PLAN DU COURS 12

- Evaluation d'une expression arithmétique
- Bilan du cours : « Programmation récursive »
  - Qu'est-ce-que l'informatique ?
  - Quelques connaissances sur les structures de données
  - Un principe de résolution de problème
    - Le mécanisme de récursion
    - La barrière d'abstraction
  - Quelques connaissances sur la programmation
  - Quelques connaissances sur les langages de programmation

EXPRESSION ARITHMÉTIQUE

Les expressions arithmétiques, bien formées, opérateurs `+` et `*`

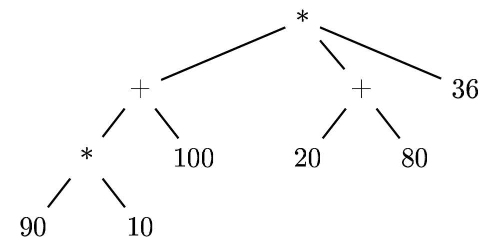
`(((90 * 10) + 100) * (20 + 80) * 36)`

- Visualisation par un arbre général (type `ArbreExpr`)
- Représentation par une S-expression (type `SchemeExpr`)

  `'(* (+ (* 90 10) 100) (+ 20 80) 36)`

- Évaluer (opérateurs `+`, `*` et constantes)
- Évaluer dans un **environnement** (opérateurs `+`, `*`, constantes et variables auxquelles sont associées des valeurs).

VISUALISATION PAR UN ARBRE GÉNÉRAL



DÉFINITION DE evaluationExprArbre

```
;;; evaluationExprArbre : ExprArbre-> Nombre
;;; (evaluationExprArbre G) rend la valeur de l'expression représentée par
G
(define (evaluationExprArbre G)
  (if (ag-feuille? G)
      (ag-etiquette G)
      (let ((LArgs  (map evaluationExprArbre (ag-foret G))))
        (application (operation (ag-etiquette G)) LArgs))))
```

Définition de operation

;;; operation : Symbole -> Opération
;;; (operation operateur) rend la fonction correspondant au symbole «operateur»

```
(define (operation operateur)
  (cond ((equal? operateur '+) +)
        ((equal? operateur '*) *)
        (else (erreur 'operation operateur "pas défini"))))
```

Définition de application

*;;; application : Opération * LISTE[Nombre] -> Nombre*
;;; (application op args) rend le résultat de l'application de l'opération «op»
;;; à la liste d'arguments «args»

```
(define (application op args)
  (if (pair? (cdr args))
      (op (car args) (application op (cdr args)))
      (op (car args))))
```

Définition de evaluationExprS

;;; evaluationExprS : ExprScheme -> Nombre
;;; (evaluationExprS E) rend la valeur de l'expression arithmétique «E»

```
(define (evaluationExprS E)
  (if (list? E)  ; pourquoi avons-nous mis list? et non pair?
      (let ((LArgs  (map evaluationExprS (cdr E))))
        (application (operation (car E)) LArgs))
      E))
```

```
(evaluationExprS '(* (+ (* 90 10) 100) (+ 20 80) 36))
```

```
(* (+ (* 90 10) 100) (+ 20 80) 36)
```

Evaluation dans un environnement

L'environnement est représenté par une liste d'associations

`(Symbole, Nombre)`

Par exemple

```
(define (MonEnv) '((x 3) (y 4) (z 5)))
```

```
(valeur-de 'y (MonEnv)) -> 4
```

La fonction d'évaluation doit donc avoir 2 arguments :

- l'expression elle même
- et la liste d'associations représentant l'environnement.

Spécification et application

*;;; evaluationExpr : ExprScheme * LISTE[N-UPLET[Symbole Nombre]] -> Nombre*
;;; (evaluationExpr E env) rend la valeur de l'expression arithmétique «E»
;;; dans l'environnement «env»

```
(define (MonEnv) '((x 3) (y 4) (z 5)))

(evaluationExpr '(+ x (* 3 y 5) (+ 1 2 y (* z 10)))
                (MonEnv))
```


Définition de evaluationExpr

```
(define (evaluationExpr E env)
  ;; evaluationExpr-env : ExprScheme -> Nombre
  ;; (evaluationExpr-env E) : même sémantique que «evaluationExpr»
  ;; mais fonction rendue unaire pour utilisation dans un map
  (define (evaluationExpr-env E)
    (evaluationExpr E env))

  (if (list? E)
      (let ((LArgs  (map evaluationExpr-env (cdr E))))
        (application (operation (car E)) LArgs))
      (if (symbol? E)
          (valeur-de E env)
          E)))
```

Qu'est-ce-que l'informatique ?

Cours d'introduction à l'informatique et non seulement d'initiation à la programmation.

D'après LE PETIT ROBERT :

- « Informatique : (1962) **Science** du traitement automatique de l'information »
- « Ensemble des **techniques** de la collecte, du tri, de la mise en mémoire, du stockage, de la transmission et de l'utilisation des informations traitées automatiquement à l'aide de programmes (logiciel) mis en œuvre sur ordinateurs. »

Traitement auto. de l'information

Pour manipuler des informations à l'aide d'un ordinateur, il faut

- une **structuration des données**,
- une description précise du traitement à effectuer : c'est l' **algorithme**.
- tout algorithme doit être écrit dans un langage « compréhensible » par l'ordinateur utilisé : c'est le **programme**.

Les structures de données

Au niveau des **concepts**, ont été vu :

- **LISTE** : séquence ordonnée d'éléments de même type.
- **N-UPLET** : structure de données regroupant n éléments.
- **Arbre** : ensemble de nœuds organisés de façon hiérarchique à partir d'un nœud distingué qui est la racine.

Structures de base en Scheme

En **Scheme** ont été vues les

- **données atomiques** (comme nombres, chaînes, booléens etc.) et
- **les données composites** : les listes et les Sexpressions

En Scheme, les listes et les Sexpressions ont été manipulées à l'aide de deux opérateurs le **car** et le **cdr** (que l'on retrouve dans tout langage permettant la manipulation de listes).

Les fonctions de base sur les listes

- Les **constructeurs** permettent de construire une liste
- Les **accesseurs** permettent d'accéder aux éléments de la liste ou à des sous-listes
- Les **reconnaisseurs** permettent de savoir si une valeur est une liste, une liste non vide

3 constructeurs list, cons et append

```
;;; list: alpha *... -> LISTE[alpha]
;;; (list v...) crée une liste dont les termes sont les arguments
;;; (list) rend la liste vide

;;; cons: alpha * LISTE[alpha] -> LISTE[alpha]
;;; (cons v L) rend la liste dont le premier élément est
;;; v et dont les éléments suivants sont les éléments de la liste L.

;;; append: LISTE[alpha] * LISTE[alpha] -> LISTE[alpha]
;;; (append l1 l2) rend la concaténation des listes l1 et l2
```

Les accesseurs d'une liste

Les **accesseurs** permettent d'accéder aux éléments de la structure de données

```
;;; car: LISTE[alpha]/non vide/ -> alpha
;;; (car L) rend le premier élément de la liste donnée
;;; ERREUR lorsque L n'est pas une liste non vide

;;; cdr: LISTE[alpha] ]/non vide/ -> LISTE[alpha]
;;; (cdr l) rend la liste des termes de L sauf son premier élément.
;;; ERREUR lorsque L n'est pas une liste non vide
```


Les reconnaisseurs d'une liste

Les **reconnaisseurs** permettent de savoir quelle est la forme d'une donnée structurée.

- Pour savoir si une valeur est une liste : le prédicat `list?`

```
;;; list?:Valeur -> bool
;;; (list? v) rend vrai ssi v est une liste
;;; (éventuellement vide).
```

- Pour savoir si une valeur est une liste non vide : le prédicat `pair?`

```
;;; pair?: valeur -> bool
;;; (pair? v) rend vrai ssi v a un car et un cdr,
;;; c'est à dire ssi v est une liste non vide.
```

Autres structures de données

Nous avons introduit d'autres structures de données :

- les lignes et paragraphes
- les arbres binaires
- les arbres généraux

On manipule ces structures de données uniquement à travers leur **barrière d'abstraction** :

- on ne vous rend visible que la **spécification** des fonctions ;
- on ne vous montre pas l'implantation de ces fonctions.

Barrière d'abstraction

Spécification du constructeur :

```
;;; ag-noeud: α * Foret[α] -> ArbreGeneral[α]
;;; où Foret[α] est du type LISTE[ArbreGeneral[α]]
;;; (ag-noeud e foret) rend l'arbre formé de la racine d'étiquette «e» et,
;;; comme sous-arbres immédiats, les arbres de la forêt «foret».
```

Spécification des accesseurs :

```
;;; ag-etiquette: ArbreGeneral[α] -> α
;;; (ag-etiquette g) rend l'étiquette de la racine de l'arbre «g».

;;; ag-foret: ArbreGeneral[α] -> Foret[α]
;;; où Foret[α] est du type LISTE[ArbreGeneral[α]]
;;; (ag-foret g) rend la forêt des sous-arbres immédiats de «g».
```

Principe de résolution de problème

- Un **principe** : un problème, non trivial, pour pouvoir être résolu, est décomposé (cassé) en sous-problèmes bien identifiés.
- Les sous-problèmes sont résolus suivant deux **règles** :
  - Première règle : essayer de résoudre chaque problème de façon simple, combinable et sans exception.
  - Seconde règle : essayer de résoudre 2 sous-problèmes distincts de façon indépendante

Le mécanisme de récursion

- **naturels**
  Ramener le pb à un n strictement plus petit et traiter les cas de base (en général 0 ou 1)
- **listes**
  Ramener le pb à une liste strictement plus petite (en général le `cdr`) et traiter les cas de base (en général la liste vide ou la liste à un élément)
- **arbres**
  Ramener le pb à des arbres strictement plus petits (les sous-arbres) et traiter les cas de base (en général soit un arbre vide soit une feuille).

Le schéma récursif

Le schéma récursif est toujours sous la même forme :

- décomposer le pb initial en pbs plus petits
- traiter les cas de bases

```
(define (f var)
   (if (trivial? var)
       (cas-de-base var)
       (appliquer-composer (f (sous-termes var))) ) )
```

Récursion sur les entiers

```
;;; puissance: nbre * nat -> nat
;;; HYPOTHESE x est un nombre, et m est un entier positif ou nul
;;; (puissance x m) rend x a la puissance m
(define (puissance x m)
  (if (= m 0)
      1
      (* x (puissance x (- m 1)))))

(define (puissance x m)
  (if (= m 0)
      1
      (if (even? m)
          (carre (puissance x (quotient m 2)))
          (* x (carre (puissance x (quotient m 2)))))))
```


Récursion sur les listes

```
;;; somme-carres: LISTE[Nombre] -> Nombre
(define (somme-carres L)
  (if (pair? L)
      (+ (carre (car L))  (somme-carres (cdr L)))
      0))

;;; liste-carres: LISTE[Nombre] -> LISTE[ Nombre]
(define (liste-carres L)
  (if (pair? L)
      (cons (carre (car L))  (liste-carres (cdr L)))
      (list)))
```

Les fonctionnelles : filtre, ...

```
;;; map: (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
(define (map fn L)
  (if (pair? L)
      (cons (fn (car L)) (map fn (cdr L)))
      (list)))

;;; reduce: (alpha * beta -> beta)  *  beta * LISTE[alpha] -> beta
(define (reduce fn base L)
  (if (pair? L)
      (fn (car L)
          (reduce fn base (cdr L)))
      base))
```


Récursion sur les arbres binaires

```
;;; nombre-noeuds :  ArbreBinaire[α] -> nat
;;; (nombre-noeuds B) rend le nombre de noeuds de B
(define (nombre-noeuds B)
  (if (ab-noeud? B)
      (+ 1
         (nombre-noeuds (ab-gauche B))
         (nombre-noeuds (ab-droit B)))
      0))
```

Récursion sur les arbres généraux

```
;;; nombre-noeuds-arbre : ArbreGeneral[α] -> nat
;;; (nombre-noeuds-arbre G) rend le nombre de noeuds de G
(define (nombre-noeuds-arbre G)
  ;; nombre-noeuds-foret : Foret[α] -> nat
  ;; (nombre-noeuds-foret F) rend le nombre de noeuds de F
  (define (nombre-noeuds-foret F)
    (if (pair? F)
        (+ (nombre-noeuds-arbre (car F))
           (nombre-noeuds-foret (cdr F)))
        0))

  (+ 1 (nombre-noeuds-foret (ag-foret G))))
```


Avec reduce et map

```
;;; nombre-noeuds-arbre: ArbreGeneral[α] -> nat
;;; (nombre-noeuds-arbre G) rend le nombre de noeuds de G

(define (nombre-noeuds-arbre G)
  (+ 1 (reduce + 0 (map nombre-noeuds-arbre (ag-foret G)))))
```

Programmation

En plus de l'initiation à la programmation (Scheme) vous avez acquis des connaissances sur la programmation.

Il faut arriver à écrire simple, efficace, esthétique, pas pour l'ordinateur, mais pour les humains.

- Spécification
- Barrière d'abstraction/interface
- Grammaire
- Efficacité
- Esthétique

Spécif. et barrière d'abstraction

- La spécification permet de décrire ce que la fonction fait (et non la façon dont elle le fait).
- La barrière d'abstraction (interface) correspond à la spécification pour un groupe de fonctions et de données.
- La spécification et la barrière d'abstraction permettent
  - d'augmenter le niveau d'abstraction,
  - de rendre les problèmes triviaux du côté utilisation,
  - de masquer ce qui est fait du côté concepteur (pbs plus complexes).
- La simplicité actuelle dans l'utilisation des ordinateurs repose sur une complexité extraordinaire en dessous.

Utilisation d'une grammaire

- La grammaire permet, avec un nombre fini de règles de génération, de générer un ensemble infini de phrases.
- Elle est importante pour décrire des données et des langages de programmation.

Complexité des algorithmes

Chaque fois que cela était possible, nous vous avons montré des possibilités d'améliorer l'efficacité, soit au niveau de la méthode utilisée soit au niveau de l'implantation, par exemple :

- dans la factorielle avec détection d'erreur,
- dans le calcul de la puissance,
- en limitant l'utilisation de la fonction `length` pour une liste,
- dans le calcul de la somme cumulée d'une liste,
- dans l'utilisation d'un arbre binaire de recherche,
- ...

Langages de programmation

- A priori, tous les langages permettent de résoudre les mêmes problèmes.
- Mais les primitives d'un langage sont fournies en fonction du domaine
- Les langages sont donc faits pour traiter un certain type de problème :
  - Ainsi Maple est fait pour manipuler des objets mathématiques.
  - Il existe des langages pour le dessin, d'autres pour dessiner des polices de caractères.
  - Scheme offre la manipulation des listes et des arbres et non des polices de caractères.

Syntaxe/sémantique

- Syntaxe/sémantique (Signifié/Signifiant)
- Il faut arriver à sortir le problème que l'on a dans la tête par un encodage :
  - dans la sémantique on considère les concepts, le sens qu'on leur attribue,
  - dans la syntaxe, la représentation des concepts.

Famille de langages

- Famille **fonctionnelle**

  On travaille en composant des fonctions

  - Scheme en fait partie (il est proche des maths).
  - Une autre catégorie de langages fonctionnels : ML où l'on traite des types

- Famille **impérative**

  On travaille en séquençant des instructions, par transformations d'état pour passer d'un état à l'autre

  - Pour ces langages, il est difficile de faire des preuves de programmes
  - Ils sont plus efficaces car ils ne cachent pas la machine