Semaine 11 : Lisp

Nous présentons maintenant les principes d'un autre langage fonctionnel: Lisp.

Lisp est le premier langage fonctionnel à avoir été implanté; cela remonte à 1959-1960 (John McCarthy).

Le nom est un acronyme pour List processor.

A l'époque, Lisp a été concu pour manipuler des structures de données nécessaires au calcul symbolique, telles que les listes ou les arbres (les langages de l'époque tels que Fortran ne manipulaient que des tableaux).

Lisp a été utilisé pour implanter de nombreuses applications conséquentes. Exemples :

- Macsyma, le premier programme informatique d'algèbre
- AutoCAD, un programme de CAO populaire

Variantes de Lisp

Durant ses 45 années d'existence, Lisp a évolué, et il en existe aujourd'hui de nombreux dialectes.

Les trois dialectes les plus répandus sont :

- Common Lisp: Allegro CL (commercial, gros), CLISP, GCL, ..
- Scheme (académique, propre): DrScheme, MIT Scheme, Scheme 48, ..
- Elisp (langage d'extension de emacs)

Nous traitons ici seulement un sous-ensemble de Scheme.

Le sous-ensemble que nous présentons est purement fonctionnel – il n'a ni variables, ni affectations (le vrai Lisp en a, avec plusieurs autres choses).

Allegro CL	http://www.franz.com/products/allegrocl/
CLISP	http://clisp.cons.org/
GCL	http://savannah.gnu.org/projects/gcl/
DrScheme	http://www.plt-scheme.org/software/drscheme/
MIT Scheme	http://www.gnu.org/software/mit-scheme/
Scheme 48	http://s48.org/

Particularités de Lisp

Comparé à Scala, il y a quatre points principaux sur lesquels Lisp diffère et qui le rendent intéressant à étudier :

- Pas de syntaxe, les programmes sont simplement des séquences imbriquées de mots.
- Pas de système de types (statique).
- Une seule forme de donnée composée : la cellule `cons` (à partir de laquelle les listes sont construites).
- Les programmes sont aussi des listes, ils peuvent donc être construits, transformés et évalués par d'autres programmes.

Comme exemple de programme Lisp, considérons la fonction `faculty` :

```
((define (faculty n) (if (= n 0)
                          1
                          (* n (faculty (- n 1)))))
 (faculty 10))
```


Observations :

- Une expression Lisp composée est une séquence de sous-expressions entre parenthèses. On appelle aussi une telle expression une *combinaison*.
- Toute sous-expression est soit un mot simple (un *atome*) soit une expression composée.
- Les sous-expressions sont séparées par des caractères d'espacement.
- La première sous-expression d'une expression composée dénote un *opérateur* (en général une fonction).
- Les autres expressions dénotent les *opérandes*.

Formes spéciales

Certaines combinaisons ressemblent à des applications de fonction mais n'en sont pas. En particulier :

`(define name expr)`	définit `name` comme un alias pour le résultat de `expr` dans le programme qui suit. analogue à ***def*** *name* = *expr* en Scala.
`(lambda (params) expr)`	la fonction anonyme qui prend les paramètres `params` et retourne `expr`, analogue à (*params* ⇒ *expr*) en Scala.
`(if cond expr1 expr2)`	retourne le résultat de `expr1` si `cond` s'évalue à vrai, et le résultat de `expr2` sinon.

De telles combinaisons sont appelées des *formes spéciales*.

Applications de fonctions

Une combinaison (`op` $\texttt{od}_1$ ... $\texttt{od}_n$) qui n'est pas une forme spéciale est traitée comme une application de fonction.

Elle est évaluée en appliquant le résultat de l'évaluation de l'opérateur `op` aux résultats de l'évaluation des opérandes $\texttt{od}_1$, ..., $\texttt{od}_n$.

C'est à peu près tout. Il est difficile d'imaginer un langage de programmation utile avec moins de règles.

En fait, si Lisp est si simple c'est qu'il était destiné à être un langage intermédiaire pour les ordinateurs, pas les humains.

A l'origine il était prévu d'ajouter une syntaxe plus conviviale à la Algol, qui serait traduite en la forme intermédiaire par un préprocesseur.

Cependant, les humains se sont habitués à la syntaxe de Lisp plutôt rapidement (au moins certains d'entre eux), et ont commencé à apprécier ses avantages, si bien que la syntaxe conviviale n'a jamais été introduite.

Données Lisp

Les données en Lisp sont les nombres, les chaînes de caractères, les symboles et les listes.

- Les nombres sont soit flottants soit entiers. Les entiers ont une taille arbitraire (pas de dépassement !)
- Les chaînes de caractères sont comme en Java.
- Les symboles sont de simples séquences de caractères non délimitées par des apostrophes. Exemples :

  ```
  x    head    +    null?    is-empty?    set!
  ```

  Les symboles sont évalués en recherchant la valeur d'une définition du symbole dans un environnement.

Il n'y a pas de type booléen séparé. A la place, on représente false par le nombre *0*, et toute valeur différente de *0* est interprétée comme true.

Les listes dans Lisp

Les listes s'écrivent comme des combinaisons, par ex.

```
(1 2 3)
(1.0 "hello" (1 2 3))
```

Remarquez que les listes sont hétérogènes ; elles peuvent avoir des éléments de types différents.

Remarquez aussi que l'on ne peut pas évaluer une liste comme celles-ci, vu que leur premier élément n'est pas une fonction.

Pour empêcher une liste d'être évaluée, on utilise la forme spéciale `quote`.

```
(quote (1 2 3))
```

L'argument de `quote` est retourné comme résultat sans être lui même évalué. On peut abbréger `quote` avec le caractère spécial '.

```
'(1 2 3)
```

Les listes en interne

Comme en Scala, la notation pour les listes en Lisp est juste du sucre syntaxique.

De façon interne, les listes sont formées à partir de :

- La liste vide, qu'on écrit `nil`.
- Des couples tête `x`, et queue `y`, qu'on écrit `(cons x y)`.

La liste (ou combinaison) (x_1 ... x_n) est représentée par

(cons x_1 (... (cons x_n nil) ...))

On accède aux listes en utilisant les trois opérations

- `(null? x)` retourne vrai si la liste est vide.
- `(car x)` retourne la tête de la liste `x`.
- `(cdr x)` retourne la queue de la liste `x`.

Les noms `car` et `cdr` remontent à l'IBM 704, le premier ordinateur sur lequel Lisp a été implanté.

Sur cet ordinateur, `CAR` signifiait "contents of address register" et `CDR` signifiait "contents of decrement register".

Ces deux registres servaient à stocker les deux moitiés d'une cellule `cons` dans l'implantation pour l'IBM 704.

Analogie avec Scala,

```
cons    ~   ::
nil     ~   Nil
null?   ~   isEmpty
car     ~   head
cdr     ~   tail
```


Les listes et les fonctions

Comme Lisp n'a pas de système de types statique, on peut représenter les listes en utilisant juste les fonctions, et un unique symbole `none` :

```
nil          =   (lambda (k) (k 'none 'none))
(cons x y)   =   (lambda (k) (k x y))
(car l)      =   (l (lambda (x y) x))
(cdr l)      =   (l (lambda (x y) y))
(null? l)    =   (l (lambda (x y) (= x 'none)))
```

L'idée est que l'on peut représenter une cellule `cons` comme une fonction qui prend une autre fonction `k` en paramètre.

La fonction `k` doit permettre de décomposer la liste.

La fonction `cons` applique simplement `k` à ses argmuments.

Ensuite, `car` et `cdr` applique simplement la fonction `cons` aux fonctions de décomposition appropriées.

Cette construction montre que, en principe, toute donnée peut être contruite à partir de fonctions pures.

Mais en pratique, on représente la cellule `cons` par un couple de pointeurs.

Un exemple

Voici la définition et une utilisation de `map` en Lisp :

```
((define (map f xs)
  (if (null? xs)
    '()
    (cons (f (car xs)) (map f (cdr xs)))))
 (map (lambda (x) (* x x)) '(1 2 3)))
```

Quel est le résultat de l'évaluation de cette expression ?

Interpréteurs

Lisp est si simple qu'il est un véhicule idéal pour étudier les règles d'évaluation d'un programme.

En particulier, il est assez simple d'écrire un interpréteur pour Lisp.

C'est ce que nous allons faire maintenant.

Plus précisément, nous allons implanter Scheme−−, un sous-ensemble restreint mais complet de Scheme, en utilisant Scala comme langage d'implantation.

Définir de nouveaux langages est quelque chose que nous avons fait souvent dans ce cours. Des exemples sont :

- Un langage pour les expressions arithmétiques,
- Un langage pour les circuits digitaux,
- Un langage de contraintes.

Ces langages étaient implantés comme bibliothèques de fonctions Scala.

Deux nouveautés maintenant :

- Nous allons implanter un langage de programmation complet, qui peut calculer n'importe quel algorithme (Turing complet).
- Ce langage aura une syntaxe externe qui ressemble à Lisp, pas à Scala. Cette syntaxe sera mise en relation avec une structure de donnée interne Scala grâce à un programme d'analyse syntaxique.

L'implantation procédera en trois étapes.

1. Définir une représentation interne d'un programme Scheme−−.
2. Définir un traducteur d'une chaîne de caractères en une représentation interne
3. Définir un programme interpréteur qui évalue les représentations internes.

Représentations internes de Scheme−−

Nos représentations internes pour Scheme−− suivent étroitement les structures de données utilisées en Scheme. Elles réutilisent également autant que possible les constructions Scala équivalentes.

Nous définissons un type *Data* représentant les données Scheme comme un alias du type *scala.Any*.

type *Data* = *Any*

- les nombres et les chaînes sont représentés comme en Scala par des éléments de type *int* et *String*,
- les listes sont représentées par des listes Scala,
- les symboles sont représentés par des instances de la classe Scala *Symbol* (voir ci-dessous).

Symboles Scala

- Les symboles en Scala sont des valeurs qui représentent des identificateurs.
- Il existe une syntaxe simplifiée pour de tels symboles : si *name* est un identificateur, alors

  'name est un raccourci pour *Symbol*(*name*).

  La classe standard *Symbol* est définie de la manière suivante dans le paquetage *scala*.

```
case class Symbol(name: String) {
  override def toString() = "'" + name
}
```


Exemple : La fonction `faculty` s'écrit en Scheme−− :

```
(define faculty (lambda (n)
  (if (= n 0)
     1
     (* n (faculty (- n 1))))))
```

Sa représentation interne peut être construite avec :

```
List('define, 'faculty, List('lambda, List('n),
  List('if, List('=, 'n, 0),
     1,
     List('*, 'n, List('faculty, List('-, 'n, 1)))))))
```


Du texte à la représentation interne

Nous allons maintenant écrire un programme d'analyse syntaxique qui associe une représentation interne à une chaîne de caractères.

Cette association procède en deux étapes.

- D'une chaîne de caractères à une séquence de mots (appelés lexèmes ou tokens)
- D'une séquence de mots vers un arbre *Data*.

Ici, un lexème peut être :

- Une parenthèse ouvrante "(" ou fermante ")".
- Une suite de caractères ne contenant pas d'espaces blancs ou de parenthèses.
- Les lexèmes qui ne sont pas des parenthèses doivent être séparés par des espaces blancs, c.-à-d. des caractères blancs, des retour à la ligne ou des caractères de tabulation.

Obtention des lexèmes

Nous représentons la suite des mots d'un programme Lisp comme un itérateur, défini par la classe suivante :

```
class LispTokenizer(s: String) extends Iterator[String] {
  private var i = 0
  private def isDelimiter(ch: Char) = ch ≤ ' ' || ch == '(' || ch == ')'
  def hasNext: boolean = {
     while (i < s.length && s.charAt(i) ≤ ' ') { i = i + 1 }
     i < s.length
  }
  ...
```

Remarques :

- L'itérateur conserve une variable privée *i*, qui dénote l'indice du prochain caractère à lire.

- La fonction *hasNext* passe tous les espaces blancs précédant le lexème suivant (s'il existe). Elle utilise la méthode *charAt* de la classe Java *String* pour accéder aux caractères d'une chaîne.

```
  ...
  def next: String =
    if (hasNext) {
      val start = i
      if (isDelimiter(s.charAt(i))) i = i + 1
      else
        do i = i + 1
        while (!isDelimiter(s.charAt(i)))
        s.substring(start, i)
    } else error("premature end of string")
}
```

- La fonction *next* retourne le lexème suivant. Elle utilise la méthode *substring* de la classe *String*.

Analyse syntaxique et construction de l'arbre

Etant donnée la simplicité de la syntaxe de Lisp, il est possible de l'analyser sans technique avancée d'analyse syntaxique.

Voici comment cela est fait.

```
def string2lisp(s: String): Data = {
  val it = new LispTokenizer(s)
  def parseExpr(token: String): Data = {
    if (token == "(") parseList
    else if (token == ")") error("unbalanced parentheses")
    else if (Character.isDigit(token.charAt(0))) Integer.parseInt(token)
    else if (token.charAt(0) == '\"' &&
             token.charAt(token.length−1)=='\"')
      token.substring(1,token.length − 1)
    else Symbol(token)
  }
```



```
  def parseList: List[Data] = {
    val token = it.next
    if (token == ")") List() else parseExpr(token) :: parseList
  }
  parseExpr(it.next)
}
```

Remarques :

- La fonction *string2list* convertit une chaîne de caractères en une expression arborescente *Data*.
- Elle commence par définir un itérateur *LispTokenizer* appelé *it*.
- Cet itérateur est utilisé par les deux fonctions *parseExpr* et *parseList*, qui s'appellent récursivement (analyseur syntaxique récursif descendant).
- *parseExpr* parse une expression simple.
- *parseList* parse une liste d'expressions entre parenthèses.

Maintenant, si nous écrivons dans l'interpréteur Scala :

```
string2lisp("(lambda (x) (+ (* x x) 1))")
```

nous obtenons (sans l'indentation) :

```
List('lambda, List('x),
  List('+,
    List('*,'x,'x),
    1))
```

Exercice : Écrire une fonction *lisp2string*$(x\colon Data)$ qui affiche les expressions Lisp au format Lisp. Par ex.

lisp2string(*string2lisp*("(lambda (x) (+ (* x x) 1))"))

devrait retourner

```
(lambda (x) (+ (* x x) 1))
```


Formes spéciales

Notre interpréteur Lisp ne sera capable d'implanter qu'une expression seule.

Cette expression peut avoir l'une des formes spéciales suivantes :

1.

```
(val x expr rest)
```

Evalue `expr`, lie le résultat à `x`, et ensuite évalue `rest`. Analogue à ***val*** *x = expr; rest* en Scala.

2.

```
(def x expr rest)
```

Lie `x` à `expr`, et ensuite évalue `rest`. `expr` est évaluée à chaque fois que `x` est utilisé. Analogue à ***def*** *x = expr; rest* en Scala.

Les vrais Lisp ont à la place une variété de lieurs appelés `define` pour le niveau externe, et `let`, `let*` et `letrec` à l'intérieur des combinaisons.

Le `define` et le `letrec` en Lisp correspondent grossièrement au `def` et le `let` Lisp correspond grossièrement au `val`, mais leur syntaxe est plus compliquée.

3.

```
(lambda (p_1 ... p_n) expr)
```

Définit une fonction anonyme avec paramètres $\texttt{p}_1$, ..., $\texttt{p}_n$ et corps `expr`.

4.

```
(quote expr)
```

retourne `expr` sans l'évaluer.

5.

```
(if cond then-expr else-expr)
```

La construction if-then-else habituelle.

Sucre syntaxique

D'autres formes peuvent être converties en celles-ci en transformant la représentation interne des données Lisp.

On peut écrire une fonction *normalize* qui élimine de l'arbre les autres formes spéciales.

Par exemple, Lisp supporte les formes spéciales

```
(and x y)
(or x y)
```

pour les or et and court-circuités. La procédure *normalize* peut les convertir en expressions if-then-else de la manière suivante.

$$
\begin{array}{l}
\textbf{def}\ \mathit{normalize}(\mathit{expr}: \mathit{Data}): \mathit{Data} = \mathit{expr}\ \textbf{match}\ \{ \\
\quad \textbf{case}\ \text{'}\mathit{and} :: x :: y :: \mathit{Nil} \Rightarrow \\
\quad\quad \mathit{normalize}(\text{'}\textbf{if} :: x :: y :: 0 :: \mathit{Nil}) \\
\quad \textbf{case}\ \text{'}\mathit{or} :: x :: y :: \mathit{Nil} \Rightarrow \\
\quad\quad \mathit{normalize}(\text{'}\textbf{if} :: x :: 1 :: y :: \mathit{Nil}) \\
\quad \ldots\ //\ \textit{further simplifications go here} \\
\}
\end{array}
$$

Formes dérivées

Notre fonction de normalisation accepte les formes dérivées ci-dessous.

```
(and x y)                       => (if x y 0)
(or x y)                        => (if x 1 y)

(def (name args_1 ... args_n) => (def name
   body                              (lambda (args_1 ... args_n) body)
 expr)                              expr)

(cond (test_1 expr_1) ...       => (if test_1 expr_1
      (test_n expr_n)                (...
      (else expr'))                    (if test_n expr_n expr')...))
```


Résumé

Lisp est un langage assez inhabituel.

- Il est dépourvu de syntaxe élaborée et d'un système de types statique.
- Il réduit toutes les données composées aux listes.
- Il identifie les programmes et les données.

Ces points ont des avantages ainsi que des inconvénients.

- Pas de syntaxe :
  + facile à apprendre, − difficile à lire.
- Pas de système de types statique.
  + flexible, − facile de faire des erreurs.
- Seules les listes sont des données composées :
  + flexible, − abstraction de donnée pauvre.
- Les programmes sont des données :
  + puissant, − difficile de conserver la sûreté.