Le Premier LISP

Didier Verna
didier@lrde.epita.fr
http://www.lrde.epita.fr/~didier

Séminaire du LRDE, 19 Juin 2002

Table des matières

Nativité

- MIT, Laboratoire d'Intelligence Artificielle, 1960
- Projet «Advice Taker» (1958) : manipulation d'expressions représentant des phrases déclaratives et impératives en vue de déductions
- IBM 704
- John Mc Carthy : «Recursive Functions of Symbolic Expressions and their Computation by Machine.»

IBM 704

S-expressions ou S-exp

Premier soucis : définir une «expression»
(S comme «symbolic»)

- Une S-exp est un atome ou une liste d'expressions
- Une liste est une séquence parenthésée d'expressions séparées par des espaces
- Un atome est une séquence de lettres

```
foo
()
(foo)
(foo bar)
(foo (bar) baz)
```

Valeurs des S-exp

Une S-exp a une *valeur* (analogie mathématique : «$1 + 1$» $\rightarrow 2$)

- Une S-exp non atomique s'écrit (`OP ARG1 ARG2 ...`)
- `OP` est un «opérateur»
- `ARGx` est un «argument»

➥ La valeur d'une S-exp est définie comme l'application de l'opérateur à ses arguments

➥ On défini 7 opérateurs «primitifs» (axiomes)

Les sept Opérateurs Primitifs

Opérateur n°1 : (**quote** x)

Retourne x

((quote x) est aussi noté ’x)

```
> (quote a)
a

> ’(a b c)
(a b c)

> (quote ’(a b c))
(quote (a b c))
```

Opérateur n°2 : (atom x)

Retourne `t` si la valeur de `x` est atomique, `()` sinon
(`t` représente *vrai*, `()` représente *faux*)
(l'atome `nil` est équivalent à `()`)

```
> (atom 'a)
t

> (atom '(a b c))
()

> (atom '())
t
```

Remarque : évaluation des arguments

Contrairement à **quote**, **atom** évalue son argument

```
> (atom (atom 'a))
t

> (atom '(atom 'a))
()
```

$\Longrightarrow$ **quote** est un élément distinctif de LISP : c'est l'opérateur permettant de faire la distinction entre code et données (conséquence de leur équivalence structurelle en LISP)

Opérateur n°3 : (eq x y)

Retourne `t` si les valeurs de x et y sont le même atome, `()` sinon

```
> (eq 'a 'a)
t

> (eq 'a 'b)
()

> (eq '() '())
t

> (eq '(a b c) '(a b c))
()
```

Opérateurs n°4 et 5 : (car x) et (cdr x)

Retournent le premier élément (le reste) de la valeur de x
(cette valeur doit donc être une liste)

```
> (car '(a b c))
a

> (cdr '(a b c))
(b c)

> (car '())
()

> (cdr nil)
()
```

La vérité sur car, cdr et les listes

Découlent de l'architecture matérielle de l'IBM 704

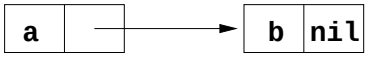
- **car** : **C**ontents of **A**ddress **R**egister
- **cdr** : **C**ontents of **D**ecrement **R**egister
- Une liste n'est en fait composée que d'un car et d'un cdr
- Le car et le cdr sont séparés par des points : `(car . cdr)`

$\Longrightarrow$ La notation par espacement est en réalité un raccourci d'écriture pour des listes de listes.

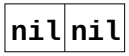
Exemples

~~()~~

(a b) <=> (a . (b)) <=> (a . (b . nil))

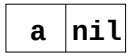


(nil . nil)

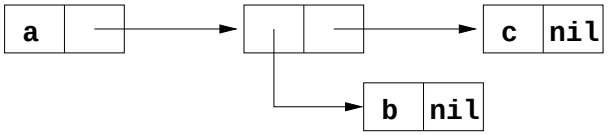


(a b . c) <=> (a . (b . c))

(a) <=> (a . nil)



(a (b) c) <=> (a . ((b) c)) <=> (a . ((b) . (c)))
<=> (a . ((b . nil) . (c . nil)))



Opérateur n°6 : (cons x y)

Retourne la liste de **car** la valeur de x et de **cdr** la valeur de y
(y est supposé être une liste)

```
> (cons 'a '(b c))
(a b c)

> (cons 'a (cons 'b (cons 'c '())))
(a b c)

> (car (cons 'a '(b c)))
a

> (cdr (cons 'a '(b c)))
(b c)
```

Opérateur n°7 : (cond (p_1 e_1) ... (p_n e_n))

Évalue les S-exp p_i jusqu'à ce que l'une d'elles retourne `t`, puis retourne l'évaluation de la S-exp e_i correspondante.

```
> (cond ((eq 'a 'b) 'first)
        ((atom 'a)  'second))
second

> (cond ((eq 'a 'b) 'first)
        ((eq 'c 'd) 'second)
        (t          'default))
default
```

Dénotation Fonctionnelle

- 5 opérateurs sur 7 évaluent toujours leurs arguments
- Les deux autres sont **quote** et **cond**
- Tout opérateur du premier type sera appelé «fonction»

Ambigüité du terme mathématique «fonction» :

- Les *formes* $x + y^2$, $f(x, y)$ etc sont appelées «fonctions»
- Mais $x + y^2$(3, 4) = 13 ou 19 ? ?

$\Longrightarrow$ Notation λ de Church (1941) : $\lambda((x_1, \ldots, x_n), \epsilon)$

$$\lambda((x, y), x + y^2)(3, 4) = 19$$

Application à LISP

Fonction LISP :

- `(lambda (`p_1 `...`p_n`)` e`)`
- Les p_i sont des atomes (paramètres)
- e est une S-exp

Appel de fonction LISP :

- `((lambda (`p_1 `...`p_n`)` e`)` a_1 `...`a_n`)`
- Les a_i sont évalués
- e est évaluée avec remplacement de chaque occurrence de p_i par la valeur de a_i

Remarque : La valeur d'un a_i peut très bien être une fonction...

Exemples

```
> ((lambda (x) (cons x '(b c))) 'a)
(a b c)

> ((lambda (x y) (cons x (cdr y))) 'z '(b c))
(z c)

> ((lambda (f) (f '(b c))) '(lambda (x) (cons 'a x)))
>> ((lambda (x) (cons 'a x)) '(b c))
(a b c)
```

Fonctions récursives

Rochester signale l'inadéquation de la notation lambda pour les fonctions récursives : `fact = (lambda (n) (* n (`*`fact???`*` (-1 n))))`

$\Longrightarrow$ Nouvelle notation :

- `(label f (lambda (`p_1 `...`p_n`) `e`))`
- `(defun f (`p_1 `...`p_n`) `e`)`

Même comportement qu'une lambda-expression, mais toute occurrence de `f` est en plus évaluée à la lambda-expression elle-même.

```
(defun fact (n) (* n (fact (-1 n))))
```

Quelques Fonctions

Commençons par des raccourcis...

- (**cadr** e) : `(car (cdr e))`
- (**cdar** e) : `(cdr (car e))`
- (**c[ad]+r** e) : ...
- (**list** $e_1 \ldots e_n$) : `(cons` e_1 `...(cons` e_n `'()) ...)`

```
> (cadr '((a b) (c d) e))
(c d)
> (cdar '((a b) (c d) e))
(b)

> (list 'a 'b 'c)
(a b c)
```

Quelques Opérateurs Logiques

- **null** :

```
(defun null (x)
  (eq x '()))
```

- **not** :

```
(defun not (x)
  (cond (x '())
        ('t 't)))
```

- **and** :

```
(defun and (x y)
  (cond (x (cond (y 't)
                 ('t '())))
        ('t '())))
```

append

Retourne la concaténation de deux listes

```
(defun append (x y)
  (cond ((null x) y)
        ('t (cons (car x) (append (cdr x) y)))))

> (append '(a b) '(c d))
(a b c d)

> (append '() '(a b))
(a b)

> (append '(nil) '(a b))
(nil a b)
```

pair

Retourne une liste associative (property list) à partir de deux listes

```
(defun pair (x y)
  (cond ((and (null x) (null y)) '())
        ((and (not (atom x)) (not (atom y)))
         (cons (list (car x) (car y))
               (pair (cdr x) (cdr y))))))

> (pair '(a b c) '(d e f))
((a d) (b e) (c f))
```

assoc

Retourne le cadr du premier élément d'une liste associative dont le car correspond au paramètre souhaité

```
(defun assoc (x y)
  (cond ((eq x (caar y)) (cadar y))
        ('t (assoc x (cdr y)))))

> (assoc 'x '((x a) (y b)))
a

> (assoc 'x '((x one) (x two) (y b) (x three)))
one
```

Le Miracle

Avec seulement ce qui précède (et même moins), il est possible d'écrire un interpréteur LISP en LISP

```
(defun eval (exp env)
  (cond
   ((atom exp) (assoc exp env))
   ((atom (car exp))
    (cond ((eq 'quote (car exp)) (cadr exp))
          ((eq 'atom  (car exp)) (atom (eval (cadr exp) env)))
          ((eq 'eq    (car exp))
           (eq (eval (cadr exp) env) (eval (caddr exp) env)))
          ((eq 'car   (car exp)) (car (eval (cadr exp) env)))
          ((eq 'cdr   (car exp)) (cdr (eval (cadr exp) env)))
          ((eq 'cons  (car exp))
           (cons (eval (cadr exp) env) (eval (caddr exp) env)))
          ((eq 'cond  (car exp)) (condeval (cdr exp) env))
          ('t (eval (cons (assoc (car exp) env) (cdr exp)) env))))
   ((eq (caar exp) 'label)
    (eval (cons (caddar exp) (cdr exp))
          (cons (list (cadar exp) (car exp)) env)))
   ((eq (caar exp) 'lambda)
    (eval (caddar exp)
          (append (pair (cadar exp) (listeval (cdr exp) env)) env)))
   ))
```

Explications

- **eval** prends deux arguments :
  - **exp** est une S-exp à évaluer
  - **env** est l'«environnement» d'évaluation

  $\Longrightarrow$ Une liste associative d'atomes et des valeurs correspondantes.
- **eval** comprends 4 clauses d'évaluation :
  - une pour les atomes
  - une pour les listes commençant par un atome
  - une pour les label-expressions
  - une pour les lambda-expressions

Clause 1 : atome

Rechercher sa valeur dans l'environnement

```
((atom exp) (assoc exp env))
```

Par exemple :

```
> (eval 'x '((a b) (x val)))
>> (assoc 'x '((a b) (x val)))
val
```

Clause 2 : (a . . .) 1ère partie

a est un opérateur primitif : appeler l'opérateur sur l'évaluation de ses arguments

```
((eq 'cons (car exp))
 (cons (eval (cadr exp) env) (eval (caddr exp) env)))
```

Par exemple :

```
> (eval '(cons x '(b c)) '((x a)))
>> (cons (eval x '((x a))) (eval (quote (b c)) '((x a))))
(a b c)
```

Exceptions

- **quote** n'évalue pas son argument :
  `((eq 'quote (car exp)) (cadr exp))`
- **cond** a un traitement spécial par **condeval** :
  `((eq 'cond (car exp)) (condeval (cdr exp) env))`

```
(defun condeval (conds env)
  (cond ((eval (caar conds) env)
         (eval (cadar conds) env))
        ('t
         (condeval (cdr conds) env))))
```

Clause 2 : (a . . .) 2ème partie

a est un atome : le remplacer par sa valeur (qui doit finir par être une lambda ou label-expression), et évaluer le résultat

```
('t (eval (cons (assoc (car exp) env) (cdr exp)) env))))
```

Par exemple :

```
> (eval '(f '(b c)) '((f (lambda (x) (cons 'a x)))))
>> (eval ((lambda (x) (cons 'a x)) '(b c))
>>       '((f (lambda (x) (cons 'a x)))))
(a b c)
```

Clause 3 : label-expression

((label f (lambda ($p_1 ... p_n$) e)) $a_1 ... a_n$)

Évaluer la lambda-expression en ajoutant la définition de la fonction elle-même dans l'environnement

```
((eq (caar exp) 'label)
    (eval (cons (caddar exp) (cdr exp))
          (cons (list (cadar exp) (car exp)) env)))

> (eval ((label f (lambda (p1...pn) e)) a1...an) env)
>> (eval ((lambda (p1...pn) e) a1...an)
>>  ((f (label f (lambda (p1...pn) e))) env))
```

Clause 4 : lambda-expression

((lambda ($p_1 ... p_n$) e) $a_1 ... a_n$)
Évaluer e avec les associations (p_i (eval a_i)) ajoutées dans l'environnement

```
((eq (caar exp) 'lambda)
 (eval (caddar exp)
       (append (pair (cadar exp)
                     (listeval (cdr exp) env)) env)))
```

```
> (eval ((lambda (p1...pn) e) a1...an) env)
>> (eval e ((p1 a1) ...(pn an) env))
```

listeval

Retourne la liste de tous les arguments évalués dans l'environnement

$$(a_1 \ldots a_n) \Longrightarrow (\overline{a_1} \ldots \overline{a_n})$$

```
(defun listeval (args env)
  (cond ((null args) '())
        ('t (cons (eval (car args) env)
                  (listeval (cdr args) env)))))
```

Bilan

La définition d'**eval** ne nécessite que les opérateurs primitifs (mais ne pas oublier de rajouter les symboles exp et env dans l'environnement...)

```
(assoc (car exp) env)

(eval '((label assoc
          (lambda (x y)
            (cond ((eq (car (car y)) x) (car (cdr (car y))))
                  ('t (assoc x (cdr y))))))
        (car exp) env)
      (cons (cons 'exp (cons exp '()))
            (cons (cons 'env (cons env '())) env))))
```

Les apports de LISP

- **Branchements Conditionnels** : inventés par Mc Carthy. Fortran n'avait qu'un **goto** conditionnel. Mc Carthy introduit les conditionnels dans Algol.
- **1er Langage Fonctionnel** : les fonctions sont des objets de 1ère classe (passées en paramètre, stockées dans des variables. . .).
- **Récursion dans les langages** : conséquence du point précédent.
- **Nouveau concept de variable** : les variables n'ont pas de type, ce ne sont que des pointeurs. Seules les valeurs ont un type.
- **Garbage Collection** («réclamation»)
- **Expressions** : pas de distinction entre expressions et déclarations. `(if foo (= x 1) (= x 2))` $\leftrightarrow$ `(= x (if foo 1 2))`

Y-combinator

Aussi appelé «point fixe de Curry». Mc Carthy ne le connaissait sans doute pas. La notation **label** n'est pas nécessaire pour obtenir la récursivité.

```
(setq Y (lambda (gen)
          (funcall
           (lambda (f) (funcall f f))
           (lambda (f)
             (funcall gen (lambda (x) (funcall (funcall f f) x)))))))
(setq dofact (lambda (func)
               (lambda (n)
                 (cond ((zerop n) 1)
                       (t (* n (funcall func (- n 1))))))))
(setq fact (funcall Y dofact))
(funcall fact 4)
```

24

Dynamic Scoping

Le tout premier exemple de fonction LISP de haut niveau donné par Mc Carthy lui-même était faux à cause du dynamic scoping

```
(defun funcall-on-list (func list)
  (let (elt n)
    (while (setq elt (pop list))
      (push (funcall func elt) n))
    (nreverse n)))

(defun increment-list (n list)
  (funcall-on-list
   (lambda (elt) (+ elt n)) list))

(increment-list 1 '(1 2 3))
```

Signaling : (wrong-type-argument number-char-or-marker-p nil)