LI101 : Programmation Récursive

©Equipe enseignante Li101

Université Pierre et Marie Curie

Semestre : Automne 2012

Cours 2 : Premiers pas

Plan du cours

Spécification et définition : rappel

Spécification :

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

Une spécification : deux définitions

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

Conforme à la spécification

- nom de la fonction
- nombre d'argument
- types des arguments
- respect des cas d'erreur ou hypothèses

$\Rightarrow$ **résultat correct**

Des exemples corrects :

```
(moyenneDe3nombres 13 18 14)
(moyenneDe3nombres (+ 4 5) 5 (- 8 1))
(moyenneDe3nombres (+ 4 5) (moyenneDe3nombres 13 18 14) 7)
```

Des exemples incorrects :

```
(moyenneDe3nombre 13 18 14)
(moyenneDe3nombres 1 2)
(moyenneDe3nombres 13 18 "toto")
```

Vocabulaire pour une application

Dans l'application `(* (+ 1 2) 3)` :

- le *paramètre en position fonctionnelle* est `*`
- la *fonction* est la multiplication
- les *paramètres d'appel* sont les expressions `(+ 1 2)` et `3`
- les *arguments* de la multiplication sont les valeurs 3 et 3
- la *valeur de l'application* est 9

Évaluation

Les étapes d'un calcul

Soit l'expression : $\frac{5(49-2)}{13}$

– Evaluer le numérateur $5(49-2)$
  – Evaluer le premier facteur 5 (immédiat)
  – Evaluer le second facteur $49-2$
    – Evaluer le premier terme 49 (immédiat)
    – Evaluer le second terme 2 (immédiat)
    = Appliquer la soustraction : $49-2=47$
  = Appliquer la multiplication : $5 \times 47 = 235$
– Evaluer de dénominateur 13 (immédiat)
= Appliquer la division :
  valeur = $235/13 = 18.076923076\ldots$ (valeur approchée)
  ou valeur = $235/13 = 18$ (division entière)

Principe d'évaluation en Scheme

3 types d'expressions en Scheme

- les constantes
- les applications de fonction
- les formes spéciales
  - la forme spéciale `if` : *Alternative*
  - les formes spéciales `and` et `or` : *Connecteurs*
  - la forme spéciale `cond` : *Conditionnelle*

3 principes d'évaluation

- les expressions simples s'évaluent en elle-même : `12` →12
- les applications de fonctions s'évaluent selon un principe de substitution
- les formes spéciales ont chacune une règle d'évaluation qui leur est propre

Évaluation d'une application de fonction

Règle d'évaluation d'une application

- Évaluer chacun des arguments
- Remplacer l'appel de la fonction par le corps de la fonction en remplaçant chaque paramètre par la valeur de l'argument correspondant
- Évaluer l'expression obtenue jusqu'à aboutir à une expression simple (valeur)

Exemple :

```
(moyenneDe3nombres (+ 4 5) 11 (- 8 1))
→  (moyenneDe3nombres 9 11 7)
→  (/ (+ 9 11 7) 3)
→  (/ 27 3)
→  9
```

La syntaxe d'une alternative

Règle de grammaire :

```
<forme-spéciale> ::=
     (if <expression> <expression> <expression> )
```

La 1ère <expression> est la *condition* ; elle a pour valeur

- soit vrai #t
- soit faux #f

La 2ème <expression> est la *conséquence*
La 3ème <expression> est l'*alternant*.

Évaluation d'une alternative

L'alternative est une **forme spéciale**, elle a une **règle d'évaluation** particulière.

- Évaluer la condition
- Si la valeur résultante est #t alors évaluer la conséquence uniquement, la valeur résultante est la valeur de l'alternative
- Sinon évaluer l'alternant uniquement, la valeur résultante est la valeur de l'alternative

`(if (> 3 2) (+ 3 2) (- 3 2))` → `(+ 3 2)` → `5`
`(if (> 2 3) (+ 3 2) (- 3 2))` → `(- 3 2)` → `1`

Alternative : structure de contrôle

Un seul de la *conséquence* ou de l'*alternant* est évalué
Soit la fonction `mystere` suivante :

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

Quel est le résultat de chacune de ces applications ?
`(mystere 400 2 4)` → 200
`(mystere 80 2 4)` → 20
`(mystere 400 2 0)` → 200
`(mystere 80 2 0)` → *Erreur de primitive '/' : Division par zéro*

Forme spéciale *vs* application

Forme spéciale ≠ application

Soit

```
(define (si c e1 e2) (if c e1 e2))
```

- pour évaluer l'application de `si`, les arguments sont évalués **tous les trois**
- pour évaluer la forme spéciale `if` seuls **deux** arguments sont évalués : le premier (la condition) ; puis le second *ou* le troisième

Questions : que fait l'évaluation

- de `(if #t (/ 6 2) (/ 6 0))` ?
- et de `(si #t (/ 6 2) (/ 6 0))` ?

Prédicat

Une *condition* est un **prédicat** (résultat booléen)

Exemple :

```
;;; negative?: Nombre -> bool
;;; (negative? x) vérifie que x est strictement négatif
```

Utilisation : la valeur absolue d'un nombre

```
;;; valeur-absolue: Nombre -> Nombre
;;; (valeur-absolue x) rend la valeur absolue de x
(define (valeur-absolue x)
  (if (negative? x)
      (- x)
      x
  )
)
```

Négation d'un prédicat

```
;;; not: bool -> bool
;;; (not b) rend la négation de b
```

Une autre définition de la valeur absolue : ≪inverser≫ le test

```
(define (valeur-absolue-alt x)
  (if (not (negative? x))
      x
      (- x) )
)
```

Même fonction ?

```
(valeur-absolue 5) → 5
(valeur-absolue-alt 5) → 5
(valeur-absolue -5) → 5
(valeur-absolue-alt -5) → 5
(valeur-absolue 0) → 0
(valeur-absolue-alt 0) → 0
```

Conjonction et disjonction de prédicats

Les **formes spéciale** `and` et `or`
Syntaxe : règles de grammaire

```
<forme-spéciale> ::=
    (if ... ) ou
    (and <expression>* ) ou
    (or <expression>* )
```

Structure des formes spéciale `and` et `or` : ressemble à une application

- `(and e1 e2 e3 ... en)`
- `(or e1 e2 e3 ... en)`

MAIS forme spéciale ⇒ **règle d'évaluation particulière**

Règles d'évaluation du `and` et du `or`

Pour évaluer `(and e1 e2 e3 ... en)` :

- évaluer l'argument courant (on commence à gauche), puis
- si le résultat est `#f` alors arrêter d'évaluer et donner le résultat `#f`
- sinon,
  - si l'argument courant est le dernier, `en`, alors donner la valeur de `en`
  - sinon, passer à l'argument suivant

Pour évaluer `(or e1 e2 e3 ... en)` :

- évaluer l'argument courant (on commence à gauche), puis
- si le résultat est `#f` alors
  - si l'argument courant est `en`, alors donner la valeur de `en`
  - sinon, passer à l'argument suivant
- sinon, arrêter et donner la valeur de l'argument courant

Conjonction ; exemple

On a les deux prédicats :

```
;;; number?: Valeur ->  bool
;;; (number? v) reconnaît si v est un nombre

;;; positive?: Nombre -> bool
;;; (positive? x) vérifie que x est strictement positif
```

Remarque : l'argument de `positive?` doit être un nombre, sinon : erreur

On veut définir la fonction suivante telle que :

```
;;; nbre-positif?: Valeur -> bool
;;; (nbre-positif? v) reconnaît que v est un nombre
;;; strictement positif
```

Fonction `nbre-positif?` 1er essai

Une définition

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

Quelques tests :
`(nbre-positif? 32.45)` → `#t`
`(nbre-positif? -1 )` → `#f`
`(nbre-positif? "douze")` → *Erreur de primitive 'positive ?' : J'attends un nombre réel*

Résultat **non conforme** à la spécification ! !

Cherchez l'erreur...

Fonction `nbre-positif?` révisitée

Règle d'évaluation du `and` !

Une définition correcte :

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

Jeu de tests
`(nbre-positif? 32.45)` → `#t`
`(nbre-positif? -1)` → `#f`
`(nbre-positif? "douze")` → `#f`

Alternatives imbriquées

On veut une fonction `signe` qui étant donné un nombre `x` renvoie :

- `-1` si `x` est négatif
- `0` si `x` est nul
- `1` si `x` est positif

La description suggère l'utilisation de la forme `if` :
si . . ., si . . ., si . . .

Analyser et comprendre la description pour la traduire avec des formes `if`

Alternatives imbriquées (suite)

On sait comparer des nombres à zéro :

`(< x 0)`, `(= x 0)`, `(> x 0)`

Organiser les comparaisons :

- Mauvaise expression (trop littérale) :

```
(if (< x 0) -1
    (if (= x 0) 0
        (if (> x 0) 1) ) )
```

  Un `if` sans *alternant* `(if (> x 0) 1)`

  Un test *inutile* : en effet, si `(< x 0)` est faux et si `(= x 0)` est faux, alors `(> x 0)` est *nécessairement vrai*.

- Meilleure définition :

```
(if (< x 0) -1 (if (= x 0) 0 1))
```

La conditionnelle : forme cond

- L'imbrication des `if` peut vite devenir illisible : **source d'erreurs**.
- La forme `cond` simplifie l'écriture de code correspondant à des alternatives imbriquées.

Syntaxe :

```
<forme-spéciale> ::=
     ... ou
     (cond <clauses> )

<clauses> ::=
     <clause>* ou
     <clause>* (else <expression> )
<clause> ::=
     ( <expression> <expression> )
```

La conditionnelle (suite)

Forme générale de la conditionnelle :

```
(cond (c1 e1) ... (cn en) (else e))
```

Application à la définition de `signe` :

```
;;; signe : Nombre -> int
;;; (signe x) rend -1, 0 ou 1 si x est respectivement,
;;; strictement négatif, nul ou strictement positif
(define (signe x)
   (cond ((< x 0) -1)
         ((= x 0) 0)
         (else 1) ) )
```

Évaluation d'un cond

Pour évaluer `(cond (c1 e1) ... (cn en) (else e))` :

- évaluer `c1` : si le résultat **n'est pas** `#f` alors donner la valeur de `e1` ;
- sinon, ...
- sinon, évaluer `cn` : si le résultat **n'est pas** `#f` alors donner la valeur de `en` ;
- sinon, donner la valeur de `e`.

⇒ **Attention à l'ordre des conditions !**

Travail avant le prochain TD/TP

- Reprendre vos notes
- Points traités :
  - Alternative
  - Spécification d'un problème
  - Règles d'évaluation
  - Formes spéciales `and`, `or`, `cond`
- Être capable d'écrire, sans l'aide des notes et du cours, la spécification et la définition de :
  - `valeur-absolue`
  - `nbre-positif?`
  - `signe`
  - `aire-couronne`