de l en suivant l'algorithme récursif du tri fusion. Exemples :

```
# merge_sort [1; 5; 3; 6; 8; 3; 9; 10; 2];;
- : int list = [1; 2; 3; 3; 5; 6; 8; 9; 10]
```

Chapitre 16

Programmation modulaire

16.1 Programmation modulaire

Définition 1. *La* programmation modulaire *consiste à découper d'un programme en plusieurs modules.*

Définition 2. *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

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

```
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 p2)
  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.

Signature par défaut

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
```

16.2 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.

Compilation d'un seul fichier

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).

Édition de liens

```
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 mycomplex.cmo zones.cmo images.cmo visu_z
```

16.3 Systèmes de compilation automatisés

`make, OMake, ocamlbuild, Oasis, Dune`

16.4 Foncteurs

Les foncteurs sont les outils de la généricité.