

IA41

**Concepts fondamentaux en Intelligence Artificielle
et langages dédiés**

CM #9a

**Primitives arithmétiques,
opérateurs logiques,
primitives conditionnelles,
affectations et primitives itératives**

Fabrice LAURI

Les primitives arithmétiques (1/6)

Fonctions arithmétiques :

$(+ \langle n_1 \rangle \dots \langle n_N \rangle)$, $(- \langle n_1 \rangle \dots \langle n_N \rangle)$: addition, soustraction

$(* \langle n_1 \rangle \dots \langle n_N \rangle)$, $(/ \langle n_1 \rangle \dots \langle n_N \rangle)$: multiplication, division

Exemples :

$(+ 5 (- 12 4 1) 2) \rightarrow 14$

$(* (+ 1 2) 3 5) \rightarrow 45$

$(1+ \langle n \rangle)$: incrémentation (opérateur unaire)

$(1- \langle n \rangle)$: décrémentation (opérateur unaire)

Exemple : $(1+ 5) \rightarrow 6$

Les primitives arithmétiques (2/6)

Fonctions arithmétiques :

(**rem** $\langle n_1 \rangle \langle n_2 \rangle$) : reste de la division entière de n_1/d

Exemple : (rem 11 3) \rightarrow 2

(**max** $\langle n_1 \rangle \dots \langle n_N \rangle$), (**min** $\langle n_1 \rangle \dots \langle n_N \rangle$) : maximum et minimum

Exemple : (min 8 (max 1 6 3) 13) \rightarrow 6

(**expt** $\langle n_1 \rangle \langle n_2 \rangle$) : mise à la puissance de n_1 par n_2

Exemple : (expt 10 2) \rightarrow 100

(**abs** $\langle n \rangle$) : valeur absolue

Exemples : (abs (+ 1 -4)) \rightarrow 3

Les primitives arithmétiques (3/6)

Prédicats numériques :

=, /=, >, <, >=, <= : symboles d'égalité, de différence, d'inégalités

Exemples : ($> 2 1$) $\rightarrow t$
($> 2 2$) $\rightarrow nil$
($>= 2 2$) $\rightarrow t$
($> 5.5 2 1$) $\rightarrow t$;; $5.5 > 2$ ET $2 > 1$?
($> 5.5 2 2$) $\rightarrow nil$;; $5.5 > 2$ ET $2 > 2$?

(**zerop** <n>) : le nombre est-il égal à zéro ?

Exemples : (**zerop** 3) $\rightarrow nil$
(**zerop** (- (+ 1 2) 3)) $\rightarrow t$
(**zerop** 'a) \rightarrow **ERREUR**

Les primitives arithmétiques (4/6)

Prédicats numériques :

(**plusp** <n>) : le nombre est-il positif ?

Exemples : (plusp 11) → t

(plusp -12.7) → nil

(plusp 0) → nil

(plusp '(a b)) → ERREUR

(**minusp** <n>) : le nombre est-il négatif ?

Exemples : (minusp 11) → nil

(minusp -12.7) → t

(minusp 0) → nil

(minusp 'a) → ERREUR

Les primitives arithmétiques (5/6)

Prédicats numériques :

(numberp <e>) : l'argument est-il un nombre ?

Exemples : (numberp 'a) → nil
(numberp (+ 5.5 1)) → t

(integerp <e>) : l'argument est-il un entier ?

Exemples : (integerp 11) → t
(integerp 12.5) → nil
(integerp '(pomme)) → nil

(floatp <n>) : l'argument est-il un réel ?

Les primitives arithmétiques (6/6)

Prédicats numériques :

(evenp <n>) : le nombre est-il pair ?

Exemples : (evenp 'a) → **ERREUR**
(evenp (* 2 3)) → *t*
(evenp (* 5 3)) → *nil*

(oddp <n>) : le nombre est-il impair ?

Exemples : (oddp 'a) → **ERREUR**
(oddp (* 2 3)) → *nil*
(oddp (* 5 3)) → *t*

Opérateurs logiques (1/2)

Les opérateurs logiques en LISP sont des semi-prédicats. Un semi-prédicat est une fonction qui retourne soit faux (*nil*), soit une autre valeur quelconque, différente de *nil*, qui sera interprétée comme vrai (*t*).

(and <e₁>...<e_N>) : **pseudo**-conjonction d'expressions
→ retourne <e_N> si aucun des e_i ne s'évalue à *nil*,
sinon renvoie *nil*.

Exemples : (and '(a b) '1 '(c)) → (c)
(and '(a b) (cdr '(a)) '(c)) → nil
(and '(a b) '1 '(c) t) → t

Opérateurs logiques (2/2)

(**or** <e₁>...<e_N>) : **pseudo**-disjonction d'expressions

→ retourne le résultat du premier <e_i> qui ne s'évalue pas à *nil*,
sinon renvoie *nil*.

Exemples : (or '(a b) '1 '(c)) → (a b)
(or '(a b) (cdr '(a)) '(c)) → (a b)
(or 'c '(a b) (cdr '(a))) → c

(**not** <e>) : *négation d'une expression*

Exemples : (not t) → nil
(not ()) → t
(not '(a b)) → nil
(not (or nil 3 4 nil 5)) → nil

Opérateur d'affectation

(**setq** <symbol₁> <expr₁> ... <symbol_n> <expr_n>) : associe l'expression <expr_i> au symbole <symbol_i>.

Exemples : vitesse → **ERREUR**
(setq vitesse 60) → 60
vitesse → 60
(+ vitesse 50) → 110
(setq prenom "Marie") → "Marie"
prenom → "Marie"

(**symbol-value** <symbol>) : force l'évaluation d'un symbole

Exemple : (symbol-value 'vitesse) → 60

Une base de données représentée à l'aide de listes

Il est possible de représenter très facilement des informations à l'aide de listes.

Exemple :

```
(setq Evenement '(Jean-Claude et Hervé se moquent de Sylvain))  
(setq Jean-Claude '(cadre commercial))  
(setq Hervé '(responsable des achats))  
(setq Sylvain '(employé au service gestion))  
(setq responsable '(chargé d'une fonction))  
  
(symbol-value (car (symbol-value (caddr (Evenement))))))
```

L'affectation de listes à des symboles permet ainsi de représenter des situations ou des relations entre objets.

Opérateurs conditionnels (1/3)

(**if** <condition> <action-then> <action₁-else>...<action_N-else>) :
si <condition> est évaluée à *non-nil*, **if** retourne l'évaluation de <action-then>. Si condition est *nil*, les expressions <action₁-else>...<action_N-else> sont évaluées séquentiellement et la valeur de <action_N-else> est retournée.

*Exemples : (if (numberp 1) '(un nombre) '(pas un nombre)) →
(un nombre)*

(if nil 1 2) → 2

Opérateurs conditionnels (2/3)

(**cond** <clause₁> <clause₂> ... <clause_n>) : une clause étant de la forme (<condition> <action₁> { <action₂>...<action_n>}), **cond** évalue la condition de chaque clause jusqu'à ce que l'évaluation retourne une valeur différente de *nil*. Les actions de <clause_i> sont évaluées séquentiellement de gauche à droite et la valeur de la dernière expression est retournée. Si toutes les évaluations des clauses sont *nil*, **cond** retourne *nil*.

Exemples : (setq forme 'sphere r 1) → 1
(cond ((eq forme 'cercle) (* pi r r))
((eq forme 'sphere (* 4 pi r r))) → 12,56637

Opérateurs conditionnels (3/3)

(**case** <var>

(<val₁> <action₁₁> { <action₁₂>...<action_{1R}>})

(<val₂> <action₂₁> { <action₂₂>...<action_{2S}>})

...

(<val_N> <action_{N1}> { <action_{N2}>...<action_{NT}>})

Evalue la valeur de <var> avec chacune des valeurs <val₁>...<val_N>. Si la valeur de <var> est égale à <val_i>, les expressions <action_{i1}> { <action_{i2}>...<action_{iU}>} sont évaluées de gauche à droite et l'expression <action_{iU}> est retournée. Sinon, **case** retourne *nil*.

Exemple : (case forme

(cercle (* pi r r)

(sphere (* 4 pi r r))) → 12,56637

Retour sur les listes : Récuratif vs Itératif

La programmation réursive est simple à appliquer, mais :

- lente, car tous les paramètres d'une fonction réursive sont passés à chaque appel réursif
- gourmande en terme d'occupation mémoire, car à chaque appel réursif, un empilement à la fois des paramètres et du résultat de la fonction réursive est réalisé.

Dans certaines situations, revenir dans le cadre de la programmation itérative permet de calculer plus rapidement et avec moins de mémoire.

Les fonctions d'application simple (1/3)

Un programme peut être considéré comme une donnée
(son code source)

Il suffit pour cela de bloquer l'évaluation avec une *quote* (').

Exemple :

(+ 1 4 9) est un programme

'(+ 1 4 9) est une donnée

Pour définir la fonction *somme*, deux solutions s'offrent à nous :

- solution récursive :

$L = () \rightarrow \text{somme } L = 0$

$L \neq () \rightarrow \text{somme } L = \text{tete } L + \text{somme } \text{reste } L$

- solution itérative :

ajout '+ L

eval 'ajout '+ L

Les fonctions d'application simple (2/3)

Par exemple en LISP :

(cons '+ '(1 2 3)) → (+ 1 2 3)

(eval '(+ 1 2 3)) → 6

est équivalent à :

(apply '+ '(1 2 3)) → 6

ou

(funcall '+ 1 2 3) → 6

Les fonctions d'application simple (3/3)

En résumé :

*(**apply** <f> <liste>) : permet de retourner la valeur de l'application de la fonction <f> à la liste <liste>.*

*(**funcall** <f> <arg₁>...<arg_n>) : permet de retourner la valeur de l'application de la fonction <f> aux n arguments <arg₁>...<arg_n>.*

Une fonction de parcours de listes

Une fonction de parcours de liste permet de créer une liste obtenue par application d'une fonction sur chaque élément d'une ou de plusieurs listes de départ.

Exemples :

terme-à-terme carré, (1 4 5) → (1 16 25)

terme-à-terme 1+, (1 4 5) → (2 5 6)

terme-à-terme impair, (1 2 3) → (vrai faux vrai)

terme-à-terme =, (1 2 3) (3 2 1) → (faux vrai faux)

La primitive LISP permettant de réaliser un parcours itératif et d'effectuer une opération sur chaque élément d'une liste est *mapcar*. Sa syntaxe est :

(mapcar <f> <liste₁>...<liste_n>)