Programmation Fonctionnelle
L2 informatique U. Bordeaux
Année 2019-2020

Rappel : les documents concernant cette unité d'enseignement sont disponibles sous https ://moodle1.u-bordeaux.fr/course/view.php ?id=4812. 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.
- Voir `https ://ocmal.org/learn-companies.html`

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.).

- `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

1.3 Organisation de l'UE

12 CI de 1h20 + 12 TM de 1h20.

Évaluation

Contrôle continu: 1 DS (1h20) et 1 TP noté (1h20)

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

- Frédérique Carrère (A2)
- Irène Durand (responsable de l'UE) (A1 + A5)
- Stefka Gueorguieva (A3)
- Thomas Place (A4 + TM A1-2))
- Simon Archipoff (TM A3-4)

Tutorat

`à compléter`

Comment réussir ?

- Être actif en CI et TM
- Chercher les exercices par soi-même
- 2 à 3h 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.

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 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. Par exemple, `add 12 30*2` est équivalent à `(add 12 30) * 2`, qui vaut `84` et non à `add 12 (30*2)` (qui vaudrait `72`).

Exemple :

```
 # max 7 3;;
- : int = 7
```

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 courants prédéfinis, en particulier les opérateurs arithmétiques, utilisent la notation *infixe* (`1 + 4`).

Exemple :

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

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. Noter la définition récursive d'une expression

```
# 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
```

Conversions

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

Exercice 2.2 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`

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'expression 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 2.3 Pour chacune des expressions suivantes, indiquer si elle est correcte et si c'est le cas 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`

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

Commentaires

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

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

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 2.5 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 2.6 Même exercice que précédemment :

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

Exercice 2.7 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 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 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 $\text{pgcd}(n, p) = \text{pgcd}(p, n \bmod p)$. 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

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

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-1}, u_n), 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 ?

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 (tuples)

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.

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.

```
# 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>
```

Exercice 4.3 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 prédicat `est_une_figure carte` de type `carte -> bool` qui retourne `true` si `carte` est une figure, `false` sinon.
- Écrire un prédicat `est_de_couleur carte couleur` de type `carte -> couleur -> bool` qui retourne `true` si `carte` est de couleur `couleur`.

Exercice 4.4 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. Lors des pics de pollution, les véhicules diesel à 4 roues au moins sont interdits. Écrire une fonction `peut_rouler : vehicule -> bool` qui teste si un véhicule est autorisé.
4. 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.
5. Ajouter un constructeur au type `carburant` de façon à prendre en compte les véhicules hybrides (pouvant fonctionner avec deux types de carburants), et donner un exemple de véhicule hybride.

Chapitre 5

Application : Zones du plan

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 M de coordonnées $(x\,,y)$ son affixe, le nombre complexe $z = x + i \cdot y$.
Pour représenter un point du plan, on utilisera donc le type

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

1. Écrire l'accesseur `realpart` qui permet de récupérer la partie réelle d'un complexe.
2. Écrire l'accesseur `imagpart` qui permet de récupérer la partie imaginaire d'un complexe.
3. Définir une variable `c_origin` contenant le point de coordonnées $(0\,,0)$.
4. Définir une variable `p12` contenant le point $(1\,.\,,2\,.)$.
5. Implémenter les opérations
   `c_sum c1 c2`, `c_dif c1 c2`, `c_mul c1 c2`,
   `c_abs c`, `c_scal lambda c`, `c_exp c`
   qui permettent respectivement de calculer la valeur absolue d'un complexe, la somme, la différence, le produit de deux complexes, la multiplication d'un complexe par un scalaire (float), l'exponentielle complexe.

Exercice 5.2 Une transformation F du plan transforme chaque point M en son image M'.
Aux points M et M', on associe respectivement leurs affixes, z et z'. L'écriture complexe de la transformation F est $z' = f(z)$ où f est la fonction $C \to C$ qui à z associe z'.

1. L'écriture complexe d'une translation de vecteur $\overrightarrow{u}$ est $z' = z + b$ où b est l'affixe du vecteur $\overrightarrow{u}$.
   Écrire une fonction `move_point point vector` de type

   ```
   val move_point : mycomplex -> mycomplex -> mycomplex = <fun>
   ```

   qui renvoie l'image de `point` par une translation de vecteur `vector`.
2. L'écriture complexe d'une rotation de centre Ω d'affixe ω et d'angle θ est $z' = e^{i\cdot\theta} \cdot (z - \omega) + \omega$.
   Écrire une fonction `rotate_point point origin angle` de type

   ```
   val rotate_point : mycomplex -> mycomplex -> float -> mycomplex = <fun>
   ```

   qui renvoie l'image de `point` par une rotation de centre `origin` et d'angle `angle`.
3. L'écriture complexe d'une homothétie de centre Ω d'affixe ω et de rapport k est $z' = k \cdot (z - \omega) + \omega$
   Écrire une fonction `scale_point point origin scale_factor` de type

   ```
   val scale_point : mycomplex -> mycomplex -> float -> mycomplex = <fun>
   ```

   qui renvoie l'image de `point` par une homothétie de centre `origin` et de rapport `scale_factor`.

5.2 Zones du plan représentées par leur fonction caractéristique

Exercice 5.3 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, une zone vide est représentée par la variable `nowhere` suivante :

```
# let nowhere = fun point -> false;;
val nowhere : 'a -> bool = <fun>
```

1. Définir une variable `everywhere` représentant la zone correspondant au plan tout entier.
2. Écrire une fonction `point_in_zone_p` telle que `point_in_zone_p point zone` retourne `true` si le point `point` appartient à la zone `zone`.

Exemples :

```
# point_in_zone_p c_origin nowhere;;
- : bool = false
# point_in_zone_p c_origin everywhere;;
- : bool = true
```

Exercice 5.4 1. 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.

2. 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).
3. Écrire une fonction `zone_complement zone1` qui retourne la zone correspondant aux points ne se trouvant pas dans la zone `zone1`.
4. Écrire une fonction `zone_intersection` telle que `zone_intersection zone1 zone2` retourne la zone correspondant aux points se trouvant à la fois dans les zones `zone1` et `zone2`.

Exercice 5.5 1. Écrire une 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 move_zone zone vector =
```

```
fun p -> point_in_zone_p (c_dif p vector) zone;;
```

Exercice 5.6 S'inspirer de la fonction `move_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 (λ_1, λ_2). La fonction applique à la zone la transformation

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

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

Exercice 5.8

```
let test =
  let c = make_disk0 1. in
  let c1 = move_zone c (C(1., 0.)) in
  assert (point_in_zone_p (C(0.0, 0.5)) c);
  assert (not (point_in_zone_p (C(1.0, 0.5)) c));
  assert (point_in_zone_p (C(0.5, 0.)) (zone_intersection c c1));;
```

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

Exercice 5.9

1. Vérifier la présence des fichiers `images.cmo` et `visu-zones.ml`.
2. Regarder le contenu du fichier `visu-zones.ml`.
3. Compiler le fichier `visu-zones.ml`.
4. Utiliser la fonction `view_zone zone` pour produire une image PNG et visualiser une partie finie d'une zone.
5. Pour adapter la taille de la zone visualisée, utiliser `view_zone zone size`.

Exercice 5.10

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 6

Types récursifs

La récursivité est utilisable dans la définition des types. Ceci permet en particulier de construire des types infinis.

On peut par exemple représenter les listes d'entiers[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 8

6.1 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 6.1 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))
```

6.2 Entiers de Peano

Exercice 6.2 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 :

$$(m+n)_{n\in\mathbb{N}} = \left\{ \begin{array}{ll} & m+0=m \\ \forall n \in \mathbb{N} & m+S(n)=S(m+n) \end{array} \right\} \tag{6.1}$$

$$(m\times n)_{n\in\mathbb{N}} = \left\{ \begin{array}{ll} & m\times 0=0 \\ \forall n \in \mathbb{N} & m\times S(n)=(m\times n)+m \end{array} \right\} \tag{6.2}$$

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 7

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

Exercice 7.2 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 8

Listes

8.1 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. 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; 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[1] 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`.

1. (en fait construire une nouvelle liste ayant les mêmes éléments que `l` et `e` à la fin

8.2 Exemples de fonctions sur les listes

Exercice 8.1 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 8.2 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.

8.3 Génération de listes

Exercice 8.3 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 8.4 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]
```

8.4 Autres exercices sur les listes

Exercice 8.5 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 8.6 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 8.7 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 8.8 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 8.9 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 8.10 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 8.11 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 8.12 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 8.13 — 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` ? Justifiez votre affirmation par une preuve par récurrence.
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 9

Application : album photo

Exercice 9.1 — 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 | Miroir_d_Eau | Mode | People | Mon_assiette_au_resto
type photo = Photo of int * (subject list)
type album = photo list
```

Un exemple d'album est le suivant :

```
let mon_album =
  [Photo(2016, [Selfie; Miroir_d_Eau]);
   Photo(2014, [Selfie; People]);
   Photo(2014, [Selfie; Monument; Mode]);
   Photo(2012, [Mon_assiette_au_resto; People])]
```

1. Écrire les accesseurs `photo_annee photo`, `photo_subjects photo` qui retournent respectivement l'année et la liste des sujets de `photo`. `photo_annee` est de type `photo -> int` `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 la photo.
3. Écrire une fonction `select_by_subject subject album` 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 People mon_album` retourne : `[Photo(2014, [Selfie; People]); Photo(2012, [Mon_assiette_au_resto; People])]`.
4. Écrire une fonction `select_by_date : (int -> bool) -> album -> album` telle que l'appel `select_by_date p` retourne la liste des photos de l'album `a` dont la date satisfait la fonction booléenne `p`. Par exemple, `select_by_date (fun x -> x >= 2014) a` doit renvoyer les liste des photos de `a` datées de 2014 ou après.

Écrire une fonction `select pred album` de type `(photo -> bool) -> album -> album)` qui retourne la liste des photos qui vérifient `pred`.

Exercice 9.2 On définit maintenant le type `critere` suivant.

```
type criteria =
  Subject of subject
```

```
| Date  of (annee -> 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 : criteria -> photo -> bool` qui teste si une photo satisfait un critère.
3. Écrire une fonction `select : criteria -> album -> album` telle que `select c a` renvoie la liste des photos de l'album `a` satisfaisant le critère `c`.

Chapitre 10

Efficacité

10.1 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 ?`

10.2 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 10.1 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 10.2 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 10.3 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 10.4 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 10.5 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 10.6 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 11

Listes (suite)

11.1 Fonctions `fold`

Exercice 11.1 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 11.2 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 12

Arbres

Les ***arbres*** sont des structures souvent utilisées en informatique, qui servent à organiser des données, représenter des documents (par exemple, documents XML), des expressions, etc. Nous allons principalement utiliser des arbres ***binaires***, c'est-à-dire dont chaque nœud a au plus deux fils. Plusieurs variantes d'arbres binaires sont couramment utilisées, définies de façon **récursive**. Formellement, un ***arbre binaire*** `T` est

- soit l'arbre vide,
- soit constitué :
  - d'un nœud `r` pouvant porter une valeur, appelé la *racine* de l'arbre,
  - d'un arbre `G`,
  - d'un arbre `D`.

  On voit que la définition est récursive, puisqu'un arbre non vide est constitué de deux sous-arbres (et de sa racine).
- L'arbre `G` s'appelle le ***sous-arbre gauche*** de `T`. Si l'arbre `G` n'est pas vide, sa racine est appelée ***fils gauche*** du nœud racine `r` de `T`.
- De même, l'arbre `D` s'appelle le ***sous-arbre droit*** de `T`. Si l'arbre `D` n'est pas vide, sa racine est appelée ***fils droit*** du nœud racine `r` de `T`.

Attention : Les sous-arbres vides ne sont pas des nœuds. En particulier, ils ne portent pas de valeur. Toutes les valeurs portées dans les nœuds de l'arbre sont souvent de même type, habituellement noté `'a` dans le code OCaml. Ces valeurs sont appelées *étiquettes*. Les feuilles sont des nœuds particuliers : leur sous-arbre gauche et leur sous-arbre droit sont vides ; elles n'ont ni fils gauche, ni fils droit. Les autres nœuds sont dits *internes*, et ont un fils gauche *ou* un fils droit (ou les deux). Sur les figures, les arbres vides sont représentés par des carrés. L'arbre le plus simple est l'arbre vide, qu'on représente de la façon suivante :

Comme la définition est récursive, un nœud d'un arbre `T` a :

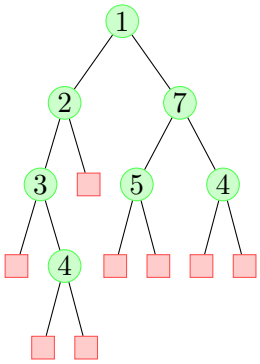
- soit un fils gauche (si l'arbre qui lui est attaché à gauche n'est pas vide).
- soit pas de fils gauche (si l'arbre qui lui est attaché à gauche est vide).

La notion de fils (gauche et droit) se généralise ainsi à tous les nœuds (et pas seulement à la racine).

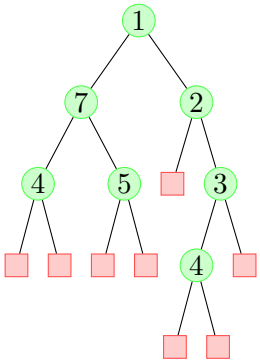
Dans ce cours, toutes les valeurs portées dans les nœuds de l'arbre seront de même type, noté `'a` par exemple dans les programmes OCaml. Ces valeurs sont appelées ***étiquettes*** des nœuds.

Une ***feuille*** est un nœud dont les deux fils sont l'arbre vide. Une feuille a donc une étiquette, mais ses deux sous-arbres attachés sont vides. Les autres nœuds sont dits *internes*, et ont donc une étiquette, et soit un fils gauche, soit un fils droit. Sur les figures, les sous-arbres vides sont représentés par des carrés, et les nœuds (feuilles ou nœuds internes) par des cercles, dans lesquels on pourra indiquer (ou non) l'étiquette. L'arbre le plus simple est donc l'arbre vide, qu'on représente de la façon suivante :

L'arbre suivant comporte 7 nœuds dont 3 feuilles et 4 nœuds internes. Notez que deux feuilles portent la même étiquette : 4. Le fils gauche de la racine est le nœud étiqueté 2. Son fils droit est le nœud étiqueté 7.



L'ordre des fils gauche et droit a une importance. Ainsi, l'arbre ci-dessus est différent de l'arbre suivant, qui est son symétrique (ou miroir), obtenu en intervertissant récursivement les fils gauche et droit.



Exercice 12.1 1. Définir un type `'a btree` pour les arbres binaires dont les nœuds portent des étiquettes de type `'a`.

2. Modifier le type de la question 1 pour que les nœuds internes portent des valeurs de type `int`.

Sauf indication contraire, dans la suite des exercices, on utilisera seulement le type `'a btree` de l'exercice précédent.

Exercice 12.2 Écrire une fonction `single x` de type `'a -> 'a btree` qui construit un arbre avec un seul nœud binaire d'étiquette `x`.

Exercice 12.3 Écrire une fonction `btree_size : 'a btree -> int` qui prend en argument un arbre et renvoie son nombre de nœuds.

Exercice 12.4 Écrire une fonction `btree_nb_leaves : 'a btree -> int` qui prend en argument un arbre et renvoie son nombre de feuilles.

Exercice 12.5 Écrire une fonction `btree_nb_internals : 'a btree -> int` qui prend en argument un arbre `t` et renvoie son nombre de nœuds internes.

Une ***branche*** d'un arbre t est une suite de nœuds allant de la racine à une feuille (sans "remonter"). Formellement, c'est une suite de nœuds $n_0, n_1, \ldots, n_k$ où n_0 est la racine de t, n_k est une feuille de t, et pour chaque i, n_{i+1} est un fils (droit ou gauche) de n_i. La ***hauteur*** de t est le nombre de nœuds de sa plus longue branche, moins 1. La hauteur de l'arbre vide est donc -1.

Exercice 12.6 Écrire une fonction `btree_height : 'a btree -> int` qui calcule la hauteur d'un arbre. Il n'est pour l'instant pas nécessaire de donner une version récursive terminale. Ce sera l'objet d'un autre exercice.

Exercice 12.7 On dit qu'un arbre est ***complet*** si tout nœud interne a exactement deux fils. Écrire une fonction `btree_is_complete : 'a btree -> bool` qui teste si un arbre binaire est complet.

Exercice 12.8 La ***profondeur*** d'un nœud est la longueur de l'unique chemin qui va de la racine à ce nœud. Un arbre est ***parfait***, s'il est complet et toutes ses feuilles ont la même profondeur. Écrire une fonction `btree_is_perfect` de type `'a btree -> bool` qui renvoie `true` si les feuilles de son arbre argument ont toutes la même profondeur, et `false` sinon.

Exercice 12.9 1. Écrire une fonction `btree_every : ('a -> bool) -> 'a btree -> bool` qui prend en paramètres :

- un prédicat `pred` sur des valeurs de type `'a`,
- un arbre `t` dont les étiquettes sont de type `'a`

et qui renvoie `true` si toutes les étiquettes de l'arbre `t` satisfont le prédicat `p`, et `false` sinon.

2. Écrire de même une fonction `btree_exists`, de même type, qui teste s'il existe une étiquette satisfaisant une propriété.

Exercice 12.10 Écrire une fonction `btree_mirror` de type `'a btree -> 'a btree` qui renvoie le symétrique (ou miroir) d'un arbre binaire.

Exercice 12.11 ⋆ (Plus difficile) Écrivez une fonction `btree_height: 'a btree -> int` récursive terminale, qui calcule la hauteur d'un arbre. **Aide** : utilisez un accumulateur contenant des couples `int * 'a btree`.

L'*ordre infixe* des nœuds internes d'un arbre ordonne les nœuds internes de l'arbre. Il est défini comme suit. Si l'arbre est une feuille, l'ordre infixe est la liste vide `[]`. Sinon, l'ordre infixe est la concaténation, dans cet ordre, des 3 listes suivantes :

- la liste en ordre infixe des nœuds internes du sous-arbre gauche,
- la liste d'un seul élément contenant l'étiquette de la racine,
- la liste en ordre infixe des nœuds internes du sous-arbre droit.

Exercice 12.12 1. Écrire une fonction `btree_to_list` de type `'a btree -> 'a list` qui renvoie la liste des étiquettes des nœuds internes d'un arbre en ordre infixe. Pour cette question, on pourra utiliser la concaténation de listes.

2. ⋆ Écrire une seconde version qui n'utilise pas la concaténation de listes.