PLAN DU COURS 2

« Programmation récursive »

- Spécification et définition d'une fonction
- Évaluation d'une application
- Forme spéciale `if`
- Formes spéciales `and` et `or`
- Forme spéciale `let`
- Travail avant le prochain TD/TP

FONCTION : SPÉCIFICATION ET DÉFINITION

1. **Spécification** (pour l'humain donc en commentaire)
   (a) **Signature** :
   - nom de la fonction
   - type des arguments
   - type du résultat

   (b) **Sémantique**
   - **QUOI**
   - ce que la fonction fait (texte en français)
2. **Définition** (pour la machine)
   - **COMMENT**
   - implantation, calcul effectué (code en Scheme)

SPÉCIFICATION ET DÉFINITION : EXEMPLE

;;; *moyenneDe3nombres: Nombre * Nombre * Nombre -> Nombre*
;;; *(moyenneDe3nombres x y z) rend la moyenne arithmétique de x, y et z*

Une première définition :

```
(define (moyenneDe3nombres x y z)
  (/ (+ x y z)
     3))
```

Une autre définition :

```
(define (moyenneDe3nombres x y z)
  (+ (/ x 3)
     (/ y 3)
     (/ z 3)))
```


APPLICATION DE FONCTION

1. Appliquer la fonction avec des arguments particuliers

```
(moyenneDe3nombres 13 18 14)
(moyenneDe3nombres (+ 4 5) 5 (- 8 1))
```

2. Évaluer une application
   - évaluer chacun des arguments,
   - évaluer l'application de la fonction aux valeurs obtenues.

```
(moyenneDe3nombres (+ 4 5)
                   (moyenneDe3nombres 5 4 6)
                   (- 8 1))
(moyenneDe3nombres (+ 4 5) (/ 5 0)(- 8 1))   !! ERREUR
```

Exemple-DEMO : moyenne.scm

Fonctions / contre / Formes spéciales

Une expression Scheme est une composition d'applications fonctionnelles

- **Évaluation d'une application**
  - évaluer chacun des arguments,
  - évaluer l'application de la fonction aux valeurs obtenues.
- **SAUF pour les FORMES SPÉCIALES : Évaluations spéciales**
  - la forme spéciale `if` : *Alternative*
  - les formes spéciales `and` et `or` : *Connecteurs*
  - la forme spéciale `let` : *Bloc lexical*

La syntaxe d'une alternative

Règle de grammaire :

<alternative> →

`(if` *<condition>* *<conséquence>* *<alternant>* `)`

La *condition* est une expression ayant pour valeur

- soit vrai `#t`
- soit faux `#f`

La *conséquence* et l'*alternant* sont des expressions.

Évaluation d'une alternative

L'alternative est une **forme spéciale**

c.a.d., a une règle d'évaluation particulière

- la condition est évaluée (retourne Vrai ou Faux)
- ensuite, selon le résultat,
  - SOIT la conséquence est évaluée
  - SOIT l'alternant est évalué

```
(if (> 3 2) (+ 3 2) (- 3 2))

(if (> 2 3) (+ 3 2) (- 3 2))
```


Exemple 1

```
(define (mystere x y z)
  (if (> x 100.0)
      (/ x y)
      (/ x z) ) )
```

`(mystere 400 2 4)` → *200*

`(mystere 80 2 4)` → *20*

`(mystere 400 2 0)` → *???*

`(mystere 80 2 0)` → *???*

Exemple 2

Spécification de la fonction `valeur-absolue`

```
;;; valeur-absolue: Nombre -> Nombre
;;; (valeur-absolue x) rend la valeur absolue de x
```

Une première définition

```
(define (valeur-absolue x)
  (if (>= x 0)
      x
      (- x) ) )
```

Une autre définition

```
(define (valeur-absolue x)
  (if (negative? x )
      (- x)
      x ) )
```


Applications de l'exemple 2

```
(valeur-absolue 5)
(valeur-absolue -5)
(valeur-absolue (- 5))
```


Condition

Les expressions conditionnelless sont construites avec les opérations logiques de négation, de conjonction et de disjonction.

En Scheme :

- Négation `not` : une *fonction prédéfinie*, une primitive
- Conjonction `and` : une *forme spéciale*, cf. la grammaire
- Disjonction `or` : une *forme spéciale*, cf. la grammaire

```
; autre définition de valeur-absolue
(define (valeur-absolue x)
  (if (not (negative? x ) )
      x
      (- x) ) )
```


and et or : des formes spéciales

- Syntaxe (les règles de grammaire)

  <*conjonction*> → `(and` <*expression*>*`)`

  <*disjonction*> → `(or` <*expression*>*`)`

- Évaluation de `and` et `or`

  `(and` *e1 e2 e3 ...*`)`

  `(or` *e1 e2 e3 ...*`)`

Évaluation d'une conjonction

```
;;; number?: Valeur -> bool
        (number? 32.45)          → #t
        (number? "essai")        → #f

;;; positive?: Nombre -> bool
        (positive? 32.45)        → #t
        (positive? "essai")      → erreur

;;; nbre-positif?: Valeur -> bool
        (nbre-positif? 32.45)    → #t
        (nbre-positif? "essai")  → #f
```


Spécif. et définition de nbre-positif?

Une autre définition (fausse)

```
(define (nbre-positif? v)                    ; FAUX
   (and (positive? v) (number? v)) )

(nbre-positif? 32.45)

(nbre-positif? "essai") ; → non pas #f mais une erreur
```


Le bloc let : une autre forme spéciale

Théorème :

Étant donné un triangle quelconque de côtés a, b et c, en nommant s la valeur $(a+b+c)/2$, son aire est égale à la racine carrée de $s(s-a)(s-b)(s-c)$

```
;;;  aire-triangle : Nombre * Nombre * Nombre -> Nombre
;;;  (aire-triangle a b c) rend l'aire d'un triangle de cotés a, b et c
(define (aire-triangle a b c)
  (let ((s (/ (+ a b c ) 2)))
    (sqrt (* s (- s a) (- s b) (- s c)))))
```


La syntaxe d'un let

<bloc> → `(let (` *<liaison>** `)` *<corps>* `)`

<liaison> → `(` *<variable>* *<expression>* `)`

<corps> → *<expression>*

L'écriture d'un `let` :

```
(let ((var1 expr1)
      (var2 expr2)
      ...
      (varN exprN) )
  corps )
```

L'Évaluation d'un `let`

Pour évaluer un `let` :

- **évaluation** des expressions *expr1*, ..., *exprN*
- **création** des variables, *var1*, ..., *varN*
- **enrichissement** de l'environnement courant en associant à chaque variable *varI* la valeur de *exprI*
- **évaluation** du *corps* dans cet environnement.

Portée des variables : corps du `let`

Des `let` imbriqués

```
(let ((a 2))
  (let ((a 3)
        (b (* a 2) ) )
    b ) )
```

Quel est le résultat de l'évaluation ?

Détection d'erreur

```
;;; aire-couronne : Nombre/>=0/ * Nombre/>=0/ -> Nombre
;;; ERREUR lorsque r1 < r2
;;; (aire-couronne r1 r2) rend l'aire de la couronne de rayon extérieur r1
;;; et de rayon intérieur r2
(define (aire-couronne r1 r2)
  (if (< r1 r2)
      (erreur 'aire-couronne
              "rayon extérieur (" r1 ") <"
              "rayon intérieur (" r2 ")")
      (- (aire-disque r1) (aire-disque r2))))
```

La détection d'erreur a un coût.

La notion d'hypothèse

```
;;; aire-couronne-sans : Nombre/>=0/ * Nombre/>=0/ -> Nombre
;;; HYPOTHÈSE : r1 >= r2
;;; (aire-couronne-sans r1 r2) rend l'aire de la couronne de rayon extérieur
;;; r1 et de rayon intérieur r2
(define (aire-couronne-sans r1 r2)
  (- (aire-disque r1) (aire-disque r2)))
```

Vocabulaire pour une fonction

Une fonction comprend :

- Une **spécification**, qui comprend elle-même :
  - sa **signature** qui indique
    - le **nom** de la fonction suivi par
    - le **type** de la fonction (type de ses arguments et de son résultat)
  - sa **sémantique** qui décrit ce que la fonction fait (et non comment elle le fait)
- Une **définition** (implantation)

Travail avant le prochain TD/TME

- Reprendre vos notes
- Lire le cours sur Internet (chez vous ou à l'UTES) ou sur le CDROM
  - Alternative
  - Nommage de valeurs
  - Spécification d'un problème
- Faire les exercices d'auto-évaluations de chaque partie de cours
- Être capable d'écrire, sans l'aide des notes et du cours, la spécification et la définition de :
  - `valeur-absolue`
  - `nbre-positif?`
  - `aire-triangle`
  - `aire-couronne`