Université Montpellier-II
UFR des Sciences - Département Informatique - Licence Informatique
Programmation Applicative et Récursive

Cours No 2 : Syntaxe des expressions.
Types prédéfinis.
Bases de l'interprétation des expression.

Notes de cours

Christophe Dony

2 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 phrases du langage de programmation Scheme sont appelées **S-expressions** (expressions symboliques). La syntaxe pour les exprimer peut être décrire en notation BNF.

Notation BNF (extraits) BNF signifie "Backus Naur Form". John Backus and Peter Naur ont imaginé une notation formelle, un méta-modèle, pour décrire la syntaxe d'un language de programmation donné. Cette notation a été utilisée en premier lieu pour Algol-60.

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

Description BNF de la syntaxe des S-expressions de Scheme - version simplifiée

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

2.1 Exemple d'expressions bien formées

```
1 123
2 true
3 "paul"
4 "coucou"
5 (+ 1 2)
6 (+ (- 2 3) 4)
7 (lambda (x y) (+ x y))
8 (sin 3.14)
```

2.2 Syntaxe préfixée et totalement parenthésée

La syntaxe de `Scheme` est dite **préfixée** et **totalement parenthésée**, ce qui se comprends intuitivement.

En conséquense, l'application d'une opération à ses opérandes etant notée de la façon suivante,

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

il n'y a jamais aucune ambiguïté liée à la priorité ou à la précédence ou à l'arité des opérations.

Par exemple, pour des opérations binaires, on ne peut pas écrire : "2 + 3 * 5".

On doit écrire "(+ 2 (* 3 5))" ou "(* (+ 2 3) 5)".

Note : Une opération est dite *unaire*, *binaire* ou *n-aire* selon son nombre d'opérandes. Le nombre d'opérande définit l'*arité* de la fonction.

3 Premières phrases

Ecrire les premières expressions du langage suppose de connaître quelques types prédéfinis, la façon dont on note leurs constantes littérales et quelques noms de fonctions. Comprendre l'interprétation des phrases, c'est à dire la façon dont sont calculées les valeurs des expressions, suppose de de connaître quelques règles d'interprétation.

3.1 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 via une représentation et un ensemble de fonctions pour les manipuler un ensemble de données utilisables dans les programmes écrits avec un langage donné.

Par exemple en *Scheme*, le type `Integer` définit une représentation en machine des nombres entiers et un ensemble de fonctions : `+`, `-`, `integer?`, etc.

3.2 Sortes de types - exemples

Type prédéfini : Type livré avec un langage donné, connu de l'interpréteur et/ou du compilateur.

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

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`.

3.3 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`

3.4 Interprétation et valeur d'une expression : 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*.[a]

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

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

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

3.5 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*.

3.6 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 (analysée syntaxiquement stockée en mémoire - fonction *read*, puis évaluée (fonction *eval)* ; la valeur résultante est alors affichée (afficheur ou *printer*).

Le processus peut être décrit ainsi :

```
1 tantque vrai faire
2   print ( eval ( read ()))
3 fin tantque
```

et être écrit en scheme :

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

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

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

3.7 Application de fonction prédéfinie

Une application de fonction à des arguments (opérandes) se présente syntaxiquement sous la forme d'une expression composée : `(nom-fonction argument1 argumentN)`

Appliquer une fonction à des arguments signifie exécuter le corps de la fonction dans un environnement ou les paramètres formels sont liés à des valeurs.

`nom-fonction` doit avoir pour valeur une fonction connue du système.

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

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 :

```
1 > sqrt
2 <primitive:sqrt>
```

Evaluation d'une application de fonction

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

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

i.e.

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

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**.

Le passage des arguments s'effectue *par valeur*. Ceci signifie que l'argument est d'abord évalué et sa valeur est ensuite passée [a].

a. L'alternative à l'appel par valeur est l'appel *par nom* peu usité - Voir le langage Algol-60 pour la définition de ce concept.

Exemples :

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

Evaluation en pas à pas montrant l'ordre des évaluations et les résultats intermédiaires

```
--> (* (+ 1 2) (* 2 3))
-->    *
<--    \#<primitive:*>
-->    (+ 1 2)
-->       +
<--       \#<primitive:+>
-->       1
<--       1
-->       2
<--       2
<--    3
-->    (* 2 3)
-->       *
<--       \#<primitive:*>
-->       2
<--       2
-->       3
<--       3
<--    6
>-- 18
```

3.8 Expressions dont l'évaluation provoque des erreurs ("erreurs à l'exécution")

```
> 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
```

La quatrième erreur illuste 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).