MODULE « INFORMATIQUE 1 »

DEUG MIAS 13 et MIAS 14

Université Paris 6

« Programmation récursive »

Anne Brygoo[a]

Michèle Soria[b]

[a] mailto:Anne.Brygoo@ufr-info-p6.jussieu.fr
[b] mailto:Michele.Soria@lip6.fr

PLAN DU COURS 1

- Objectifs du module « Programmation récursive »
- Étude des expressions
  - Lecture d'une expression
  - Évaluation d'une expression
  - Différentes écritures linéaires
  - Écriture Scheme des expressions
- Premiers pas en Scheme
  - Mécanismes d'un langage
  - Application de fonction
  - Définition de fonction : `define`
  - Alternative : `if`

OBJECTIFS DU MODULE

« Programmation récursive »

- Initiation à la programmation fonctionnelle (en Scheme),
- Écriture de fonctions récursives,
  - Récursion sur les entiers
  - Récursion sur les listes
  - Récursion sur les arbres
- Fonctionnement d'un évaluateur d'expressions.
  Texte source → Résultat

ÉTUDE DES EXPRESSIONS

Une expression ?

- En **mathématique ou en informatique**
  Une expression est une formule exprimant la façon de calculer une valeur
- précision, règles à appliquer
- Un exemple :
  pour calculer la valeur de $a * f(a-1)$,
  appliquer une multiplication aux valeurs des 2 sous-expressions.

LECTURE D'UNE EXPRESSION

Caractéristiques sous-entendues

$$3x^2 + 2x + 4$$

- pas de signe pour la multiplication
- Écriture non linéaire (exposant)
- Ordre de priorité des opérateurs sous-entendus
  on « oublie » des parenthèses
  $(3 * (x \dagger 2)) + ((2 * x) + 4)$

ÉVALUATION D'UNE EXPRESSION

Des certitudes et des doutes

- a + b*c peut se comprendre :
  - (a + b)*c ou
  - a + (b*c)
- a - b + c peut se comprendre
  - (a - b) + c ou
  - a - (b + c)
- a / b / c peut se comprendre
  - (a / b) / c ou
  - a / (b / c)

DIFFÉRENTES ÉCRITURES LINÉAIRES

- en **préfixe**, l'opérateur précède ses opérandes
  OP a b
- en **infixe**, un opérateur binaire se situe entre ses arguments
  a OP b
- en **suffixe**, l'opérateur suit ses opérandes
  a b OP

UNE EXPRESSION DOIT ÊTRE NON AMBIGUË

Lemme de Lukasiewicz : si l'on connait l'**arité** des opérateurs, on peut retrouver l'expression à partir de l'une des notations postfixe et préfixe.

Les deux notations **postfixe** et **préfixe** représentent de façon non-ambiguë l'expression.

En revanche, la notation **infixe** est ambiguë et il est nécessaire pour la rendre non-ambiguë d'utiliser des parenthèses.

ÉCRITURE SCHEME DES EXPRESSIONS

Caractéristiques de l'écriture des expressions en Scheme :

- écriture préfixe,
- complètement parenthésée,
- les parenthèses entourent toute l'expression,
- le séparateur est l'espace.

Exemple-DEMO : expr1.scm

LES PREMIERS PAS EN SCHEME

- Les mécanismes d'un langage
- L'application de fonction
- La définition de fonction : `define`
- L'alternative : `if`

LES MÉCANISMES D'UN LANGAGE

Le texte d'un programme est un moyen :

- de demander à l'ordinateur le résultat d'un calcul
- de noter et d'organiser ses idées
- de les partager avec d'autres personnes

Pour cela, un langage de programmation doit :

- permettre de manipuler des objets primitifs
- posséder des moyens de compositions (constructions) d'objets composés à partir des objets primitifs

À VOTRE DISPOSITION

Pour programmer vous allez donc disposer :

- d'objets primitifs,
- d'une bibliothèque de fonctions qui vous permet, dès le début, de bénéficier du travail d'autres personnes (primitives et non- primitives),
- d'une grammaire qui vous donne les règles de construction.

LES OBJETS PRIMITIFS

Le langage permet de manipuler :

- des constantes entières, 42, -5
- des constantes flottantes, 2.3
- les valeurs booléennes, `#t` pour *true* et `#f` pour *false*
- les constantes chaînes de caractères, `"toto"`

LES FONCTIONS PRÉDÉFINIES

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

- son nom,
- son type,
- ce qu'elle calcule,
- la signification de ses arguments.

Exemple de fonction prédéfinie :

```
;;; quotient: int*int/non nul/ -> int
;;; (quotient n1 n2) retourne le quotient de la
;;; division euclidienne de n1 par n2
```


UN PRÉDICAT

Un prédicat est une fonction qui a, pour résultat, un booléen

```
;;; number?: Valeur -> bool

;;; positive?: Nombre -> bool
```

Exemple-DEMO : primitive1.scm

RÈGLES DE COMPOSITION

Grammaire : ensemble des règles à suivre

Une expression est **syntaxiquement** correcte si elle respecte la grammaire

Une expression syntaxiquement correcte peut être **sémantiquement** incorrecte, ex (+ 12 10) au lieu de (+ 1 2 10)

<programme> → *<expression-ou-définition>**

<expression> → *<constante>*
<variable>
<forme-spéciale>
<application>

APPLICATION DE FONCTION

<application> → (*<fonction>* *<argument>**)

<fonction> → *<expression>*

<argument> → *<expression>*

```
(+ (quotient 7 2) (* 5 6 2) 8 (abs -3))

(positive? (abs -3))
```


DÉFINITION DE FONCTION

La définition de fonction permet :

- de définir de nouvelles fonctions
- de nommer ces nouvelles fonctions.
- (pour pouvoir ensuite les appliquer)
  Exemple-DEMO : def1.scm

QUATRE ÉTAPES POUR UNE DÉFINITION

1. Donner la spécification de la fonction
2. Décrire (inventer) la suite d'opérations qui fournissent le résultat cherché (algorithme)
3. Traduire l'algorithme en expression syntaxiquement correcte
4. Tester la fonction

RÈGLE DE GRAMMAIRE

<définition> →

`(define (` *<nom-fonction>* *<variable>** `)` *<corps>* `)`

<corps> →

<définition>**<expression>*

LA SPÉCIFICATION D'UNE FONCTION

Rappel : Pour pouvoir utiliser une fonction prédéfinie, il faut connaître sa *spécification* : son nom, son type, ce qu'elle calcule, la signification de ses arguments.

Quand on définit une nouvelle fonction il faut donc donner sa spécification (pour pouvoir l'appliquer correctement)

- son nom,
- son type,
- ce qu'elle calcule,
- la signification de ses variables.

; UN COMMENTAIRE POUR LES HUMAINS

Règles pour utiliser les commentaires :

Voir la charte pour l'écriture de programmes en Scheme

`;;;` *la spécifi cation d'une fonction*

`;;` *la spécifi cation d'une fonction interne*

Différence entre **grammaire** et **convention d'écriture**

Exemple-DEMO : def2.scm

L'ALTERNATIVE

Exemple d'alternative : DEMO : if1.scm

La syntaxe d'une alternative

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

<condition> → *<expression>*

<conséquence> → *<expression>*

<alternant> → *<expression>*

Une condition est une expression ayant pour valeur

- soit vrai c'est-à-dire `#t` pour *true*
- soit faux c'est-à-dire `#f` pour *false*

TRAVAIL AVANT LE PROCHAIN TD/TP

- Reprendre vos notes ce soir
- Lire sur Internet ou sur le CDROM (chez vous ou à l'UTES) les paragraphes suivants du cours :
  - Introduction
  - Étude des expressions
  - Écriture Scheme d'une application
  - Définition de fonctions
  - Alternative
- Être capable d'écrire, sans l'aide des notes et du cours, la spécification et la définition de :
  - `aire-disque`
  - `volume-cylindre`
  - `negative?`
  - `valeur-absolue` (de deux manières)