

LI101 : Programmation Récursive

©Equipe enseignante Li101

Université Pierre et Marie Curie

Semestre : Automne 2013

Cours 1 : Introduction

Plan du cours

- 1 Introduction / Présentation de l'UE
- 2 Objectifs de l'UE
- 3 Expressions
- 4 Fonctions : définition, application
- 5 Valeurs et types
- 6 Spécification et définition
- 7 Plus d'expression : l'alternative

Qu'est-ce que l'informatique ?

« *L'informatique n'est pas plus la science des ordinateurs que l'astronomie n'est celle des télescopes* » (Edsger W. Dijkstra)

Définition : science du traitement automatisé de l'information

- **Science** : « Hiérarchisation, organisation et synthèse des connaissances au travers de principes généraux (théories, lois, etc.) » donc un aspect théorique
- **Traitement automatisé** : transformation (dictée par un programme) faite par l'ordinateur
- **Information** : les données cibles du traitement, peut être tout ce qui est *numérisable* (texte, musique, voix, image, films, ADN, ...)

L'informatique regroupe un grand nombre de domaines : théorie, langage et programmation, architecture et système, réseau, sécurité, base de données, intelligence artificielle, etc.

Présentation de l'UE

UE d'initiation à la programmation

Il y a différents paradigmes de programmation :

- impératif (C, FORTRAN...)
- fonctionnel (Lisp, Scheme, Caml, Haskell)
- logique (Prolog,...)
- orienté objet (C++, Java, Python, ...)
- ...

Dans ce cours : programmation fonctionnelle

Objectifs de l'UE

Une initiation à la programmation

- ① Programmation *fonctionnelle*
- ② Écriture de *fonctions récursives*
 - Récursion sur les entiers
 - Récursion sur les listes
 - Récursion sur des structures d'arbres

Langage support : Scheme

Environnement de programmation : MrScheme

Expressions

En mathématique comme en informatique

- **formule** désignant une **valeur**
- **formule** décrivant le calcul d'une **valeur**
- composition **d'opérateurs** et **d'opérandes** décrivant un calcul
- composition d'applications de **fonctions** à leurs arguments décrivant un calcul

Deux éléments :

- lecture/écriture de l'expression : **syntaxe**
- obtenir le résultat/valeur : **évaluation**

Lecture et valeur d'une expression

Soit $(7a - f(2))(2b + 1)$

Pour évaluer, il faut savoir lire : **syntaxe**

- 1 Opérateurs implicites : la multiplication

$$(7 \times a - f(2)) \times (2 \times b + 1)$$

- 2 Parenthésage implicite : priorité des opérateurs

$$((7 \times a) - f(2)) \times ((2 \times b) + 1)$$

Pour évaluer, il faut aussi connaître

- 1 le rôle des opérateurs/fonctions : $\times, +$

- 2 la valeur des inconnues/*variables*

- numériques : a, b
- fonctionnelles : f

Écriture **S** d'une expression

Notez le pluriel.

- Notation **infixe** (usuelle en arithmétique)
 - Opérateur *entre* les arguments : $1 + 2 \times 3$
 - Ambiguïté :
réglée par la priorité des opérateurs : d'abord \times , puis $+$;
ou les parenthèses : $1 + (2 \times 3)$
- Notation **préfixe**
 - Opérateurs *avant* les arguments : $+ 1 \times 2 3$
 - Non ambiguë : **arité** des opérateurs connue
 $+ \times 2 3 1 = + (\times 2 3) 1$
- Notation **postfixe**
 - Opérateurs *après* les arguments : $1 2 3 \times +$
 - Non ambiguë : $1 2 3 \times + = 1 (2 3 \times) +$

Expressions en Scheme

Choix d'une notation

Une écriture *minimaliste et régulière* :

- Notation **préfixe**
- Notation **complètement parenthésée**
- Composants séparés par des espaces, tabulation ou retour à la ligne

La formule $(7x - f(2))(2x + 1)$ s'écrit en Scheme :

```
.....
(* (- (* 7 x) (f 2))
   (+ (* 2 x) 1))
.....
```

En Scheme, la multiplication se note *****

Règles de conversion en notation préfixe

Deux cas peuvent se produire :

- L'expression est simple : l'écrire telle quelle
- L'expression est composée :
 - 1 Chercher l'opérateur à effectuer en dernier
 - 2 Écrire une parenthèse ouvrante (
 - 3 Écrire le nom de l'opérateur suivi d'un espace
 - 4 Convertir chacun des arguments de l'opérateur en suivant les règles de conversion (en reprenant à l'étape 1 pour chaque argument) et les séparer par des espaces
 - 5 Écrire une parenthèse fermante)

Question : convertir $3(2 + 1)^2 + 2(4 - 1), (7x - 2)(2x - 1)$

Expressions en Scheme

Définition et syntaxe

Définition d'un premier *ensemble* d'expressions

- ① les **constantes** sont des expressions.
- ② les **variables** sont des expressions.
- ③ les **applications** d'un symbole de fonction **f** aux expressions e_1, \dots, e_n sont des expressions.

Notez ici la définition *réursive*.

Syntaxe, grammaire :

$\langle \text{expression} \rangle ::= \text{variable ou constante}$
 ou $\langle \text{application} \rangle$

$\langle \text{application} \rangle ::= (\text{nom-fonc } \langle \text{expression} \rangle^*)$

Notez ici encore la définition *réursive*.

Attention :

syntaxe de l'application

Fonction vs expression

La valeur de $(* (- (* 7 x) 2) (+ (* 2 x) 1))$

- dépend de la valeur de x ;
- est *fonction* de la valeur de x .

Définir une fonction : *forme spéciale* **define**

```
(define (h x)
  (* (- (* 7 x) 2) (+ (* 2 x) 1)))
```

Pose que : pour toute valeur de x ,

la valeur de *l'application* $(h x)$ est égale à
la valeur de $(* (- (* 7 x) 2) (+ (* 2 x) 1))$

Exemple :

```
(h 42) = (* (- (* 7 42) 2) (+ (* 2 42) 1))
      = (* (- 294 2) (+ 84 1))
      = (* 292 85)
      = 24820
```

Valeurs de base en Scheme

Résultats possibles de l'évaluation d'une expression

- les valeurs booléennes : `#t`, `#f`
- des valeurs entières : `42`, `-5`, etc.
- des valeurs réelles (*flottants*) : `2.3`, `3.141592653589793`, etc.
- des chaînes de caractères : `"toto"`, `"une chaine"`, `"2008"`, etc.

Valeur d'une expression

On a vu : l'application `(h 42)` a pour valeur l'entier `24820`.

Mais *quid* de l'application : `(h "bonjour")` ?

→ **Erreur de primitive '*'** : *J'attends un nombre*

Pour toute valeur de `x` qui est un **nombre**

la valeur de `(h x)` est égale à
la valeur de `(* (- (* 7 x) 2) (+ (* 2 x) 1))`

et c'est un **nombre**.

En résumé : **si** `x` est un nombre **alors** `(h x)` est un nombre.

Types

Chaque valeur appartient à un **type**.

Notations pour les types de base :

- **bool** pour les booléens ;
- **nat** pour les entiers naturels (positifs) ;
- **int** pour les entiers signés (positifs ou négatifs) ;
- **Nombre** pour les nombres quelconques (réels, entiers ou relatifs) ;
- **string** pour les chaînes de caractères.

Les fonctions ont aussi un type : on note,

$h: \text{Nombre} \rightarrow \text{Nombre}$

C'est une partie de la **spécification** des fonctions.

Spécification

Pour pouvoir utiliser une fonction, il faut connaître sa **spécification**

- sa **signature**
 - son nom
 - son type
- ce qu'elle calcule et la signification de ses variables
- les indications éventuelles d'hypothèses ou d'erreur

Spécification

Pour pouvoir utiliser une fonction, il faut connaître sa **spécification**

- sa **signature**
 - son nom
 - son type
- ce qu'elle calcule et la signification de ses variables
- les indications éventuelles d'hypothèses ou d'erreur

```
.....  
;;; quotient: int*int -> int  
;;; (quotient a b) rend le quotient de la  
;;; division euclidienne de a par b.  
;;; ERREUR lorsque b est égal à 0  
.....
```

(extrait de la *carte de référence*)

Spécification

Pour pouvoir utiliser une fonction, il faut connaître sa **spécification**

- sa **signature**
 - son nom : **quotient**
 - son type
- ce qu'elle calcule et la signification de ses variables
- les indications éventuelles d'hypothèses ou d'erreur

```
.....  
;;; quotient: int*int -> int  
;;; (quotient a b) rend le quotient de la  
;;; division euclidienne de a par b.  
;;; ERREUR lorsque b est égal à 0  
.....
```

(extrait de la *carte de référence*)

Spécification

Pour pouvoir utiliser une fonction, il faut connaître sa **spécification**

- sa **signature**
 - son nom : `quotient`
 - son type : `int*int -> int`
- ce qu'elle calcule et la signification de ses variables
- les indications éventuelles d'hypothèses ou d'erreur

```
.....  
;;; quotient: int*int -> int  
;;; (quotient a b) rend le quotient de la  
;;; division euclidienne de a par b.  
;;; ERREUR lorsque b est égal à 0  
.....
```

(extrait de la *carte de référence*)

Spécification

Pour pouvoir utiliser une fonction, il faut connaître sa **spécification**

- sa **signature**
 - son nom : `quotient`
 - son type : `int*int -> int`
- ce qu'elle calcule et la signification de ses variables : `(quotient a b)` rend le quotient ... a par b
- les indications éventuelles d'hypothèses ou d'erreur

```
.....  
;;; quotient: int*int -> int  
;;; (quotient a b) rend le quotient de la  
;;; division euclidienne de a par b.  
;;; ERREUR lorsque b est égal à 0  
.....
```

(extrait de la *carte de référence*)

Spécification

Pour pouvoir utiliser une fonction, il faut connaître sa **spécification**

- sa **signature**
 - son nom : `quotient`
 - son type : `int*int -> int`
- ce qu'elle calcule et la signification de ses variables : `(quotient a b) rend le quotient ... a par b`
- les indications éventuelles d'hypothèses ou d'erreur :
`ERREUR lorsque b est égal à 0`

```
.....  
;;; quotient: int*int -> int  
;;; (quotient a b) rend le quotient de la  
;;; division euclidienne de a par b.  
;;; ERREUR lorsque b est égal à 0  
.....
```

(extrait de la *carte de référence*)

Spécification : autres exemples

```
.....  
;;; substring: string * nat * nat -> string  
;;; (substring s i1 i2) rend la sous-chaîne de s com-  
;;; mençant à l'indice i1 et terminant à l'indice i2-1.  
;;; Le premier indice est 0.  
;;; ERREUR lorsque i1 ou i2 n'est pas un indice dans s  
;;; ou si i2<i1.  
.....
```

Exercices : quelles sont les valeurs de

- (substring "Hello world!" 0 5) →
- (substring "Hello world!" 6 12) →
- (substring "Hello world!" 3 8) →
- (substring "Hello world!" 3 27) →

- (substring "Hello world!" 5 3) →

Spécification : autres exemples

```
.....  
;;; substring: string * nat * nat -> string  
;;; (substring s i1 i2) rend la sous-chaîne de s com-  
;;; mençant à l'indice i1 et terminant à l'indice i2-1.  
;;; Le premier indice est 0.  
;;; ERREUR lorsque i1 ou i2 n'est pas un indice dans s  
;;; ou si i2<i1.  
.....
```

Exercices : quelles sont les valeurs de

- (substring "Hello world!" 0 5) → "Hello"
- (substring "Hello world!" 6 12) →
- (substring "Hello world!" 3 8) →
- (substring "Hello world!" 3 27) →

- (substring "Hello world!" 5 3) →

Spécification : autres exemples

```
.....  
;;; substring: string * nat * nat -> string  
;;; (substring s i1 i2) rend la sous-chaîne de s com-  
;;; mençant à l'indice i1 et terminant à l'indice i2-1.  
;;; Le premier indice est 0.  
;;; ERREUR lorsque i1 ou i2 n'est pas un indice dans s  
;;; ou si i2<i1.  
.....
```

Exercices : quelles sont les valeurs de

- (substring "Hello world!" 0 5) → "Hello"
- (substring "Hello world!" 6 12) → "world!"
- (substring "Hello world!" 3 8) →
- (substring "Hello world!" 3 27) →

- (substring "Hello world!" 5 3) →

Spécification : autres exemples

```

.....
;;; substring: string * nat * nat -> string
;;; (substring s i1 i2) rend la sous-chaîne de s com-
;;; mençant à l'indice i1 et terminant à l'indice i2-1.
;;; Le premier indice est 0.
;;; ERREUR lorsque i1 ou i2 n'est pas un indice dans s
;;; ou si i2<i1.
.....

```

Exercices : quelles sont les valeurs de

- (substring "Hello world!" 0 5) → "Hello"
- (substring "Hello world!" 6 12) → "world!"
- (substring "Hello world!" 3 8) → "lo wo"
- (substring "Hello world!" 3 27) →
- (substring "Hello world!" 5 3) →

Spécification : autres exemples

```

.....
;;; substring: string * nat * nat -> string
;;; (substring s i1 i2) rend la sous-chaîne de s com-
;;; mençant à l'indice i1 et terminant à l'indice i2-1.
;;; Le premier indice est 0.
;;; ERREUR lorsque i1 ou i2 n'est pas un indice dans s
;;; ou si i2<i1.
.....

```

Exercices : quelles sont les valeurs de

- (substring "Hello world!" 0 5) → "Hello"
- (substring "Hello world!" 6 12) → "world!"
- (substring "Hello world!" 3 8) → "lo wo"
- (substring "Hello world!" 3 27) → **Erreur de primitive 'substring' : L'indice de fin est en dehors de la chaîne**
- (substring "Hello world!" 5 3) →

Spécification : autres exemples

```

.....
;;; substring: string * nat * nat -> string
;;; (substring s i1 i2) rend la sous-chaîne de s com-
;;; mençant à l'indice i1 et terminant à l'indice i2-1.
;;; Le premier indice est 0.
;;; ERREUR lorsque i1 ou i2 n'est pas un indice dans s
;;; ou si i2<i1.
.....

```

Exercices : quelles sont les valeurs de

- (substring "Hello world!" 0 5) → "Hello"
- (substring "Hello world!" 6 12) → "world!"
- (substring "Hello world!" 3 8) → "lo wo"
- (substring "Hello world!" 3 27) → **Erreur de primitive 'substring'** : L'indice de fin est en dehors de la chaîne
- (substring "Hello world!" 5 3) → **Erreur de primitive 'substring'** : L'indice de fin doit être supérieur au début

La notion d'hypothèse

Fonction partielle : ne s'applique pas à toutes les valeurs d'un type.

Ajouter une clause **HYPOTHESE** à la spécification :

- restreindre le domaine de définition
- indiquer une dépendance (relation) entre les arguments

Exemple :

```
;;; aire-couronne : Nombre * Nombre -> Nombre
;;; (aire-couronne r1 r2) rend l'aire de la couronne
;;; de rayon extérieur r1 et de rayon intérieur r2
;;; HYPOTHÈSE : r1 et r2 sont positifs et r1 >= r2
```

```
(define (aire-couronne r1 r2)
  (- (aire-disque r1) (aire-disque r2)))
```

- Si les hypothèses sont respectées alors **résultat correct**
- sinon, on ne garantit rien : **résultat non spécifié**

Erreur vs Hypothèse

- Contrôler la validité des arguments
- Ne pas exécuter le calcul : signaler une ERREUR

⇒ modification de la spécification et du code

```
;;; aire-couronne : Nombre * Nombre -> Nombre
;;; (aire-couronne r1 r2) rend l'aire de la couronne de
;;; rayon extérieur r1 et de rayon intérieur r2
;;; ERREUR lorsque r1 ou r2 négatif ou r1 < r2
```

```
(define (aire-couronne r1 r2)
  (if (or (negative? r1) (negative? r2) (< r1 r2))
      (erreur "aire-couronne: arguments invalides")
      (- (aire-disque r1) (aire-disque r2))))
```

La détection d'erreur a un coût.

Quatre étapes pour une définition

- 1 **Donner** la spécification de la fonction
 - Signature et description

Quatre étapes pour une définition

- 1 **Donner** la spécification de la fonction
 - Signature et description
- 2 **Inventer** la composition d'opérations menant au résultat cherché : l'algorithme

Quatre étapes pour une définition

- 1 **Donner** la spécification de la fonction
 - Signature et description
- 2 **Inventer** la composition d'opérations menant au résultat cherché : l'algorithme
- 3 **Traduire** l'algorithme en une expression Scheme syntaxiquement correcte et **poser** la définition

Quatre étapes pour une définition

- ① **Donner** la spécification de la fonction
 - Signature et description
- ② **Inventer** la composition d'opérations menant au résultat cherché : l'algorithme
- ③ **Traduire** l'algorithme en une expression Scheme syntaxiquement correcte et **poser** la définition
- ④ **Tester** la fonction
 - Vérifier son comportement en l'appliquant sur des exemples pertinents

Quatre étapes pour une définition : exemple

Les première étapes :

Quatre étapes pour une définition : exemple

Les première étapes :

Spécification

```
.....  
;;; carre: Nombre -> Nombre  
;;; (carre n) donne la valeur de n à la puissance 2  
.....
```

Quatre étapes pour une définition : exemple

Les première étapes :

Spécification

```
.....  
;;; carre: Nombre -> Nombre  
;;; (carre n) donne la valeur de n à la puissance 2  
.....
```

Algorithme, ici, formule du calcul :

multiplier n par lui-même

Quatre étapes pour une définition : exemple

Les première étapes :

Spécification

```
.....  
;;; carre: Nombre -> Nombre  
;;; (carre n) donne la valeur de n à la puissance 2  
.....
```

Algorithme, ici, formule du calcul :

multiplier n par lui-même

Traduire en Scheme :

`(* n n)`

Quatre étapes pour une définition : exemple

Les première étapes :

Spécification

```
.....  
;;; carre: Nombre -> Nombre  
;;; (carre n) donne la valeur de n à la puissance 2  
.....
```

Algorithme, ici, formule du calcul :

multiplier n par lui-même

Traduire en Scheme :

`(* n n)`

(à suivre)

Quatre étapes pour une définition : exemple (suite)

Poser la définition : **forme spéciale** Scheme

$\langle \text{définition} \rangle ::=$

```
(define ( nom-fonc <nom-args> )
  <expression> )
```

$\langle \text{nom-args} \rangle ::=$ variable
 ou variable $\langle \text{nom-args} \rangle$

Pour notre exemple :

```
.....
(define (carre n)
  (* n n))
.....
```

Quatre étapes pour une définition : exemple (fin)

Tester choisir des exemples d'application et *prévoir* le résultat.

.....

```
(carre -6); -> 36  
(carre -4.2); -> 17.64  
(carre 0); -> 0  
(carre 4.2); -> 17.64  
(carre 6); -> 36
```

.....

Exemple : résumé

Ce qu'il faut rendre

.....

Exemple : résumé

Ce qu'il faut rendre

```
;;; carre: Nombre -> Nombre .....  
;;; (carre n) donne la valeur de n à la puissance 2
```

Exemple : résumé

Ce qu'il faut rendre

;;; carre: Nombre -> Nombre

;;; (carre n) donne la valeur de n à la puissance 2

```
(define (carre n)
  (* n n))
```

Exemple : résumé

Ce qu'il faut rendre

```

;;; carre: Nombre -> Nombre .....
;;; (carre n) donne la valeur de n à la puissance 2

(define (carre n)
  (* n n))

;;; jeu de tests
(verifier
  carre
    (carre -6) => 36
    (carre -4.2) => 17.64
    (carre 0) => 0
    (carre 4.2) => 17.64
    (carre 6) => 17.64
  )

```

Plus d'expression : l'alternative

Choisir un calcul en fonction d'une condition :

Si *ceci* **alors** *cela* **sinon** *plutôt cela*

C'est une **forme spéciale** en Scheme

Syntaxe :

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

Exemple : la valeur absolue

```
.....
(if (>= x 0)
    x
    (- x))
.....
```