Programmation fonctionnelle

1er semestre 2000–2001

Laura Kallmeyer

laura.kallmeyer@linguist.jussieu.fr

Page web du cours: http://talana.linguist.jussieu.fr/∼lkallmey/LI034.html

Litérature:

Winston & Horn (1989): très bien livre de LISP (Common LISP), mais traite pas les applications linguistiques

Gazdar & Mellish (1989): applications linguistiques avec LISP, mais n'est pas une introduction en LISP

Wertz (1985): en français, mais utilise une autre variante de LISP que Common LISP

Abelson et Sussman (1996): très bien livre, mais se porte sur Scheme. Ne traite pas les applications linguistiques

LISP: langage de programmation, conçu par John McCarthy pour le traitement d'expressions symboliques.

Il existe un grand nombre de versions différentes de LISP dont une est Common-LISP.

LISP = acronyme de LISt Processor.

LISP traite des listes. Les éléments d'une liste sont des atomes ou à nouveau des listes. Un atome est un nombre ou un nom.

Quelques exemples d'atomes:

```
124
-1000
je-suis-un-atome
1a
```

Quelques exemples de listes:

```
()
(a b c)
((a b) cd (e))
```

Quelques exemples d'objets qui sont ni listes ni atomes:

```
(a) a
(a b) c)
)(
(((a b) cd (e))
```

La liste vide, `()`, s'appelle `NIL`.

Les programmes en LISP sont aussi des listes.

Exemple:

```
(if (member 'Marie '(Anne Marie Sandrine))
    'Anne 'Sandrine)
```

Ce programme est une liste de longueur 4.

1 Exemple: un simple automate fini

1.1 Le problème

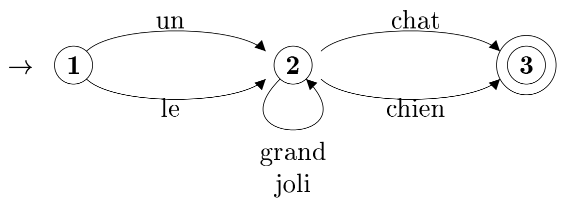
Premier exemple d'un programme très simple:

Ecrire une fonction `np-check` qui prend comme argument une liste et retourne `T` (true) si cette liste se compose d'un déterminant `un` ou `le`, suivi d'un nombre arbtitraire d'adjectives `grand` et `joli` suivi d'un nom `chien` ou `chat`. Sinon, la fonction rend `NIL` (l'équivalent LISP de la valeur `false` dans d'autres langages de programmation).

Les séquences acceptées peuvent être décrites par l'expression régulière suivante:

`(un+le)(grand+joli)`*`(chien+chat)`

L'automate fini correspondant à cela:



1 est l'état initial, **3** est le seul état final.

Idée: une fonction `np-step` determine à partir de l'état actuel de l'automate et du prochain mot le prochain état à choisir. Si `np-step` se trouve dans un état final et s'il ne reste plus de mots, `np-step` peut retourner `T`.

`np-check` appelle la fontion `np-step` avec l'état initial et la liste complète des mots.

Algorithme pour `np-step`:

```
np-step[wordlist, state]
   if      (state=3 and wordlist is empty)
           return T
   else if (state=1 and first element in wordlist is 'un' or 'le')
           then step[wordlist without first word, 2]
   else if (state=2 and first element in wordlist is 'grand' or 'joli')
           then step[wordlist without first word, 2]
   else if (state=2 and first element in wordlist is 'chat' or 'chien')
           then step[wordlist without first word, 3]
   else    return NIL
```

1.2 Quelques fonction de base: `QUOTE`, `CAR`, `CDR`, `CONS`

L'argument de `np-check` est une liste, comme par exemple `(le joli grand chat)` ou `(un chien)` ou `(pas de syntagme nominal)`.

LISP ne distingue pas entre `(un chien)` et le programme `(null nil)`. C'est-à-dire le systeme prend l'expression `(un chien)` pour un programme et essaie de l'évaluer, d'appliquer une fonction qui s'appelle `un` sur la valeur de la variable `chien`. Comme tous les deux probablement n'existent pas, le système retourne un message disant que la fonction `un` n'est pas définie.

Pour empêcher le systeme d'interpreter `(un chien)` comme un programme et de l'évaluer, on utilise `QUOTE` ou `'` tout court.

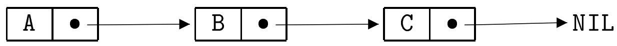
```
> (null nil)
T
> '(null nil)
```

```
(NULL NIL)
> '(un chien)
(UN CHIEN)
```

Pour vérifier si le premier élément de la liste est un déterminant, le deuxième un adjective etc., nous devons avoir accès aux différentes parties d'une liste.

L'organisation interne d'une liste en LISP est en doublets. L'adresse d'une liste est l'adresse de son premier doublet qui se compose du premier élément de la liste et de l'adresse du reste de la liste.

Exemple: Organisation interne de la liste `(A B C)`.



Les deux éléments d'un doublet peuvent être des listes ou des atomes. Quand le deuxième est un atome on sépare les deux par un point, par exemple `(A . B)` est le doublet avec `A` comme premier et `B` comme deuxième élément.

Les fonctions `CAR` et `CDR` donnent accès aux deux éléments d'un doublet, `CAR` retourne le premier et `CDR` le deuxième élément. Dans le cas d'une liste, `CAR` extrait le premier élément et `CDR` extrait le reste (la liste sans son premier élément).

```
> (CAR '(A B C))
A
> (CDR '(A B C))
(B C)
> (CAR '(UN CHIEN))
UN
> (CDR '(c))
NIL
> (CAR '(A . B))
A
> (CDR '(A . B))
B
> (car '(cdr (a b)))
CDR
```

Une suite d'appels de `CAR` et `CDR` peut être abrégée: `(CADR liste)` signifie `(CAR (CDR liste))`, `(CDADR liste)` signifie `(CDR (CAR (CDR liste)))`. Le maximum sont quatre `a` ou `d` dans une commande `c...r`.

Au lieu de `car` et `cdr`, on peut aussi utiliser `first` et `rest` qui fonctionnent exactement de la même façon.

Une fonction très liée à `CAR` et `CDR` est `CONS` qui construit un doublet à partir de son `CAR` et `CDR`.

```
> (CONS 'A '(B C))
(A B C)
> (CONS (CAR '(A)) (CDR '(A B C)))
(A B C)
> (CONS 'C NIL)
(C)
> (CONS 'A 'B)
(A . B)
```

Maintenant on peut définir la syntaxe du langage LISP. Les éléments de LISP s'appellent des expressions symboliques ou s-expressions. Une s-expression peut être un atome ou un doublet (une liste est aussi un doublet).

Définition de la syntaxe de LISP:

```
<sexpr> ::= <atome> | (<sexpr> . <sexpr>)
<atome> ::= <nom> | <nombre>
<nom> ::= <lettre> | <nom><lettre> | <nom><chiffre>
<nombre> ::= <chiffres> | +<chiffres> | -<chiffres>
<chiffres> ::= <chiffre> | <chiffre><chiffres>
<lettre> ::= A|B|C|D|E|F| ... Z ...
             symboles exclus comme <lettre>: (, ), ;, ',
             espace, tabulation, retour chariot
<chiffre> ::= 0|1|2|3|4|5|6|7|8|9
```

En général, les systèmes LISP permettent plus de symboles dans les noms que seulement les lettres `A` ... `Z`, et ils permettent aussi des nombres comme `-3.41`.

1.3 Prédicats et séléction

Pour savoir si deux atomes sont égaux, par exemple pour tester si le premier élément de notre liste est égal au déterminant `un`, on utilise `EQ`. Attention: `EQ` teste l'égalité des adresses en mémoire et ne marche pas pour des listes.

```
> (EQ 'a 'a)
T
> (EQ 'a (CAR '(a b c)))
T
> (EQ 'a 'b)
NIL
> (EQ '(a b c) '(a b c))
NIL
```

(Pour comparer des listes, il faut la fonction `equal`.)

En cas des nombres, l'égalité peut être testée aussi par `=`.

```
> (= 0 1)
NIL
> (= (+ 2 1) 3)
T
```

Un autre prédicat dont nous avons besoin pour l'exemple de 2.1 est le prédicat `NULL` qui s'applique à une liste et qui retourne `T` si la liste est vide, et `NIL` si elle ne l'est pas.

Les opérations logiques de conjonction et disjonction sont réalisées en LISP par les fonction `AND` et `OR`. `AND` prend un nombre arbitraire d'aguments. Si l'évaluation d'un de ces arguments ramène `NIL`, la fonction retourne `NIL`. Sinon, la fonction retourne la valeur ramenée par l'évaluation de son dernier argument. `OR` prend un nombre arbitraire d'arguments et retourne NIL si l'évaluation de tous ses arguments retourne `NIL`. Sinon, la première valeur différent de `NIL` est retourné.

Attention: tout ce qui n'est pas `NIL` est considéré comme vrai, par conséquent comme équivalent à `T`.

```
> (AND (EQ 'a 'a) T)
T
> (AND 'A (CAR '(b c)))
B
```

```
> (OR (EQ 'A 'B) (EQ 'A 'A))
T
> (OR 'A 'B)
A
> (AND (EQ 'A 'B) (EQ 'A 'A))
NIL
```

Les structures de séléction dans `np-step` sont exprimées en LISP par la fonction `COND`. `COND` est syntaxiquement définie comme suit:

```
(COND   (test_1  action_11 action_12 ... action_1n)
        (test_2  action_21 action_22 ... action_2n)
                           ...
        (test_k  action_k1 action_k2 ... action_kn))
```

L'évaluation de la fonction `COND` procède comme ceci: D'abord `test_1` est évalué. Si le résultat n'est pas `NIL`, `action_11`, `action_12`, ..., `action_1n` sont évaluées et `COND` retourne la valeur de la derniere action. Si l'évaluation de `test_1` a ramené `NIL`, LISP continue avec le deuxième argument etc. Si tous les `test_i` ramènent la valeur `NIL`, la fonction retourne `NIL` (dans ce cas aucune des actions est évaluée).

```
> (COND ((EQ 'A 'A) 'A))
A
> (COND ((EQ 'A 'A) 'A 'B 'C))
C
> (COND ((EQ 'A 'B) 'B) (T 'A))
A
> (COND ((EQ 'A 'B) 'B))
NIL
```

1.4 Définir une fonction

On utilise `DEFUN` pour définir une fonction en LISP. `DEFUN` a trois arguments, un nom (le nom de la nouvelle fonction), une liste (la liste de ses paramètres) et le corps de la fonction. Le corps est une suite d'appels de fonctions.

Une fois une fonction est définie, on peut l'appeller avec des arguments pour ses paramètres. En ce moment, les paramètres de la fonction sont liés aux arguments de cet appel et le corps de la fonction est évalué.

```
> (DEFUN add1 (n) (+ 1 n))
ADD1
> (add1 2)
3
```

(Les s-expressions sont évaluées une après l'autre.)

Maintenant nous avons tout ce qu'il faut pour définir les deux fonction `np-check` et `np-step` de 2.1:

```
(DEFUN np-step (wordlist state)
  (COND ((AND (= state 3) (NULL wordlist))
         T)
        ((AND (= state 1)
              (OR (EQ (CAR wordlist) 'un)
```

```
                 (EQ (CAR wordlist) 'le)))
         (np-step (CDR wordlist) 2))
        ((AND (= state 2)
              (OR (EQ (CAR wordlist) 'grand)
                  (EQ (CAR wordlist) 'joli)))
         (np-step (CDR wordlist) 2))
        ((AND (= state 2)
              (OR (EQ (CAR wordlist) 'chat)
                  (EQ (CAR wordlist) 'chien)))
         (np-step (CDR wordlist) 3))
        (T NIL)))

(DEFUN np-check (wordlist) (np-step wordlist 1))
```

(Ce n'est pas la meilleure façon de définir un automate fini. On verra des définitions alternatives plus tard.)

La fonction `np-step` est une fonction recursive, elle s'appelle elle-même.

Typiquement, une fonction recursive a un argument qui est réduit dans chaque appel. (Ici c'est la liste `wordlist` dont `np-step` enlève un élément chaque fois que `np-step` est appellé.) La recursion se termine quand cet argument est réduit à une certaine valeur terminale. Pour une liste, c'est souvent la liste vide, pour un nombre c'est souvent 0 ou 1.

Exemple:

```
(defun fac (n)
   (cond ((= n 0) 1)
         (T (* n (fac (- n 1))))))
```

Attention: Il faut vérifier si l'argument en question a déjà la valeur terminale avant l'appel recursif.