L1 IMA UGA Valence
Programmation Fonctionnelle
INF251

Frédéric Prost
frederic.prost@univ-grenoble-alpes.fr

UGA

2021

Intervenants

- Cours (12*1h30) : Frédéric Prost ; e-mail: Frederic.Prost@univ-grenoble-alpes.fr
- TD/TP (12*1h30 chacun) :
  - Xavier Girod : Xavier.Girod@univ-grenoble-alpes.fr
  - Emeric Malevergne : Emeric.Malevergne@univ-grenoble-alpes.fr

Ressources pédagogiques du cours :

- Page web : `http://membres-lig.imag.fr/prost/L1_INF251/`

Evaluations

$\Longrightarrow$ TOUT DEPENDRA IN FINE DE LA SITUATION SANITAIRE

Evaluations

$\Longrightarrow$ TOUT DEPENDRA IN FINE DE LA SITUATION SANITAIRE

- 1 contrôle continu. (0.2)
- 1 projet : par binôme en fin de semestre. (0.2)
- 1 examen terminal (0.6)

Plan

Programmation

- La programmation est au coeur de la société numérique.
- L'informatique a plusieurs aspects
  - Pratiques.
  - Scientifiques.
  - Industriels.
- Fondamentalement différent des sciences naturelles : l'ordinateur n'est pas un objet issu de la nature.
- Il existe une manière de penser "informatique".
- Utilisation de `Ocaml` mais pas un cours de `OCaml`.

Programmation Fonctionnelle

- Approche très maths de la programmation :
  - tout programme est vu comme une fonction.
  - une éxécution est un calcul de la valeur rendue par la fonction.
- Différent de la programmation impérative : contrôle de l'état d'une machine.
- Avantages de la programmation fonctionnelle :
  - Plus abstraite (moins sensible aux détails technologiques) : Erlang.
  - Ecriture plus compacte/lisible.
  - Plus puissante : ordre supérieur, manipulation de programmes.
- Inconvénients :
  - Moins efficace (moins important aujourd'hui) : utilisation mémoire, rapidité.
  - Demande une tournure d'esprit particulière :récursivité.

Exemples

DEMONSTRATION

Installation de Ocaml

- Instructions plus précises sur la page du cours.
- Il faut installer un évaluateur Ocaml sur votre plateforme. Page officielle :

  `https://ocaml.org/docs/install.fr.html`

- Il existe des évaluateurs en ligne (dans un premier temps seulement) :
  - Try Ocaml : `https://try.ocamlpro.com/`
  - Repl It : `https://repl.it/languages/ocaml`

Plan du cours

1. Expressions, types, fonctions.
2. Récursivité.
3. Ordre supérieur.
4. Structures arborescentes.

Plan

Plan

Objets de base en Ocaml

Objets préféfinis du langage de programmation.

Type	nom en Ocaml	constantes	opérations prédéfinies
Entiers	`int`	1 -34 2345	`+ - * / ...`
Réels	`float`	0.5 0. 2e7	`+. -. *. /. ...`
Caractères	`char`	`'e' '0' '\n'`	`...`
Booléens	`bool`	`true false`	`|| && not`

Opérateurs de comparaison dont le résultat est un booléen

`e1 < e2, e1 <> e2,`

Conversion de types : `int_of_float, int_of_char, ...`

Expression, valeur et type

- Un programme est une expression dont on veut calculer la valeur.
- Une expression arithmétique :

$$e1 = 234 + (45 * 7) - 5 = 544$$

$e1$ est une expression dont la valeur est 544 et le type entier.

- Une expression algébrique :

$$e2 = 4 * x + (y/z) = ?$$

sa valeur dépend d'un environnement d'évaluation.

environnement	valeur($e2$)
$[x \leftrightarrow 0, y \leftrightarrow 6, z \leftrightarrow 3]$	2
$[x \leftrightarrow 1, y \leftrightarrow 5, z \leftrightarrow 2]$	6
$[x \leftrightarrow -3, y \leftrightarrow 12, z \leftrightarrow 3]$	-8

Typage

- En Ocaml toute expression a une valeur, et un type.
- Le langage est fortement typé :
  - `1+2.5` n'est pas une expression correcte.
  - `1.+.2.5` est une version correcte.
  - `1+ (int_of_float 2.5)` est une autre version correcte.
- Aux types de bases sont ajoutés des types que l'utilisateur peut définir.
- Certains opérateurs sont polymorphes : comparaison et égalité avec la contrainte que les deux opérandes soient de même type :
  - Corrects : `1=1, 1.0=1.0, true = false, 'e' <'r', true < false`
  - Incorrects : `1 = 1.0, 2<3<4 , 4 < 6.8`

Nommer une expression - changer l'environnement d'évaluation

- Définition globale : `let ident=expr;;`
  A pour effet d'ajouter : `ident` $\leftrightarrow$ `val`, dans l'environnemnt d'évaluation avec `val` la valeur de `expr` dans l'environnement d'évaluation précédent.

Nommer une expression - changer l'environnement d'évaluation

- Définition globale : `let ident=expr;;`
  A pour effet d'ajouter : `ident` $\leftrightarrow$ `val`, dans l'environnemnt d'évaluation avec `val` la valeur de `expr` dans l'environnement d'évaluation précédent.
- Définition locale : `let ident=expr_1 in expr_2;;`
  l'association `ident` $\leftrightarrow$ `val_1` n'est définie que pour l'évaluation de `expr_2`.

Nommer une expression - changer l'environnement d'évaluation

- Définition globale : `let ident=expr;;`
  A pour effet d'ajouter : `ident` $\leftrightarrow$ `val`, dans l'environnemnt d'évaluation avec `val` la valeur de `expr` dans l'environnement d'évaluation précédent.
- Définition locale : `let ident=expr_1 in expr_2;;`
  l'association `ident` $\leftrightarrow$ `val_1` n'est définie que pour l'évaluation de `expr_2`.
- Définitions simultanées : `let id_1=exp_1 and id_2=exp_2 and ... id_n=exp_n in exp;;`
  tous les `exp_i` sont évalués dans l'environnement courrant et produisent `val_i`, cela produit l'environnement `id_i` $\leftrightarrow$ `val_i` dans lequel est évalué `expr`

DEMONSTRATION

Définition et utilisation de fonctions

- Définir de nouvelles opérations.
- Réutilisation des fonctions à différents endroits du code.
- Lisibilité du code.
- Trois points de vues :
  - **Spécification** : ce qu'il faut faire.
  - **Définition** : comment le faire.
  - **Utilisation** : quand et où le faire.

Définir une fonction

- Schéma de définition de fonction :

```
let nom_fonction (arg1:type1)...(argn:typen) : typeres =
    expr ;;
```

- Conditions :
  - `expr` est une expression algébrique qui dépend des `argi`.
  - Le type de `expr` est le type `typeres`.
- Evaluation d'une fonction :

```
nom_fonction a_1 a_2 ...  a_n;;
```

vaut l'évaluation de `expr` dans l'environnement d'évaluation courant étendu avec :

$$[a\_1 \leftrightarrow v\_1, \ldots, a\_n \leftrightarrow v\_n]$$

Définir une fonction

- Schéma de définition de fonction :

```
let nom_fonction (arg1:type1)...(argn:typen) : typeres =
    expr ;;
```

- Conditions :
  - `expr` est une expression algébrique qui dépend des `argi`.
  - Le type de `expr` est le type `typeres`.
- Evaluation d'une fonction :

```
nom_fonction a_1 a_2 ...  a_n;;
```

vaut l'évaluation de `expr` dans l'environnement d'évaluation courant étendu avec :

$$[a\_1 \leftrightarrow v\_1, \ldots, a\_n \leftrightarrow v\_n]$$

DEMONSTRATION

Expression conditionnelle

- Schéma d'expression conditionnelle : `if cond then e_1 else e_2`
- Typage :
  `cond` est une expression de type `bool`.
  `e_1, e_2` sont deux expression de même type `t`.
  L'expression conditionnelle a pour type `t`.
- La valeur de l'expression est la valeur de `e_1` si `cond` est `true`, et la valeur de `e_2` sinon.
- La partie `else` EST OBLIGATOIRE.

Plan

Traitement de données

- Un programme peut être vu comme un manipulateur de données.
- A la fin des fins tout est écrit en binaire '0' et '1'.
  $\Longrightarrow$ Il faut abstraire pour que ce soit humainement compréhensible (code ASCII par exemple).
- Bien organiser les données permet de mieux y penser et de résoudre les problèmes plus proprement.
- Notion d'abstractions successives :

$$\begin{aligned} \text{pixel (rouge, vert, bleu)} &\Longrightarrow \text{image} = \text{tableau de pixels} \\ &\Longrightarrow \text{vidéo} = \text{liste d'images} \\ &\Longrightarrow \text{catalogue Netflix} = \text{liste de vidéos} \\ &\Longrightarrow \ldots \end{aligned}$$

- On peut définir ses propres types !

Définir un type synonyme

- Renommer un type existant.
  - types spécifiques à un contexte.
  - plus facile à se rappeler, à réutiliser.
  - rend le code plus lisible.
- Schéma de définition : `type nouveau_type=type_existant;;`
- Par exemple :
  - `type age = int; (*entre 0 et 130*)`
  - `type euros = float;;`
  - `type dollars = float;;`

  on peut définir une fonction qui convertit les euros en dollars dont le type ser explicite :

  `let conversion (x:euros) : dollars = ...`

Types énumérés

- Correspond à la définition en extension d'un ensemble en mathématiques.
- Par exemple pour exprimer les couleurs (rouge, noir), hauteurs (Roi, Dame, Valet, ...) et enseignes (Pique, Coeur, Carreaux, Trèfle) des cartes.
- Schéma de définition d'un type énuméré :
  `type nouveau_type = Valeur_1 | Valeur_2 | ... | Valeur_n`
- Les noms des valeurs doivent débuter par une majuscule.
- `Valeur_i` sont des constantes symboliques de type `nouveau_type`
- Egalité et ordre implicite sur les valeurs définies (suivant l'ordre dans la définition).

Une nouvelle expression : le filtrage

- Correspond à une analyse par cas.
- Schéma d'une expression de filtrage :

```
match expression with
     |pattern_1 -> expr_1
     |pattern_2 -> expr_2
     ....
     |pattern_n -> expr_n
```

- `expression` est évaluée et comparée aux `pattern_i` dans l'ordre.
- Les `expr_i` sont tous du même type `t`
- La valeur est celle pour laquelle la première correspondance est trouvée.

Types produits

- Produit cartésien permet de représenter un couple de valeurs. typiquement le plan $\mathbb{N}^2$. Notion de vecteur.
- Schéma de définition d'un type produit :
  `type nouveau_type = type_existant1 * type_existant2;;`
- Un opérateur de construction : si `a` est de type `ta` et `b` de type `tb` alors
  `(a,b)` est de type `ta*tb`
- Deux opérateurs définis sur les types produits : `fst, snd`.
- Nommer les expressions d'un produit cartésien :
  `let (x_1,x_2)=v in expr_avec_x_1_et_x_2`

Types sommes (ou union)

- Comment peut on avoir des objets qui peuvent avoir plusieurs types ?
- Un nombre peut être soit un entier, soit un réel, soit un complexe.
- On peut définir un type somme en donnant des constructeurs pour identifier les différents case.

```
type nombre = Entier of int |
              Reel of float |
             Complexe of (float*float);;
```

- Pour filtrer sur un type somme on utilise une expression `Match`

`DEMONSTRATION`

Types sommes

- Schéma de définition des types sommes :

```
type nouveau_type =
          | Identificateur_1 of type_1
          | Identificateur_2 of type_2
          ...
          | Identificateur_n of type_n
```

- Remarques :
  - Identificateur_i est appelé un constructeur de nouveau_type
  - la définition `of type_i` est optionnelle
  - `type_i`, peut être n'importe quel type déjà défini
  - Identificateur_i doit commencer par une lettre majuscule
- Déclaration d'une expression de type union `t` :

  `let expression = Identificateur v`

  où `Identificateur of t` est un constructeur du type `t` et `v` est une valeur de type `t`

Filtrage de Types sommes

- Le type énuméré est une version sans paramètre des types sommes.
- L'utilisation d'expressions de type somme se fait par filtrage comme pour les types énumérés.
- Par exemple :

```
type nombre = Entier of int | Reel of float;;
```

- Etudions une fonction qui indique si un nombre est plus grand que 2.
  - Spécification : `pgd_deux` est une fonction qui prend en argument un `nombre` et rend le booléen "vrai" si ce `nombre` est plus grande que 2 et "faux" sinon.
  - Profil : `pgd_deux : nombre -> booléen`
  - Réalisation :

```
let pgd_deux (x:nombre) : bool =
        match x with
            Entier(y) -> y > 2
            Reel(y) -> y > 2.0;;
```

Plan

Plan

Définitions récursives

- Très similaire au raisonnement par récurrence en mathématiques. Plutôt que de partir de 0 et d'expliquer comment on passe de n à n+1 on explique comment calculer `f(n+1)` en fonction de la valeur de `f(n)` et en donnant la valeur pour `f(0)`.
- Une définition de fonction est récursive si pour définir `f` on utilise `f` dans l'expression qui définit `f`...

  Soit `f(x) = ... f(y) ...`
- Pourquoi cela ne tourne t il pas en rond ? Par exemple `f(x) = f(x)`...
  - Il faut des cas d'arrêt.
  - Il faut que les appels récursifs soient faits sur des arguments "plus petits".

$$n! = \begin{cases} 1 & \text{si n=0 (cas d'arrêt)} \\ \text{n*(n-1)!} & \text{sinon (appel récursif)} \end{cases}$$

Exemples introductifs

- Somme des n premiers entiers :

$$\Sigma_{i=0}^{n} i = (\Sigma_{i=0}^{n-1} i) + n$$

implantation Ocaml :

```
let rec sum (n:int) :  int = if i=0 then 0 else (sum
(n-1))+n;;
let rec sum_deux (n:int) :  int =
     match n with
           0 -> 0
         | _ -> (sum (n-1))+ n;;
```

Exemples introductifs

- Somme des n premiers entiers :

$$\Sigma_{i=0}^{n} i = (\Sigma_{i=0}^{n-1} i) + n$$

implantation Ocaml :

```
let rec sum (n:int) :  int = if i=0 then 0 else (sum
(n-1))+n;;
let rec sum_deux (n:int) :  int =
     match n with
           0 -> 0
         | _ -> (sum (n-1))+ n;;
```

- Calcul :

$$\begin{aligned} \texttt{sum } 4 &\rightarrow \texttt{sum } 3+4 \\ &\rightarrow \texttt{sum } 2+3+4 \\ &\rightarrow \texttt{sum } 1+2+3+4 \\ &\rightarrow \texttt{sum } 0+1+2+3+4 \\ &\rightarrow 0+1+2+3+4 \end{aligned}$$

Exemples introductifs - Pas aussi simple

- La récurrence ne se limite pas à un seul cran : suite de Fibonnacci

$$F(n+2) = F(n+1) + F(n)$$

.

- On peut faire la récurrence sur plusieurs arguments en même temps.

  $\Longrightarrow$ Algorithme d'Euclide pour le pgcd.

- Il est possible de faire des appels récursifs imbriqués :

```
let rec mac (n:int) :  int =
        if n>100 then (n-10) else mac(mac(n+11));;
```

  que vaut `mac(n)` ?

- Récurrence croisée (paire, impaire) ...

Plan

Une manière d'analyser les problèmes

- Une manière de penser particulière : supposons que je sache résoudre le problème pour une petite taille, que dois-je faire pour résoudre le problème pour une plus grande taille ?
- Question : combien de mouvement faire pour casser une tablette de chocolat de taille n (sans superposer les plaques coupées) pour n'avoir plus que des carreaux individuels ?
- Les définitions de fonctions récursives sont en faits basées sur des types récursifs.
- En regardant comment le type récursif a été défini on peut en tirer une grille d'analyse de manière plus ou moins systématique.

Types récursifs par l'exemple : les entiers en unaire

- On peut définir les entiers par deux constructeurs :
  - Un élément de base : 0
  - Un constructeur, la fonction successeur : Succ telle que Succ(x) correspond à l'entier $x + 1$.
- Ainsi $Succ(Succ(Succ(0)))$ est le terme qui représente "3".
- En Ocaml on peut définir un tel type par :
  `type naturel = Zero | Succ of naturel;;`
- Toute l'arithmétique peut être reconstruite sur ce type défini.

Types récursifs

- Schéma de définition d'un type récursif :

```
type nom_de_type = Constr_1 of type_1
                 | ...
                 | Constr_n of type_n;;
```

- C'est la même forme que pour les types produits à la différence que `nom_de_type` peut apparaître dans les `type_i`, mais au moins un i est tel que ce ne soit pas le cas.
- Comme pour les types sommes l'égalité est définie sur les valeurs des types récursifs.
- Les valeurs du type sont les objets suivants :
  - Atome : constructeurs dont le type n'est pas récursif.
  - Objets obtenus par application de Constructeurs à des objets déjà construits.

Analyse récursive basée sur les types récursifs

- La structure d'une fonction récursive dépend du type récursif.
- A chaque constructeur du type récursif associer une équation.
- Exemple des naturels : une récurrence sur un argument donne deux équations. Par exemple la fonction qui multiplie par trois.

  `Profile : mult_trois naturel -> naturel`

  `Sémantique : mult n = trois fois n`

  `Equations de récurrences :`
  - Si `n = Zero` alors `mul_trois n = Zero`
  - Su `n = (Succ p)` alors `mul_trois n = Succ(Succ(Succ(mul_trois p)))`

  `Implantation :`

```
let rec mul_trois (n:naturel) : naturel =
      match n with
          Zero -> Zero
          | Succ(p) -> Succ(Succ(Succ(mul_trois p)))
```

Plan

Un type récursif particulier : les listes

- Les listes permettent de faire des collections d'objets de taille non fixée.
- A la différence des tableaux on ne peu tpas accéder directement au n-ième élément de la liste.
- Une liste n'est pas un ensemble car l'ordre est important.
- Une manière d'implanter "à la main" le type récursif liste d'entiers :

```
type liste_nat = Nil (*la liste vide *)
               | Cons of int * liste_nat;;
```

- Autrement dit une liste est soit la liste vide soit la liste qui contient un élément et le reste.

Exemples de listes et fonction sur les listes

- La liste $1, 2, 3$ est représentée dans le type `liste_nat` par :
  `Cons (1,Cons(2,Cons(3,Nil)))`
- Compter le nombre d'éléments d'une liste :

```
Profil -- compte :  liste d'entiers -> un entier
Sémantique -- compte(l)= le nombre d'élémént dans la
liste l
Equations de récurrence --
        compte Nil = 0
        compte Cons(x,reste) = 1+compte(reste)
Implantation --
let rec compte (l:liste_nat) :  int =
        match l with Nil -> 0
        | Cons(x,reste) -> 1 + (compte reste);;
```

Listes natives en Ocaml

- en Ocaml [ ] dénote la liste vide (équivalent du `Nil`
- en Ocaml :: est l'équivalent de `Cons`.
- $[a; b; c; d]$ dénote la liste contenant a, b, c et d.
  Contrainte : a, b, c et d doivent avoir le même type. On peut faire des listes de n'importe quel type.
  type : si a, b, c et d sont du type t alors $[a; b; c; d]$ est du type `t list`.
- L'implantation de la fonction `compte` devient :

```
let rec compte (l:int list) :  int =
        match l with [ ] -> 0
        | x::reste -> 1 + (compte reste);;
```

Analyses récursives de fonctions - récurrence sur 1 paramètre

- Somme des éléments d'une liste d'entiers.
- Produit des éléments d'une liste d'entiers.
- Ajouter un élément à droite.
- Inverser une liste.
- Concaténer deux listes.

Analyses récursives de fonctions - récurrence sur 2 paramètres

Première idée faire toutes les combinaisons possibles pour l'analyse récurrente. Par exemple :

- Savoir si un élément *e* apparaît au moins *n* fois dans une liste *l*.
- Supprimer *n* occurrences d'une élément *e* dans une liste *l*.
- Interclasser deux listes ordonnées par ordre croissant pour en faire une liste en ordre croissant.

Plan

Toutes les récursivités ne sont pas si simples

- Un exemple simple : une suite est elle ordonnée par ordre croissant ?

Toutes les récursivités ne sont pas si simples

- Un exemple simple : une suite est elle ordonnée par ordre croissant ?
- L'ensemble des parties d'un ensemble ?

Toutes les récursivités ne sont pas si simples

- Un exemple simple : une suite est elle ordonnée par ordre croissant ?
- L'ensemble des parties d'un ensemble ?
- Le compte est bon.

Toutes les récursivités ne sont pas si simples

- Un exemple simple : une suite est elle ordonnée par ordre croissant ?
- L'ensemble des parties d'un ensemble ?
- Le compte est bon.
- Quelques algorithmes de tri (récurrence sur propriété abstraite) :
  - Par insertion.

Toutes les récursivités ne sont pas si simples

- Un exemple simple : une suite est elle ordonnée par ordre croissant ?
- L'ensemble des parties d'un ensemble ?
- Le compte est bon.
- Quelques algorithmes de tri (récurrence sur propriété abstraite) :
  - Par insertion.
  - Par fusion.

Toutes les récursivités ne sont pas si simples

- Un exemple simple : une suite est elle ordonnée par ordre croissant ?
- L'ensemble des parties d'un ensemble ?
- Le compte est bon.
- Quelques algorithmes de tri (récurrence sur propriété abstraite) :
  - Par insertion.
  - Par fusion.
  - Par pivot.

Plan

Introduction

- Les fonctions sont des valeurs "comme les autres". Ou plutôt toute définition est une définition de fonction :
  `let x:int = 3;;`
  peut être vu comme une fonction à 0 argument qui répond toujours 3.
- Comment noter "la valeur" associée à une fonction ? En mathématiques on note :

$$\begin{cases} f & : & \mathbb{N} \to \mathbb{N} \\ & & x \mapsto x+1 \end{cases}$$

  la valeur de la fonction f est l'expression "$x \mapsto x+1$".
  En Ocaml on note cela par :
  `fun x -> x+1`
- Ce type d'expression peut être passé en argument ou rendu comme résultat !
- Exemple de la dérivée.

Ordre supérieur

- Introduit une grande puissance d'abstraction.
- Très difficile à utiliser correctement :
  "With great powers comes great responsabilities"
  $\Longrightarrow$ Nous nous limiterons à des schémas sur les listes.
- Deux autres aspects liés :
  - Polymorphisme
  - Curryfication

Plan

Notion de variables de types

- Les types des objets de base sont fixés : par exemple le type de 4 est `int`.
- Dans l'expression suivante on sait que x est forcément un entier:
  `let f x = x+4;;`
- Dans l'expression suivante on n'a pas de contraintes sur `x`:
  `let f x = (x,x)`
- On note avec un quote unique " ' ", les types sur lesquels il n'y a pas de contrainte.

Notion de variables de types

- Polymorphisme "simple" : la fonction ne doit pas dépendre du type particulier.
- Seule contrainte : si on utilise le même nom de type alors il représente toujours le même type.
  Par exemple `'a list` est une liste polymorphe, mais tous les éléments doivent avoir le même type.
  - ne dépendent pas du type : la taille d'une liste, inverser une liste,...
  - dépendent du type : somme des éléments, moyenne de la liste, ...
- La seule chose importante est le partage des noms de type.
  `let f = fun x:'a -> fun y:'a -> (x,y);;`
  n'est pas équivalent à
  `let f = fun x:'a -> fun y:'b -> (x,y)`

Plan

Schéma sur les listes

- On se limite à des schémas pré-implantés dans Ocaml dans le module `List`.
- Le but est de définir des fonctions sur les listes qu'on pourrait définir par récurrence en utilisant ces schémas.
- Deux types de schémas:
  1. `List.map` : appliquer une fonction à tous les éléments d'une liste.
  2. `List.fold` : traiter les éléments d'une liste un par un en allant de gauche à droite (ou droite à gauche).

Schéma : map

- Modification d'une liste de notes :
  `List.map (fun x-> (if (x<8) then 1 else 0)+x) notes`
- Analyse du schéma `map` :
  Profil: `map : ('a ->'b) -> 'a list -> 'b list`
  Sémantique : `map f l` est la liste contenant tous les éléments de la liste `l` auxquels on a appliqué la fonction `f`. Donc `map f [a1;a2;a3;a4]` est la liste `[ f(a1);f(a2);f(a3);f(a4)]`
  Equations de récurrences :

```
map f [ ] = [ ]
map f x::reste = f(x)::(map f reste)
```

Utilisation du schéma map

- Un map ne change pas la structure de la liste sur lequel il est appliqué (nombre d'éléments et places respectives).

Utilisation du schéma map

- Un map ne change pas la structure de la liste sur lequel il est appliqué (nombre d'éléments et places respectives).
- Ajouter un élément à chaque liste d'une liste de liste.

Utilisation du schéma map

- Un map ne change pas la structure de la liste sur lequel il est appliqué (nombre d'éléments et places respectives).
- Ajouter un élément à chaque liste d'une liste de liste.
- Produire la liste des noms des personnes ayant réussie une UE.

Utilisation du schéma map

- Un map ne change pas la structure de la liste sur lequel il est appliqué (nombre d'éléments et places respectives).
- Ajouter un élément à chaque liste d'une liste de liste.
- Produire la liste des noms des personnes ayant réussie une UE.
- Transformer en minuscule un texte (vu comme une liste de char).
- etc.

Schéma de repliage : fold

Qu'ont en commun ces problèmes suivants ?

- Somme des éléments d'une liste.
- Somme des carrés des éléments d'une liste.
- Produit des éléments d'une liste.
- Existance d'un élément vérifiant une propriété dans une liste ?
- Tous les éléments d'une liste vérifient ils une propriété dans une liste ?

$\Longrightarrow$ dans tous les cas il s'agit d'appliquer une fonction élément par élément, le résultat étant la combinaison de ce qui a été calculé avec le nouvel élément.

Repliage par la droite

- Somme d'une liste $[a_1; a_2; \ldots; a_n]$:

$$+(a_1, +(a_2 \ldots a_{n-1}, +(a_n, 0) \ldots))$$

Repliage par la droite

- Somme d'une liste $[a_1; a_2; \ldots; a_n]$:

$$+(a_1, +(a_2 \ldots a_{n-1}, +(a_n, 0) \ldots))$$

- Produit d'une liste $[a_1; a_2; \ldots; a_n]$:

$$*(a_1, *(a_2 \ldots a_{n-1}, *(a_n, 1) \ldots))$$

Repliage par la droite

- Somme d'une liste $[a_1; a_2; \ldots; a_n]$:

$$+(a_1, +(a_2 \ldots a_{n-1}, +(a_n, 0) \ldots))$$

- Produit d'une liste $[a_1; a_2; \ldots; a_n]$:

$$*(a_1, *(a_2 \ldots a_{n-1}, *(a_n, 1) \ldots))$$

- Un élément dans $[a_1; a_2; \ldots; a_n]$ vérifie-t-il P ?:

$$||(P(a_1), ||(P(a_2) \ldots P(a_{n-1}), ||(P(a_n), true) \ldots))$$

Repliage par la droite

- Somme d'une liste $[a_1; a_2; \ldots; a_n]$:

$$+(a_1, +(a_2 \ldots a_{n-1}, +(a_n, 0) \ldots))$$

- Produit d'une liste $[a_1; a_2; \ldots; a_n]$:

$$*(a_1, *(a_2 \ldots a_{n-1}, *(a_n, 1) \ldots))$$

- Un élément dans $[a_1; a_2; \ldots; a_n]$ vérifie-t-il P ?:

$$||(P(a_1), ||(P(a_2) \ldots P(a_{n-1}), ||(P(a_n), true) \ldots))$$

- Tous les éléments dans $[a_1; a_2; \ldots; a_n]$ vérifient-ils P ?:

$$\&\&(P(a_1), \&\&(P(a_2) \ldots P(a_{n-1}), \&\&(P(a_n), false) \ldots))$$

fold_right Analyse

- Profil :
  `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
- Equations de récurrence :
  - `fold_right f [ ] b = b`
  - `fold_right f x::s b = f (fold_right f s b) x`

fold_right Analyse

- Profil :
  `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

- Equations de récurrence :
  - `fold_right f [ ] b = b`
  - `fold_right f x::s b = f (fold_right f s b) x`

Essentiellement `List.fold_right f` $[a_1; \ldots; a_n]$ `b` est :

$$f(a_1, f(a_2 \ldots a_{n-1}, f(a_n, b) \ldots))$$

fold_right Analyse

- Profil :
  `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

- Equations de récurrence :
  - `fold_right f [ ] b = b`
  - `fold_right f x::s b = f (fold_right f s b) x`

Essentiellement `List.fold_right f` $[a_1; \ldots; a_n]$ `b` est :

$$f(a_1, f(a_2 \ldots a_{n-1}, f(a_n, b) \ldots))$$

fold_right : utilisation

- Application directe : produit, somme, existence d'un élément, tous vérifient ils une propriété ?

fold_right : utilisation

- Application directe : produit, somme, existence d'un élément, tous vérifient ils une propriété ?
- Comptez le nombre de chiffres positifs dans une liste.

fold_right : utilisation

- Application directe : produit, somme, existence d'un élément, tous vérifient ils une propriété ?
- Comptez le nombre de chiffres positifs dans une liste.
- Maximum d'une liste

fold_right : utilisation

- Application directe : produit, somme, existence d'un élément, tous vérifient ils une propriété ?
- Comptez le nombre de chiffres positifs dans une liste.
- Maximum d'une liste
- Trier un liste.

fold_left : replions de la droite vers la gauche

- Profil :
  `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
- Equations de récurrence :
  - `fold_left f b [ ] = b`
  - `fold_left f b x::s = (fold_left f (f x b) s)`

fold_left : replions de la droite vers la gauche

- Profil :
  `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

- Equations de récurrence :
  - `fold_left f b [ ] = b`
  - `fold_left f b x::s = (fold_left f (f x b) s)`

Essentiellement `List.fold_left f` $[a_1; \ldots; a_n]$ `b` est :

$$f(a_n, \ldots f(a1, f(a_0, b)) \ldots)$$

fold_left : replions de la droite vers la gauche

- Profil :
  `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

- Equations de récurrence :
  - `fold_left f b [ ] = b`
  - `fold_left f b x::s = (fold_left f (f x b) s)`

Essentiellement `List.fold_left f` $[a_1; \ldots; a_n]$ `b` est :

$$f(a_n, \ldots f(a1, f(a_0, b)) \ldots)$$

fold_left : replions de la droite vers la gauche

- Séquence des valeurs cumulées :

$$[3; 7; 2; 9; 12] \rightarrow [3; 10; 12; 21; 33]$$

- Raisonnement par récurrence plus naturelle en allant vers la gauche : en prenant les éléments en allant vers la gauche :

$$[3; 7; 2; 9; 12] = [3; 7; 2; 9]@[12]$$

l'appel récursif à valeur cumulés sur la liste à gauche donnerait :

$$[3; 10; 12; 21]$$

il suffit de prendre le dernier élément de lui ajouter 12.

fold_left : replions de la droite vers la gauche

- Séquence des valeurs cumulées :

$$[3; 7; 2; 9; 12] \rightarrow [3; 10; 12; 21; 33]$$

- Raisonnement par récurrence plus naturelle en allant vers la gauche : en prenant les éléments en allant vers la gauche :

$$[3; 7; 2; 9; 12] = [3; 7; 2; 9]@[12]$$

l'appel récursif à valeur cumulés sur la liste à gauche donnerait :

$$[3; 10; 12; 21]$$

il suffit de prendre le dernier élément de lui ajouter 12.

-
```
List.fold_left (fun e -> fun valc -> if (valc= [])
        then [e]
        else valc@[(dernier valc)+e]) [ ] l
```

Plan

A propos de Curryfication

- Exemple : `lef f (x1:int)(x2:int)(x3:int):int = x1+x2+x3;;`
  - `f` est du type "fonction à trois paramètres".
  - le résultat de `(f 1 2 3)` est un entier.

A propos de Curryfication

- Exemple : `lef f (x1:int)(x2:int)(x3:int):int = x1+x2+x3;;`
  - `f` est du type "fonction à trois paramètres".
  - le résultat de `(f 1 2 3)` est un entier.
- Que vaut `(f 1)` ?

A propos de Curryfication

- Exemple : `lef f (x1:int)(x2:int)(x3:int):int = x1+x2+x3;;`
  - `f` est du type "fonction à trois paramètres".
  - le résultat de `(f 1 2 3)` est un entier.
- Que vaut `(f 1)` ?
  c'est une fonction à deux argument dont le résultat est un entier.

A propos de Curryfication

- Exemple : `lef f (x1:int)(x2:int)(x3:int):int = x1+x2+x3;;`
  - `f` est du type "fonction à trois paramètres".
  - le résultat de `(f 1 2 3)` est un entier.
- Que vaut `(f 1)` ?
  c'est une fonction à deux argument dont le résultat est un entier.
  `f x1 x2 ... xn` est en fait une suite d'applications
  `((((f x1) x2) x3) ... xn)`
- Typage :
  - si `f` est de type `t1 -> t2 -> ... -> tn`
  - alors `f x1 x2 ... xj` est de type
    `t(j+1) -> ... -> tn`

Intérêt de la Curryfication

- Soit `f` une fonction qui prend un objet de `A`, un de `B` et rend un résultat de type `C`.
- Deux manières de l'implanter :
  - Sans Curryfication : `f1 : A * B -> C`
  - Avec Curryfication : `f2 : A -> B -> C`
- `f1(a,b);;` et `(f2 a b);;` sont des expressions correctes.
- `f2 a;;` est une expression correcte.
  `f1 a;;` n'est pas une expression correcte.
- La Curryfication permet une application partielle des fonctions, elle est plus souple d'utilisation.

Intérêt de la Curryfication

- Soit `f` une fonction qui prend un objet de `A`, un de `B` et rend un résultat de type `C`.
- Deux manières de l'implanter :
  - Sans Curryfication : `f1 : A * B -> C`
  - Avec Curryfication : `f2 : A -> B -> C`
- `f1(a,b);;` et `(f2 a b);;` sont des expressions correctes.
- `f2 a;;` est une expression correcte.
  `f1 a;;` n'est pas une expression correcte.
- La Curryfication permet une application partielle des fonctions, elle est plus souple d'utilisation.
  Attention à ne pas oublier de paramètre...

Plan