Chapitre 1

La Récursivité

Introduction

La définition d'un objet est dite **récursive** si elle fait référence à l'objet lui-même. Voici quelques objets dont la définition est récursive :

- Les **suites récurrentes** en mathématiques (Fibonacci...)
- L'**ordre lexicographique** (celui du dictionnaire). Il est défini en posant qu'un mot a est plus petit qu'un mot b (c'est-à-dire placé avant dans le dictionnaire) si a est vide (c'est-à-dire formé d'aucun caractère), si le premier caractère de a précède le premier caractère de b dans l'alphabet ou si le mot formé des autres caractères de a est plus petit que le mot formé des autres caractères de b.
- La **description syntaxique d'un langage** par une grammaire. Par exemple pour décrire les identificateurs autorisés dans le langage Caml, on dira qu'un identificateur est : soit une lettre, soit un identificateur suivi d'une lettre, d'un chiffre ou du caractère souligne.
  Cela peut s'exprimer par une notation dite BNF (Bachus Naur Form., du nom de ses inventeurs).

  ```
  <identificateur> ::= <lettre> |
                   <identificateur>[ <lettre> | <chiffre>  | _]
  ```

  De même pour décrire un langage de manipulation d'expressions arithmétiques, on dira qu'une expression arithmétique est : soit un nombre, soit la somme, le produit ou la négation d'une expression: ce qui en BNF s'exprime :

  ```
  <exp> ::= <exp> + <exp> | <exp> * <exp> | - <exp> | <nombre>
  <nombre> ::= <chiffre> <nombre>
  <chiffre> ::= 0...9
  ```

On retrouve également la notion de récursivité dans de très nombreux domaines, des tableaux d'Escher aux dessins des boites de "Vache qui rie" :-) en passant par les fractals ou les fugues de Bach.

En Caml, la récursivité intervient dans la définition de **type récursifs** et de **fonctions récursives**. Lorsque l'on définit de tels objets, on est confronté à un certain nombre de problèmes : Dans le cas des types récursifs, il faut être sûr de pourvoir créer des objets de ce type par la suite et donc être très prudent avec les déclarations de la forme :

```
type infini= Infini of infini;;
Type infini defined.
let rec t = Infini t;;
```

Dans le cas des fonctions récursives, il faut éviter d'entrer dans des boucles infinies comme dans l'exemple suivant :

```
let rec f x = f x;;
f : 'a -> 'b = <fun>
f 2;; (* boucle infinie *)
```

Nous introduisons maintenant un peu de terminologie pour étudier plus formellement les problèmes liés à la récursivité.

1.1 Principes de récurrence sur N

On rappelle les principes de récurrence sur les entiers naturels.

Théorème 1. *Soit P un prédicat sur N. Si*

- $P(0)$ *est vrai*
- $\forall n \in N, P(n) \Longrightarrow P(n+1)$,
  alors pour tout $n \in N$, $P(n)$ est vrai.

Preuve. Elle s'appuie sur le fait que **toute partie non vide de N admet un plus petit élément**. Supposons que :

- $P(0)$ est vrai
- $P(n) \Longrightarrow P(n+1)$

Supposons alors qu'il existe un entier n_0 telque $P(n_0)$ ne soit pas vérifiée. Soit alors n_1 le plus petit des entiers pour lequel $P(n)$ est faux ($n_1 \neq 0$). Alors $P(n_1 - 1)$ est vrai et donc $P(n_1)$ serait également vrai.

Théorème 2. *Soit P un prédicat sur N. Si*

- $P(0)$ *est vrai,*
- $\forall n \in N, (\forall k < n, P(k)$ *est vrai*$) \Longrightarrow P(n)$,
  alors pour tout $n \in N$, $P(n)$ est vrai

Preuve. Il suffit de considérer le prédicat $Q(n) = P(0) \wedge P(1) \wedge \ldots \wedge P(n-1)$ et d'appliquer le théorème 1

1.2 Utilisation de la récursivité sur N

L'idée de base est de considérer un prédicat de la forme :

P(n)=" Nous savons résoudre le problème pour `n` ".

Puis d'utiliser le principe de récurrence avec ce prédicat. Si on sait résoudre le problème pour 0 et que sachant le problème résolu pour k nous savons le résoudre pour $k+1$ alors nous savons le résoudre pour toute valeur de n. En fait la preuve de ce prédicat est un moyen de construire une solution du problème donc un programme. Voyons quelques exemples.

Définition de la fonction puissance

Notre problème est de définir la fonction qui, étant donnés deux entiers a et b calcule a^b. On considère donc le prédicat

P(n)=" Nous savons calculer a^{n} ".

Prouvons ce prédicat par récurrence sur n :

- P(0) est vrai, nous savons calcule a^0, cela fait 1 (1 est une preuve de $P(O)$).
- Supposons que $P(n-1)$ soit vrai donc que nous sachions calculer a^n. Alors $P(n)$ est vrai car nous savons calculer a^n, il suffit de multiplier a^n par a (la fonction qui à $a^{(n-1)}$ associe $a^{(n-1)} * a$ $(= a^n)$ est une preuve de $P(n-1) \Longleftarrow P(n)$).

Par conséquent pour tout n, $P(n)$ est vrai, nous savons calculer a^n. La preuve de $\forall n, P(n)$ donne un moyen effectif d'associer à un n donné une preuve de $P(n)$ donc d'associer à n la valeur a^n. c'est donc une fonction qui à n associe a^n.

Cette fonction se traduit en Caml par :

```
let rec puis a n =
       if b=0
        then 1                    (* preuve cas de base *)
       else  a * (puis a (n-1));; (* preuve de l'étape de récurrence *)
```

La présence **obligatoire** du mot clef `rec` dans les définitions récursives sera expliquée au chapitre 4.2.

Le problème des tours de Hanoi

On dispose de trois tours que nous appellerons `Dep`, `Int` et `Arr`. Sur la tour `Dep` sont empilés `n` disques de tailles décroissantes. Il s'agit de transférer ces disques sur la tour `Arr` en respectant les règles suivantes :

1. On ne déplace qu'un disque à la fois.
2. Un disque ne peut être empilé que sur un disque plus grand ou sur une tour vide.

La position de départ des tours de Hanoi avec 3 disques est la suivante:

```
        |                    |                  |
      --|--                  |                  |
     ---|---                 |                  |
   -----|-----               |                  |
--------|--------    --------|--------  --------|--------
      Dep                   Int                Arr
```

Lorsque l'on a déplacé `n-1` disques sur la tour intermédiaire, on peut déplacer le plus grand disque de `Dep` vers `Arr` On retrouve le problème avec `n-1` disques, `Int` comme tour de départ et `Dep` comme tour d'arrivée.

Avec 3 disques après avoir déplacé les deux plus petits disques sur la tour centrale on a la position suivante :

```
        |                    |                  |
        |                    |                  |
        |                  --|--                |
   -----|-----            --- | ---             |
--------|--------    --------|--------  --------|--------
      Dep                   Int                Arr
```

Nous pouvons déplacer le disque restant sur la tour `Arr` et obtenir la position suivante :

```
        |                    |                  |
        |                    |                  |
        |                  --|--                |
        |                 --- | ---        -----|-----
--------|--------    --------|--------  --------|--------
      Dep                   Int                Arr
```

Il reste alors à déplacer deux disques de la tour `Int` vers la tour `Arr` en utilisant `Dep` comme pivot.

Comme les tours `Dep`, `Int` et `Arr` sont interchangeable, on peut considérer le prédicat suivant **P(n)=" Nous savons déplacer n disques d'une tour vers une autre en respectant les règles ".** on se trouve dans la situation suivante :

- $P(0)$ est vrai. (I n'y a rien à faire, super non ?)
- si $P(k-1)$ est vrai alors nous savons montrer $P(k+1)$. (On déplace les $(k-1)$ premiers disques vers `Int` ce que l'on sait faire par hypothèse, on déplace le disque restant vers `Arr` ce que l'on sait faire par hypothèse, enfin on re-déplace les $(k-1)$ premiers disques de `Int` vers `Arr`.)

On peut donc exprimer cela en Caml par la fonction `Hanoi` ci-dessous :

```
let mouvement de vers =
 print_string(de^"-->"^vers);print_newline();;

let rec hanoi dep int arr n = match n with
0 -> () (* preuve du cas de base (rien a faire) *)
|n-> hanoi dep arr int (n-1); (* deplacer les n-1 premiers au milieu *)
     mouvement dep arr;       (* deplacer le disque restant *)
     hanoi int dep arr (n-1);; (* deplacer les n-1 disques poses au milieu
                                  vers la tour de droite *)
```

Donc l'éxécution produit :

```
#hanoi "A" "B" "C" 3;;
A-->C
A-->B
C-->B
```

```
A-->C
B-->A
B-->C
A-->C
- : unit = ()
```

1.3 Complexité en temps des fonctions récursives

Les problèmes de "complexité" seront abordés en détail en cours d'algorithmique néanmoins dès que l'on écrit des programmes, il est bon de pouvoir estimer leur efficacité et prévoir leur comportement. Par exemple si nous écrivons un algorithme de tri et qu'il trie une liste de 10 éléments en 1 seconde (c'est très très long !), il est bon de pouvoir dire si il triera une liste de 100 éléments en 2, 10, 100 ou 22026 secondes.

Définition 1. *La notation $f = O(g)$ exprime qu'il existe k tel que $f \leq k \times g$. Grossièrement, cela signifie que f est proportionnelle à g dans le pire des cas.*

Exemple 1. Calculons la complexité de notre fonction `puissance`. Soit TC le temps pour exécuter le if, et TM le temps pour la multiplication. On note U_n le temps pour exécuter (`puis x n`).

$$U_0 = TC$$

$$U_n = TC + TM + U_{n-1} = (n+1) * TC + n * TM$$

On dit que **l'algorithme est en $\mathbf{O(n)}$**.

Pour des raisons d'efficacité, on cherche évidement à produire des programmes de complexité minimale. Dans le cas de la fonction puissance il est possible d'écrire une version plus efficace en raisonnant par dichotomie:

Exemple 2 (Une version plus efficace de la fonction puissance). On sépare les cas ou `b` est paire (even) des cas ou `b` est impaire.

```
let rec puisDicho a b =
       if b=0 then 1
       else if (even b) then carre  (puisDicho a (b /2))
       else  a * carre (puisDicho a (b/2));;
trace "puisDicho";;
```

On peut monter que **cet algorithme est en $\mathbf{O(\log n)}$** (exercice :-)).

1.4 Récursion profonde et récursion terminale

Considérons l'archétype des fonctions récursives, la fonction factorielle.

Une récurrence évidente permet d'écrire :

```
let rec fact n = if n=0
            then 1               (* cas de base *)
             else n*fact (n-1);; (* cas de récurrence *)
```

Détaillons l'évaluation de `fact 3`

```
fact 3
 -> if n=0  then 1 else  n*fact (n-1)                       avec la liaison n=3
 ->  n*fact (n-1)                                           avec la liaison n=3
    -> 3*fact 2
    ->3 * ( if n=0  then 1 else  n*fact (n-1))              avec la liaison n=2
    ->3 * (  n*fact (n-1))                                  avec la liaison n=2
       ->3 * ( 2 * (fact 1))
       ->3 * ( 2 * ( if n=0  then 1 else  n*fact (n-1)))    avec la liaison n=1
       ->3 * ( 2 * (  n*fact (n-1) ))                       avec la liaison n=1
           -> 3 * ( 2 * (1 *(fact 0)))
           -> 3 * ( 2 * ( 1 * ( if n=0  then 1 else  n*fact (n-1))))   avec la liaison n=0
           -> 3 * ( 2 * ( 1 * 1))
       <- 3 * ( 2 * ( 1))
    <- 3 *  2
<-6
```

On constate que la multiplication `3*(fact 2)` ne peut être faite avant de connaître le résultat de `(fact 2)` ; puis que pour évaluer `( 2 * (fact 1))` on doit attendre de connaître le résultat de `(fact 1)` ; et enfin que la multiplication (1 *(fact 0)) ne peut être faite qu'après le calcul de `(fact 0)`; on peut alors remonter en cascade (c'est les flèches <-) les multiplications restées en attente. Pour pouvoir effectuer cette remontée, il faut donc avoir stocké les opérations en attente. Ce stockage nécessite l'intervention d'une pile. On remarque par ailleurs que le nombre de résultats à stocker est proportionnel à l'argument `n` donc que plus cet argument est grand plus on doit stocker de résultats sur la pile. Dans certains cas, ce stockage peut saturer la mémoire[1] (vous aurez alors le délicieux message `Uncaught exception: Out_of_memory`).

Ce type de récurrence où l'appelle récursif est enveloppé (c'est-à-dire est un argument d'une opération) et qui nécessite l'utilisation d'une pile s'appelle une récurrence profonde.

Nous allons voir une autre définition récursive de factorielle mais qui n'a pas les inconvénients de la première. Il va s'agir d'accumuler dans un second paramètre les valeurs des appels récursifs de sorte qu'il n'y ait plus de calculs en attente.

On donne ici une petite méthode pour passer d'une version à l'autre[2] :

On regarde la ligne de l'appel récursif et on accepte de perdre de l'information pour généraliser :
`( ? * (fac ?))`
On introduit une fonction auxiliaire :
`(aux p q) = (q * (fac p))`

[1] essayez `fact 1987076;;`
[2] cette méthode sera justifiée dans la section sur les preuves de programme

On cherche une définition intrinsèque de `aux`, c'est-à-dire que l'on élimine `fac` de la définition (dans le cas de base on renvoie toujours l'accumulateur) :

```
(aux 0 acc) == acc
(aux p acc) == (acc*(fac p))
            == (acc*( p*(fac ( p- 1))))
            == (acc* p)* (fac (p- 1))
            == (aux ( p-1) (acc*p))
```

d'où une nouvelle définition de la factorielle[3]

```
let facIt n =
  aux n 1;;
```

```
let rec aux p acc =
  if p=0 then
        acc
        else aux (p-1) ( p* acc);;
```

ou en utilisant une définition locale

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

Détaillons maintenant l'évaluation de `factIt 3`

En résumé :

Existence d'une enveloppe ⇔ "récurrence profonde" ⇔ pile ⇔ Mémoire nécessaire dépendant de l'argument

Pas d'enveloppe ⇔ "récurrence terminale" ⇔ pas besoin de pile ⇔ Mémoire nécessaire indépendante de l'argument

Toute fonction récursive terminale peut s'écrire de façon itérative.

```
let factFor n =
   let result = ref 1 in
   for i=1 to n do result :=i* !result; done;!result;;
```

[3] vous pouvez réessayer `factIt 1987076;;`, vous ne saturerez plus la pile (mais le résultat sera faux car vous aurez dépassé la taille du plus grand entier et même du plus grand float ...)

```
factIt 3
->(aux 3 1)
 -> match p with  0 -> acc |_-> aux (p-1) (acc * p)          avec  p=3 et acc =1
   ->aux 2 3
   ->match p with  0 -> acc |_-> aux (p-1) (acc * p)         avec  p=2 et acc =3
      ->aux 1 6
      ->match p with  0 -> acc |_-> aux (p-1) (acc * p)      avec  p=1 et acc =6
        -> aux 0 6
        ->match p with  0 -> acc |_-> aux (p-1) (acc * p)    avec  p=0 et acc =6
            ->6
```

Ainsi le calcul de `aux 3 1` se ramène à celui de `aux 2 3` qui lui même se ramène à `aux 0 6` qui renvoie `6`. Les résultats intermédiaires sont stockés dans l'accumulateur `acc` et à chaque étape le nouveau résultat intermédiaire vient remplacer l'ancien. A aucun moment on a des calculs en suspens à mémoriser, on n'a donc pas besoin de pile. **La place mémoire nécessaire est** donc **constante et indépendante de l'argument** `n`.
Dans ce second type de récursivité, l'appel récursif n'est pas enveloppé (c'est-à-dire qu'il n'est pas argument d'une opération) , il est "en position terminale".

1.5 Principes d'induction

Le principe d'induction est une généralisation du principe de récurrence. Le principe de récurrence découle du fait que N est défini inductivement. C'est-à-dire défini comme **le plus petit ensemble** vérifiant les deux propriétés suivante :

1. 0 est élément de N.
2. Si x est élément de N alors x+1 est également élément de N.

Ce qui assure l'existence d'un bon ordre et d'un élément minimal 0.

Nous allons maintenant voir que l'on peut par le même procédé définir d'autres ensembles. Et qu'il existe pour ces ensembles des principes similaires au principe d'induction sur les entiers.[4].

Intuitivement pour établir des principes similaires au principe de récurrence sur d'autres ensembles que N, il faut que l'on puisse les munir d'un ordre et qu'ils possèdent un élément minimal pour cet ordre. Cela conduit aux définitions suivantes :

Définition 2. *Soit E un ensemble et $<$ une relation d'ordre sur E.*

- $<$ *est* **un bon ordre** *si* **toute partie non vide a un plus petit élément**
- $<$ *est un ordre* **bien fondé** *si* **toute partie non vide a un élément minimal**
- *L'ordre $<$ sur E est* **bien fondé** *s'il n'existe pas de suite strictement décroissante d'éléments de E.*

[4] Par abus de langage on emploi souvent indifféremment induction et récurrence

Théorème 3 (Principe d induction).
Soit E un ensemble, P un prédicat sur E, et soit x_o le plus petit élément de E si

- *$P(x_o)$ est vrai.*
- *$\forall n \in E, P(x) \Longrightarrow P(n+1)$*
  Alors pour tout $x \in E$, $P(x)$ est vrai

Théorème 4 (Principe d'induction généralisé).
Soit E un ensemble et soit P un prédicat sur E, Si

- *P est vrai pour tous les éléments minimaux de E*
- *$\forall x \in E, (\forall k < x, P(k)$ est vrai$) \Longrightarrow P(x)$*
  Alors pour tout $x \in E$, $P(x)$ est vrai

1.6 Ensembles définis récursivement

Définition 3. *Soit E un ensemble, B une partie de E et K un ensemble d'opérations $\Phi : E^{ar(\Phi)} \to E$. On appelle ensemble X défini récursivement (ou inductivement) par les relation :*

1. *$B \subset X$.*
2. *$\Phi \in K, x_1 \dots x_{ar(\Phi)} \in X \Rightarrow \Phi(x_1, \dots, x_{ar(\Phi)}) \in X$.*

la plus petite partie X de E qui est K stable et contient B.

C'est aussi l'intersection de toutes les parties K-stables[5] contenant B. On dit que X est la **clôture** de B par les opérations de K.

1.6.1 Types inductifs

Dans le monde de la programmation on s'intéresse plutôt à la notion de type qu'à celle d'ensemble. La définition précédente permet de définir des types inductifs[6]. Ils décrivent des objets composés à partir du même type. De telles définitions se traduisent presque immédiatement en Caml. Les éléments des ensemble B et K s'appellent alors des constructeurs de type.

Exemple 3.

1. Les entiers de Peano $O \in N$ et $\forall n \in N, n+1 \in N$

   ```
   type nat = O | S of nat;;
   ```

   `O` et `S` sont les constructeurs du type `nat`.
2. Les listes d'entiers. Une liste est soit vide, soit formée en ajoutant un entier au début d'une liste.

[5] Une partie P d'un ensemble E est K-stable si $\forall x \in P \ \wedge \ \forall \phi \in K \phi(x) \in P$
[6] On dit aussi types récursifs

```
type listeEntier = nil | cons of int*listEntier;;
```

`nil` et `cons` sont les constructeurs du type `listeEntier`

Remarque 1. Dans le cas des listes, le Constructeur `cons` n'est pas à proprement parler un opérateur de type `list`→`list` mais une famille de tels opérateurs indicée par `int`.

1.6.2 Autres exemples de définitions inductives

- Les circuits électriques :
  Un circuit est formé d'une résistance, de deux circuits mis en série ou de deux circuits mis en parallèle. Ce qu'on modélise en Caml par :

  ```
  type circuit = R of float
       | S of circuit*circuit | P of circuit*circuit;;
  ```

- Une généalogie :

  ```
  type personne = {nom:string;prenom:string};;
  type genealogie = Inconu
           | Arbregen of personne*genealogie*genealogie;;
  ```

- Les arbres binaires d'entiers :

  ```
  type  arbre = Arbre_vide | Noeud of arbre * int * arbre;;
  ```

- Une grammaire pour la logique des propositions :
  On modélise les variables par une chaîne de caractères. Une proposition est soit une constante `Vrai` ou `Faux`, soit une variable propositionelle, soit la négation d'une proposition, soit la conjonction de deux propositions, soit la disjonction de deux propositions, soit enfin l'implication.

  ```
  type prop = Vrai | Faux
             |Var of string
             |Non of prop
             |Et of prop*prop
             |Ou of prop*prop
             |Imp of prop*prop;;
  ```

1.6.3 Enregistrements récursifs et autres bizarreries

Il est aussi possible dans une certaine mesure de donner des définitions récursives dont l'utilité est moins évidente comme les enregistrements récursifs, la définition de valeurs qui ne sont pas des types ou des fonctions mais des structures de données. Par exemple :

- Les enregistrements récursifs :

```
type personne=
   {nom:string;prenom:string;père:personne;mere:personne}

let rec creator =
       {nom="ee";prenom="eee";pere=creator;mere=creator};;
```

– Listes infinies...

```
let rec l = 1::l;;
l : int list =
 [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  ...]
#
```

Cela crée une structure en boucle (il y a du pointeur cache...). Comme c'est infini, on ne peut afficher toute la liste d'où les `....`

```
let x = ref 1;;
let rec l = (x:=!x+1); !x::l
let rec l = (last (1::l))::l ;;....
This kind of expression is not allowed in right-hand sides of "let rec".
```

1.6.4 Retour sur le polymorphisme

On a déjà vu des fonctions et des types polymorphes, par exemple :

```
let id x = x;;
id : 'a -> 'a = <fun>

[];;
- : 'a list = []
```

De la même manière, on peut définir des types récursifs polymorphes :

```
type  'a arbre = Arbre_vide | Noeud of 'a arbre * 'a  * 'a arbre;;
```

On vérifie que l'arbre vide est polymorphe :

```
Arbre_vide;;
- : 'a arbre = Arbre_vide
```

Le type des arbres non vides dépend du type associé au paramètre `'a`.

```
 Noeud (Arbre_vide,"Réveillez VOUS BORDEL !",Arbre_vide);;
- : string arbre = Noeud (Arbre_vide, "REVEILLEZ VOUS BORDEL !", Arbre_vide)
```

1.7 Le Principe d'induction structurelle et son utilisation

1.7.1 Le Principe d'induction structurelle

Théorème 5 (Principe d'induction structurelle).
Soient E un ensemble, B une partie de E et K un ensemble d'opérations, X la clôture de B par les opérations de K et soit P un prédicat sur E
Si P est vrai pour tout $x \in B$ et que pour tout $\Phi \in K$ et tout $x_1 \dots x_{ar(\Phi)} \in E$,

$$P(x_1), \dots, P(x_{ar(\Phi)}) \ \Rightarrow P(\Phi(x_1 \dots x_{ar(\Phi)}))$$

Alors $P(x)$ est vrai pour tout $x \in X$.

Preuve. Soit l'ensemble $Y = \{x \in E | P(x) \ est\ vrai\}$, on sait que $B \subset Y$, et que Y est stable car :

$$\Pi \in K, x_1 \dots, x_{ar(\Phi)} \in Y \Rightarrow \Phi(x_1, dots, x_{ar(\Phi)})$$

Donc Y contient X puisque X est l'intersection des parties stables.

1.7.2 Retour sur le filtrage

Le mécanisme de filtrage permet de décomposer un objet pour savoir comment (à partir de quel constructeur) il est construit. Plus généralement il permet de mettre en correspondance un motif avec un terme. C'est-à-dire de trouver quelles valeurs doivent prendre les variables du motif pour qu'il soit égal au terme. (Cela revient donc à trouver une substitution s telque $s(motif) = trerme$). Voici quelques exemples :

1. Un motif simple, les paires :

   ```
   #let terme = (2,"c'est pas ta note !");;
   terme : int * string = 2, "c'est pas ta note !"

   #let (m1,m2)=terme;;
   m1 : int = 2
   m2 : string = "c'est pas ta note !"
   ```

   la variable `m2` du motif à été liée à la valeur du second membre de terme. On peut vérifier :

   ```
   #m2;;
   - : string = "c'est pas ta note !"
   ```

   Il est évident que si le motif est une paire de variables, ce filtrage est toujours possible, il suffit d'associer à chaque variable le membre correspondant dans le terme. Mais que se passe-t-il si on choisit comme motif non plus une paire mais un triplet de variables :

```
#let (x,y,z)=terme;;
Toplevel input:
>let (x,y,z)=terme;;
>           ^^^^^
This expression has type int * string,
but is used with type int * string * 'a.
```

Le compilateur détecte une erreur de typage. On constate donc que le type du terme doit être le même que celui du motif (en fait nous verrons plus loin que celui du motif peut être plus général).
Essayons maintenant de prendre comme motif une paire constituée d'un entier et d'une variable :

```
#let (2,x)=terme;;
Toplevel input:
>let (2,x)=terme;;
>^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
x : string = "c'est pas ta note !"
```

Ici le filtrage est possible mais ne le reste pas si sans changer la forme du motif on modifie la valeur de l'entier comme on le voit ci-dessous :

```
#let (5,x)=terme;;
Toplevel input:
>let (5,x)=terme;;
>^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
Uncaught exception: Match_failure ("", 1176, 1193)
```

Le message `Warning: this matching is not exhaustive` sert à signaler ce type de risque.

2. Motif sur des enregistrements :

```
type prof = {nom:string;prenom:string;numero:int};;
#let terme3 = {nom="Pons";prenom="Olivier";numero=1};;
terme3 : prof = {nom = "Pons"; prenom = "Olivier"; numero = 1}

(*
remarquez que l'ordre des champs est sans importance et
   que vous n'etes pas obliger de les donner tous
*)
#let {numero=n;nom=x}=terme3;;
n : int = 1
x : string = "Pons"
```

3. Motif sur des types Somme :

```
type nombre = Int of int | Float of float;;

#
let (Int x)=terme4;;
Toplevel input:
>let (Int x)=terme4;;
>^^^^^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
x : int = 5

#let (Float y)=terme4;;
Toplevel input:
>let (Float y)=terme4;;
>^^^^^^^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
Uncaught exception: Match_failure ("", 1249, 1271)
```

Remarque 2. Les parenthèses autour du motif sont obligatoires :

```
#let Int x=terme4;;
Toplevel input:
>let Int x=terme4;;
>    ^^^^^^^^^^^^
The constructor Int requires an argument.
```

4. Motif sur les types récursifs :

```
#type 'a arbre = Vide | Noeud of 'a*'a arbre*'a arbre;;
Type arbre defined.
#let term2 = Noeud(3,Noeud(1,Vide,Vide),Noeud(3,Noeud(4,Vide,Vide),Vide));;
term2 : int arbre =
 Noeud (3, Noeud (1, Vide, Vide), Noeud (3, Noeud (4, Vide, Vide), Vide))
```

on essaie de filtrer `terme2` par `Vide`

```
#let Vide=term2;;
Toplevel input:
>let Vide=term2;;
>^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
Uncaught exception: Match_failure ("", 548, 564)
```

Comme `Vide` est une constante, le seul terme qu'elle peut filtrer est la constante `Vide` elle même.

Remarque 3. Attention à ne pas confondre le filtrage tel que nous le voyons et un test d'égalité comme ci dessous :

```
# Vide=term2;;
- : bool = false
```

```
#let (Noeud(v,fg,fd))=term2;;
Toplevel input:
>let (Noeud(v,fg,fd))=term2;;
>^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
v : int = 3
fg : int arbre = Noeud (1, Vide, Vide)
fd : int arbre = Noeud (3, Noeud (4, Vide, Vide), Vide)
#
let (Noeud(v,fg,fd))=term2 in fg;;
Toplevel input:
>let (Noeud(v,fg,fd))=term2 in fg;;
>^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
- : int arbre = Noeud (1, Vide, Vide)
```

Remarque 4. Vous pouvez toujours ne récupérer que l'information qui vous intéresse, en utilisant un `let ...in` :

```
#let (Noeud(v,fg,fd))=term2 in v;;
Toplevel input:
>let (Noeud(v,fg,fd))=term2 in v;;
>^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
- : int = 3
```

Le filtrage apparaît aussi dans la définition de fonction :
Ecrivons une fonction d'impression pour le type `nombre` défini plus haut.

```
let print_nombre = fun
    (Int x)->print_int x;
    |(Float x)->print_float x;;
```

Il est possible de rendre le filtrage plus explicite grâce à la construction `match terme with`. Elle est indispensable si le terme que l'on veut filtrer n'est pas exactement l'argument de la fonction :

```
let add_num n1 n2 = match  (n1,n2) with
  (Int x,Int y) -> Int (x+y)
  |(Int x,Float y)->Float ((float_of_int x)+.y)
 |(Float x,Int y)->Float (x+.(float_of_int y))
 | (Float x,Float y) -> Float (x+.y);;
```

Pour construire ce type de fonction on fait **un raisonnement par cas**, le nombre de cas étant fini. Lorsque le nombre cas n'est pas fini c'est que le type du terme filtré n'est plus un type somme mais un type défini par récurrence. On aura alors recours à un raisonnement par récurrence comme nous allons le voir maintenant :

1.7.3 Fonctions définies par récurrence

Soit `objet` une instance d'un type récursif `T`, pour résoudre un problème sur les objets de ce type il faut considérer le prédicat

P(objet)=" Nous savons résoudre le problème pour objet ".

Puis utiliser le principe d'induction structurelle sur `objet`.

Si on sait résoudre le problème pour tous les constructeurs de base de T et que sachant le problème résolu pour k nous savons le résoudre pour tous les $\phi_i(k)$ où les ϕ_i sont les autres constructeurs de T, alors nous savons le résoudre pour tous objets de type T.

A titre d'exemple définissons une fonction de concaténation sur les listes d'entiers. Nous voulons donc concaténer deux listes, cela se fait par exemple par récurrence sur la première :

- Si elle est vide (Constructeur `Nil`) il n'y a rien à faire
- Supposons que nous sachions concaténer `l1` avec une autre liste `l2` obtenant ainsi `l3`, alors pour concaténer `Cons x l1` avec `l2`, il suffit d'ajouter `x` en tête de la liste `l3`.

Cela ce traduit en Caml par :

```
type listeEntier =Nil | Cons of int*listeEntier;;

let rec concate l l2= match l1 with
  Nil ->l2 (* cas de base *)
 |Cons(x,l1)->x::(concate l1 l2);; (* cas de récurrence*)
```

1.8 Preuves de programme

Le but d'un programmeur est en premier lieu de fournir des programmes corrects, c'est à dire qui réponde bien aux besoins pour lesquels ils ont été conçus. La description de ces besoin s'appelle la spécification et peux être plus ou moins formelle. On dit qu'un programme est correct si il satisfait sa spécification. La satisfaction de quelques tests est une condition nécessaire à la correction d'un programme mais on apprend souvent à ses dépends qu'elle est loin d'être suffisante. (Rappelez vous le crash d'Ariane 5 et imaginez l'accueil fait aux programmeurs responsables du bogue et à ceux qui ont affirmé que le code était correct...)

Remarque 5. Il peut paraître étrange de prouver la correction de fonctions si on les déjà construites en faisant une preuve par récurrence. En fait les deux démarches sont orthogonales. En pratique on ne prend pas toujours le temps de faire les preuves complètes pendant la conception des programmes. D'autre part les preuves de correction à posteriori peuvent êtres faites par d'autres personnes que le programmeur, on parle alors de vérification de programme. Cette activité se fait rarement à la main mais en utilisant des logiciels spécialisés pour la vérification de programme. La spécification, et le programme étant alors exprimés dans des formalismes logiques. Ces logiciel permette aussi parfois de partir d'un spécification, et de prouver qu'il existe un programme qui la vérifie. On peut alors extraire de la preuve le programme en question.

1.8.1 Preuves de Terminaison de fonction

Une première chose est d'être sûr que les fonctions que l'on écrit ne vont pas produire des boucles infinies. Pour cela on va faire une preuve de terminaison. Si une fonction termine pour tout les éléments x d'un ensemble E on dit que c'est une fonction totale.

La méthode de preuve consiste à prouver que les appels récursifs se font sur des arguments "plus petits" que l'argument de la fonction. Pour cela il faut définir "une mesure" sur le domaine de l'argument. Pour pouvoir raisonner par récurrence, cette "mesure" doit être un ordre bien fondé.

Un cas simple: la somme des `n` premiers entiers

```
let rec somme n =
      if n=0 then 0
             else n+(somme (n-1));;
```

On raisonne par récurrence sur N :

- Si $n = O$ alors `somme n=0` donc termine.
- Supposons que `somme (n-1)` termine. Alors `somme n =n+(somme (n-1))` termine.
  Donc pour tout `n`, `somme n` termine.

Un cas plus compliqué, la fonction d'Ackerman

```
let rec ackerman x= match x with
      (0,m)->m+1  (* 1 *)
      |(n,0)->ackerman ((n-1),1)  (* 2 *)
      |(n,m)->ackerman ((n-1),(ackerman (n,(m-1)))) (* 3 *);;

ackerman : int * int -> int = <fun>
ackerman (5,3);; (* c'est long mais si on sait que ca va terminer *)
                 (*...on peut attendre *)
```

On munit N^2 de l'ordre lexicographique suivant :

$$(n, m) \preceq (n_1, m_1) \Leftrightarrow (n < n_1 \vee (n = n_1 \wedge m \leq m_1))$$

Vérifions qu'il s'agit d'un bon ordre : On vérifie que toute une partie non vide A de N^2, a un plus petit élément (n_0, m_0). Il suffit de considérer :

$n_0 = min\{n | \exists m \in N, (n,m) \in A\}$ et $m_0 = min\{m | (n_0, m) \in A\}$. Toute partie non vide de N^2 ayant un élément minimal pour l'ordre lexicographique, c'est un bon ordre sur N^2.

On peut introduire ensuite le prédicat :

"$P(n,m)$=`ackerman (n,m)` termine".

Nous pouvons maintenant montrer que $\forall\ (n,m) \in\ N^2,\ P(n,m)$ est vérifier. Nous raisonnons par l'absurde :
supposons que l'ensemble $A = \{(n,m) \in N^2 | P(n,m)\ est\ faux\}$ soit non vide, alors il a un plus petit élément (n_0, m_0).

- si $n_0 = 0$, par `(* 1 *)`, pour tout m, `(ackerman` (n_0,m)`)` termine (et vaut $m+1$) donc en particulier $P((n_0,m_0))$ est vrai ce qui contredit $(n_0,m) \in A$.
- sinon si $m_0 = 0$, comme $n_0 > 0$, il a un prédécesseur et $(n_0-1,1) \preceq (n_0,m_0)$. Donc par `(* 2 *)` on a `(ackerman` $(n_0,0)$`)=(ackerman` $(n_0-1,1)$`)`.
  Or $(n_0-1,1) \notin A$ donc $P((n_0-1,1))$ est vérifié et `(akerman` $(n_0-1,1)$`)` termine. Mais alors `(akerman` $(n_0,0)$`)` termine et $P((n_0,0))$ est vrai ce qui contredit $(n_0,0) \in A$.
- sinon si $m_0 > 0$ il a un prédécesseur et $(n_0, m_0-1) \preceq (n_0,m_0)$. Donc par `(* 3 *)` on a :
  `(akerman` (n_0,m_0)`)=(akerman ((`n_0-1`),(akerman` (n_0,m_0-1)`)))`. Or $(n_0-1,1) \notin A$ donc $P((n_0-1,1))$ et `(akerman` (n_0,m_0-1)`)` termine. Quelque soit m=`(akerman` (n_0,m_0-1)`)`, $(n_0-1,m) \preceq (n_0,m_0)$ donc $P((n_0-1,m))$ est vérifié et `(akerman ((`n_0-1`),(akerman` (n_0,m_0-1)`)))` termine. Mais alors `(akerman` (n_0,m_0)`)` termine et $P((n_0,m_0))$ est vrai ce qui contredit $(n_0,m_0) \in A$.

Remarque 6. La fonction `somme` et la fonction d'Ackerman sont totales sur N. mais elles ne le sont pas sur Z, elles ne terminent pas avec des entrées négatives.

Exercice 1.

1. Définir une fonction qui calcule le pgcd de deux entiers et prouvez en la terminaison.
2. On donne la fonction suivante (dite fonction f91 de Mac carthy)

```
let rec f91 x =
   if x<=100 then f91 (f91(x+11)) else x-10;;
f91 : int -> int = <fun>
```

   - Essayer d'expliquer avec vos mots les résultats possibles de cette fonction.
   - Faire la preuve de sa terminaison.

Indécidabilité de la terminaison On peut montrer qu'il n'existe pas d'algorithme capable de décider automatiquement de la terminaison des fonctions récursives. En effet supposons qu'un tel algorithme existe. On peut alors le coder par un `termine` prenant en argument une fonction et renvoyant `vrai` si elle termine et `faux` sinon (donc de type `('a->'b)->bool`). Mais on peut aussi écrire la fonction suivante :

```
let rec paradox() = if (termine paradox) then paradox() else ()
```

qui conduit à un paradoxe puisqu'elle termine si et seulement si elle ne termine pas.

1.8.2 Preuves de correction

Lorsqu'une fonction termine, elle produit toujours un résultat, il reste reste maintenant à voir comment être certain que ce résultat correspond bien a ce que l'on attend. Pour cela on va exprimer formellement (pour nous a l'aide d'un prédicat) ce que l'on attend de la fonction. Puis nous allons raisonner sur la définition de la fonction.

Si la fonction n'est pas récursive mais travaille sur un type somme ou contient un `if`, on fera un raisonnement par cas. Si la fonction est récursive et travaille sur des entiers on fera un raisonnement par récurrence.

Exemple de preuve par récurrence : Reprenons la fonction `somme`. on considère le prédicat :
P(n)=" `somme n`$=\sum_{i=O}^{n} i$ ".

On prouve par récurrence que `P(n)` est vérifié pour tout `n`

1. si `n=0` trivial.
2. supposons que `P(n)` soit vérifié. Alors `somme (n+1)=n+(somme n)`. Par hypothèse, `(somme n)=`$\sum_{i=O}^{n} i$ donc `somme (n+1)=n`$+\sum_{i=O}^{n} i$=$\sum_{i=O}^{n+1} i$

Si la fonction est récursive et travaille sur d'autres types récursifs on fera un raisonnement par induction structurelle.

Exemple de preuve par induction structurelle . Preuve de la fonction d'insertion dans une liste triée :

```
let rec insert s l = match l with
     []->[s]
     |x::xs  ->if s<=x then s::l else x::(insert s xs);;
```

On considère le prédicat :

P(l) = "pour tout élément `s`, `insert s l` est triée en ordre croissant et contient tout les éléments de `l` plus une occurrence de `s`"[7]

On raisonne maintenant par induction sur la structure de la liste pour montrer que pour toute liste croissante `l`, `P(l)` est vérifié.

– `P([])` est évident
– Supposons que si `l` est croissante `P(l)` soit vérifiée et montrons qu'alors pour tout `y` telque `y::l` soit croissante,`P(y::l)`.
  - si `s <= y` alors `(insert s y::l)=s::y::l` est trié en ordre croissant, chacun des éléments de `s::y::l` différent de `s` a le même nombre d'occurences dans `s::y::l` que dans `y::l` et `s` a une occurrence de plus. Donc `P(y::l)` est vrai.
  - si `s > y` alors `(insert s y::l)= y::(insert s l)`. Comme par hypothèse `y::l` est croissante, `l` est croissante et donc `P(l)` est vérifié. Donc `insert s l` est triée en ordre croissant et contient tout les éléments de `l` plus une occurrence de `s`; donc `y::(insert s l)` est triée en ordre croissant et contient tout les éléments de `l` plus une occurrence de `s` et une occurrence de `y` c'est-à-dire, contient tout les éléments de `y::l` plus une occurrence de `s`; donc `P(y::l)`.

  (ouf...plus délayé impossible :-))

Remarque 7. On peut aussi raisonner sur la longueur de la liste est se ramener à une récurrence.

Exercice 2 (Preuve de correction de la fonction `supprime`*).*

On considère la fonction `supprime` définie par :

```
let rec supprime s l = match l with
   []->[]
   |x::xs->if s=x then x::(supprime s xs) else (supprime s xs);;
```

Elle prend en entrée un élément `s` et une liste `l` et est sensée renvoyer une copie de `l` dans laquelle on a supprimé toutes les occurrences de `s`.

La terminaison est évidente (par exemple, en raisonnant sur la longueur de la liste et en remarquant que les appels récursifs se font toujours sur la queue de la liste courante.)

Prouver par récurrence la correction de cette fonction.

[7] Plus formellement on écrirait : $\forall s \in Element, (trie\ (insert\ s\ l)) \wedge ((noccence\ s\ (insert\ s\ l)) = (noccence\ s\ l){+}1) \wedge (\forall ynot = s(noccence\ s\ (insert\ s\ l)) = (noccence\ s\ l))$

Preuve de propriétés

On peut aussi ne vouloir vérifier que certaines propriétés d'une fonction :
si nous avons défini la fonction de concaténation des listes par

```
let rec concate l1 l2 = match l1 with
 []->l2 (* 1 *)
 |x::xs->x::(concate xs l2)    (* 2 *);;
```

on peut vouloir vérifier que la fonction obtenue est associative, c'est-à-dire que $\forall l, k, h$ `concate(l,concate(k,h))=concate(concate(l,k),h)` On va vérifier cela par récurrence sur `l` :

- Cas de base :
  concate(Nil (concate k h))=(concate k h) par définition (1)
  et concate((concate nil k) h)=(concate k h) par définition (1)
  donc $\forall k, h$`concate(nil (concate k h))=concate((concate nil k) h)`

- Cas de récurrence :
  On suppose que :
  $\forall k, h$ `concate(l (concate k h))=concate((concate l k) h).`

  ```
  concate(s::l (concate k h))=s::concate(l (concate k h)) par definition (2)
                             =s::concate((concate l k) h) par hypothèse de recurrence

  concate((concate s::l  k) h)=concate(s::(concate l k) h) par definition (2)
                             =s::concate((concate l k) h) par definition (2)
  ```

  donc $\forall k, h$ `concate(s::l (concate k h))=concate((concate s::l k) h).`

Exercice 3.

1. Soit la fonction suivante :

   ```
   let rec double l = match l with
       []->[]
      |x::xs->(2*x)::(double xs);;
   ```

   Monter que pour toutes listes d'entiers, `l` et `k`. `double(concate k l)=concate((double k) (double l))`