L1, Informatique, semestre 2

Notes de cours

Avertissement au lecteur :

Ce polycopié n'est pas un document scolaire de référence sur le cours d'informatique, c'est seulement l'ensemble de mes propres notes de cours mises en forme. Il ne contient donc pas les explications détaillées qui ont été données en cours. En particulier tous les développements systématiques des exemples, expliquant comment le langage ML effectuerait les traitements, sont absents de ces notes. On trouvera également parfois quelques simples allusions à des concepts élémentaires, largement développés en cours mais qui sont routiniers pour tout informaticien.

G. Bernot

COURS 1

1. Rappels sur le langage ML
2. Les constructeurs d'un type de données
3. Les types « somme »
4. Usage de `match` sur les types somme
5. Les stratégies de parcours dans les arbres binaires

COURS 2

1. Programmation sur les arbres binaires
2. Les arbres binaires de recherche

COURS 3

1. Les arbres binaires complets
2. Preuves par induction structurelle sur les arbres binaires complets
3. Preuves par induction en général

COURS 4

1. Programmation à l'ordre supérieur
2. Les types fonctionnels
3. Premiers exemples

COURS 5

1. Fonctions à plusieurs « vrais » arguments
2. Exemples de fonctions d'ordre supérieur

COURS 6

1. Exemples supplémentaires de fonctions d'ordre supérieur
2. Fonctions d'ordre supérieur sur les listes

COURS 7

1. Programmation des fonctions d'ordre supérieur sur les listes
2. Les arbres généraux

Cours No. 1

Rappels sur le langage ML

Un langage est défini par les structures de données qu'il offre et par son contrôle. Un langage fonctionnel comme ML possède peu de dispositifs de contrôle car la notion même de fonction qui est à la base du langage offre gratuitement presque toutes les structures de contrôle utiles. On fonde donc cette révision sur les structures de données et le contrôle sera vu au passage en fonction des besoins.

Les types de données permettent de classer et caractériser proprement les structures de données en ML. On passe donc en revue les différents types de données vus l'an dernier. On commence par les types de base, c'est-à-dire ceux qui ne servent pas à construire de nouveaux types de données à partir d'autres types de données. On continuera par les types de données composés, c'est-à-dire ceux qui se construisent de manière générique à partir d'un ou plusieurs autres types de données quelconques.

1 Rappels sur les types de base

On révise rapidement les types suivants
- Les booléens (`bool`).
- Les nombres entiers relatifs (`int`).
- Les nombres réels (`float`)
- Les chaînes de caractères (`string`)

2 Rappels sur le contrôle

On profite de ces rappels pour donner des exemples et à cette occasion on revoit les éléments de contrôle suivants :
- Les déclarations avec `let`.
- Les fonctions récursives avec `let rec`.
- Les conditionnelles `if..then..else..` qui peuvent en fait être vues comme une fonction générique de type `bool` $\times \alpha \times \alpha \rightarrow \alpha$.

3 Rappels sur les types de données composés

Une structure de données est définie par 2 choses :
1. un ensemble, dont les éléments sont les *données* de la structure de données ; et le nom de cet ensemble est appelé le *type* de ces données
2. des *fonctions* (au sens mathématique du terme), que l'on appelle aussi les *opérations* de la structure de données car elles sont utilisées par l'ordinateur pour effectuer des « calculs » sur les données précédentes.

On revoit avec plusieurs exemples les deux types de données suivants :
- Les listes dont les éléments sont d'un type quelconque α, c'est-à-dire les α `list`.
- Les types enregistrement.

On donne en exemple la gestion d'une liste d'étudiants, chaque étudiant ayant un `nom`, un `numero` de carte et une `moyenne`.

4 Rappels complémentaires sur le contrôle

À cette occasion on révise :
- L'usage de `match` sur les listes
- Le traitement d'exceptions avec `failwith`

Ceci termine les révisions. On aborde maintenant une nouvelle manière de définir de nouveaux types de données en ML : les *types « somme »*. Tout d'abord, il nous faut bien comprendre la notion de constructeurs d'un type de données.

5 Les constructeurs d'un type de données

Certaines constantes et fonctions jouent un rôle particulier au sein d'un type de données car elles permettent de construire inductivement toutes les données de ce type.

Dans le cas des listes par exemple, la constante `[]` et l'opération `_::_` permettent de construire toutes les listes. En effet, étant donné un type de données α quelconque, supposé déjà fourni, on peut construire toutes les listes avec ces deux fonctions :

- la constante `[]` fournit la seule liste de longueur 0 qui existe ;
- on peut obtenir toutes les listes de longueur 1 en utilisant l'opération `_::_` comme suit : `a::[]` et en considérant toutes les valeurs possibles de `a` au sein du type α ;
- plus généralement, on peut obtenir toutes les longueurs de liste possibles en utilisant `_::_` autant de fois que necessaire (longueur de la liste) : `a_1::a_2::...a_n::[]`

On fait le lien entre cette remarque et l'usage de l'opérateur `match` qui décompose les listes selon ce principe exactement. C'est le fait que ces deux fonctions sont les constructeurs choisis pour les listes qui autorise à faire `match` de cette façon dans les programmes.

On pourrait tout à fait définir le type des entiers naturels, de manière encore plus simple, avec les constructeurs 0 et `succ_`, où l'opération successeur est intuitivement celle qui ajoute 1 à son argument. Tout entier naturel est obtenu en appliquant `succ` autant de fois que nécessaire à 0. En fait, par définition, l'entier n est égal à n applications successives de `succ` au-dessus de 0.
D'un point de vue purement mathématique, l'ensemble des entiers naturels, $I\!N$, est défini par les axiomes de Peano :

1. 0 existe
2. si n existe alors $\texttt{succ}(n)$ existe
3. `succ` est injective
4. 0 n'est le successeur d'aucun entier naturel
5. quelle que soit la propriété P, pour prouver $\forall n \in I\!N,\ P(n)$ il suffit de prouver $p(0)$ et $\forall i \in I\!N,\ P(i) \Rightarrow P(\texttt{succ}(i))$

Les deux premiers axiomes disent exactement que 0 et `succ` sont constructeurs de $I\!N$. Le cinquième n'est autre que la récurence et dit en réalité qu'il n'y a pas d'autre entier naturel que ceux *engendrés* par 0 et `succ`. Enfin le troisième et le cinquième évitent des structures de données circulaires. Ces axiomes fondamentaux de l'arithmétique traduisent en fait les fondements mathématiques des fonctions récursives.

Les entiers relatifs sont un exemple intéressant. On a en tête facilement deux ensembles de constructeurs « raisonnables » : $\{0, succ, pred\}$ et $\{0, succ, opp\}$. Cet exemple nous permet d'insister sur le fait que plusieurs choix de constructeurs peuvent être faits pour une même structure de données. Ici on constate qu'il y a des « équations entre constructeurs », à savoir, d'une part, $succ(pred(n)) = n = pred(succ(n))$ et, d'autre part, $opp(0) = 0$ et $opp(opp(n)) = n$.

On remarque donc que le choix des constructeurs d'un type de données n'est pas unique : cas d'école liste vide `[]`, liste de taille 1 `[_]` et concaténation `@`. Certains choix de constructeurs entrainent des équations entre constructeurs parfois lourdes et c'est souvent cela qui donne une préférence pour un choix de constructeurs plutôt qu'un autre.

Enfin il existe un cas particulier de types de données (fini) où les constructeurs ne sont que des constantes. C'est le cas des types `bool` et `char`. On aura envie bien sur de définir d'autres types de ce genre comme les nucléotides `{A,T,G,C}`, les couleurs `{R,V,B}`, etc.

6 Les types « somme »

Comment on les déclare :

```
type α1 ··· αn truc =
      cst_0 | ... | cst_k
     | c_1 of τ1 | ... | c_p of τp ;;
```

où les τ_j peuvent non seulement contenir les α_i mais aussi de manière récursive le type ($\alpha_1\ \cdots\ \alpha_n$ `truc`) lui-même.

Exemples :

- nucléotides
- couleurs
- une redéfinition des listes
- arbres binaires : avec deux possibilités
  - admettre qu'un arbre *vide* est une donnée acceptable
    ```
    type 'a Abin = vide
                 | node of 'a * ('a ABin) * ('a ABin) ;;
    ```
  - ne pas accepter d'arbres vides, auquel cas il faut distinguer les noeuds n'ayant qu'un seul fils
    ```
    type 'a Abin = feuille of 'a
                 | birac of 'a * ('a ABin) * ('a ABin)
                 | monorac of 'a * ('a ABin) ;;
    ```

7 Usage de `match` sur les types somme

Forme générale, utilisée dans une expression :

... (`match` *expr* with
  $motif_1 \rightarrow result_1$
  | $motif_2 \rightarrow result_2$
  | ...
  | $motif_n \rightarrow result_n$)
...

où les motifs sont des termes qui ne peuvent faire intervenir que des variables ou des constructeurs de types. On comprend mieux sur des exemples...

Les nocléotides :

```
type base = A | T | G | C | N ;;

let complementaire b = match b with
  A -> T | T -> A | G -> C | C -> G | N -> N ;;
```

Les listes :

```
let rec est_compl (l1,l2) = match (l1,l2) with
      ([],[])      -> true
 |     ([],_)      -> false
 |     (_,[])      -> false
 | (b1::r1,b2::r2) -> (b2 = (complementaire b1))
                        && est_compl(r1,r2) ;;
```

On remarque l'usage de « _ » qui remplace une variable inutilisée. La faiblesse de cette solution est que `N` devrait être considéré comme complémentaire de n'importe quelle base dans le cas de la comparaison de brins :

```
let rec est_compl (l1,l2) = match (l1,l2) with
      ([],[])      -> true
 | (N::r1,b2::r2)  -> est_compl(r1,r2)
 | (b1::r1,N::r2)  -> est_compl(r1,r2)
 | (b1::r1,b2::r2) -> (b2 = (complementaire b1))
                        && est_compl(r1,r2)
 |      _      -> false
```

On rappelle que le premier motif qui généralise la valeur traitée est celui qui sera choisi losque la fonction est exécutée. Ainsi l'avant dernière ligne n'a lieu que si le premier nucléotide des deux brins est connu (différent de `N`). De même la dernière ligne généralise les deuxième et troisième motifs de la version précédente.

Les piles (`stack`) : constructeurs `emptystack` et `push`. fonctions `isEmpty`, `top`, `pop`, `height`. On remarque que c'est un sous-ensemble des choses que l'on peut faire avec les listes mais on mentionne rapidement les questions d'encapsulation qui justifient l'existence de types plus pauvres que les autres.

Les files FIFO : constructeurs `emptyfifo` et `add`. Fonctions `length`, `first`, `remove`. En insistant sur la technique des appels récursifs, et ne pas oublier de remettre les éléments non traités pour `supprime`.

8 Stratégies de parcours dans les arbres binaires

```
type 'a ABin = vide
             | noeud of 'a * ('a ABin) * ('a ABin) ;;
```

On a commencé à voir les arbres binaires de recherche en TD. Ils sont aux arbres binaires ce que sont les listes triées aux listes. Le lien entre les deux peut être vu comme un parcours de l'arbre binaire pour produire une liste, c'est-à-dire une énumération des éléments de l'arbre dans un ordre bien choisi pour préserver l'ordre induit par le fait que l'arbre binaire soit « de recherche ».

Puisqu'un arbre binaire de recherche est tel qu'en tout nœud, la valeur du nœud est supérieure ou égale à toutes celles du sous-arbre de gauche et strictement inférieure à toutes celles du sous-arbre de droite, il est naturel d'énumérer d'abord le sous-arbre de gauche, puis la valeur du nœud, puis le sous-arbre de droite. ceci afin de préserver un ordre croissant des éléments dans la liste résultante.

```
let rec enum1 a = match a with vide -> []
   | noeud(r,g,d) -> (enum1 g) @ [r] @ (enum1 d) ;;
enum1 : 'a ABin -> 'a list = <fun>
```

Remarquer le typage sur les listes dans `...@[r]@....`

Développement d'un exemple sur un ABR, puis sur un arbre binaire quelconque.

On voit bien qu'il existe d'autres ordres d'énumération des éléments selon l'ordre de traitement des sous-arbres et de la racine. De plus, le fait de traiter « d'un bloc » chaque sous-arbre fait que l'on effectue l'énumération en parcourant en profondeur chaque sous-arbre choisi. Il s'agit d'un ensemble de stratégies de parcours « en profondeur ». Il existe 6 stratégies de parcours en profondeur.

Les stratégies en profondeur de gauche à droite

On a vu *racine au milieu.*

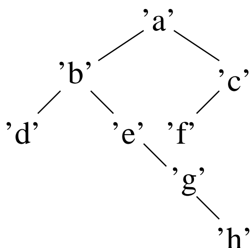
Racine à gauche :

```
let rec enum2 a = match a with vide -> []
   | noeud(r,g,d) -> r :: (enum2 g) @ (enum2 d) ;;
enum2 : 'a ABin -> 'a list = <fun>
```

Remarquer le typage sur les listes qui autorise `r] : :(...)`.

Développement d'un exemple :

```
let e = noeud(‘a‘,
              noeud(‘b‘,
                    noeud(‘d‘,vide,vide),
                    noeud(‘e‘,vide,
                          noeud(‘g‘,vide,
                                noeud(‘h‘,vide,vide)))),
              noeud(‘c‘,noeud(‘f‘,vide,vide),vide)) ;;
e : char ABin =  noeud(‘a‘,.....
```



Racine à droite :

```
let rec enum3 a = match a with vide -> []
   | noeud(r,g,d) -> (enum3 g) @ (enum3 d) @ [r] ;;
enum3 : 'a ABin -> 'a list = <fun>
```

Remarquer le typage sur les listes. . .

Exemple. . .

Les stratégies en profondeur de droite à gauche

Racine à gauche :

```
let rec enum4 a = match a with vide -> []
   | noeud(r,g,d) -> r :: (enum4 d) @ (enum4 g) ;;
enum4 : 'a ABin -> 'a list = <fun>
```

Exemple…

Racine au milieu :

```
let rec enum5 a = match a with vide -> []
   | noeud(r,g,d) -> (enum5 d) @ [r] @ (enum5 g) ;;
enum5 : 'a ABin -> 'a list = <fun>
```

Exemple…

Racine à droite :

```
let rec enum6 a = match a with vide -> []
   | noeud(r,g,d) -> (enum6 d) @ (enum6 g) @ [r] ;;
enum6 : 'a ABin -> 'a list = <fun>
```

Exemple…

Les stratégies en largeur

On constate qu'aucune énumération ne redonne l'ordre alphabétique sur l'exemple du `char ABin e` précédent. La stratégie « de remplissage » de l'arbre lorsqu'on l'a tracé n'était pas *en profondeur* mais *en largeur, de gauche à droite et de haut en bas.*

On montre que ce n'est pas trivial, que toutes les tentatives de programmation naturelles échouent.

Les stratégies en largeur nécessitent l'usage d'une file d'attente des sous-arbres qui restent à traiter.

Rappelons que l'on a défini le type `fifo` avec les constructeurs `emptyfifo` et `add`, et les fonctions `length`, `first`, `remove`.

En largeur, de gauche à droite et de haut en bas :

```
let rec enum7fifo f = match f with emptyfifo -> []
 | f' -> ( match (first f) with
                     vide -> enum7fifo (remove f')
           | noeud(r,g,d) -> r::(enum7fifo (add(d,add(g,(remove f')))))
         ) ;;
enum7fifo : 'a ABin fifo -> 'a list = <fun>

let enum7 a = enum7fifo (add(a,emptyfifo)) ;;
```

Exemple :

```
enum7 e ;;
- : char list = [`a`; `b`; `c`; `d`; `e`; `f`; `g`; `h`]
```

En largeur, de droite à gauche et de bas en haut :

```
let rec enum8fifo f = match f with emptyfifo -> []
```

```
 | f' -> ( match (first f) with
                     vide -> enum8fifo (remove f')
           | noeud(r,g,d) -> (enum8fifo (add(d,add(g,(remove f')))))@[r]
         ) ;;
enum8fifo : 'a ABin fifo -> 'a list = <fun>
```

Exemple...

En largeur, de droite à gauche et de haut en bas :

```
let rec enum9fifo f = match f with emptyfifo -> []
 | f' -> ( match (first f) with
                     vide -> enum9fifo (remove f')
           | noeud(r,g,d) -> r::(enum9fifo (add(g,add(d,(remove f')))))
         ) ;;
```

Exemple...

En largeur, de gauche à droite et de bas en haut :

```
let rec enum10fifo f = match f with emptyfifo -> []
 | f' -> ( match (first f) with
                     vide -> enum10fifo (remove f')
           | noeud(r,g,d) -> (enum10fifo (add(g,add(d,(remove f')))))@[r]
         ) ;;
```

Cours No. 2

1 Programmation sur les arbres binaires

Rappel du type :

```
type 'a ABin = empty
             | node of 'a * ('a ABin) * ('a ABin) ;;
```

On développe des exemples :

- somme des éléments d'un ABin
- appartenance d'un élément à un arbre
- nombre d'occurrences d'un élément à un arbre
- suppression d'un élément d'un arbre (toutes les occurences, puis seulement une qui dépendra de la stratégie)
- insertion d'un élément en une feuille là où son chemin depuis la racine est le plus court (le plus à gauche en cas d'égalité des hauteurs). L'idée est d'équilibrer l'arbre pour le remplir en minimisant sa hauteur.
- élément maximal
- longueur de cheminement externe
- longueur de cheminement interne

Retour sur l'insertion minimisant la hauteur : l'usage de la fonction `hauteur` à chaque appel récursif est toitalement inefficace. On refait avec le type suivant qui mémorise en chaque nœud la hauteur du sous-arbre dont il est racine :

```
type 'a ABeq = vide
             | noeud of int * 'a * ('a ABin) * ('a ABin) ;;
```

- On programme la fonction `correct` qui vérifie si les hauteurs sont correctes.
- la fonction d'insertion précédente
- la suppression d'un élément

2 Les arbres binaires de recherche

Un arbre binaire est dit *de recherche* si en tout nœud de l'arbre :

- la valeur qui est en ce nœud est supérieure ou égale à toutes les valeurs contenues dans son sous-arbre de gauche
- et cette valeur est strictement inférieure à toutes les valeurs contenues dans le sous-arbre de droite

On écrit la fonction qui recherche un élément dans l'arbre. On rappelle la fonction correspondante sur les listes triées et l'on remarque que l'une passe un temps de calcul proportionnel au nombre d'éléments, l'autre à son log_2.
On renvoit au prochain TD pour plus de choses sur les arbres binaires.

1 Les arbres binaires complets

Définition : un *arbre binaire complet* (ABC) est un arbre binaire tel qu'en tout nœud, il y a 0 ou 2 fils.

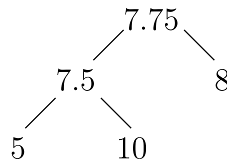
Exemple et contre-exemple.

On remarque que l'on ne peut pas avoir un arbre binaire complet possédant 2 nœuds. On passe de 1 à 3 (puis 5 d'ailleurs). On verra que les ABC ont diverses propriétés de ce genre.

En fait, on utilise généralement les ABC lorsque seules les feuilles contiennent des données d'intérêt et les nœuds internes peuvent contenir des valeurs intermédiaires annexes. On peut définir en ML les ABC non vides en ce sens, donc avec des types différents aux feuilles et dans les nœuds internes :

```
type ('a,'b) ABC = feuille of 'a
  | interne of 'b * (('a,'b) ABC) * (('a,'b) ABC) ;;
```

On peut par exemple avoir un arbre contenant des entiers aux feuilles et les moyennes des deux fils en chaque nœud interne.




```
let c = interne(7.75,
               interne(7.5,(feuille 5),(feuille 10)),
               feuille 8) ;;
c : (int, float) ABC = interne(7.75, .......
```

2 Preuves par induction structurelle sur les ABC

Un ABC possède toujours un nombre impair de nœuds

On peut en faire la preuve de cette propriété d'une manière assez naturelle :
- Un ABC est par définition soit une feuille, soit construit avec une racine au-dessus de deux feuilles, soit avec une racine au-dessus de deux sous arbres qui sont eux-mêmes construits comme cela par étapes.
- Si l'ABC ne contient qu'une feuille, alors c'est évident.
- Si l'ABC est une racine au-dessus de deux feuilles, alors c'est aussi évident.
- Si l'ABC est construit par une racine au-dessus de deux sous-ABC `g` et `d`, qui sont eux-mêmes construits par les étapes précédentes, alors `g` et `d` vérifient la propriété. Le nombre de nœuds de l'ABC est la somme de celui de `g` et de celui de `d` plus 1 (pour la racine). On a donc affaire à la somme de trois nombres impairs, qui est elle même impaire.
- De proche en proche, par constructions successives comme les étapes précédentes, tous les ABC qu'on construit ont donc obligatoirement un nombre impair de nœuds.

Cette méthode de preuve est l'*induction structurelle* : pour prouver une propriété $\mathcal{P}$ sur les ABC,

1. on prouve que $\mathcal{P}$ est vrai si l'ABC est réduit à une feuille, c'est-à-dire de la forme `feuille(x)`,
2. on suppose que $\mathcal{P}$ est vraie pour deux ABC `g` et `d`, et l'on prouve que $\mathcal{P}$ est vraie pour l'ABC obtenu en ajoutant une racive au-dessus de ces deux sous-arbres, c'est-à-dire pour `interne(r,g,d)`.

Exemple : *Le nombre de feuilles d'un ABC est égal au nombre de nœuds internes plus 1.* (preuve. . .)

3 Preuves par induction structurelle en général

En fait, l'induction structurelle peut être utilisée pour tous les types somme. Par exemple, nous avions défini le type des entiers naturels :

```
type nat = zero | succ of nat ;;
```

Ici `zero` joue le même rôle que `feuille` (c'est le cas de base) et `succ` joue le même rôle que `interne` (c'est la construction récursive des valeurs). Ainsi, pour prouver une propriété sur `nat`$=I\!N$:

1. on prouve la propriété pour `zero`,
2. on suppose la propriété vraie pour `n` est on prouve qu'elle est vraie pour `succ(n)`.

On retrouve donc les preuves par récurrence classiques sur les entiers naturels comme un cas particulier de l'induction structurelle.

Ça marche sur les listes, dont les constructeurs sont `[]` et `_::_` : pour prouver une propriété $\mathcal{P}$ sur les listes,

1. on prouve que $\mathcal{P}$ est vrai pour la liste `[]`,
2. on suppose que $\mathcal{P}$ est vraie pour une liste `r`, et l'on prouve que $\mathcal{P}$ est vraie pour `x::r` (quel que soit l'élément `x`).

Exemple :

```
let rec concat (l1,l2) = match l1 with [] -> l2
     | x1::r1 -> x1::(concat (r1,l2)) ;;

let rec reverse l = match l with [] -> []
     | x::r -> concat((reverse r),[x]) ;;
```

On prouve la propriété $\mathcal{P}$ suivante par induction sur `l1` :

```
reverse(concat(l1,l2))=concat((reverse l2),(reverse l1))
```

On passe par le lemme `concat(l,[])=l` pour le cas de base. On passe par le lemme d'associativité de `concat` pour le cas d'induction. Ces deux lemmes sont eux même prouvés par induction sur la liste `l`.

Cours No. 4

1 Programmation à l'ordre supérieur

Bilan de ce qu'on a vu dans la première partie :

Types somme : – Avec les types énumérés où il n'y a que des constantes,
- les autres où l'on peut distinguer, d'une part, les constantes (comme `empty`) ou les constructeurs qui ne prennent pas le type en cours de définition dans leurs arguments (comme `feuille`), et d'autre part, les autres constructeurs qui servent à construire les éléments du type en augmentant leur taille
- on remarque l'importance de `match` qui propage cette façon de penser à la programmation en développant les algorithmes récursivement selon le schéma des constructeurs
- enfin on a beaucoup utilisé les arbres binaires pour mettre tout ça en pratique

Induction structurelle : cela propage encore cette façon de penser à la preuve de programme.

Maintenant on va aborder une puissance de programmation fonctionnelle encore cachée dans tout ce qui précède, mais ultra-puissante.

Ordre supérieur est une terminologie venant de la logique qui qualifie toute théorie qui est assez expressive pour établir des résultats à propos d'elle-même. Ici, il s'agit de faire des programmes capables de faire des programmes, donc dans le cadre fonctionnel, écrire des fonctions capables de fabriquer des fonctions. Cela passe par une chose très simple à la base mais très puissante : les types de foncions.

2 Les types fonctionnels

Rappelons qu'on définissait le type produit cartésien en disant que si α et β sont des types de données, alors automatiquement le langage ML fournit le type α`*`β dont les éléments sont les couples `(a,b)` tels que `a` est de type α et `b` est de type β.

Ici c'est similaire.

Si α et β sont des types existants, alors $\alpha \rightarrow \beta$ est un type existant automatiquement fourni par le langage ML. Ses éléments sont les fonctions dont l'ensemble de départ est α, l'ensemble d'arrivée est β et telles qu'il existe une expression en ML pour les définir.

Vulgarisation sur la calculabilité, l'équivallence de pouvoir d'expression de tous les langages ayant la récursivité ou le `while`. La différence entre pouvoir d'expression et expressivité (e.g. 10 lignes en un langage, 100 ou 1000 dans un autre pour un même problème).

Rappel : un type de données est défini par un nom de type, l'ensemble des valeurs de ce type et enfin l'ensemble des fonctions qui travaillent sur ce type.

Il nous reste donc à donner les fonctions. Il n'y en a qu'une nouvelle :

`function` *variable* -> *expression*

Exemple :

```
let double f = function x -> 2.0 *. (f x) ;;
```

La fonction `double` produit donc un programme (une fonction) à partir d'un autre. + Explications en tous genres, exemples avec différents `f`...

On fait le typage de `double`. On remarque que finalement « `function` » fabrique des flèches dans le type comme « `( , )` »fabrique des étoiles dans le type.

En réalité il y a une autre chose qui travaille sur le type fonctionnel, c'est l'application d'un argument à une fonction : Si `u` est de type $\alpha \rightarrow \beta$ et si `v` est de type α, alors `(u v)` est de type β.

3 Premiers exemples de fonctions d'ordre supérieur

Exemple 1 : concaténer le résultat de deux fonctions qui retournent des listes :

```
let succede (f,g) = function (x,y) -> (f x) @ (g y) ;;
```

et typage de cette fonction. En exemple d'application on se donne :

```
let rec pairs n = if n < 0 then failwith "positifs SVP"
                  else if n=0 then [0]
                       else (2*n)::(pairs (n-1)) ;;
(* nombres pairs, ordre décroissant *)
let rec impairs n = if n < 0 then failwith "positifs SVP"
                  else if n=0 then [1]
                       else (impairs (n-1)) @ [2*n+1] ;;
(* impairs, ordre croissant *)
```

et on génère la liste contenant les pairs inférieurs à `n` en ordre décroissant suivis des impairs en ordre croissant :

```
let genere n = ( succede (pairs,impairs) ) ( n/2 , (n-1)/2 ) ;;
```

et on explique l'application « double », le typage, etc. On développe un exemple, etc.

Exemple 2 : extraire d'une liste selon un prédicat quelconque :

```
let rec extrait (p,l) = match l with [] -> []
     | x::r -> if (p x) then x::(extrait (p,r))
               else extrait (p,r) ;;
```

typage, commentaires sur le typage de `p`, etc.
Exemple d'application :

```
let extraitpairs l = extrait ((function n -> n mod 2 = 0) , l) ;;
```

avec commentaires sur le fait qu'il n'est pas nécessaire de nommer le prédicat de parité qui joue le rôle de `p`, ceci grâce à `function` qui construit le type fonctionnel.
. . . et développer des exemples de tout ça.

Cours No. 5

1 Fonctions à plusieurs « vrais » arguments

On commence par expliquer la distinction entre une fonction à plusieurs arguments dont le type est de la forme $\alpha_1 \to \alpha_2 \to \cdots \alpha_n \to \beta$ et une fonction à un seul argument n-uplet de type $\alpha_1 * \alpha_2 * \cdots * \alpha_n \to \beta$.

On remarque le parenthésage par défaut à droite pour le type : $\alpha_1 \to (\alpha_2 \to \cdots (\alpha_n \to \beta)..)$, à gauche pour l'application des arguments : `(..((f a_1) a_2)...) a_n`, et la correspondance logique de ces deux conventions.

À partir de maintenant on préfèrera programmer des fonctions avec plusieurs « vrais » arguments plutôt qu'en utilisant un produit cartésien. La raison en est la possibilité de faire des applications partielles (c'est-à-dire ne pas donner tous les arguments). On donne des exemples comme la fonction `double` pour laquelle `double f` a un intérêt en soi, sans lui donner pour autant le dernier argument.

Donc les fonctions « à la manière du premier semestre » sont en fait des fonctions à un seul argument, mais il existe une bijection entre l'ancienne et la nouvelle façon de programmer :

```
let curry f = function x -> function y -> f (x,y) ;;

let uncurry g = function (x,y) -> g x y ;;
```

avec toutes les explications qui vont avec. On fait le typage de ces deux fonctions. On donne un exemple sur une fonction simple et l'on remarque que pour prouver que `curry` et `uncurry` sont inverses l'un de l'autre (sur cette fonction), on utilise un principe de transformation appelé la « β-réduction » :

`(function v ->` $expr_1$`)` $expr_2$

est égal à

$expr_1$ dans laquelle on remplace toutes les occurences de `v` par $expr_2$

C'est en fait un principe de preuve au même titre que l'induction structurelle.

2 Exemples supplémentaires de fonctions d'ordre supérieur

Hormis le fait qu'une fonction puisse renvoyer une fonction, et donc « programmer à notre place », un autre avantage est qu'une fonction peut prendre en argument un autre fonction. Cela permet d'écrire des algorithmes généraux, par exemple sur les listes ou sur les arbres binaires, sans se soucier de critères ou de comparaisons élémentaires sur les données contenues dans ces listes ou arbres ou autres.

- Après rappel de recherche du nombre d'entiers pairs d'une liste, puis du nombre de chaines de caractères de plus de 4 caractères, on met le critère en paramètre :
```
let rec nbc c l = match l with [] -> 0
  | x::r -> if c x then 1 + (nbc c r)
                   else nbc c r ;;
```
- Le prédicat `deRecherche` sur un arbre binaire, paramétré par la relation d'ordre strict.
- Pour toutes ces fonctions on calcule le type et on développe un ou deux exemples d'utilisation.

Cours No. 6

1 Exemples de fonctions d'ordre supérieur (suite)

- Insertion aux feuilles dans un ABR toujours avec relation d'ordre en argument.
- Autres fonctions sur les ABR laissées en exo
- `let rond f g = function x -> f (g x) ;;` Cette fonction est en fait la loi de composition de fonction des maths, on donne des exemples, etc.
- Pour toutes ces fonctions on calcule le type et on développe un ou deux exemples d'utilisation.

On fait remarquer que la fonction `partout` développée en TD (= le "map" sur les arbres binaires) est une fonction de « ravallement » de l'arbre en utilisant les « transformations de couleurs » indiquées par la fonction `f` donnée en premier argument. Ça ne change pas la structure de l'arbre. On peut aussi vouloir faire des travaux... On reprend trois ou quatres fonctions qui en fait faisaient des travaux : les énumérations `enum` du début du cours qui transforment un arbre en liste, la somme des éléments d'un arbre qui peut être vue comme un compactage violent de l'arbre, enfin la fonction `partout` elle-même qui en fait reconstruit à l'identique l'arbre parcouru.

Toutes ces fonctions suivent le même schéma : il y a un `germe` qui est le résultat renvoyé pour l'arbre vide, il y a une fonction d'`exploitation` locale qui combine les constructions partielles `accu`mulées dans les sous-arbres gauche et droite avec l'information apportée par la racine. Bref :

```
let rec travaux exploitation germe = function empty -> germe
 | node(r,g,d) -> exploitation ( r,
                                 (travaux exploitation germe g),
                                 (travaux exploitation germe d)
                               ) ;;
```

Par la suite, la difficulté n'est plus que dans la définition de la fonction `exploitation`. On en profite pour introduire la construction `fun` :

```
(fun r accuG accuD -> ...)
```

On obtient alors :

```
let enum = travaux (fun r accG accD -> r::accG@accD) [] ;;
let somme = travaux (fun r i j -> r+i+j) 0 ;;
let partout f = travaux (fun r accG accD -> node((f r),accG,accD) empty ;;
```

On type tout ça, on fait plein d'exemples d'utilisation, etc.

2 Fonctions d'ordre supérieur sur les listes

- `map` : $(\alpha \to \beta) \to \alpha$ list $\to \beta$ list
- `list_it` : $(\alpha \to \beta \to \beta) \to \alpha$ list $\to \beta \to \beta$
- `it_list` : $(\alpha \to \beta \to \alpha) \to \alpha \to \beta$ list $\to \alpha$

avec moultes explications, moultes dessins, moultes exemples.

Cours No. 7

1 Programmation des fonctions d'ordre supérieur sur les listes

```
let rec map f = function [] -> []
 | x::r -> (f x)::(map f r) ;;

let rec list_it op l e = match l with
     [] -> e
 | x::r -> op ( x, (list_it op r e) ) ;;

let rec it_list op e = function [] -> e
 | x::r ->
```

avec typage...

2 Les arbres généraux

```
type 'a arbre = feuille of 'a
              | interne of 'a * (('a arbre) list) ;;
```

etc.