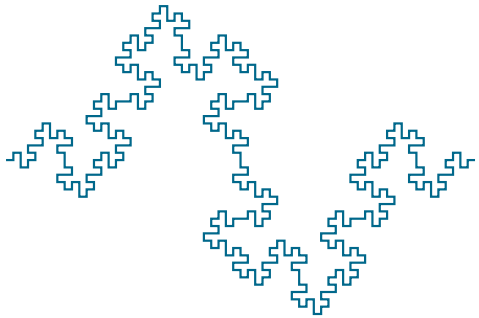
HLIN403 – Programmation Applicative

Syntaxe des expressions. Types prédéfinis. Bases de l'interprétation des expressions.

Christophe Dony – Annie Chateau
Université Montpellier – Faculté des Sciences



Introduction

Syntaxe de Scheme

Syntaxe : ensemble de règles de combinaison correcte des mots d'un langage pour former des phrases. (Syntaxe du français ou syntaxe de C ou de Scheme).

Les expressions de `scheme` sont appelées des **S-expressions** (expressions symboliques). Voici une définition simplifiée de leur syntaxe en forme BNF.

BNF est un acronyme pour "Backus Naur Form". John Backus et Peter Naur ont introduit cette notation formelle pour décrire la syntaxe d'un langage donné. Ce formalisme a été utilisé notamment pour Algol-60.

Notation BNF

: :=	se définit comme
|	ou
`<symbol>`	symbole non terminal
{ }	répétition de 0 à n fois
...	suite logique

Syntaxe des S-expressions

```
s_expression ::= <atom> | <expression_composee>
expression_composee ::== (s_expression s_expression)
atom ::= <atomic_symbol> | <constant>
constant ::= <number> | <litteral_constant>
atomic_symbol ::== <letter>{<atom_part>}
atom_part ::== empty | letter{atom_part} |
digit{atom_part}
letter ::= "a" | "b" | ... | "z"
digit ::= "1" | "2" | ... | "9"
empty = ""
```

Exemple d'expressions bien formées

```
123
true
"paul"
"coucou"
(+ 1 2)
(+ (- 2 3) 4)
(lambda (x y) (+ x y))
(sin 3.14)
```

Syntaxe préfixée et totalement parenthésée

La notation mathématique n'est pas homogène.

`Scheme` propose pour l'application d'une fonction à des opérandes une syntaxe dite **préfixée** et **totalement parenthésée** :

```
(operateur opérande1 ... opérandeN)
```

- où *operateur* doit désigner une fonction connue du système
- et où les *opérandes* doivent désigner des valeurs dont les *types* sont compatibles avec la fonction.

Syntaxe préfixée et totalement parenthésée

Cette syntaxe supprime les problèmes de priorité des opérateurs.

Un opérateur (une fonction) est dit *unaire*, *binaire* ou *n-aire* selon son nombre d'opérandes. Le nombre d'opérande définit l'*arité* de la fonction.

```
(+ 2 3 7)
(* (- 2 3) 4)
(f 2)
```

Types de données

Donnée : entité manipulée dans un programme, définie par un type.

ex : Les entiers 1, 3 et 5 sont des données de type `integer`.

Type de donnée : entité définissant comment un ensemble de données sont représentées en machines et quelles sont les fonctions qui permettent de les manipuler.

Sortes de types

Par exemple, le type `Integer` fournit une représentation des nombres entiers (on peut les manipuler) et un ensemble de fonctions : `+`, `-`, `integer?`, etc.

Type prédéfini : Tout langage de programmation est fourni avec un ensemble de types dits types prédéfinis.

En scheme les principaux types prédéfinis sont : `integer`, `char`, `string`, `boolean`, `list`, `procedure`, `...`

Sortes de types

Type scalaire : Type dont tous les éléments sont des constantes.

exemple : `integer, char, boolean`.

Type structuré : Type dont les éléments sont une aggrégation de données

exemple : `pair, array, vector`.

Les constantes littérales des types prédéfinis

Pour chaque type prédéfini d'un langage il existe des valeurs constantes qu'il est possible d'insérer dans le texte des programmes, on les appelle des constantes littérales.

ex : 123, `#\a`, "bonjour tout le monde", `#t`, `#f`

Règles d'évaluation

Interpréteur : programme transformant un texte de programme syntaxiquement correct en actions. L'interpréteur fait faire à l'ordinateur ce que dit le texte.

Si le texte de programme est une expression algébrique, l'interpréteur calcule sa valeur ou l'**évalue**. On peut alors parler au choix d'*interpréteur* ou d'*évaluateur*.[1]

La fonction d'évaluation est usuellement nommée `eval`.

Notation. On écrira `val(e) = v` pour signifier que la valeur d'une expression `e`, calculée par l'évaluateur, est `v`.

1. Dans le cas d'un langage non applicatif, comme Java par exemple, on parle d'interpréteur des instructions.

Evaluation des constantes littérales

Soit `c` une constante littérale, `val(c) = c`.

Par exemple, `33` est un texte de programme représentant une expression valide (une constante littérale) du langage scheme, dont la valeur de type entier est *33*.

Apparté : le *Toplevel*

Un toplevel d'interprétation (ou d'évaluation) est un outil associé à un langage applicatif qui permet d'entrer une expression qui est lue puis analysée syntaxiquement (lecteur), puis évaluée (évaluateur) et dont la valeur est finalement affichée (afficheur ou *printer*).

Le processus peut être décrit algorithmiquement ainsi :

```
tantque vrai faire
  print ( eval ( read ()))
fin tantque
```

et être écrit en scheme :

```
(while #t (print (eval (read))))
```

Apparté : le *Toplevel*

Exemple d'utilisation du *toplevel* pour les constantes littérales :

```
> #\a
= a
> "bonjour"
= "bonjour"
> 33
= 33
```

Evaluation d'un appel de fonction

Un appel de fonction se présente syntaxiquement sous la forme d'une expression composée : `(fonction argument1 argumentN)` dont la première sous-expression doit avoir pour valeur une fonction connue du système.

Exemples : `(sqrt 9)` ou `(+ 2 3)` ou `(modulo 12 8)`

Evaluation d'un appel de fonction

Valeur de l'expression `sqrt` : `<primitive:sqrt>`

ou `<primitive:sqrt>` est ce qui est affiché par la fontion *print* pour représenter la fonction prédéfinie calculant la racine carrée d'un nombre.

Ce que confirme l'expérimentation suivante avec le *toplevel* du langage :

```
> sqrt
<primitive:sqrt>
```

Evaluation d'un appel de fonction

`val ((f a1 ... aN)) = apply (val (f), val(a1), ..., val(aN))`

où `apply` applique la fonction `val (f)` aux arguments `val(a1) ... val(aN)`

c'est-à-dire :

évalue les expressions constituant le corps de la fonction dans l'environnement constitué des liaisons des paramètres de la fonction à leurs valeurs

Evaluation d'un appel de fonction

L'ordre d'évaluation des arguments n'est généralement pas défini dans la spécification du langage. Cet ordre n'a aucune importance tant qu'il n'y a pas d'**effets de bord**.

Exemples :

```
> (* 3 4)
= 12
> *
#<primitive:*>
> (* (+ 1 2) (* 2 3))
= 18
> (* (+ (* 1 2) (* 2 3)) (+ 3 3))
= 48
```

Evaluation d'un appel de fonction

Evaluation en pas à pas montrant la suite des évaluations effectuées.

```
--> (* (+ 1 2) (* 2 3))
--> *
<-- #<primitive:*>
--> (+ 1 2)
--> +
<-- #<primitive:+>
--> 1
<-- 1
--> 2
<-- 2
<-- 3
--> (* 2 3)
--> *
<-- #<primitive:*>
--> 2
<-- 2
--> 3
<-- 3
<-- 6

>-- 18
```

Erreurs durant l'évaluation

```
> foo
reference to undefined identifier: foo
> (f 2 3)
reference to undefined identifier: f
> (modulo 3)
modulo: expects 2 arguments, given 1: 3
> (modulo 12 #\a)
modulo: expects type <integer> as 2nd argument,
given: #\a;
> (log 0)
log: undefined for 0
```

Erreurs durant l'évaluation

La quatrième erreur illustre ce qu'est un langage dynamiquement typé.

Toutes les entités manipulées ont un type mais ces derniers sont testés durant l'exécution du programme (l'interprétation des expressions) et pas durant l'analyse syntaxique ou la génération de code (compilation).