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

Université Montpellier-II - FDS
GLIN302 - Programmation Applicative et Récursive
Cours no 2

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 expressions de `scheme` sont appelées des **S-expressions** (expressions symboliques). Voici une définition simplifiée de leur syntaxe en forme BNF[1].

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

Syntaxe des S-expressions - 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)
```

[1] "BNF is an acronym for "Backus Naur Form". John Backus and Peter Naur introduced for the first time a formal notation to describe the syntax of a given language. Notation inventée by John Backus et Peter Naur et utilisée pour Algol-60."

2.2 Syntaxe préfixée et totalement parenthésée

On remarque la non homogénéïté de la notation mathématique ...

`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.

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.

3 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.

3.1 Sortes de types

Par exemple en DrScheme, 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 fournit avec un ensemble de types dits types prédéfinis.

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`.

3.2 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`

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*.[2]

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`.

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

4.1 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*.

4.2 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))))
```

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

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

4.3 Appel de fonction

Un appel de fonction se présente syntaxiquement sous la forme d'une expression composée : `(fonction argument1 ar`
dont la première sous-expression 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 :

```
> 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)`,
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**.

Exemples :

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

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
```

4.4 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
```

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).