SU/FSI/licence/info/LU2IN019
Programmation fonctionnelle

P. Manoury

Juin. 2019

1 Types de base et expressions typées

1.1 Type `bool`

Type des *valeurs booléennes.*

- 2 *constantes* : `true` et `false`
- 3 *opérateurs* : `&&`, `||` et `not`
  Application :
  - notation *préfixe* : `(not` e`)`
  - notations *infixe* : `(`e_1 `&&` e_2`)` et `(`e_1 `||` e_2`)`

Une *expression booléenne* est une expression *de type* `bool`.

Type des constantes et opérateurs :

```
true   :  bool
false  :  bool
not    :  bool -> bool
&&     :  bool -> bool -> bool
||     :  bool -> bool -> bool
```

Si e_1 et e_2 sont des expressions booléennes alors `true`, `false`, `(not` e_1`)`, `(`e_1 `&&` e_2`)` et `(`e_1 `||` e_2`)` sont des expressions booléennes, et réciproquement.

Expressions fonctionnelles La fonction `xor` («ou exclusif») n'est pas définie.
L'expression booléenne `((`e_1 `||` e_2`) && (not (`e_1 `&&` e_2`)))` a pour valeur `true` si et seulement si

1. e_1 a pour valeur `true` et e_2 a pour valeur `false`

   ou

2. e_1 a pour valeur `false` et e_2 a pour valeur `true`

L'*expression fonctionnelle*

`(`**`fun`** e_1 e_2 `-> ((`e_1 `||` e_2`) && (not (`e_1 `&&` e_2`))))`

a pour *valeur* la fonction `xor`. Elle est de type `bool -> bool -> bool`.

Une expression fonctionnelle est construite avec le *mot clé* **`fun`** et le *symbole réservé* `->`.

Dans l'expression fonctionnelle ci-dessus, e_1 et e_2 sont des *paramètres formels*. Les paramètres formels sont des *identificateurs* (noms de variables), ils peuvent varier. L'expression

`(fun x y -> (x || y) && (not (x && y)))`

dénote à la même fonction que `(`**`fun`** e_1 e_2 `-> (`e_1 `||` e_2`) && (not (`e_1 `&&` e_2`)))`.

Application et évaluation L'application des expressions fonctionnelles utilise la notation préfixe. Pour obtenir la valeur d'une application, les paramètres formels sont remplacés par les *paramètres d'appel* lors de l'application des fonctions. Par exemple :

— l'application `((fun x y -> (x || y) && (not (x && y))) true false)` a pour valeur `true` ;

— l'application `((fun x y -> (x || y) && (not (x && y))) true true)` a pour valeur `false`.

Les paramètres d'appels sont n'importe quelle expression, pourvu qu'elle soit du type attendu (ici, `bool`).

La valeur de l'application `((fun x y -> (x || y) && (not (x && y))) true false)` est la valeur de l'expression `(x || y) && (not (x && y))` où `x` à la valeur `true` et `y` la valeur `false` ; c'est-à-dire, la valeur de l'expression `(true || false) && (not (true && false))`.

Définition En donnant un nom à une expression fonctionnelle, on définit une fonction. On donne un nom à une expression avec la construction **`let`**. Par exemple :

```
let xor = fun x y -> (x || y) && (not (x && y))
```

On peut vérifier que

— l'application `(xor true false)` a pour valeur `true` ;

— l'application `(xor true true)` a pour valeur `false`.

Sucre syntaxique pour les définitions de fonctions On peut écrire la définition

```
let xor x y = (x || y) && (not (x && y))
```

On lit cette définition comme : *«la valeur de l'application `(xor x y)` est égale à la valeur de l'expression `((x || y) && (not (x && y)))`, pour toute valeur de `x` et `y`.»*

L'expression à droite du symbole = est appelé le *corps de la fonction* définie.

Explicitation des types dans une définition de fonction On peut écrire, et on écrira

```
let xor (x:bool) (y:bool) : bool =
  (x || y) && (not (x && y))
```

Expression alternative C'est une expression qui correspond à la construction logique *«si . . . alors . . . sinon . . .»*. Elle s'obtient en utilisant les mots clé `if`, `then` et `else`. Elle a la forme

if e **then** e_1 **else** e_2

où e est une expression booléenne. Les expressions e_1 et e_2 peuvent avoir n'importe quel type, mais il faut que e_1 et e_2 soient du même type.

La valeur d'une expression alternative est définie ainsi :

— si e a la valeur `true` alors (**if** e **then** e_1 **else** e_2) a la valeur de e_1 ;

— si e a la valeur `false` alors (**if** e **then** e_1 **else** e_2) a la valeur de e_2.

On peut dire également qu'une expression alternative vér**if**ie les équations :

(**if** `true` **then** e_1 **else** e_2) $=$ e_1

(**if** `false` **then** e_1 **else** e_2) $=$ e_2

Avec une expression alternative, on obtient une autre définition de la fonction `xor` :

```
let xor (b1:bool) (b2:bool) : bool =
  if b1 then (not b2)
  else b2
```

Opérateur polymorphe Puisqu'il peut recevoir en argument des valeurs d'un type quelconque, l'opérateur `if-then-else` est dit *polymorphe*. On peut écrire son type de cette manière : `bool -> 'a -> 'a -> 'a` ; où `'a` est une *variable de type* dont la valeur (un type) est déterminé à chaque utilisation de l'alternative. Le fait d'utiliser, dans le type du `if` le même symbole de variable indique que les types de deux alternants et du résultat doivent être le même.

Les oéprateurs de comparaison (= <> < <= > <=) sont également des opérateurs polymorphes, de type `'a -> 'a -> bool`. On ne compare que des valeurs appartenant au même type.

Nous verrons que nous pourrons également définir des fonctions polymorphes.

Filtrage Les langages ML offrent une construction particulière pour distinguer entre plusieurs *cas* de valeur d'une expression. C'est la construction de *filtrage*. Elle s'obtient avec les mots clé `match` et `with` ainsi que les symboles -> et `|`. On peut la rapprocher de la construction d'alternative généralisée de C ou JAVA : le `switch`. Mais elle est bien plus riche, comme nous le verrons par la suite.

Avec les booléens, son utilisation est simple et très proche de l'alternative car il n'y a que deux cas de valeurs pour une expression booléenne : `true` ou `false`. On peut l'utiliser pour définir de manière exhaustive, à la manière d'une table de vérité, la fonction `xor` :

```
let xor (b1:bool) (b2:bool) : bool =
  match b1, b2 with
      true, true -> false
    | true, false -> true
    | false, true-> true
    | false, false -> false
```

Pour évaluer une expression de filtrage, le résultat de l'évaluation des expressions analysées est comparé séquentiellement avec les motifs de filtrage. La valeur de l'expression de filtrage est celle de la valeur qui se trouve à droite du symbole -> du premier motif qui correspond aux valeurs analysées.

On peut inverser l'ordre des motifs de filtrage lorsque les valeurs désignées sont exclusives les unes des autres. Par exemple

```
let xor (b1:bool) (b2:bool) : bool =
  match b1, b2 with
      true, false -> true
    | false, true-> true
    | false, false -> false
    | true, true -> false
```

Il existe, pour la construction de filtrage une espèce de «sinon» sous la forme d'un *motif universel* noté `_` (caractère «souligné»). Il permet de faire l'économie de comparaisons inutiles. Par exemple :

```
let xor (b1:bool) (b2:bool) : bool =
  match b1, b2 with
      true, false -> true
    | false, true -> true
    | _ -> false
```

Enfin, les motifs peuvent mentionner des noms de variables. Dans ce cas, la variable prend la valeur analysées correspondante et elle peut figurer dans l'expression se trouvant à droite du symbole `->` associé au motif. Par exemple :

```
let xor (b1:bool) (b2:bool) : bool =
  match b1, b2 with
```

```
  true, true -> false
| true, false -> true
| false, r -> r
```

ATTENTION : l'usage du filtrage est assez délicat. Il nécessite de savoir quel ensemble de valeurs analysées correspond à un motif pour placer ceux-ci dans un ordre adéquat.

Couples, n-uplets Il est parfois utile qu'une fonction calcule «plusieurs valeurs» à la fois. Pour prendre un exemple avec les circuits booléens : le *semi-additionneur* calcul un *bit* de somme *s* et un *bit* de retenue *c*. Le résultat de l'opération est *le couple* formé de *s* et *c* que l'on note `(`*s* `,` *c*`)`. Notez la virgule qui sépare les deux valeurs. Un couple est une valeur qui en contient deux. Ici, son type sera `bool * bool`.

La «demi-somme» des *bits* `b1` et `b2` est donnée par le ou exclusif (xor) et la retenue par la conjonction. On définit

```
let half_adder (b1:bool) (b2:bool) : (bool * bool) =
   ( xor b1 b2 , b1 && b2 )
```

Filtrage et nommage On obtient un additionneur complet avec une fonction à trois arguments : les deux *bits* à additionner (`b1` et `b2`) et une retenue initiale (`c0`). Le calcul est alors le suivant
- — calculer `(half_adder b1 b2)` qui donne un `(s1,c1)`
- — calculer `(halft_adder s1 c0)` qui donne un`(s2,c2)`

Le résultat de l'additionneur complet est le couple `(s2, c1 || c2)`.

Traduisons cela dans notre langage :

```
let adder (b1:bool) (b2:bool) (c0:bool) : (bool*bool) =
   match (half_adder b1 b2) with
      (s1,c1) -> (
        match (half_adder s1 c0) with
            (s2,c2) -> (s2, c1 || c2)
      )
```

Le résultat de `(half_adder b1 b2)` est *décomposé* avec la construction `match-with` pour accéder aux deux composantes du couple que l'on nomme `s1` et `c1`. Ces noms peuvent être utilisés (comme des noms déclarés par `let-in`) dans l'expression associée à la valeur filtrée ; ici, un autre filtrage.

On peut également utiliser la construction `let-in` pour décomposer un couple :

```
let adder (b1:bool) (b2:bool) (c0:bool) : (bool * bool) =
  let (s1,c1) = (half_adder b1 b2) in
  let (s2,c2) = (half_adder s1 c0) in
    (s2, c1 || c2)
```

Un couple, plus généralement un n-uplet, peut être argument d'une fonction. On utilise notre aditionneur pour définir une fonction d'addition de deux quartets (4 *bits*). On représente un quartet par un quadruplet de booléens, de type `bool*bool*bool*bool`.

On peut définir un racourci pour ce type :

```
type quartet = bool * bool * bool * bool
```

L'additionneur 4 *bits* prend en argument deux quartets `q1` et `q2`, et donne le couple formé du quartet qui contient la somme de `q1` et `q2` et de la retenue de cette somme :

```
let add4bits (q1:quartet) (q2:quartet) : (quartet*bool) =
  match q1, q2 with
    (a1,a2,a3,a4), (b1,b2,b3,b4) ->
      let (s1,c1) = adder a1 b1 false in
      let (s2,c2) = adder a2 b2 c1 in
      let (s3,c3) = adder a3 b3 c2 in
      let (s4,c4) = adder a4 b4 c3 in
        ((s1,s2,s3,s4), c4)
```

1.2 Type `int`

Type des valeurs entières, positives ou négatives (entiers relatifs). On appelle *expression entière* les expressions de type `int`.

- Les constantes du type `int` peuvent être écrites en notation *décimale*. Par exemple : `-42`, `0` ou `42`. On peut aussi utiliser la notation *hexadécimale* en mentionnant le préfixe `0x`. Par exemple, 42 s'écrit `0x2a`. Il existe également une notation *binaire* des entiers, avec le préfixe `0b`. Par exemple, 42 s'écrit `0b101010`.
- Les opérateurs arithmétiques de base sont `+ - * / mod abs succ pred`. Les opératieurs binaires utilisent la notation infixe : `(`e_1 `+` e_2`)` `(`e_1 `-` e_2`)` `(`e_1 `*` e_2`)` `(`e_1 `/` e_2`)` `(`e_1 `mod` e_2`)`. Les opérateurs unaires, la notation préfixe : `(succ` e`)` `(pred` e`)` `(abs` e`)`.
  Attention au signe `-` qui est aussi utilisé comme opérateur de la soustraction. Il faut souvent utiliser des parenthèse pour désambigüer son utilisation :
  `(abs -1)` n'est pas correct ;
  `(abs (-1))` est correct.
- Les opérateurs «bit à bit» sont des opérateurs de bas niveau qui reposent sur la *représentation binaire* des entiers. On y trouve la opérateurs logiques : `land lor lxor lnot`, en notation infixe pour les trois premiers et préfixe pour `lnot`. On y trouve également les opérateurs de décalage : `lsr lsl asr` ; notation infixe.

L'ensemble des valeurs de type `int` est *fini*. Une valeur de type `int` est en effet contenue dans un *mot mémoire*. Le nombre de valeurs entière varie donc selon l'architecture de la machine où sont exécutés les programmes (32 bits ou 64 bits, par exemple). On a deux constantes prédéfinies qui donnent la valeurs du plus petit et du plus grans entiers représentable, respectivement : `min_int` et `max_int`.

Les opérations arithmétiques sont effectuées *modulo* cette limite de taille : `max_int + 1` a pour valeur celle de `min_int`.

Fonctions partielles et exception La division est une *fonction partielle* : elle n'est pas définie si le diviseur est égal à 0. L'expression entière `1/0` n'a pas de valeur. Elle est syntaxiquement correcte et bien typée (`1` et `0` sont des expressions entières, la division `/` est de type `int -> int -> int`, donc `1/0` est une expression entière). Mais son évaluation *déclenche l'exception* : `Division_by_zero`. Lorsqu'une exception est déclenchée, le programme est interrompu. L'exception signale ici une *erreur d'exécution*.

Opérateurs de comparaison et polymorphisme On peut utiliser avec les entiers les opérateurs de comparaison : `< <= = >= >` et `<>`, en notation infixe.

Toutefois, les opérateurs de comparaison ne sont pas propres aux entiers. En effet, si on peut comparer deux entiers (par exemple, `(0 = 1)` a la valeur `false`), on peut également comparer deux booléens : par exemple, `(false < true)` a la valeur `true`.

Ainsi, les opérateurs de comparaison sont des opérateurs *polymorphes*. Ils peuvent prendre en argument des valeurs de n'importe quel type pour peu que ce soit le même : on ne peut pas comparer un entier avec un booléen.

Pour noter le type des opérateurs polymorphe, on utilise des *variables de type*. Syntaxiquement, les variables de type sont reprérées par l'usage du symbole `'` (caractère «apostrophe»). Par exemple, les opérateurs de comparaison ont le type `'a -> 'a -> bool`. La mention répétée de la variable de type `'a` force l'identité de type entre les deux arguments des opérateurs de comparaison.

Fonctions récursives La fonction d'élévation à la puissance n'est pas définie. Il n'y a pas d'expression qui donne la valeur de «x à la puissance n». Mais :

«x à la puissance 0»	$=$	1
«x à la puissance 1»	$=$	x
«x à la puissance 2»	$=$	$x \times x$
«x à la puissance 3»	$=$	$x \times x \times x$
etc.		

Schématiquement :

$$\text{«}x \text{ à la puissance } n\text{»} \quad \text{est égal à} \quad \underbrace{x \times \ldots \times x}_{n\ fois}$$

Il est alors crucial de faire la remarque suivante. Lorsque n est plus grand que 0 :

$$\underbrace{x \times \ldots \times x}_{n\ fois} \quad \text{est égal à} \quad x \times \underbrace{x \times \ldots \times x}_{n-1\ fois}$$

Cette remarque nous permet de poser *l'équation récursive suivante* :

$$\text{«}x \text{ à la puissance } n\text{»} = x \times \text{«}x \text{ à la puissance } (n-1)\text{»}$$

Et on a par convention que «x à la puissance 0» $= 1$. De cela on déduit la *définition récursive* :

$$\begin{array}{lcll} \text{«}x \text{ à la puissance } n\text{»} & = & 1 & \text{, si } (n = 0) \\ & = & x \times \text{«}x \text{ à la puissance } (n-1)\text{»} & \text{, sinon} \end{array}$$

Écrivons $pow(x, n)$ pour «x à la puissance n». Les *équations récursives* ci-dessus deviennent :

$$\begin{array}{lcll} pow(x, n) & = & 1 & \text{si } n = 0 \\ & = & x \times pox(x, n-1) & \text{sinon} \end{array}$$

On peut transcrire cette définition en Ocaml. Pour poser une *définition récursive*, on utilise les mots clé `let` et `rec`, l'un après l'autre :

```
let rec pow (x:int) (n:int) : int =
   if (n = 0) then 1
   else x * (pow x n)
```

Cette définition donne les schémas d'évaluations attendus :

$$
\begin{array}{lcl}
\texttt{(pow } x \texttt{ 0)} & = & \texttt{1} \\
\texttt{(pow } x \texttt{ 1)} & = & x \texttt{ * (pow } x \texttt{ 0)} \\
 & = & x \texttt{ * 1} \\
 & = & x \\
\texttt{(pow } x \texttt{ 2)} & = & x \texttt{ * (pow } x \texttt{ 1)} \\
 & = & x \texttt{ * } x \texttt{ * (pow } x \texttt{ 0)} \\
 & = & x \texttt{ * } x \texttt{ * 1} \\
 & = & x \texttt{ * } x
\end{array}
$$

etc.

Fonction partielle et exception La fonction `pow` est de type `int -> int -> int`. Comme `(-1)` est de type `int`, l'application `(pow` e `(-1))` est correcte, pour toute expression entière e. Mais,

$$\texttt{(pow } e \texttt{ (-1))} = e \texttt{ * (pow } e \texttt{ (-2))} = e \texttt{ * } e \texttt{ * (pow } x \texttt{ (-3))} = \ldots$$

La fonction `pow` est partielle en son second argument : il y a des arguments d'appel e_2 de type `int` pour lesquelles, quelle que soit l'expression entière e_1, l'application `(pow` e_1 e_2`)` n'a pas de valeur. En langage courant, on dit que «la fonction boucle», sous entendu, «infiniment».

Pour éviter ce cas de figure, on peut adopter une attitude de *programmation défensive* qui consiste à vérifier que le second argument ne risquera pas de provoquer une évaluation indéterminée. Pour la fonction `pow`, on vérifie que la valeur du second argument n'est pas strictement négative.

On a deux manières de mettre en œuvre la programmation défensive :

a) on complète le domaine de la fonction, en convenant, par exemple, que pour les puissances négatives, la valeur de l'application est 1. Ce qui donne la définition :

```
let rec pow (x:int) (n:int) : int =
  if (n <= 0) then 1
  else x * (pow x (n-1))
```

On aura ici que `(pow x -1)` a pour valeur 1. Cette première manière a toutefois l'inconvénient que l'application d'un argument impropre peu passer inapercue.

b) on intercepte les applications hors domaine en *déclenchant* une exception. L'utilisation d'un arguement impropre ne passera pas alors inaperçue.
On peut pour cela utiliser la *fonction prédéfinie* `failwith` :

```
let rec pow (x:int) (n:int) : int =
  if (n < 0) then failwith "invalid exponent"
  else if (n = 0) then 1
  else x * (pow x (n-1))
```

La fonction prédéfinie `failwith` déclenche l'exception `Failure` accompagnée d'un message sous forme de chaîne de caractère (voir **??** pour les chaînes de caractères).
On peut également utiliser la *fonction primitive* `raise` avec une exception prédéfinie du langage. L'exception `Invalid_argument` est d'ailleurs prévue à cet effet :

```
let rec pow (x:int) (n:int) : int =
  if (n < 0) then raise (Invalid_argument "pow: negative exponent")
  else if (n = 0) then 1
  else x * (pow x (n-1))
```

Définition locale (de fonction) Notre mise en œuvre de la programmation défensive pour la fonction `pow` à l'inconvénient d'avoir à évaluer à chaque appel récursif de la fonction un test qui n'est utile qu'une seule fois : lors du «premier» appel de la fonction. Pour palier cet inconvénient, on divise le calcul de la fonction en deux temps :

— Tester si le second argument est négatif ou non.
— Calculer récursivement l'élévation à la puissance, si le second argument n'est pas négatif. Dans ce cas, on pourra utiliser une définition «incomplète».

Pour réaliser cela, on utilise une *définition locale* de fonction. On obtient une définition locale avec les mots clé `let` (éventuellement suivi de `rec`) et `in`. Voici comment cela se présente dans notre cas :

```
let pow (x:int) (n:int) : int =
  let rec loop (n:int) =
    if (n = 0) then 1
    else x * (loop (n-1))
  in
    if (n < 0) then raise (Invalid_argument "pow: negative exponent")
    else (loop n)
```

Notez que la définition de `pow` elle-même n'est pas une définition récursive : on n'a pas écrit `let rec pow ...` La fonction récursive est la fonction locale `loop`. Celle-ci est utilisée dans l'expression qui vient après le mot clé `in`. L'ensemble syntaxique `let rec loop ...in ...` forme l'expression qui définit la fonction `pow`.

La forme générale d'une déclaration locale est `let` x `=` e_1 `in` e_2 (ou `let rec` x `=` e_1 `in` e_2). C'est une expression dont la valeur est la valeur de e_2 dans laquelle x a la valeur de e_1.

Portée des variables Il y a plusieurs remarques à formuler concernant la *portée* des variables dans notre dernière définition de `pow`.

— la déclaration locale de la fonction `loop` a pour portée l'expression qui vient après le mot clé `in`. C'est-à-dire que dans cette expression, le nom `loop` est connu et peut être utilisé. Cette définition de `loop` ne peut être utilisée qu'à cet endroit : c'est sa *portée lexicale*. Aucune autre portion de programme ne peut utiliser cette définition.
— la fonction locale `loop` utilise la variable `x` alors que celle-ci ne fait pas partie des arguments de `loop`. Ceci est possible car la définition de `loop` est *dans la portée* des arguments de la définition de `pow`.
— enfin, l'argument de `loop` s'appelle `n`, comme le second argument de `pow`. Il ne peut toutefois pas y avoir ici de confusion pour le processus d'évaluation : la mention de `n` dans la définition de `loop` *masque* celle de `n` dans la définition de `pow`.

Récurrence terminale Le schéma d'évaluation de `(pow 5 3)` se développe ainsi :

```
(pow 5 3) = (loop 3)
          = 5 * (loop 2)
          = 5 * (5 * (loop 1))
          = 5 * (5 * (5 * (loop 0)))
          = 5 * (5 * (5 * 1))
          = 5 * (5 * 5)
          = 5 * 25
          = 125
```

On y observe deux phases :

la première s'achève lorsque les appels récursifs de `loop` ont tous été «résolus». dans cette phase, les multiplications par 5 sont «mises en attente».

la seconde consiste à effectuer les multiplications jusqu'à obtenir le résultat final.

On appelle ces deux phases, respectivement la *descente récursive*[1] et la *remontée récursive*. Dans certains cas, on peut fusionner ces deux phases en utilisant une forme de définition *récursive terminale*.

1. Nous empruntons cette expression au domaine de l'analyse syntaxique.

Pour transformer la définition de `loop` en forme récursive terminale, on ajoute à celle-ci un argument appelé *accumulateur*. Cet argument supplémentaire nous permettra d'effectuer les multiplications que notre première définition de `loop` laisse en attente. Voici comment les choses se présentent :

```
let pow (x:int) (n:int) : int =
  let rec loop (n:int) (r:int) =
    if (n = 0) then r
    else (loop (n-1) (x*r))
  in
    if (n < 0) then raise (Invalid_argument "pow: negative argument")
    else (loop n 1)
```

Si l'on développe le schéma d'évaluation de `(pow 5 3)`, on obtient à présent :

$$\begin{array}{lcl} \texttt{(pow 5 3)} & = & \texttt{(loop 3 1)} \\ & = & \texttt{(loop 2 5)} \\ & = & \texttt{(loop 1 25)} \\ & = & \texttt{(loop 0 125)} \\ & = & 125 \end{array}$$

L'accumulateur contient à chaque appel récursif une valeur correspondant au résultat des multiplications mises en attente dans la version non terminale. Lorsqu'il n'y a plus d'appel récursif, la valeur de l'accumulateur correspond bien à toutes les multiplications qu'il fallait effectuer. En général, on a que `(loop` n r`)` a pour valeur $5^n \times r$. Puisqu'à l'appel initial de `loop`, on donne à r la valeur 1, on obtient bien que `(loop` n `1)` a pour valeur $5^n \times 1$, soit 5^n.

Attention : nous avons pris la peine de stipuler que *«dans certains cas, on peut fusionner»* les phases d'évaluation de l'application d'une fonction récursive. Cela n'est pas toujours facile. Ce peut même devenir inutilement complexe.

Fonctionnelle On peut définir une fonction qui réalise une «boucle récursive» générique contrôlée par une valeur entière. Étant donné une fonction f et une valeur a et un entier n, elle donnera la valeur de $(\underbrace{f\ldots(f}_{n\ fois}\ a)\ldots)$ – ce que l'on note aussi $f^n(a)$. On peut définir cette itération d'applications, récursivement, en fonction de n :

$$\begin{array}{lcll} f^n(a) & = & a & \text{si } n = 0 \\ & = & f(f^{n-1}(a)) & \text{sinon} \end{array}$$

Ce qui donne la définition Ocaml :

```
let rec iter (n:int) (f: 'a -> 'a) (a:'a) : 'a =
  if (n > 0) then (f (iter (n-1) f a))
  else a
```

On peut aussi poser les équations récursives suivantes :

$$\begin{array}{lcll} f^n(a) & = & a & \text{si } n = 0 \\ & = & f^{n-1}(f(a)) & \text{sinon} \end{array}$$

Ce qui donne la définition Ocaml :

```
let rec iter (n:int) (f: 'a -> 'a) (a:'a) : 'a =
  if (n > 0) then (iter (n-1) f (f a))
  else a
```

qui est récursive terminale.

La fonction `iter` a deux particularités :

— la fonction `iter` est de type : `int -> ('a -> 'a) -> 'a -> 'a`. C'est une fonction polymorphe ;
— c'est une *fonction d'ordre supérieure*, appelée aussi *fonctionnelle*, car l'un de ses arguments est lui même une fonction (`f: 'a -> 'a`).

Cette fonction permet de donner une définition compacte de l'élévation à la puissance. En effet, pour obtenir la valeur de «x à la puissance n», il faut itérer n fois la fonction «multiplier par x» sur la valeur neutre 1.

En utilisant une expression fonctionnelle comme argument de `iter`, on peut poser :

```
let pow (x:int) (n:int) : int =
  if (n < 0) then raise (Invalid_argument "pow")
  else (iter n (fun r -> x * r) 1)
```

où l'expression fonctionnelle `(fun r -> x * r)` représente l'opération «multiplier par x» ; elle est de type `int -> int`.

1.3 Type `float`

Type des «nombres à virgule». On appelera *flottant* les valeurs du type `float`.

Les constantes utilisent la *notation pointée*. Par exemple : `...-6.2832 ...-0.1 ...0.0 ...0.1 ...3.1416 ...`. On peut aussi utiliser une *exponentiation* (puissances de 10). On aura `56789e+3` a pour valeur 5678.9 ou `789e-3` a pour valeur 0.789.

Les opérateurs arithmétiques sont ceux que l'on a sur les entiers avec une petite différence de notation : les symboles sont suivi du caractère «point» : `+. -. *. /.`. Ils sont de type `float -> float -> float`.

Les types numériques `int` et `float` sont incompatibles :
l'expression `1 * 0.5` n'est pas correctement typée ;
l'expression `1.0 *. 0.5` est correctement typée.

Pour calculer avec des entiers ou des flottants on utilise des primitives de conversion explicites :
`int_of_float : float -> int` donne la partie entière de son argument ;
`float_of_int : int -> float`.

Il existe d'autres fonctions primitives ou prédéfinie sur les flottants, telles, par exemple, les fonctions trigonométriques. Nous n'en dirons pas plus sur ce sujet et vous invitons à consulter la documentation du langage.

2 Structures linéaires

2.1 Type `('a list)`

Type des listes *paramatrées*. Les listes peuvent recevoir des valeurs appartenant à n'importe quel type de données. C'est en ce sens que le type des listes est *«paramétré»*. On peut ne pas connaître le type des éléments d'un liste et on utilise alors la notation d'un *type polymorphe* : `('a list)` où apparaît une variable de type, ici ; `'a`.

Les listes sont des *structures linéaires dynamiques*. Les chaînes de caractères sont des structures *statiques* : une fois crées, leur taille ne change plus. Pour ajouter un caractères à une chaîne, il faut en créer une nouvelle pour y recopier les éléments de l'ancienne chaîne plus celui, que l'on veut ajouter. L'ajout d'un élément dans une listes sera beaucoup plus économique ; si l'on ajoute en début de liste.

Constantes Il y a potentiellement une infinité de listes. E ce, à deux titres :
— la taille d'une liste n'est *a priori* pas limitée ;

— il y a autant de liste d'une taille donnée qu'il y a de types de valeurs.

La notation pour les constantes de listes utilise les crochets `[` et `]` ainsi que le point virgule `;`. La *liste vide* qui ne contient aucun élément est notée `[]`. On peut former des listes de booléens : `[true]`, `[true,false]`, `[true,false,false],...` ; des listes d'entiers : `[1]`, `[1;2]`, `...` ; de chaînes de caractères : `["hello"]`, `["hello"; "world"]`, `...` ; ou encore des listes de listes d'entiers : `[[]]`, `[[1]]`, `[[1],[1,2]]`, `...`, etc. Les potentialités sont infinies. Les seules contraintes sont physiques : la mémoire de la machine doit être suffisante pour contenir la structure ; logique : tous les éléments d'une liste appartiennent au même type. Les types de ces valeurs sont compètement déterminé. On a, dans nos exemples, respectivement : `(bool list)`, `(int list)`, `(string list)` et `((int list) list)`.

Opérateur L'opérateur privilégié des listes est, à l'instar des chaînes de caractères, la concaténation. Elle est notée `@`, en infixe. Elle est de type `'a list -> 'a list -> 'a list`. Elle satisfait les schémas d'équation suivants :

`[] @ [x1; ..; xn]`	=	`[x1; .., xn]`
`[x1; ..; xn] @ []`	=	`[x1; ..; xn]`
`[x1; ..; xn] @ [y1; ..; ym]`	=	`[x1; ..; xn; y1; ..; ym]`

Constructeurs Les *constructeurs* des listes sont des opérations distinguées sur les listes. Ce sont les *primitives* qui sont à la base de la construction de toutes les structures de listes. Nous verrons, par exemple, comment on les utilisent pour définir la concaténation.

Le type des listes possède deux constructeurs. Le premier est une constante ; la liste vide que l'on note `[]`. Le second est l'opérateur qui permet d'ajouter un élément en tête de liste. Il est symbolisé par `::`, en notation infixe. Si `x` est une valeur, d'un certain type, et `xs` une liste d'éléments de cette valeur alors `x::xs` représente la liste qui commence par `x` et se poursuit par les éléments de `xs`. On prononce `x::xs` : `x` *«conse»* `xs`. Exemples :

— l'expression `x1::[]` a pour valeur celle de `[x1]` ;

— l'expression `x2::x1::[]` a pour valeur celle de `x2::[x1]` qui a pour valeur celle de `[x2; x1]`

— l'expression `x3::x2::x1::[]` a pour valeur celle de `x3::x2::[x1]` qui a pour valeur celle de `x3::[x2; x1]` qui, elle même a pour valeur celle de `[x3; x2; x1]`.

Ainsi toute expression de type `('a list)` s'évalue soit vers la constante `[]`, soit vers une structure *de la forme* `x::xs`.

Listes et filtrage La *structure de contrôle* `match-with` qui nous avait permis de distinguer entre plusieurs cas de valeurs des booléens peut également servir à distinguer les *cas de construction* des listes. Par exemple, on peut l'utiliser pour définir une fonction booléenne valant `true` si et seulement si son argument est une liste vide :

```
let is_empty (xs : 'a list) : bool =
  match xs with
    [] -> true
  | x::xs' -> false
```

On peut ici ne pas exprimer le deuxième cas, en utilisant le *motif universel* :

```
let is_empty (xs : 'a list) : bool =
  match xs with
    [] -> true
  | _ -> false
```

Définition récursive de la fonction de concaténation sur les listes.

La fonction de concaténation des listes satisfait les deux schémas d'équations suivants :

```
[] @ [y1; ..; ym]                =  [y1; .., ym]
[x1; x2; ..; xn] @ [y1; ..; ym]  =  [x1; x2; ..; xn; y1; ..; ym]
```

Appelons `ys` la liste notée `[y1; ..; ym]`. La première équation dit que : pour toutes liste `ys`, l'expression `[]@ys` a pour valeur celle de `ys` :

```
[] @ ys  =  ys
```

Pour la seconde équation : remarquons que la liste notée `[x1; x2; ..; xn]` est égale à `x1::[x2; ..; xn]` ; de même, la liste notée `[x1; x2; ..; xn; y1; ..; ym]` est égale à `x1::[x2; ..; xn; y1; ..; ym]`. La seconde équation peut donc se réécrire :

```
(x1::[x2; ..; xn]) @ [y1; ..; ym]  =  x1::[x2; ..; xn; y1; ..; ym]
```

Appelons `xs` la liste notée `[x2; ..; xn]` (attention : elle comence avec `x2`). Ici, intervient la remarque essentielle : la liste notée `[x2; ..; xn; y1; ..; ym]` correspond à la concaténation des listes `[x2; ..; xm]` et `[y1; ..; ym]`, c'est-à-dire à la valeur de l'expression `[x2; ..; xm] @ [y1; ..; ym]`. Ainsi, en utilisant les noms `xs` et `ys`, la seconde équation s'écrit :

```
(x1::xs) @ ys  =  x1::(xs @ ys)
```

En résumé, la concaténation est définie par les deux équations

$$\begin{cases} \texttt{[] @ ys} & = \texttt{ys} \\ \texttt{(x::xs) @ ys} & = \texttt{x::(xs @ ys)} \end{cases}$$

Ces deux équations nous donnent une définition par cas de constructeur de la fonction de concaténation des listes. C'est une définition récursive puisque l'opération définie apparaît à gauche et à droite dans la seconde équation. Cette définition équationnelle est implémentée dans la bibliothèque standard de OCAML (module `Pervasives`) de la manière suivante :

```
let rec ( @ ) l1 l2 =
  match l1 with
    [] -> l2
  | hd :: tl -> hd :: (tl @ l2)
```

Le module `List` de la bibliothèque standard contient un nombre importants de fonctions utilitaires sur les listes. Elles sont en général de type polymorphe sur les éléments des listes. Citons :

`List.length : 'a list -> int` qui donne le nombre d'éléments de son argument, sa *longueur*.

`List.mem : 'a -> 'a list -> bool` qui donne la valeur `true` si et seulement si son premier argument appartient à la lsite donnée en second argument.

`List.nth : 'a list -> int -> 'a` telle que `(nth xs i)` donne l'éléménet en `i`-ème position dans la liste `xs`, s'il existe. Le premier élément est à la position 0. On obtient l'exception `Failure "nth"` si `i` est supérieur ou égal à la longueur de `xs` ou l'exception `Invalid_argument "List.nth"` si `i` est négatif.

`List.rev : 'a list -> 'a list` telle que `(List.rev [x1; ..; xn])` donne la liste `[xn; ..; x1]`.

etc.

À titre d'exemples, voici les définitions de ces fonctions.

La fonction `length` : calculer la longueur d'une liste, c'est compter son nombre d'éléments. Si l'on considère les deux cas de construction des listes, on a :

— la liste `[]` ne contient aucun élément ;
— la liste `x::xs` contient un élément de plus que la liste `xs`.
La fonction `length` satisfait donc les deux équations :

$$\left\{\begin{array}{lcl} \texttt{(length [])} & = & 0 \\ \texttt{(length (x::xs))} & = & 1 + \texttt{(length xs)} \end{array}\right.$$

D'où la définition :

```
let rec length (xs : 'a list) : int =
  match xs with
    [] -> 0
  | _::xs -> 1+(length xs)
```

Comme on n'a pas eu besoin ici de la valeur du premier élément, on a utilisé le motif universel (`_`).

Cette fonction admet la définition récursive terminale suivante :

```
let length (xs : 'a list) : int =
  let rec loop (xs : 'a list) (r:int) : int =
    match xs with
      [] -> r
    | _::xs -> (loop xs (r+1))
  in
    (loop xs 0)
```

La fonction `mem` : pour déterminer si un élément `z` appartient ou non à une liste, on raisonne par cas de construction de la liste :
— si la liste est vide, alors `z` n'appartient pas à la liste ;
— si la liste est de la forme `x::xs` alors, il y a deux possibilités :
  — le premier élément de la liste est égal à `z` et donc `z` appartient à `x::xs`
  — `z` appartient à la liste `xs`
On peut donc poser que `mem` satisfait les *équations conditionnelles* suivantes :

$$\left\{\begin{array}{lcll} \texttt{(mem z [])} & = & \texttt{false} & \\ \texttt{(mem z (x::xs))} & = & \texttt{true} & \text{si } \texttt{x} = \texttt{z} \\ \texttt{(mem z (x::xs))} & = & \texttt{(mem z xs)} & \text{sinon} \end{array}\right.$$

D'où la définition :

```
let rec mem (z:'a) (xs:'a list) : bool =
  match xs with
    [] -> false
  | x::xs -> if (x=z) then true else (mem z xs)
```

On peut également dire que `z` est un élément de `x::xs` si et seulement si `x`=`z` ou `z` est un élément de `xs`. Ce qui donne la définition :

```
let rec mem (z:'a) (xs:'a list) : bool =
  match xs with
    [] -> false
  | x::xs -> (x=z) || (mem z xs)
```

La fonction (partielle) `nth` : quoique de réalisation assez simple, l'élaboration de cette fonction réclame un peu d'attention.

Intuitivement, l'application `(nth [x0; ..; xn] i)` a pour valeur `xi`, si `i` est compris entre 0 et n. Pour `i` négatif ou strictement supérieur à n, l'application n'a pas de valeur ; de même, `(nth [] i)` pas de valeur.

On peut remarquer que :
- l'indice du premier élément d'une liste est 0, donc l'application `(nth [x0; ..; xn] 0)` a pour valeur `x0` ;
- si l'indice d'un élément dans la liste `[x1; ..; xn]` est i alors, il est $i+1$ dans la liste `[x0; x1; ..; xn]` ; c'est-à-dire, si un élément est d'indice i dans la liste `[x0; x1; ..; xn]`, il est d'indice i dans la liste `[x1; ..; xn]`.

De ces remarques, on déduit les équations conditionnelles suivantes

`(nth (x::xs) i)`	=	`x`	si `i`=0
`(nth (x::xs) i)`	=	`(nth xs (i-1))`	sinon

On obtient de cette première analyse la définition suivante qui déclenche une exception lorsque la valeur n'est pas définie :

```
let rec nth (xs:'a list) (i:int) : 'a =
  match xs with
    [] -> (failwith "nth")
  | x::xs -> if (i=0) then x else (nth xs (i-1))
```

Si l'on regarde la spécification de la fonction `List.nth`, il y a en fait deux cas où la fonction n'a pas de valeur : le cas où l'indice est supérieur ou égal à la longueur de la liste et le cas où l'indice est négatif. Le premier cas est signalé par l'exception `(Failure "nth")` ; le second, par l'exception `(Invalid_argument "List.nth")`.

On peut intercepter le second cas avant d'activer la recherche récursive par l'application d'une fonction locale, comme nous l'avions fait, par exemple, pour la fonction `pow` (??). Cela donne :

```
let nth (xs:'a list) (i:int) : 'a =
  let rec loop (xs:'a list) (i:int) : 'a =
    match xs with
      [] -> (failwith "nth")
    | x::xs -> if (i=0) then x else (loop xs (i-1))
  in
    if (i < 0) then raise (Invalid_argument "List.nth")
    else (loop xs i)
```

Le déclenchement de l'exception `(Failure "nth")` est cohérent avec la spécification : « On obtient l'exception `Failure "nth"` si `i` est supérieur ou égal à la longueur de `xs` ». En effet si, `i` étant positif, la succession d'appels récursifs atteint la liste vide sans avoir annulé `i`, puisque l'indice `i` est décrémenté de 1 à chaque appel récursif, c'est que l'indice était supérieur à la longueur de la liste.

La fonction `rev` : schématiquement, l'application `(rev [x0; x1; ..; xn])` a pour valeur `[xn; ..; x1; x0]`. On peut remarquer que :
- `(rev [])` a pour valeur `[]` ;
- la liste `[xn; ..; x1]` est la valeur de `(rev [x1; ..; xn])` ;
- la liste `[xn; ..; x1; x0]` est la valeur de la concaténation de `[xn; ..; x1]` et `[x0]`.

On peut donc poser les deux équations suivantes :

$$\left\{\begin{array}{lcl}\texttt{(rev [])} & = & \texttt{[]} \\ \texttt{(rev (x::xs))} & = & \texttt{(rev xs) @ (x::[])}\end{array}\right.$$

Cela donnerait la définition suivante :

```
let rec rev (xs:'a list) : 'a list =
  match xs with
    [] -> []
  | x::xs -> (rev xs) @ [x]
```

Cette définition n'est toutefois pas très satisfaisante. Illustrons le en déroulant les étapes d'évaluation de `(rev [x1; x2; x3; x4])` :

(1)	`(rev [x1; x2; x3; x4])`	=	`(rev [x2; x3; x4]) @ (x1::[])`
(2)		=	`((rev [x3; x4]) @ (x2::[])) @ (x1::[])`
(3)		=	`(((rev (x4::[])) @ (x3::[])) @ (x2::[])) @ (x1::[])`
(4)		=	`((((rev []) @ (x4::[])) @ (x3::[])) @ (x2::[])) @ (x1::[])`
(5)		=	`((([] @ (x4::[])) @ (x3::[])) @ (x2::[])) @ (x1::[])`
(6)		=	`(((x4::[]) @ (x3::[])) @ (x2::[])) @ (x1::[])`
(7)		=	`((x4::([] @ (x3::[]))) @ (x2::[])) @ (x1::[])`
(8)		=	`((x4::(x3::[])) @ (x2::[])) @ (x1::[])`
(9)		=	`(x4::((x3::[]) @ (x2::[]))) @ (x1::[])`
(10)		=	`(x4::x3::([] @ (x2::[]))) @ (x1::[])`
(11)		=	`(x4::x3::(x2::[])) @ (x1::[])`
(12)		=	`x4::((x3::(x2::[])) @ (x1::[]))`
(13)		=	`x4::x3::((x2::[]) @ (x1::[]))`
(14)		=	`x4::x3::x2::([] @ (x1::[]))`
(15)		=	`x4::x3::x2::(x1::[])`

Les étapes (1) à (5) correspondent au développement de la fonction `rev` en terme de concaténations. Les étapes (6) à (15) sont les étapes d'évaluation des différentes concaténation mises en attentente. C'est ici que le bât blesse : chaque évaluation d'une concaténation reparcours des valeurs déjà envisagées ; par exemple, `x4` est repris 3 fois avant d'arriver à sa place définitive.

Il est donc intéressant ici d'envisager l'utilisation d'un accumulateur, et d'une fonction récursive locale, pour éviter la mise en attente de concaténations :

```
let rev (xs:'a list) : 'a list =
  let rec loop (xs:'a list) (r:'a list) =
    match xs with
      [] -> r
    | x::xs -> (loop xs (x::r))
  in
    (loop xs [])
```

Schémas d'itération Le module `List` contient également nombre de fonctionnelles qui correspondent à des schémas génériques de traitement ou d'exploration de listes.

Schéma d'application
`List.map : ('a -> 'b) -> 'a list -> 'b list` telle que `(List.map f [x1; ..; xn])` a pour valeur la liste `[(f x1); ..; (f xn)]`.

Schéma de filtrage
`List.filter : ('a -> bool) -> 'a list -> 'a list` telle que `(List.filter p xs)` donne la liste des éléments `xi` de `xs` pour lesquels `(p xi)` vaut `true`.

Schémas d'accumulation
`List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` telle que `(List.fold_right f`

[x1; ..; xn] a) donne la valeur de l'expression (f x1 .. (f xn a)..).
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a telle que (List.fold_left f a [x1; ..; xn]) donne la valeur de l'expression (f (.. (f a x1) ..) xn).

À titre d'exemples, voici les définitions de ces itérateurs :

```
let rec map (f : 'a -> 'b) (xs : 'a list) : ('b list) =
  match xs with
    [] -> []
  | x::xs -> (f x)::(map f xs)
let rec filter (p : 'a -> bool) (xs : 'a list) : ('a list) =
  match xs with
    [] -> []
  | x::xs -> if (p x) then x::(filter p xs)
             else (filter p xs)
let rec fold_right (f : 'a -> 'b -> 'b) (xs : 'a list) (b : 'b) : 'b =
  match xs with
    [] -> b
  | x::xs -> (f x (fold_right f xs b))
let rec fold_left (f : 'a -> 'b -> 'a) (a : 'a) (xs : 'b list) : 'a =
  match xs with
    [] -> a
  | x::xs -> (fold_left f (f a x) xs)
```

Notez que fold_left est récursive terminale.

Un itérateur pour les gouverner tous On peut (re)définir les itérateurs List.map et List.filter comme cas d'application de List.fold_right.

La fonction itérée par List.map est simplement le constructeur *cons* :

```
let map (f : 'a -> 'b) (xs : 'a list) : 'b list =
  List.fold_right (fun x r -> (f x)::r) xs []
```

La fonction itérée par List.filter sélectionne les éléments à retenir pour le résultat final :

```
let filter (p : 'a -> bool) (xs : 'a list) : 'a list =
  List.fold_right (fun x r ->  if (p x) then x::r else r) xs []
```

L'itérateur List.fold_right peut permettre une expression très compacte de certaines fonctions. Par exemple, on obtient la fonction de recherche de l'élément maximal d'une liste de la manière suivante :

```
let find_max (xs : 'a list) : 'a =
  match xs with
    [] -> raise (Invalid_argument "find_max")
  | x::xs -> List.fold_right max xs x
```

Notez que find_max est une fonction partielle, non définie pour la liste vide.

On peut comparer cette définition à une définition plus directe :

```
let find_max (xs : 'a list) : 'a =
  let rec loop (xs : 'a list) (r : 'a) =
```

```
    match xs with
      [] -> r
    | x::xs -> (loop xs (max x r))
  in
    match xs with
      [] -> raise (Invalid_argument "find_max")
    | x::xs -> (loop xs x)
```

On peut noter que la fonction locale `loop` est terminale récursive. Cela indique que, dans ce cas, on peut tout aussi bien utiliser l'itérateur `List.fold_left` :

```
let find_max (xs : 'a list) : 'a =
  match xs with
    [] -> raise (Invalid_argument "find_max")
  | x::xs -> List.fold_left max x xs
```

L'utilisation de `List.fold_right` ou `List.fold_left` est indiférent dès lors que la fonction itérée est associative et commutative. On peut alors préfèrer `List.fold_left` qui est récursive terminale.

`List.fold_left` accumule *«à l'envers»* : on peut définir la fonction `rev` directement avec `fold_left`, version récursive terminale :

```
let rev (xs:'a list) : ('a list) =
  List.fold_left (fun r x -> x::r) [] xs
```

Pour avoir un `map` récursif terminal, il faut *2 passes* :

```
let map (f:'a -> 'b) (xs:'a list) : ('b list) =
  List.fold_left (fun r x -> x::r) []
      (List.fold_left (fun r x -> (f x)::r) [] xs)
```

Itération non bornée Parmi les fonctionnelles fournies par la bibliothèque standard, il y a celle-ci :

`find : ('a -> bool) -> 'a list -> 'a` telle que `(find p xs)` donne le premier élément `x`de `xs` pour lequel `(p x)` prend la valeur `true`, s'il existe, et déclenche l'exception `Not_found`, sinon.

On pourrait définir cette fonction en utilisant l'itérateur `List.filter` : en effet, le premier élément de `xs` pour lequel `p` prend la valeur `true` et le premier élément du résultat de `(List.filter p xs)`, si ce résultat n'est pas la liste vide. On aurait :

```
let find (p : 'a -> bool) (xs : 'a list) : 'a =
  match (List.filter p xs) with
    [] -> raise Not_found
  | x::_ -> x
```

Notez comment ici, on analyse le résultat de l'application d'une fonction et non plus simplement la forme d'un argument.

Toutefois, cette manière de faire n'est pas très optimale. En effet, `List.filter` explore toute la liste pour construire un résultat dont on ne conserve que le premier élément. Il eût été plus optimal d'arrêter la recherche dès que le premier `x` de `xs` pour lequel `p` prend la valeur `true` a été trouvé. Ce qui donne la définition :

```
let rec find (p : 'a -> bool) (xs : 'a list) : 'a =
  match xs with
    [] -> raise Not_found
  | x::xs -> if (p x) then x else (find p xs)
```

Il existe deux *conditions d'arrêt* pour cette fonction :

1. La liste est vide, au quel cas la recherche a échoué.
2. La fonction `p` prend la valeur `true` pour le premier élément de la liste, auquel cas, ce premeir élément est le résultat recherché.

De surcroît, cette définition est maintenant récursive terminale.

Les itérateurs `List.map` et `List.filter` implémentent des *itération bornées* sur les listes. Le nombre d'étapes d'itérations (*i.e.* le nombre d'appels récursifs) de l'application de ces itérateurs est égale à la taille des listes traitées.

En revanche, les fonctions de recherche d'une valeur dans une structure sont des exemples d'itérations *non bornées*. C'est-à-dire, d'itérations qui n'ont pas nécessairement besoin d'explorer la totalité des structures traitées pour produire un résultat.

3 Structures arborescentes

L'ensemble des listes est défini à l'aide de deux *constructeurs* : la constante `[]` pour la liste vide et l'opérateur `::` pour une liste non vide. Les listes sont des *structures récursives* : le second argument du constructuer de liste `::` est lui-même une liste. Cette manière d'envisager les structures de données permet d'en définir de plus complexes.

3.1 Arbres binaires

En calquant la définition récursive des listes, on peut définir les structures d'arbres binaires étiquetés de la manière suivante :

— on se donne un arbre vide ;
— on construit un arbre en ajoutant un premier élément (la *racine*) à deux arbres.

Pour appliquer ce principe dans le langage, on se donne deux symboles :

— `Empty`, constante pour désigner l'arbre vide ;
— `Node`, opérateur ternaire pour désigner l'opération d'ajout d'une racine pour connecter deux arbres.

3.2 Définition de type

Se donner ces symboles revient à *définir un nouveau type* dans le langage. La clause de définition d'une type est `type`. On écrit :

```
type 'a btree =
  Empty
| Node of 'a * ('a btree) * ('a btree)
```

Cette définition déclare le nom du nouveau type (`btree`). Comme les listes, il s'agit d'un type paramétré (variable de type `'a`). Le constructeur `Node` réclame trois arguments : l'élément placé à la racine, de type `'a` et les deux arbres réunis dans la structure construite, donc de type `'a btree`. Le mot clé `of` indique, si nécessaire le type des argument du constructeur. S'il y en a plusieurs, leur types sont séparés par une étoile (caractère `*`).

Le type ainsi définit appartient à la catégorie des *types somme*[2]. Nous y reviendrons en **??**.

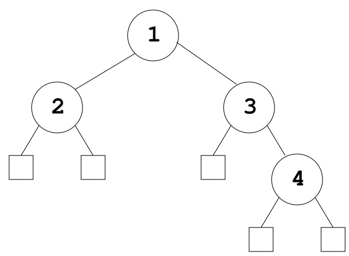
Toute valeur appartenant au type `'a btree` est égale soit à `Empty` soit à la valeur d'une expression de la forme `Node(x,bt1,bt2)` où `bt1` et `bt2` sont des expressions de type `'a btree`.

Notez : la manière dont les arguments d'un constructeur sont réunis dans un *n-uplet* (ici, un triplet) placé entre parenthèses.

Par exemple l'expression

```
Node(1,
     Node(2, Empty, Empty),
     Node(3,
          Empty,
          Node(4, Empty, Empty)))
```

a pour la valeur l'arbre (de type `int btree`) que l'on peut dessiner ainsi :



Les carrés représentent les (sous)arbres vides.

3.3 Filtrage et récurrence

Le principe de programmation (récursive) avec les arbres ressemble à celui que l'on a utilisé pour les listes. Il repose sur l'analyse des formes possible d'un arbre à l'aide le la structure de contrôle `match-with`.

Illustrons le avec la définition de la fonction `btree_mem`, de type `'a -> 'a btree -> bool` telle que `(btree_mem x bt)` vaut `true` si `x` apparaît dans l'arbre `bt` et `false` sinon. C'est l'analogue de la fonction `List.mem` pour les listes. Les propriétés de `btree_mem` seront donc également similaires : un arbre binaire est comme une liste, sauf qu'un arbre binaire non vide a deux suites, là où une liste non vide n'en a qu'une.

Par analogie avec les listes, en suivant une analyse par acs de constructeur des arbres, on peut donc dire
- — que `(btree_mem x Empty)` vaut `false`, pour tout `x`
- — que `x` apparaît dans `Node(y,bt1,bt2)` si et seulement si `x` est à la racine de `Node(y,bt1,bt2)` (c'est-à-dire, `x`=`y`), ou `x` apparaît dans `bt1` ou `bt2`.

La fonction `btree_mem` doit donc satisfaire les équations suivantes :

$$\begin{cases} \texttt{(btree\_mem Empty)} & = \texttt{false} \\ \texttt{(btree\_mem x (Node(y,bt1,bt2)))} & = \texttt{(x=y) || (btree\_mem x bt1) || (btree\_mem x bt2)}. \end{cases}$$

D'où la définition :

2. La terminologie est assez fluctuante pour ces types. On les nomme égaement *type union (étiquetée)*, *type algébrique* et dans la communauté ML, on utilise le terme *type variants*.

```
let rec btree_mem (x:'a) (bt:'a btree) : bool =
  match bt with
    Empty -> false
  | Node(y, bt1, bt2) -> (x=y) || (btree_mem x bt1) || (btree_mem x bt2)
```

3.4 Récurrence terminale

Considérons la fonction `btree_size` qui donne la taille d'un arbre (son nombre d'éléments). C'est ici l'analogue de la fonction `List.length`. Par analogie avec les propriétés de `List.length`, on pose :

$$\begin{cases} \texttt{size Empty} & = & \texttt{0} \\ \texttt{(size (Node(x,bt1,bt2)))} & = & \texttt{1 + (size bt1) + (size bt2)} \end{cases}$$

Ce qui donne la définition :

```
let rec size (bt:'a btree) : int =
  match bt with
    Empty -> 0
  | Node(_, bt1, b2) -> 1 + (size bt1) + (size bt2)
```

Si l'on veut obtenir une définition récursive terminale de cette fonction, on peut tenter de s'inspirer de la définition récursive terminale de `List.length` en introduisant une fonction récursive locale `loop` munie d'une *accumulateur*. Mais la chose est ici plus compliquée. En effet, pour calculer la taille de `Node(x,bt1,bt2)`, il faut calculer la taille de `bt1` et la taille de `bt2`. Il faut donc nécessairement procéder à deux appels récursifs. Ce qui est contradictoire avec la notion de récurrence terminale.

On peut toutefois obtenir quelque chose en analysant comment une expression telle que `1 + (size bt1) + (size bt2)` est évaluée. La valeur de cette expression ne sera obtenue que lorsque `(size bt1)` et `(size bt2)` auront été «résolues». Pour ce, le mécanisme d'évaluation commencera par l'évaluation de `(size bt1)` [3] ce qui provoque la «mise en attente» de l'évaluation de l'addition (`1 + ..`) et de `(size bt2)`. On avait vu comment supprimer la mise en attente de l'addition en ajoutant un accumulateur. Pour supprimer la mise en attente de `(size bt2)`, il suffit de se souvenir de `bt2` afin de le traiter à son tour lorsque `bt1` aura été traité. Et pour cela, il suffit d'introduire un nouvel accumulateur qui contient une liste d'arbres à traiter. Le calul se tremine lorsque l'arbre à traiter est vide et la liste d'arbres en attente égalenet. Cette idée donnera :

```
let size (bt:'a btree) : int =
  let rec loop (bt:'a btree) (bts:('a btree) list) (r:int) =
    match bt with
      Empty -> (
        match bts with
          [] -> r
        | bt::bts -> (loop bt bts r)
      )
      | Node(_,bt1,bt2) -> (loop bt1 (bt2::bts) (r+1))
  in
    (loop bt [] 0)
```

Dans la fonction auxiliaire `loop` le premier argument (`bt`) est tout simplement le premier arbre à traiter. On peut faire l'économie de cet argument en l'intégrant dans la liste des arbres à traiter. la fonction est alors définie par récurrence sur la liste d'arbres à traiter, et par analyse, par filtrage, du premier élément de la liste. On obtient alors la définition suivante :

3. Ce choix est arbitraire, mais faire l'autre choix ne change rien à l'argument.

```
let size (bt:'a btree) : int =
  let rec loop (bts:('a btree) list) (r:int) =
    match bts with
      [] -> r
    | (Empty::bts) -> (loop bts r)
    | ((Node(x,bt1,bt2))::bts) -> (loop (bt1::bt2::bts) (r+1))
  in
    (loop [bt] 0)
```

Notez le filtrage «en profondeur» : sur la liste et sur son premier élément.

Ce que nous avons réalisé avec cette dernière version de `size` n'est rien d'autre que la gestion explicite dans notre fonction du mécanisme d'empilement induit par l'évaluation d'une fonction récursive, non terminale. Le gain, en terme de consomation mémoire, est assez faible, puisque nous construisons une liste là où le mécanisme d'évaluation utilise la pile des appels de fonctions.

Recherche dans un arbre On veut rechercher dans un arbre binaire une valeur satisfaisant une certaine propriétés (exprimée par une fonction booléenne). On réalise cette recherche avec la fonction de signature `search (p:'a -> bool) (bt:'a btree) : 'a`.

L'idée générale du processus de recherche est le suivant :
- si l'arbre est vide, la recherche échoue
- sinon,l 'arbre a la forme `Node(x, bt1, bt2)` :
  - si `(p x)`, on a trouvé ; la valeur de la fonction est `x`
  - sinon, on cherche dans `bt1` ; puis, si besoin, dans `bt2`

C'est l'idée générale de `btree_mem`, à cette différence que on ne peut ici combiner le résultat avec une disjonction.

On peut envisager deux solution pour «combiner» la séquence : ou bien c'est `x`, ou bien c'est dans txtbt1 ou bien c'est dans `bt2`. La première consiste à utiliser le type `option` pour marquer que l'on a trouvé ou non ; la deuxième consiste à utiliser ce que nous avons fait pour obtenir une définition récursive de `size` : une liste des arbres mis en attente.

Utilisation du type `option`

```
let search (p:'a -> bool) (bt:'a btree) : 'a =
  let rec loop (bt:'a btree) : 'a option =
     match bt with
       Empty -> None
     | Node(x, bt1, bt2 -> (
         if (p x) -> (Some x)
         else (
           match (loop bt1) with
             Some x -> x
           | None -> (loop bt2)
         )
   )
 in
   match (loop bt) with
     None -> raise Not_found
   | Some x -> x
```

Notez l'utilisation du `match` comme une alternative : si on n'a pas trouvé dans `bt1`, on cherche dans `bt2`.

Utilisation d'une liste d'attente

```
let search (p:'a -> bool) (bt:'a btree) : 'a =
  let rec loop (bts:'a btree list) =
     match bts with
       [] -> raise Not_found
     | (Empty::bts) -> (loop bts)
     | (Node(x, bt1, bt2))::bts ->
        if (p x) then x
        else (loop (bt1::bt2::bts))
  in
    (loop [bt])
```

Nous donnons, au paragraphe suivant une troisième manière de procéder qui exploite la fait qu'en cas d'échec, on déclenche une exception.

3.5 Exception et contrôle

Le mécanisme des exceptions permet de les «déclencher» avec la primitive `raise`. Il offre également la possibilité de les intercepter avec la construction `try ... with ...`

L'évaluation de l'expression `1/0` provoque le déclenchement de l'exception `Division_by_zero`. Le programme dans lequel cette expression aura été évaluée sera stopé à moins qu'un traitement pour cette exception n'y ait été prévu. Pour traiter une exception, on utilise la constrcution `try-with`. Par exemple `try (1/0) with Division_by_zero -> Printf.eprint "Divison par zéro"`. Dans ce cas, le programme n'est pas interrompu, et un message est affiché sur la sortie en erreur (`stderr`).

On peut utiliser le mécanisme des exceptions à la manière dont on utilise une expression alternative pour contrôler le flot d'évaluation d'un programme. Par exemple, on cherche dans un arbre binaire un élément satisfait une certaine propriété (exprimée comme une fonction booléenne). Le schéma de ce processus est proche de celui de la fonction `btree_mem` à ceci près que la fonction de recherche peut échouer (fonction partielle). Voici comment on peut décrire ce schéma :

- si l'arbre est vide, la recherche échoue ;
- si l'arbre a la forme `Node(x,bt1,bt2)` :
  - si `x` satisfait la propriété, la recherche réussit ;
  - sinon, essayer de cherche dans `bt1` :
    - si cela réussit, on a terminé ;
    - sinon, chercher dans `bt2`.

Ce qui se transcrit comme la définition de la fonction suivante qui donne un élément de l'arbre en cas de réussite :

```
let rec search (p:'a -> bool) (bt:'a btree) : 'a =
  match bt with
    Empty -> raise Not_found
  | Node(x,bt1,bt2) ->
      if (p x) then x
      else (
        try (search bt1)
        with Not_found -> (search bt2)
      )
```

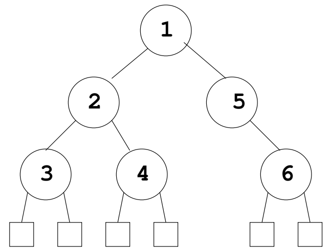
Avec cette fonction, on peut donner une autre définition de la fonction `btree_mem`. Par exemple :

```
let btree_mem (x:'a) (bt:'a btree) : bool =
```

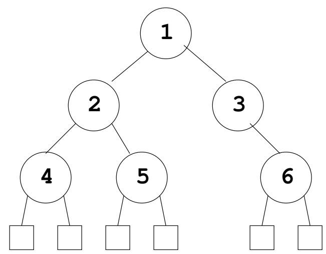
```
try (x = search (fun y -> x=y) bt)
with Not_found -> false
```

3.6 Parcours en largeur

La stratégie de recherche implémentée dans la fonction `search` ci-dessus est dite *en profondeur*. La figure ci-dessous illustre la succession selon laquelle les éléments de l'arbre sont examinés :



Les noeuds de l'arbre sont numérotés dans l'ordre dans lequel ils sont examinés. Il est parfois utile d'examiner les éléments dans l'ordre suivant :



Une telle manière de parcourir la structure arborescente est dite *en largeur*.

On ne peut réaliser cela directement avec une définition récursive reposant sur la structure des arbres. Il faut, à l'instar de ce qui a été réalisé pour une version récursive terminale de `size`, gérer explicitement l'ensemble des sous-arbres restants à traiter. Mais, pour obtenir un parcours en largeur, l'ordre dans lequel les (sous)arbres à traiter sont mémorisé n'est pas celui des listes (qui implante en fait un *mécanisme de pile* : le dernier élément ajouté est le premier accessible). Il faut utiliser une structure de *file d'attente* : le premier élément ajouté est le premier accessible.

Les files d'attentes quoique cela soit très inefficace, on peut utiliser une structure de liste pour réaliser la gestion de files d'attente. On définit alors les fonctions de base des gestion des files d'attente :

- reconnaître qu'une file est vide :

```
let is_empty (xs:'a list) : bool =
  match xs with
    [] -> true
  | _ -> false
```

- ajouter un élément dans un file d'attente

```
let add (x:'a) (xs:'a list) : 'a list =
  xs@[x]
```

- obtenir le premier élément de la file d'attente (si il existe)

```
let top (xs:'a list) : 'a =
  match xs with
    x::_ -> x
  | _ -> raise Not_found
```

- supprimer le premier élément de la file d'attente

```
let pop (xs:'a list) : 'a list =
  match xs with
    _::xs -> xs
  | _ -> xs
```

Munis de ces utilitaires, on définit ainsi un parcours en largeur d'un arbre binaire

```
let bfsearch (p:'a -> bool) (bt:'a btree) : 'a =
  let rec loop (bts:('a btree) list) =
    if (is_empty bts) then
      raise Not_found
    else (
        match (top bts) with
          Empty -> (loop (pop bts))
        | Node(x,bt1,bt2) -> (
            if (p x) then x
            else (loop (add bt2 (add bt1 (pop bts))))
        )
    )
  in
    (loop (add bt []))
```

Le module `Queue` il existe une réalisation bien meilleure des files d'attentes dans le module `Queue` de la bibliothèque standard. Le type de données des files d'attentes défini par ce module est `'a Queue.t`.

Toutefois les fonctions fournies par ce module ne sont pas... fonctionnelles. Par exemple, la fonction d'ajout d'un élément dans une file d'attente a pour signature `Queue.add: 'a -> 'a Queue.t -> unit`. La valeur de retour de cette «fonction» n'est pas une file d'attente mais la valeur de type `unit` que l'on note `()`. Elle procède à l'ajout par *effet de bord*, par *modification en place* de la structure. On a donc affaire à une «fonction» qui fait usage des traits impératifs des langages de programmation comme l'affectation. On avait usage d'appeler de telles fonctions des *procédures*[4]. Dans la même veine, `Queue.top`, de type `'a Queue.t -> 'a` supprime l'élément en tête de la file **et** délivre également la valeur de cet élément.

Le caractère impératif de `Queue.add` nous interdit l'usage d'expressions telles que `(loop (Queue.add bt2 (Queue.add bt1 (pop bts))))`. Il faut remplacer la *composition* fonctionnelle d'expressions par leur mise en *séquence*. Là où l'on a écrit `(loop (add bt2 (add bt1 bts)))`, on écrit `(Queue.add bt1 bts); (Queue.add bt2 bts); (loop bts)`.

Avec l'utilisation des files d'attentes du module `Queue`, la fonction de recherche devient

```
let bfsearch (p:'a -> bool) (bt:'a btree) : 'a =
  let rec loop (bts: ('a btree) Queue.t) =
    if (Queue.is_empty bts) then
      raise Not_found
    else (
      match (Queue.pop bts) with
```

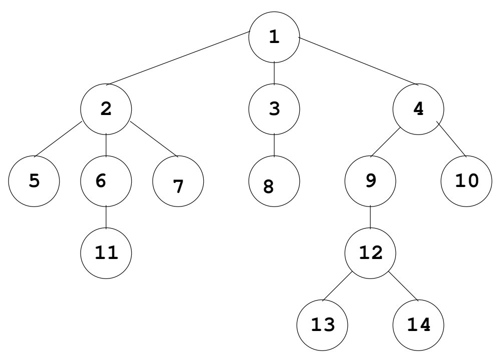
4. D'où le titre de ce cours :)

```
          Empty -> (loop bts)
        | Node(x,bt1,bt2) -> (
            if (p x) then x
            else (
              Queue.add bt1 bts;
              Queue.add bt2 bts;
              loop bts)
            )
    )
  in
  let bts = Queue.create () in
    Queue.add bt bts;
    (loop bts)
```

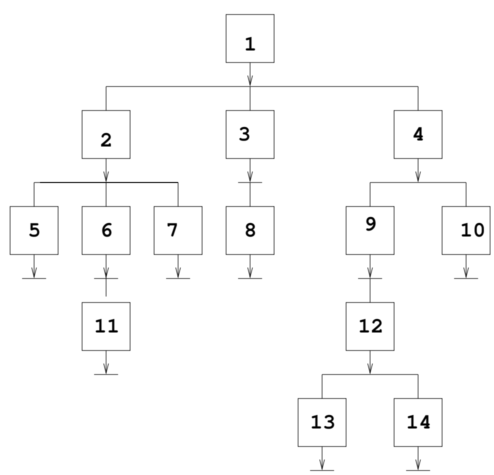
Par rapport à sa version purement fonctionnelle, on perd ici la *transparence référentielle* du code : dans la définition de `loop`, l'identificateur `bts` ne désigne pas toujours la même valeur.

3.7 Arbres généraux

Structure arborescente dont le nombre de branchements est variable



Autre représentation graphique



Dans cette dernière, chaque nœud est constitué d'un nombre fixé de composants :

1. L'étiquette (les carrés).

2. La suite des sous-arbres (les barres horizontales).

Pour réaliser ces suites, de taille variable, possiblement vides, on utilise la structure dynamique de listes. D'où la définition :

```
type 'a gtree =
  GEmpty
| GNode of 'a * ('a gtree) list
```

L'arbre donné en exemple est représenté par d'expression

```
GNode(1,
  [ GNode(2, [ GNode(5, []);
               GNode(6, [ GNode(11, [])]);
               GNode(7, []) ]);
    GNode(3, [ GNode(8, []) ] );
    GNode(4, [ GNode(9, [ GNode(12, [ GNode(13, []);
                                      GNode(14, []) ]) ]);
               GNode(10, []) ])
  ])
```

Notez que le constructeur `Empty` est ici inutile. En fait, ce constructeur est un peu parasite car on assimile les expressions `GNode(x,[])` et `GNode(x, [Empty])`. Il ne sert réellement que si l'arbre entier est vide. D'ailleurs, la littérature ignore en général l'arbre vide, dans le cas des arbres généraux.

La liste des sous-arbres attachée à un nœud est appelée *forêt*. Ainsi une arbre général est :
- soit vide ;
- soit composé d'une étiquette et d'une forêt ;

et une forêt est ;
- soit vide ;
- soit composée d'un arbre et d'une forêt.

Les arbres et les forêt sont ainsi *mutuellement définis*, de manière récursive. En conséquence, un schéma simple de traitement récursif des arbres généraux est le schéma de *récurrence mutuelle*. Par exemple la fonction de test d'appartenance d'un élément à un arbre est réalisée par une fonction sur les arbres associée à une fonction sur les forêts ; ces deux fonctions s'appelant l'une, l'autre :

```
let rec gtree_mem (z:'a) (gt : 'a gtree) : bool =
  match gt with
      GEmpty -> false
    | GNode(x, gts) -> (z=x) || (forest_mem z gts)
and forest_mem z (gts:('a gtree) list) : bool =
  match gts with
      [] -> []
    | gt::gts -> (gtree_mem z gt) || (forest_mem z gts)
```

La récurrence mutuelle est réalisée dans le langage avec les mots clé `let rec` (pour l'aspect «récursif») et `and` (pour l'aspect «mutuel»).

Itérateurs Ici, on peut tirer profit du fait que la *forêt* est une valeur de type `list` et utiliser des itérateurs prédéfinis sur les listes. En effet, le test d'appatenance, dans le cas d'un arbre non vide – de la forme `GNode(x, gts)` peut aussi s'exprimer comme : ou bine `x=z` ou bien *il existe* un arbre dans `gts` pour lequel `gtree_mem` est vérifié pour `z`. Un itérateur sur les listes nous donne le *il existe* : `List.exists : ('a -> bool) -> 'a list -> bool`. On peut alors se dispenser de la récurrence mutuelle :

```
let rec gtree_mem (z:'a) (gt : 'a gtree) : bool =
```

```
match gt with
  GEmpty -> false
| GNode(x, gts) -> (x=z) || (List.exists (gtree_mem z) gts)
```