

# Programmation 3

Chargée de cours : Irène Durand

Chargé de TD : Kahn Le Ngoc Kim

Cours 7 séances de 3h :

Cours (1) 8h30-11h30, du 8/12 au 11/12

Cours (2) 13h00-16h00 les 7/12, 8/12 et 10/12

<http://dept-info.labri.fr/~idurand/Enseignement/PFS/vietnam.html>

# Objectifs

**Maîtriser** un certain nombre de méthodes et techniques de programmation

- symbolique, fonctionnelle
- impérative, objet

dans le but de

**Écrire** des applications

- maintenables, réutilisables,
- lisibles, modulaires,
- générales, élégantes.

Aspects de la programmation **non enseignés** :

- Efficacité extrême
- Temps réel
- Applications particulières (jeu, image, numérique, ...)

# Contenu

## Langage support Langage Common Lisp

- SBCL: Steele Bank Common Lisp <http://www.sbcl.org/>

## Support de cours

- Robert Strandh et Irène Durand :  
  Traité de programmation en Common Lisp
- Transparents

## Pourquoi Common Lisp ?

- Langage très riche (fonctionnel, symbolique, objet, impératif)
- Syntaxe simple et uniforme
- Sémantique simple et uniforme
- Langage programmable (macros, reader macros)
- Représentation de programmes sous la forme de données

- Normalisé par ANSI
- Programmation par objets plus puissante qu'avec d'autres langages

## Bibliographie

- Peter Seibel                      *Practical Common Lisp*  
Apress
- Paul Graham :                  *ANSI Common Lisp*  
Prentice Hall
- Paul Graham :                  *On Lisp*  
*Advanced Techniques for Common Lisp*  
Prentice Hall
- Sonya Keene :                  *Object-Oriented Programming in Common Lisp*  
*A programmer's guide to CLOS*  
Addison Wesley
- David Touretzky :              *Common Lisp :*  
*A Gentle introduction to Symbolic Computation*  
The Benjamin/Cummings Publishing Company, Inc

## Autres documents

The HyperSpec (la norme ANSI complète de Common Lisp, en HTML)

<http://www.lispworks.com/documentation/HyperSpec/Front/index.htm>

SBCL User Manual

<http://www.sbcl.org/manual/>

CLX reference manual (Common Lisp X Interface)

Guy Steele : Common Lisp, the Language, second edition  
Digital Press, (disponible sur WWW en HTML)

David Lamkins : Successful Lisp (Tutorial en-ligne)

# Historique de Common Lisp

Langage conçu par John McCarthy entre 1956 et 1959 au MIT pour des applications liées à l'intelligence artificielle (avec Fortran l'un des plus vieux langages toujours utilisés)

- Issu de la théorie du Lambda-Calcul de Church
- Dans les années 1970, deux dialectes : Interlisp et Maclisp
- Aussi : Standard Lisp, NIL, Lisp Machine Lisp, Le Lisp
- Travail pour uniformiser les dialectes : Common Lisp
- Normalisé par ANSI en 1994

# Common Lisp aujourd'hui

## Conférences

- ELS (Bordeaux 08, Milan 09, Lisbonne 10, Hambourg 11)  
<http://www.european-lisp-symposium.org/>
- Lisp50@OOPSLA <http://www.lisp50.org/>, 08
- ILC (Stanford 05, Cambridge 07, MIT 09, Reno 10)  
<http://www.international-lisp-conference.org/10>

Forums [fr.comp.lang.lisp](http://fr.comp.lang.lisp)

Chat (avec xchat par exemple) /serveur: [irc.freenode.net](http://irc.freenode.net)  
/join #lisp  
<http://webchat.freenode.net/> #lisp



# Logiciels/Entreprises utilisant CL

## — Entreprises et Logiciels commerciaux

ITA Software <http://www.itasoftware.com>

Igor Engraver Éditeur de partition musicales  
<http://www.noteheads.com>

RavenPack International  
<http://www.ravenpack.com/aboutus/employment.htm>

## — Plate-forme Web :

BioCyc Plate-forme d'accès BD biologiques (voies métaboliques/génome)  
<http://www.biocyc.org>

## — Logiciels libres

CARMA Case-based Range Management Adviser  
<http://carma.johnhastings.org/index.html>

BioBike Base de connaissance programmable pour la biologie  
<http://biobike.csbc.vcu.edu>

OpenMusic Langage visuel pour la composition musicale  
<http://repmus.ircam.fr/openmusic/home>

GSharp Éditeur de partitions musicales  
<http://common-lisp.net/project/gsharp>

Liste de logiciels libres <http://www.cliki.net/index>

# Calcul Symbolique

numérique/symbolique

Avec des bits on peut coder des nombres mais aussi des objects de type mot ou phrase

En Lisp,

- objects de base : sortes de mots appelés **atomes**,
- groupes d'atomes : sortes de phrases appelées **listes**.

Atomes + Listes = **Expressions symboliques (S-expr)**

- Lisp manipule des S-expr
- un programme Lisp est une S-expr, donc même représentation pour les programmes et les données.
- **Conséquence** : possibilité d'écrire des programmes qui se modifient ou modifient ou produisent des programmes.

# Applications de Calcul Symbolique

Toute application non numérique, en particulier

- Intelligence artificielle (systèmes experts, interfaces en langages naturel,...)
- Raisonnement automatique (preuves de théorèmes, preuves de programmes,...)
- Systèmes (implémentation de langages, traitement de texte,...)
- Calcul formel
- Jeux

Voir <http://www.cl-user.net>

# Comment faire pour apprendre à programmer ?

Il faut surtout lire beaucoup de code écrit par des experts.

Il faut lire la littérature sur la programmation. Il n'y en a pas beaucoup (peut-être 10 livres).

Il faut programmer.

Il faut maintenir du code écrit par d'autres personnes.

Il faut apprendre à être bien organisé.

# Standards de codage

Il faut s'habituer aux standards de codage

Pour Common Lisp, suivre les exemples dans la littérature, en particulier pour l'indentation de programmes qui est très standardisée (et automatisée).

Il faut pouvoir comprendre le programme sans regarder les parenthèses. L'indentation n'est donc pas une question de goût personnel.

Il faut utiliser SLIME (Superior Lisp Interaction Mode for Emacs) <http://common-lisp.net/project/slime/>

# Common Lisp est interactif

Common Lisp est presque toujours implémenté sous la forme de système **interactif** avec une **boucle d'interaction** (read-eval-print loop ou REPL).

Une interaction calcule la valeur d'une S-expression, mais une S-expression peut aussi avoir des effets de bord.

En particulier, un effet de bord peut être de modifier la valeur d'une variable, de créer une fonction, d'écrire sur l'écran, dans un flot...

Le langage n'a pas la notion de programme principal. Il est néanmoins possible de préciser la fonction à exécuter quand l'application est lancée.

Normalement, on lance Lisp une seule fois par séance.

## Common Lisp est interactif (suite)

Au CREMI, une séance est un TD ou une demi-journée de travail. Sur un ordinateur personnel, une séance peut durer des mois.

Le langage est conçu pour le développement interactif. Les instances d'une classes sont mises à jour quand la définition d'une classe change, par exemple.

La programmation fonctionnelle (sans effets de bord) est elle-même adaptée à l'écriture d'applications interactives.

# Lancer le système Lisp

```
irdurand@mcgonagall:~$ sbcl
```

```
This is SBCL 1.0.15, an implementation of ANSI Common Lisp.
```

```
More information about SBCL is available at <http://www.sbcl.org/>.
```

```
SBCL is free software, provided as is, with absolutely no warranty.  
It is mostly in the public domain; some portions are provided under  
BSD-style licenses. See the CREDITS and COPYING files in the  
distribution for more information.
```

```
* 1234
```

```
1234
```

```
* (+ 3 4)
```

```
7
```

```
*
```



## Quitter le système Lisp

```
* hello
```

```
debugger invoked on a UNBOUND-VARIABLE in thread #<THREAD "initial thread">  
The variable HELLO is unbound.
```

```
Type HELP for debugger help, or (SB-EXT:QUIT) to exit from SBCL.
```

```
restarts (invokable by number or by possibly-abbreviated name):  
  0: [ABORT] Exit debugger, returning to top level.
```

```
(SB-INT:SIMPLE-EVAL-IN-LEXENV HELLO #<NULL-LEXENV>)  
0] 0
```

```
* (quit)
```

```
irdurand@mcgonagall.emi.u-bordeaux1.fr:
```

# Lisp sous Emacs avec le mode SLIME

Superior Lisp Interaction Mode for Emacs

Beaucoup plus riche que le mode Lisp d'Emacs

Même aspect et fonctionnalités quelque soit le Lisp utilisé

Pour entrer, `M-x slime`

Pour sortir, taper une virgule (,) puis `quit` dans le mini-buffer

- Aide à l'indentation et à la syntaxe
- Compilation interactive de fonctions, de fichiers
- Documentation, complétion de symboles
- Débogage

Voir les modes (REPL) et (Lisp Slime) avec `c-h m`

# Programmation fonctionnelle

Common Lisp est un langage **mixte** (fonctionnel, impératif, orienté objets) mais ses ancêtres étaient **purement** fonctionnels.

Programmation fonctionnelle :

- entité de base = **fonction**
- **pas d'effets de bord** (pas de variables)
- structure de contrôle = **si-alors-sinon** + **récurtivité**

Pour obtenir rapidement des programmes corrects, utiliser la programmation fonctionnelle le plus possible

- Les programmes sont plus faciles à tester
- Programmation ascendante (bottom-up)
- Paramètres sont souvent des fonctions (fermetures)

**Inconvénients** : efficacité

# Expressions

Un **atome** peut-être

- un objet auto-évaluant,
- ou un symbole

Une **expression** (en anglais **form**) Common Lisp peut être :

- un atome
- ou une expression composée (avec des parenthèses).

# Expressions, analyse syntaxique

Une expression tapée à la boucle d'interaction est d'abord lue et **analysée syntaxiquement**.

Le résultat de cette analyse est une **représentation interne** de l'expression (S-expression).

La fonction responsable de cette analyse s'appelle **read**.

Cette fonction est **disponible** à l'utilisateur.

# Expressions, évaluation

La S-expression est ensuite **évaluée**, c'est à dire que sa valeur est calculée.

Cette évaluation peut donner des effets de bord.

Le **résultat** de l'évaluation est **un ou plusieurs objets Lisp**.

La fonction responsable de l'évaluation de S-expressions s'appelle **eval**.

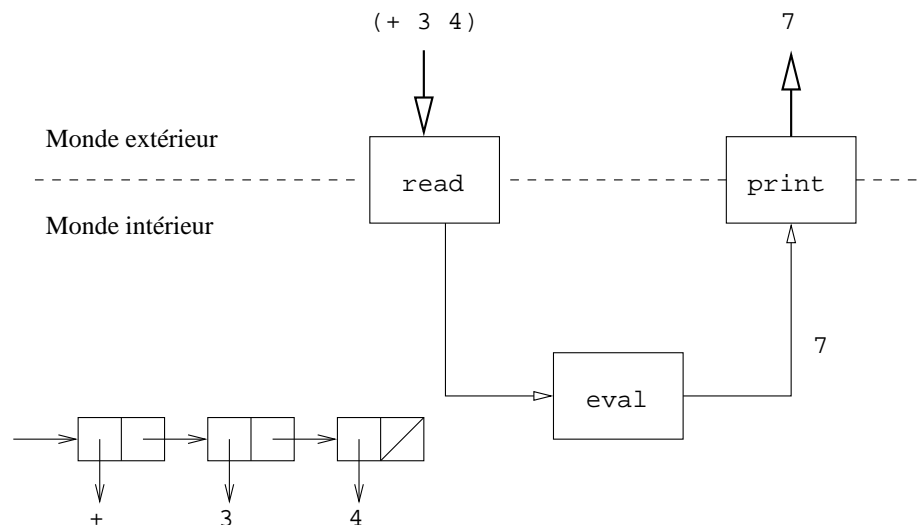
Cette fonction est **disponible** à l'utilisateur.

# S-expressions, affichage

Les objets résultant de l'évaluation sont ensuite **affichés** (ou imprimés) en **représentation externe**.

La fonction responsable de l'affichage s'appelle **print**.

Cette fonction est **disponible** à l'utilisateur.



# Objets auto-évaluants

Un objet auto-évaluant est la même chose qu'une constante. Le plus souvent, il s'agit de nombres, de caractères ou de chaînes de caractères.

```
CL-USER> 1234
```

```
1234
```

```
CL-USER> 6/8
```

```
3/4
```

```
CL-USER> #\c
```

```
#\c
```

```
CL-USER> "bonjour"
```

```
"bonjour"
```

```
CL-USER> #(1 2 3 4)
```

```
 #(1 2 3 4)
```



# Symboles

Si l'expression est un symbole, il sera considéré comme le nom d'une variable. la fonction `eval` va donc renvoyer sa valeur.

```
CL-USER> *standard-output*
```

```
#<SWANK-BACKEND::SLIME-OUTPUT-STREAM B1A0041>
```

```
CL-USER> nil
```

```
NIL
```

```
CL-USER> t
```

```
T
```

```
CL-USER> *features*
```

```
(:ASDF :SB-THREAD :ANSI-CL :COMMON-LISP :SBCL :UNIX :SB-DOC ...)
```

# Expressions composées

Une **expression composée** est une liste de sous-expressions entourées de parenthèses : `(op e1 e2 ... en)`

Le plus souvent, la première sous-expression est un symbole qui est le nom d'une **fonction**, d'une **macro** ou d'un **opérateur spécial**.

Les autres sous-expressions sont les **arguments** de la fonction, de la macro ou de l'opérateur spécial.

Liste des opérateurs spéciaux :

block	let*	return-from
catch	load-time-value	setq
eval-when	locally	symbol-macrolet
flet	macrolet	tagbody
function	multiple-value-call	the
go	multiple-value-prog1	throw
if	progn	unwind-protect
labels	progv	
let	quote	

## Expressions composées

```
CL-USER> (+ 3 4)
```

```
7
```

```
CL-USER> (length "hello")
```

```
5
```

```
CL-USER> (+ (* 3 4 2) (- 5 4) (/ 5 3))
```

```
80/3
```

```
CL-USER> (if (> 5 4) "oui" "non")
```

```
"oui"
```

```
CL-USER> (length *features*)
```

```
37
```

```
CL-USER> (floor 10.3)
```

```
10
```

```
0.3000002
```

Ici, `if` est un opérateur spécial, alors que `+`, `*`, `-`, `/`, `>`, `length` sont des fonctions.

# Définition de fonction

L'utilisateur peut définir des fonctions en utilisant la macro `defun` :

```
CL-USER> (defun doubler (x)
            (* 2 x))
```

**DOUBLER**

```
CL-USER> (doubler 10)
```

**20**

```
CL-USER> (doubler 3/2)
```

**3**

```
CL-USER> (doubler 1.5)
```

**3.0**

## Définition de fonction

```
CL-USER> (defun my-gcd (x y)
           (if (= x y)
               x
               (if (> x y)
                   (my-gcd y x)
                   (my-gcd x (- y x))))))
```

### MY-GCD

Cette indentation est obligatoire, car les programmeurs Lisp ne regardent pas les parenthèses. De plus, elle doit être automatique.

# Opérateurs booléens

Fonction : `not`

Macros : `or`, `and`

permettent de former toutes les expressions booléennes

```
CL-USER> (not 3)
```

```
NIL
```

```
CL-USER> (or nil (- 2 3) t 2)
```

```
-1
```

```
CL-USER> (and (= 3 3) (zerop 3) t)
```

```
NIL
```

```
CL-USER> (and (= 3 3) (zerop 0) t)
```

```
T
```

```
CL-USER> (or (and t nil) (or (not 3) 4))
```

```
4
```

# Définition de variables globales

```
CL-USER> (defvar *x* 1)
*X*
CL-USER> *x*
1
CL-USER> (defvar *x* 2)
*X*
CL-USER> *x*
1
CL-USER> (setf *x* 2)
2
CL-USER> (defparameter *y* 1)
*Y*
CL-USER> *y*
1
CL-USER> (defparameter *y* 2)
*Y*
CL-USER> *y*
2
```

Les \* autour des noms de variables globales font partie des standards de codage.

## Expressions avec effets de bord

Certaines expressions peuvent avoir des effets de bord. Il s'agit par exemple de l'affectation d'une variable ou de l'affichage (autre que par la boucle REP)

```
CL-USER> (setf x 3)
```

```
3
```

```
CL-USER> x
```

```
3
```

```
CL-USER> (+ (print 3) 4)
```

```
3
```

```
7
```

`setf` : nom d'une macro

`(setf x 3)` : *expression macro* (anglais : macro form)

`print` : nom d'une fonction avec un effet de bord

`(print 3)` : *expression fonction* (anglais : function form).



## Définition de constantes

```
CL-USER> (defconstant +avogadro-number+ 6.0221353d23)
```

```
+AVOGADRO-NUMBER+
```

```
CL-USER> (setf +avogadro-number+ 89)
```

```
Can't redefine constant +AVOGADRO-NUMBER+ .
```

```
[Condition of type SIMPLE-ERROR]
```

Les + autour des noms de constantes font partie des standards de codage.

## Documentation des symboles

```
CL-USER> (defvar *smic-horaire* 9.61 "smic horaire 01/01/2015")
```

```
*SMIC-HORAIRE*
```

```
CL-USER> (documentation '*smic-horaire*' 'variable)
```

```
"smic horaire 01/01/2015"
```

```
CL-USER> (defun delta (a b c)
```

```
    "discriminant of a quadratic equation"
```

```
    (- (* b b) (* 4 a c)))
```

```
DELTA
```

```
CL-USER> (documentation 'delta 'function)
```

```
"discriminant of a quadratic equation"
```

Raccourcis Emacs-Slime c-c c-d d, c-c c-d f ... (voir le mode)

## Retour sur les Symboles

Constantes : `abc`, `234hello`, `|ceci n'est pas un tube|`

Sauf avec la syntaxe `|...|`, le nom est en majuscules

```
CL-USER> (defvar abc 22)
```

```
ABC
```

```
CL-USER> (defvar 234abc 11)
```

```
234ABC
```

```
CL-USER> (defvar |ceci n'est pas un tube| 8)
```

```
|ceci n'est pas un tube|
```

```
CL-USER> (+ |ceci n'est pas un tube| 3)
```

```
11
```

# Quote

Souvent, en programmation symbolique, le **symbole** n'est utilisé que pour son **nom**.

On ne lui attribue **pas de valeur**.

Comment affecter à un symbole **s1**, un symbole **s2** et non pas la valeur du symbole **s2** ?

La solution est un **opérateur spécial** appelé **quote**

De manière générale, **quote** empêche l'évaluation de son argument (expression ou atome)

```
CL-USER> (defparameter s1 (quote s2))
```

**s1**

```
CL-USER> s1
```

**s2**

# Quote

```
CL-USER> (quote hello)
```

```
HELLO
```

```
CL-USER> (defparameter *symbole* (quote |ceci n'est pas un tube|))
```

```
*SYMBOLE*
```

```
CL-USER> *symbole*
```

```
|ceci n'est pas un tube|
```

Au lieu de taper `(quote expr)` on peut taper `'expr`.

```
CL-USER> 'hello
```

```
HELLO
```

```
CL-USER> (setf *symbole* 'hello)
```

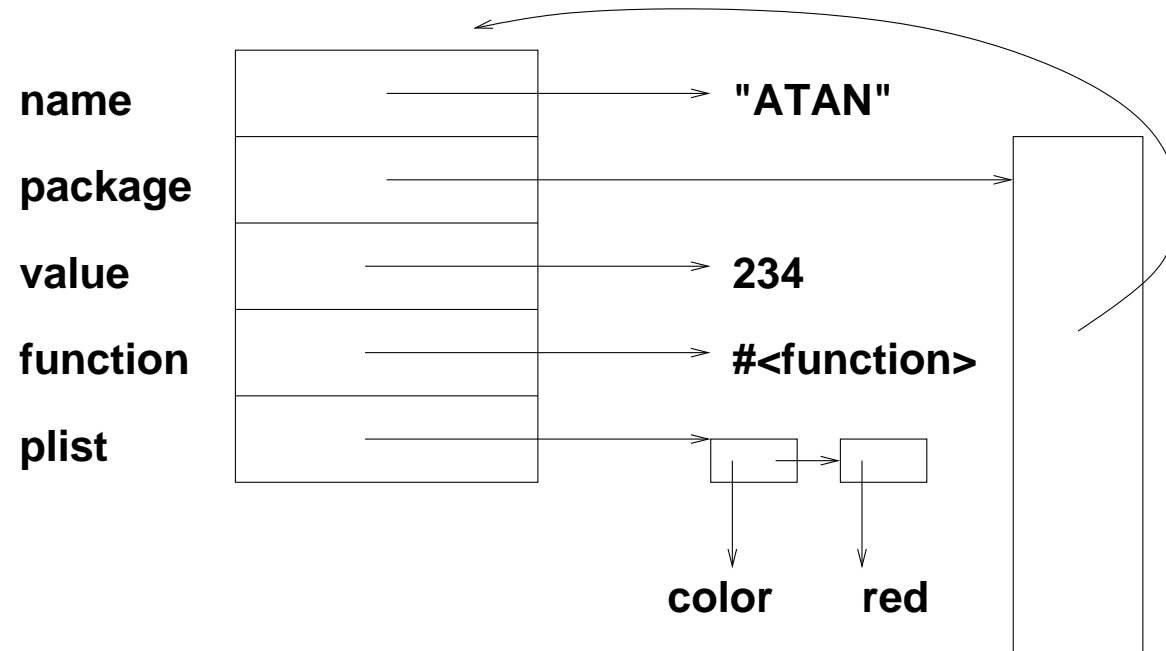
```
HELLO
```

```
CL-USER> '(+ 1 2 3)
```

```
(+ 1 2 3)
```

# Symboles

Représentation:



(f a1 a2 a2 ...) : valeur fonctionnelle du symbole f

Autres cas (+ f g), f : valeur de la variable f

# Symboles

```
CL-USER> length
```

```
Error in KERNEL::UNBOUND-SYMBOL-ERROR-HANDLER:  the variable LENGTH is  
[Condition of type UNBOUND-VARIABLE]
```

```
CL-USER> (setf length 100)
```

```
Warning:  Declaring LENGTH special.
```

```
100
```

Si on veut la fonction associée à un symbole dans une autre position que suivant une parenthèse ouvrante : opérateur spécial `function` ou raccourci `#'`.

```
CL-USER> (function length)
```

```
#<Function LENGTH 1049C581>
```

```
CL-USER> #'length
```

```
#<Function LENGTH 1049C581>
```

## Comparaison de symboles

Tester l'égalité entre deux symboles est une opération très rapide.

Tableau de hachage dans le paquetage (package) courant.

C'est la fonction `read` qui cherche dans le tableau de hachage et éventuellement crée le symbole.

```
CL-USER> (defvar *c* '|ceci n'est pas un tube|)
```

```
*C*
```

```
CL-USER> (eq *c* '|ceci n'est pas un tube|)
```

```
T
```

```
CL-USER> (eq *c* 'hello)
```

```
NIL
```



## Conditionnelles (1)

```
(if (f ...)
    (g ...)
    "hello")
```

```
(cond ((> x 3) (setf y (g ...)) (+ x y))
      (finished (+ x 15))
      (t 0))
```

```
(case (f ...)
      ((apr jun sept nov) 30)
      (feb (if (leap-year) 29 28))
      (t 31))
```

Comparaison des clés avec le prédicat `eq1`

## Définition de variables locales

Un **contexte** est une suite d'instructions dans un environnement définissant des variables locales

```
(defun f (n)
  (let ((v1 (sqrt n))
        (v2 (log n)))
    (* (+ v1 v2) (- v1 v2))))
```

```
(let* ((v1 (sqrt n))
       (v2 (log v1)))
  (* (+ v1 v2) (- v1 v2)))
```

la dernière expression est équivalente à :

```
(let ((v1 (sqrt n)))
  (let ((v2 (log v1)))
    (* (+ v1 v2) (- v1 v2))))
```

## Objets de première classe

Un objet est de **première classe** s'il peut être : la valeur d'une variable, l'argument d'un appel de fonction et retourné par une fonction.

Dans la plupart des langages les types de base (nombres, caractères,...) sont des objets de première classe.

En Lisp, les fonctions sont des objets de première classe.

```
CL-USER> #'1+
```

```
#<Function 1+ 103744B9>
```

```
CL-USER> (mapcar #'1+ '(4 3 4 8))
```

```
(5 4 5 9)
```

```
CL-USER> (reduce #'max '(4 3 5 8))
```

```
8
```

# Fonctions anonymes ou Abstractions

```
CL-USER> (lambda (x) (+ x 2))
```

```
#<FUNCTION (LAMBDA (X) (+ X 2)) 48907DD9>
```

```
CL-USER> (mapcar (lambda (x) (+ x 2)) '(3 4 5 6))
```

```
(5 6 7 8)
```

```
CL-USER> (find-if (lambda (x) (> x 5))
```

```
          '(5 8 3 9 4 2) :from-end t)
```

```
9
```

```
CL-USER> (defparameter *l* (list "fait" "il" "chaud"))
```

```
*L*
```

```
CL-USER> (sort *l*
```

```
          (lambda (x y) (> (length x) (length y))))
```

```
( "chaud" "fait" "il")
```

```
CL-USER> *l*
```

```
("il")
```

## Fonctions anonymes(suite)

```
CL-USER> (complement #'<)
```

```
#<Closure Over Function "DEFUN COMPLEMENT" 4891B851>
```

Comment appeler une fonction anonyme sans lui donner de nom ?

On ne peut pas écrire `((complement #'<) 1 3)`

Il faut utiliser la fonction `funcall` :

```
CL-USER> (funcall (complement #'<) 1 3)
```

```
NIL
```

```
CL-USER> (funcall (lambda (x) (* x x)) 5)
```

```
25
```

## Fonction retournant une fonction anonyme

```
CL-USER> (defun composition (f g)
            (lambda (x)
              (funcall g (funcall f x)))))
```

COMPOSITION

```
CL-USER> (composition #'sin #'asin)
```

#<CLOSURE (LAMBDA (X)) AFA5935>

```
CL-USER> (funcall (composition #'sin #'asin) 1)
```

0.99999994

```
CL-USER>
```

## Fonctions nommées

Les fonctions **nommées** sont des fonctions anonymes associées à (la valeur fonctionnelle d')un symbole.

On peut nommer automatiquement une fonction en la définissant avec la macro **defun**.

```
CL-USER> (defun plus-deux (x) (+ 2 x))
```

```
PLUS-DEUX
```

```
CL-USER> (plus-deux 4)
```

```
6
```

ou manuellement en affectant le champs fonction du symbole

```
CL-USER> (setf (symbol-function 'plus-trois) (lambda (x) (+ 3 x)))
```

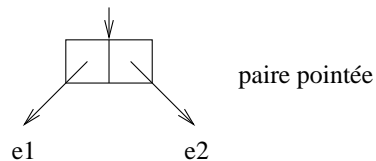
```
#<FUNCTION (LAMBDA (X) (+ 3 X)) 488FC3F9>
```

```
CL-USER> (plus-trois 4)
```

```
7
```

# Paires pointées

La **paire pointée** ou le **cons** est le type de base qui va servir à construire des **listes** ou des **structures arborescentes**.



Opérations : **constructeur cons**, **accesseurs car**, **cdr**

```
CL-USER> (cons 'a 'b)
```

```
(A . B)
```

```
CL-USER> (defparameter *p* (cons 'a 'b))
```

```
*P*
```

```
CL-USER> (car *p*)
```

```
A
```

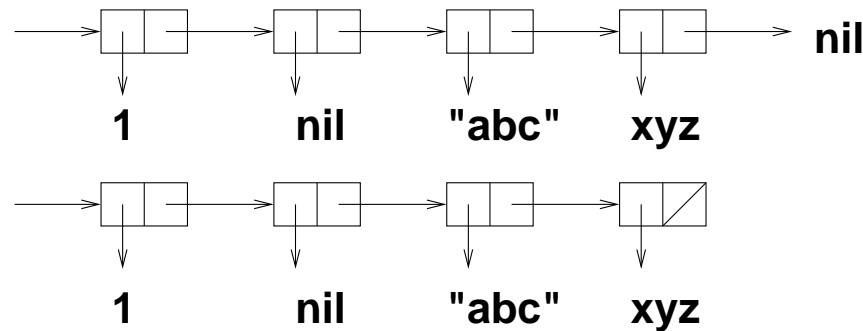
```
CL-USER> (cdr *p*)
```

```
B
```



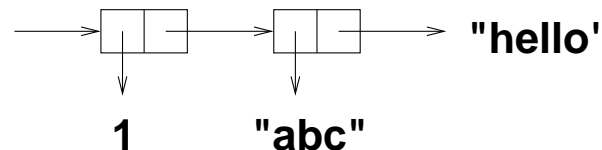
# Listes

Une liste est soit la liste vide `()` ou `nil`, soit une paire pointée donc le `cdr` est une liste.



Affichage par `print` : (1 NIL "abc" XYZ)

Une liste généralisée (anglais : dotted list) est terminée par un objet autre que `nil`.



Affichage par `print` : (1 "abc" . "hello")

## Listes (suite)

Une liste tapée à la boucle REPL est considérée comme une expression composée, et sera donc évaluée. Pour obtenir une liste sans l'évaluer, utiliser `quote`.

```
CL-USER> (+ 3 4)
```

```
7
```

```
CL-USER> '(+ 3 4)
```

```
(+ 3 4)
```

```
CL-USER> (defvar *l* '(+ 3 4))
```

```
*L*
```

```
CL-USER> *l*
```

```
(+ 3 4)
```

## Listes (suite)

Opérations de base : `cons`, `car`, `cdr`

```
CL-USER> (cons 1 (cons 2 (cons 3 nil)))
```

```
(1 2 3)
```

```
CL-USER> (cons 'hello '(how are you))
```

```
(HELLO HOW ARE YOU)
```

```
CL-USER> (setf *l* '(how are you))
```

```
(HOW ARE YOU)
```

```
CL-USER> (cons 'hello *l*)
```

```
(HELLO HOW ARE YOU)
```

```
CL-USER> *l*
```

```
(HOW ARE YOU)
```

```
CL-USER> (car *l*)
```

```
HOW
```

```
CL-USER> (cdr *l*)
```

```
(ARE YOU)
```

## Listes (suite)

Opérations plus complexes : `list`, `append`, `reverse`, ...

```
CL-USER> (defparameter *a* 'hello)
```

```
*a*
```

```
CL-USER> (setf *l* '(*a* 3 (+ 1 4)))
```

```
(*A* 3 (+ 1 4))
```

```
CL-USER> (setf *l* (list *a* 3 (+ 1 4)))
```

```
(HELLO 3 5)
```

```
CL-USER> (append *l* '(a b c))
```

```
(HELLO 3 5 A B C)
```

```
CL-USER> *l*
```

```
(HELLO 3 5)
```

```
CL-USER> (reverse *l*)
```

```
(5 3 HELLO)
```

```
CL-USER> *l*
```

```
(HELLO 3 5)
```

## Listes (suite)

Structure de contrôle de base : la récursivité

Utiliser `endp` pour terminer la récursion

```
(defun greater-than (l x)
  (if (endp l)
      '()
      (if (> (car l) x)
          (cons (car l) (greater-than (cdr l) x))
          (greater-than (cdr l) x))))
```

```
CL-USER> (greater-than '(5 3 6 4 5 3 7) 4)
(5 6 5 7)
```

## Listes (suite)

Mais on n'a presque jamais besoin de récursion sur les listes.

```
CL-USER> (remove-if-not (lambda (x) (> x 4)) '(5 3 6 4 5 3 7))  
(5 6 5 7)
```

```
CL-USER> (some (lambda (x) (and (zerop (rem x 3)) x)) '(1 3 5 7))  
3
```

```
CL-USER> (every #'oddp '(1 3 5 7))  
T
```

Attention :

```
CL-USER> (car nil)  
NIL
```

```
CL-USER> (cdr nil)  
NIL
```

# Atomes et listes

`(atom x) ≡ (not (consp x))`

`(listp x) ≡ (or (null x) (consp x))`

`(endp l) ≡ (null l)` mais utiliser `endp` pour une liste

CL-USER> `(atom #(1 2 3))`

T

CL-USER> `(atom "toto")`

T

CL-USER> `(atom '(a))`

NIL

CL-USER> `(listp #(1 2))`

NIL

CL-USER> `(listp '(1 2))`

T

## Listes (suite)

Construction : `cons`, `list`, `list*`, `append`, `copy-list`, `copy-tree`,  
`revappend`, `butlast`, `ldiff`, `subst`, `subst-if`, `subst-if-not`,  
`adjoin`, `union`, `intersection`, `set-difference`, `set-exclusive-or`,  
`subsetp`

Accès : `car`, `cdr`, `member`, `nth`, `nthcdr`, `last`, `butlast`, `cadadr`,  
`cdaaar`, ... `first`, `second`, ..., `tenth`

Autres : `length`, `subseq`, `copy-seq`, `count`, `reverse`, `concatenate`,  
`position`, `find`, `merge`, `map`, `some`, `every`, `notany`, `notevery`,  
`search`, `remove`, `remove-duplicates`, `elt`, `substitute`, `mismatch`



## Réversivité terminale

Un appel est dit **terminal** si aucun calcul n'est effectué entre son retour et le retour de la fonction appelante

Une fonction récursive est dite **réversive terminale**, si tous les appels récursifs qu'elle effectue sont terminaux.

Fonction RT  $\Rightarrow$  pas besoin d'empiler les appels récursifs

Si récursion **linéaire**, transformation possible en RT

Le plus souvent, ajout d'argument(s) jouant un rôle d'**accumulateur**

```
(defun fact-aux (n p)
  (if (zerop n)
      p
      (fact-aux (1- n) (* n p))))
```

```
(fact-aux 6 1)
```

## Fonctions funcall, apply, reduce

funcall function &rest args+ => result\*

```
CL-USER> (funcall #' + 3 5 2 7 3)
```

```
20
```

apply function &rest args+ => result\*

```
CL-USER> (apply #' + '(3 5 2 7 3))
```

```
20
```

```
CL-USER> (apply #' + '())
```

```
0
```

```
CL-USER> (apply #' + 3 5 '(2 7 3))
```

```
20
```

reduce function sequence &key key from-end start end initial-value => result

```
CL-USER> (reduce #'cons '(1 2 3 4 5))
```

```
(((((1 . 2) . 3) . 4) . 5)
```

```
CL-USER> (reduce #'cons '(1 2 3 4 5) :from-end t :initial-value nil)
```

```
(1 2 3 4 5)
```

```
CL-USER> (reduce #'cons '(0 1 2 3 4 5 6) :start 2 :end 5)
```

```
((2 . 3) . 4)
```

# Fonctions d'application fonctionnelles

mapcar, mapcan

```
CL-USER> (mapcar (lambda (x) (list (* 2 x))) '(1 2 3 4))  
((2) (4) (6) (8))
```

```
CL-USER> (mapcan (lambda (x) (list (* 2 x))) '(1 2 3 4))  
(2 4 6 8)
```

```
CL-USER> (mapcar #'append '((1 2) (3 4) (5 6)) '((a b) (c d)))  
((1 2 A B) (3 4 C D))
```

```
CL-USER> (mapcan #'append '((1 2) (3 4) (5 6)) '((a b) (c d)))  
(1 2 A B 3 4 C D)
```

# Listes d'association

Une liste d'association est une liste de paires (clé,valeur).

```
CL-USER> (defparameter *la*  
            '((blue . bleu) (red . rouge) (yellow . jaune)))
```

**\*la\***

```
CL-USER> (assoc 'red *la*)
```

**(RED . ROUGE)**

```
CL-USER> (assoc 'green *la*)
```

**NIL**

```
CL-USER> (defparameter *la1* '(("un" . 1) ("deux" . 2) ("trois" . 3)))
```

**\*LA1\***

```
CL-USER> (assoc "un" *la1*)
```

**NIL**

```
CL-USER> (assoc "un" *la1* :test #'equal)
```

**("un" . 1)**

Fonctions spécifiques : `assoc`, `acons`, `assoc-if`, `copy-alist`,  
`assoc-if-not`, `rassoc`, `rassoc-if`, `rassoc-if-not`, `pairlis`, `sublis`

# Égalités

Pour comparer les valeurs, utiliser :

`=` pour comparer des nombres  
`eq1` pour les atomes simples (hors tableaux)  
`equal` pour les listes et les chaînes de caractères  
`equalp` pour les structures, les tableaux, les tables de hachage

$= \subset \text{eq1} \subset \text{equal} \subset \text{equalp}$

```
CL-USER> (= #C(1 2) #C(1 2))
```

```
T
```

```
CL-USER> (eq1 nil ())
```

```
T
```

```
CL-USER> (equal '(1 2 3) '(1 2 3))
```

```
T
```

Pour tester si deux objects sont identiques, utiliser `eq`, (en particulier pour les symboles).

## Égalités (suite)

```
CL-USER> (defparameter *x* (list 'a))
```

```
*X*
```

```
CL-USER> (eq *x* *x*)
```

```
T
```

```
CL-USER> (eq1 *x* *x*)
```

```
T
```

```
CL-USER> (equal *x* *x*)
```

```
T
```

```
CL-USER> (eq *x* (list 'a))
```

```
NIL
```

```
CL-USER> (eq1 *x* (list 'a))
```

```
NIL
```

```
CL-USER> (equal *x* (list 'a))
```

```
T
```

# Retour sur la définition et l'appel de fonction

Lors d'un appel de fonction **tous** les arguments sont évalués.  
Le passage de paramètres se fait toujours par **valeur**.  
Les paramètres se comportent comme des variables **lexicales** (locales).

```
CL-USER> (defun f (l)
            (dotimes (i (length l))
              (format t "~A:~A  " i (car l))
              (pop l)))
```

**F**

```
CL-USER> (defparameter *l* '(a b c))
```

**\*L\***

```
CL-USER> (f *l*)
```

**0:A 1:B 2:C**

**NIL**

# Liste des paramètres d'une fonction

4 sortes de paramètres

1. paramètres **requis** (obligatoires)
2. éventuellement paramètres **facultatifs**, introduits par le mot **&optional**
3. éventuellement : paramètre **reste**, introduit par le mot **&rest**
4. éventuellement : paramètres **mot-clés**, introduits par le mot **&key**

```
CL-USER> (defun g (a b &optional c d)
           (list a b c d))
```

**G**

```
CL-USER> (g 1 2 3)
(1 2 3 NIL)
```



```
CL-USER> (defun h (e &rest l)
           (list e l))
```

H

```
CL-USER> (h 1)
```

```
(1 NIL)
```

```
CL-USER> (h 1 2 3 4)
```

```
(1 (2 3 4))
```

```
CL-USER> (defun k (&key (couleur 'bleu) image
                    (largeur 10) (hauteur 5))
           (list couleur largeur hauteur image))
```

K

```
CL-USER> (k)
```

```
(BLEU 10 5 NIL)
```

```
CL-USER> (k :image 'fleur :hauteur 20)
```

```
(BLEU 10 20 FLEUR)
```

## Macro assert

```
assert test-form [(place*) [datum-form argument-form*]]
```

```
CL-USER> (assert (zerop 0))
```

```
NIL
```

```
CL-USER> (assert (zerop 10))
```

```
The assertion (ZEROP 10) failed.  
[Condition of type SIMPLE-ERROR]
```

```
Restarts:
```

```
0: [CONTINUE] Retry assertion.
```

```
1: [ABORT-REQUEST] Abort handling SLIME request.
```

```
...
```

```
(defun fact (n)  
  (assert (and (integerp n) (> n -1)))  
  (if (zerop n)  
      1  
      (* n (fact (1- n)))))
```

## Macro `assert` (suite)

Utilisation pour l'écriture de fonctions de test

Une `fonction de test` :

- retourne `NIL` si ok
- signale une erreur sinon

```
(defun f (x)
  (make-list x :initial-element 0))
```

```
(defun test-f (x)
  (let ((l (f x)))
    (assert (listp l))
    (assert (= (length l) x))
    (assert (every #'numberp l))
    (assert (every #'zerop l)))))
```

```
CL-USER> (test-f 10)
```

```
NIL
```

# Programmation impérative : affectation

`setf pair* => result*`

Affectation de variables

```
CL-USER> (defparameter *v* 10)
```

```
*v*
```

```
CL-USER> (setf *v* 20)
```

```
20
```

```
CL-USER> (setf *v* '(1 2 3))
```

```
(1 2 3)
```

```
CL-USER> *v*
```

```
(1 2 3)
```

```
CL-USER>
```

```
(setf *v1* 3 *v2* (* 5 *v1*))
```

```
15
```

```
CL-USER> (list *v1* *v2*)
```

```
(3 15)
```

Affectation parallèle

```
CL-USER> (psetf *v1* 0 *v2* (* 2 *v1*))
```

```
NIL
```

```
CL-USER> (list *v1* *v2*)
```

```
(0 6)
```

`pair := place new-value`

Affectation de places

```
CL-USER>
```

```
(setf (car (last *v*)) 'fin)
```

```
FIN
```

```
CL-USER> *v*
```

```
(1 2 FIN)
```

```
CL-USER> (setf (cadr *v*) 'deux)
```

```
DEUX
```

```
CL-USER> *v*
```

```
(1 DEUX FIN)
```

`psetf pair* => nil`

# Opérations destructives sur les listes

`rplaca, rplacd, replace, nconc, nreconc, nreverse, push, pushnew,  
pop, nbutlast, nsubst, fill, delete, delete-duplicates, nsubst-if,  
nsubst-if-not, nset-difference, nset-exclusive-or, sort, nsubstitute.`

## Définitions de fonctions locales (`labels`)

```
(defun f (n)
  (assert (and (integerp n) (> n -1)))
  (labels ((aux (i fact)
            (if (zerop i)
                fact
                (aux (1- i) (* i fact))))))
    (aux n 1)))
```

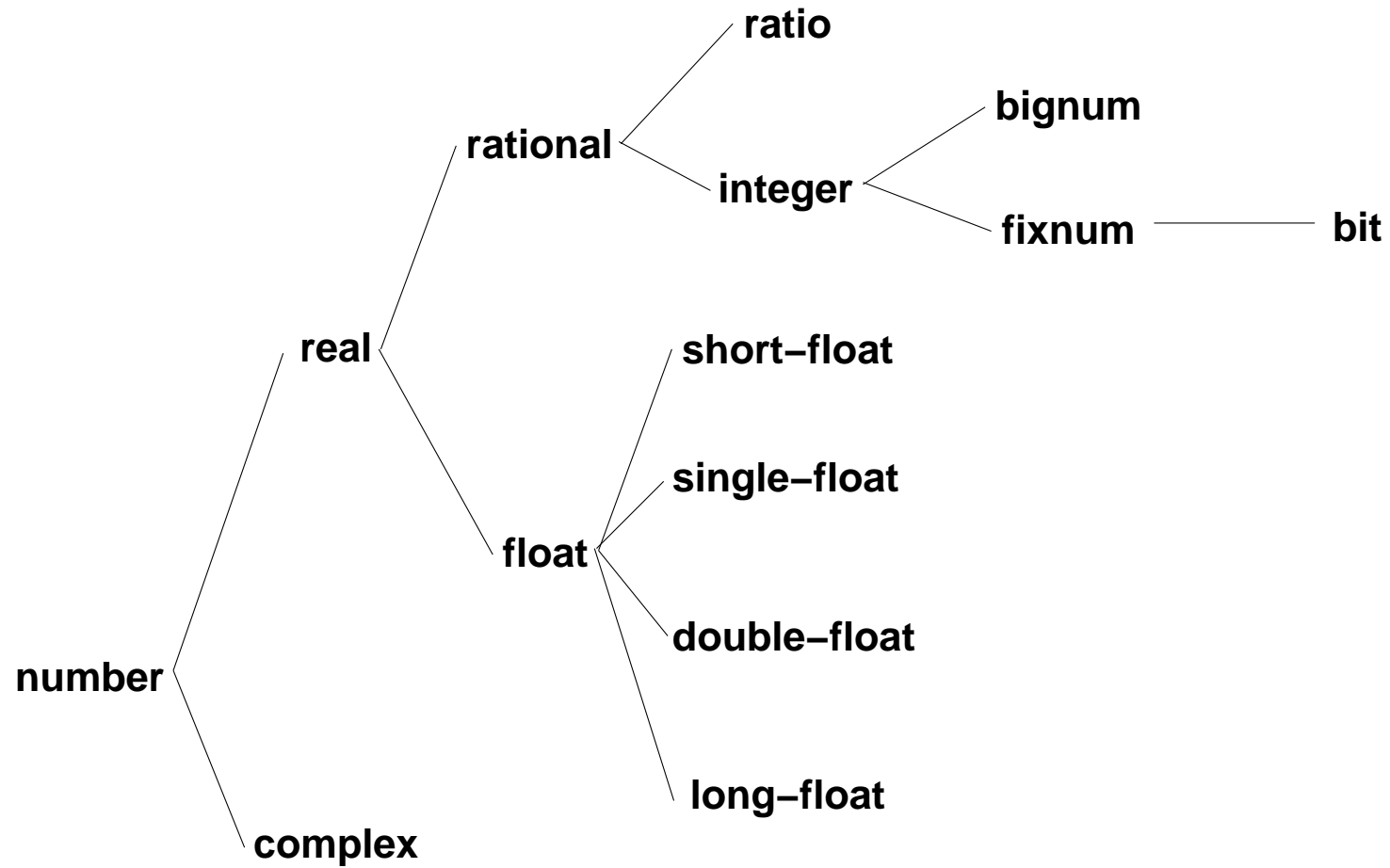
`Portée` des fonctions locales définies :

`labels` : la fonction est aussi connue dans son propre corps  
(récursivité possible)

## labels (example)

```
(defun leaves (tree)
  (let ((l '()))
    (labels ((aux (st)
              (if (atom st)
                  (push st l)
                  (progn (aux (car st)) (aux (cdr st))))))
      (aux tree)
      l)))
```

# Hiérarchie des nombres





# Nombres entiers

Précision arbitraire (**bignums**)

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

CL-USER> (fact 5)

120

CL-USER> (fact 40)

815915283247897734345611269596115894272000000000

# Nombres complexes

Constantes : `#c(4 5)`, `#c(1/2 3/4)`, `#c(4/6 2.0)`

Les opérations habituelles marchent `exp`, `expt`, `log`, `sqrt`, `isqrt`, `abs`, `phase`, `signum`, `sin`, `cos`, `tan`, `cis`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`

```
CL-USER> (sin (/ pi 6))
```

```
0.499999999999999994d0
```

```
CL-USER> (sin #c(3 4))
```

```
#C(3.8537378 -27.016813)
```

Exemples d'utilisation : transformée de Fourier

# Nombres flottants

```
CL-USER> 20.03
20.03
CL-USER> 20.4e3
20400.0
CL-USER> 20.4e-3
0.0204
CL-USER> (typep 20.03 'single-float)
T
CL-USER> (typep 20.03 'short-float)
T
CL-USER> (typep 20.03 'double-float)
NIL
CL-USER> (typep 20.4d-3 'double-float)
T
CL-USER> (typep 1.0d0 'double-float)
T
CL-USER> (typep 1.0s0 'short-float)
T
CL-USER>
```

# Tableaux

```
CL-USER> (make-array '(4 3))
```

```
#2A((0 0 0) (0 0 0) (0 0 0) (0 0 0))
```

```
CL-USER> (make-array '(4 3) :initial-element 5)
```

```
#2A((5 5 5) (5 5 5) (5 5 5) (5 5 5))
```

```
CL-USER> (defparameter *t*
```

```
  (make-array
```

```
    4
```

```
    :initial-contents (list (+ 3 4) "hello" 'hi t)))
```

```
#(7 "hello" HI T)
```

```
CL-USER> (aref *t* 1)
```

```
"hello"
```

```
CL-USER> (aref (make-array '(2 2) :initial-contents '((a1 a2) (b1 b2)))
```

```
A2
```

Les **vecteurs** sont des tableaux à une dimension.

```
CL-USER> (vectorp *t*)
```

```
T
```

## Valeurs multiples

```
CL-USER> (floor 75 4)
```

```
18
```

```
3
```

```
CL-USER> (multiple-value-bind (x y) (floor 75 4)  
      (list x y))
```

```
(18 3)
```

```
CL-USER> (values (cons 1 2) 'hello (+ 3 4))
```

```
(1 . 2)
```

```
HELLO
```

```
7
```

```
CL-USER> (multiple-value-call #'(+ (floor 75 4))
```

```
21
```

```
CL-USER> (multiple-value-list (floor 75 4))
```

```
(18 3)
```

# Tables de hachage

Création d'une table de hachage

```
make-hash-table &key test size rehash-size rehash-threshold => hash-table
```

```
CL-USER> (defparameter *ht* (make-hash-table))
```

```
*HT*
```

```
CL-USER> *ht*
```

```
#<HASH-TABLE :test EQL :COUNT 0 A005611>
```

Par défaut, les clés sont comparées à l'aide de `eql`

Si clés pas comparables avec `eql`, préciser la fonction de comparaison à l'aide du paramètre mot-clé `test` :

```
(make-hash-table :test #'equal)
```

par exemple si les clés sont des chaînes de caractères.

## Tables de hachage (suite)

Ajout d'entrées

```
CL-USER> (setf (gethash 'weight *ht*) 234)
```

234

```
CL-USER> (setf (gethash 'shape *ht*) 'round)
```

ROUND

```
CL-USER> (setf (gethash 'color *ht*) 'nil)
```

NIL

Nombre d'entrées : fonction `hash-table-count`

```
CL-USER> (hash-table-count *ht*)
```

3

## Tables de hachage (suite)

Accès à une entrée

```
CL-USER> (gethash 'weight *ht*)
```

```
234
```

```
T ; trouvé
```

```
CL-USER> (gethash 'size *ht*)
```

```
NIL
```

```
NIL ; pas trouvé
```

```
CL-USER> (gethash 'color *ht*)
```

```
NIL
```

```
T ; trouvé mais de valeur NIL
```

Fonction d'application : `maphash`

```
CL-USER> (let ((l '()))
```

```
    (maphash (lambda (k v) (push k l)) *ht*)
```

```
    l)
```

```
(WEIGHT SHAPE COLOR)
```



## Tables de hachage (suite)

Suppression d'une entrée

```
CL-USER> (remhash 'toto *ht*) ; clé inexistante
```

```
NIL
```

```
CL-USER> (remhash 'shape *ht*)
```

```
T ; suppression réussie
```

Macro `with-hash-table-iterator`

```
CL-USER> (with-hash-table-iterator (suivant *ht*)  
        (do () (nil) ;; boucle infinie  
            (multiple-value-bind (trouve? k v) (suivant)  
                (unless trouve? (return))  
                (format t "Cle: ~A, Valeur: ~A~%" k v )))))
```

```
Cle: COLOR, Valeur: NIL
```

```
Cle: WEIGHT, Valeur: 234
```

```
NIL
```

## Définition de fonctions locales (`flet`)

```
(defun print-couple (couple s)
  (format s "[~A,~A]" (first couple) (second couple)))
```

```
(defun print-couples (couples &optional (s t))
  (dolist (couple couples)
    (print-couple couple s)))
```

```
(defun print-couples (couples &optional (s t))
  (flet ((print-couple (couple)
           (format s "[~A,~A] " (first couple) (second couple))))
    (dolist (couple couples)
      (print-couple couple))))
```

Portée des fonctions locales définies :

`flet` : la fonction n'est connue que dans le corps du `flet`

## flet vs labels

```
(defun polynome (x)
  (flet ((carre (x) (* x x))
        (cube (x) (* x x x))))
  (+ (* 2 (cube x)) (* 3 (carre x)))))
```

```
(defun polynome (x)
  (labels ((carre (x) (* x x))
          (cube (x) (* x (carre x)))))
  (+ (* 2 (cube x)) (* 3 (carre x)))))
```

# Itération

```
(do ((i 0 (1+ i))
      (j n (1- j)))
    ((<= j 0) (+ x y))
    (format t "~a ~a~%" i j)
    (f i j))
```

```
(do* ...)
```

```
(dotimes (i 10 (+ x y))
  (format t "~a~%" i))
```

```
(dolist (elem (mapcar ...) 'done)
  (format t "~a~%" elem))
```

Voir aussi

sur des listes : `mapc`, `mapl`

sur des séquences : `map`

## Conditionnelles (2)

<code>(when (f ...)</code>	<code>(unless (f ...)</code>
<code>(g ...)</code>	<code>(g ...)</code>
<code>(setf x 23)</code>	<code>(setf x 23)</code>
<code>(+ x y))</code>	<code>(+ x y))</code>
<code>(typecase ...)</code>	

# Blocs (1)

Trois types de **blocs** : **tagbody**, **block**, **progn**, **prog1**, **prog2**

**tagbody** est la construction de base, mais n'est jamais utilisée directement par un utilisateur

**progn** (resp **prog1**, **prog2**) évalue les expressions en séquence et retourne la valeur de la dernière (resp. première, deuxième).

```
CL-USER> (progn
            (setf *l* '(5 2 6 3))
            (setf *l* (sort *l* #'>))
            (car *l*))
```

6

```
CL-USER> *l*
```

```
(6 5 3 2)
```

```
CL-USER> (prog1
            (pop *l*)
            (print *l*))
```

```
(5 3 2)
```

6

## Blocs (2)

`block` est comme un `progn` avec un nom.

```
CL-USER> (block found
           (print "hello")
           (when (> 3 2)
             (return-from found 23))
           (print "hi"))
```

"hello"

23

Certains blocs sont définis *implicitement*.

```
(defun f (l)
  (dolist (e l 'fin)
    (when (null e)
      (return-from f 'echap)))))
```

```
CL-USER> (f '(1 2 3 4))
```

FIN

```
CL-USER> (f '(1 2 nil 4))
```

ECHAP

# Variables

Variable : place mémoire nommée

Variables CL : dynamiquement typées (type vérifié à l'exécution)

2 sortes de variables : lexicale, spéciale

```
CL-USER> (defvar *x* 10)
```

```
*X*
```

```
CL-USER> (setf *x* (1- *x*))
```

```
9
```

```
CL-USER> (defparameter *l* '(1 2 3))
```

```
*L*
```



# Variables Lexicales

Est **lexicale**, toute variable qui n'est **pas** spéciale.

Paramètres d'une fonction (sauf si variable spéciale) :

```
(defun f (x y)
  (+ (* -3 x x) y 4))
```

Variables locales définies dans un **let**, **let\***, **do**, **do\*** :

```
(defun my-reverse (l)
  (let ((r1 '()))
    (dolist (e l r1)
      (push e r1))))
```

## Variables lexicales : exemples

Variable locale *permanente* (cf *static* en C)

```
CL-USER> (let ((l (list 'do 'mi 'sol)))  
           (nconc l l)  
           (defun note ()  
             (pop l))))
```

NOTE

```
CL-USER> (note)
```

DO

```
CL-USER> (note)
```

MI

```
CL-USER> (note)
```

SOL

```
CL-USER> (note)
```

DO

# Variables Spéciales

Une variable spéciale est globale et dynamique.

Une variable spéciale fonctionne comme une Pile.

```
CL-USER> *print-base*
```

```
10
```

```
CL-USER> (let ((*print-base* 2))  
           (format t "~A" 15))
```

```
1111
```

```
NIL
```

```
CL-USER> (defun affiche-nombre (n *print-base*)  
           (format t "~A" n))
```

```
AFFICHE-NOMBRE
```

```
CL-USER> (affiche-nombre 15 3)
```

```
120
```

```
NIL
```

## Variables Spéciales : attention

```
CL-USER> (defparameter x 10)
```

**X**

```
CL-USER> (describe 'x)
```

X is an internal symbol in #<PACKAGE "COMMON-LISP-USER">.

It is a special variable; its value is 10.

; No value

```
(defun f (x) ...)
```

**x** est spéciale !

Par convention, toujours utiliser des noms entourés d'astérisques (\*) pour les identificateurs de variables spéciales.

**\*print-base\*, \*print-circle\*, \*gc-verbose\*.**

## Variables spéciales : exemples

(Touresky p156-157)

```
CL-USER> (defparameter *a* 100)
```

```
*A*
```

```
CL-USER> (defun f (*a*) (list *a* (g (1+ *a*)))))
```

```
F
```

```
CL-USER> (defun g (b) (list *a* b))
```

```
G
```

```
CL-USER> (f 3)
```

```
(3 (3 4))
```

Changer temporairement la valeur d'une variable spéciale :

```
(let ((*standard-output* un-autre-flot))
```

```
...
```

```
...)
```

## Variables lexicales ou spéciales

```
CL-USER> (let ((x 10))  
           (defun f (y)  
             (+ x y)))
```

F

```
CL-USER> (defun g (x)  
          (f x))
```

G

```
CL-USER> (g 3)
```

13

Si on avait auparavant défini `x` comme variable spéciale

```
CL-USER> (defparameter x 5)
```

X

... idem ci-dessus

```
CL-USER> (g 3)
```

6

# Chaînes de caractères

Constantes : "abc", "ab\"c", ...

Une chaîne de caractères est un vecteur, un vecteur est un tableau mais aussi une séquence.

```
CL-USER> (aref "abc def" 3)
```

```
#\Space
```

```
CL-USER> (lower-case-p (aref "abc def" 4))
```

```
T
```

```
CL-USER> (char< (aref "abc def" 1) #\a)
```

```
NIL
```

## Tableaux ajustables

```
CL-USER> (setf *tab* (make-array '(2 3)))
```

```
#2A((0 0 0) (0 0 0))
```

```
CL-USER> (adjustable-array-p *tab*)
```

```
NIL
```

```
CL-USER> (setf *tab* (make-array '(2 3) :adjustable t))
```

```
#2A((0 0 0) (0 0 0))
```

```
CL-USER> (adjustable-array-p *tab*)
```

```
T
```

```
CL-USER> (setf (aref *tab* 0 0) 5)
```

```
5
```

```
CL-USER> *tab*
```

```
#2A((5 0 0) (0 0 0))
```

```
CL-USER> (adjust-array *tab* '(3 4))
```

```
#2A((5 0 0 0) (0 0 0 0) (0 0 0 0))
```



## Vecteurs avec `fill-pointer`

```
CL-USER> (setf *tab* (make-array 6 :fill-pointer 4))
```

```
#(0 0 0 0)
```

```
CL-USER> (length *tab*)
```

```
4
```

```
CL-USER> (setf (fill-pointer *tab*) 6)
```

```
6
```

```
CL-USER> (length *tab*)
```

```
6
```

```
CL-USER> (setf *tab* (make-array 6 :fill-pointer 4 :adjustable t))
```

```
#(0 0 0 0)
```

```
CL-USER> (adjust-array *tab* 20 :fill-pointer 10)
```

```
#(0 0 0 0 0 0 0 0 0 0)
```

# Macros

```
(defmacro pi! (var)  
  (list 'setf var 'pi))
```

Évaluation en deux phases : **macro-expansion** puis évaluation.

Important : macro-expansion faite (par le compilateur) avant la compilation proprement dite.

L'expression : **(pi! x)**

est transformée en l'expression : **(setf x pi)**

qui sera compilée à la place de l'expression initiale.

Les macros permettent donc de programmer de **nouvelles** instructions.

## Macros (suite)

Exemples de macros prédéfinies : `defun`, `defvar`, `defparameter`, `defclass`, `defmacro`, `push`, `pop`, `incf`, `decf`, `loop`,...

Résultat de la macro-expansion : `macroexpand-1`, `macroexpand`

```
CL-USER> (macroexpand-1 '(incf x))
```

```
(LET* ((#:G9838 1) (#:G9837 (+ X #:G9838)))  
  (SETQ X #:G9837))
```

```
T
```

```
CL-USER> (macroexpand-1 '(pop l))
```

```
(LET* ((#:G9836 L))  
  (PROG1 (CAR #:G9836) (SETQ #:G9836 (CDR #:G9836)) (SETQ L #:G9836)))
```

```
T
```

# Écrire de nouvelles macros

Avec la macro `defmacro`

```
CL-USER> (defparameter *x* 0)
```

```
*x*
```

```
CL-USER> (defmacro set-x (exp)
           (list 'setf '*x* exp))
```

```
SET-X
```

```
CL-USER> (macroexpand-1 '(set-x (1+ 2)))
```

```
(SETF *X* (1+ 2))
```

```
T
```

```
CL-USER> (set-x (1+ 2))
```

```
3
```

```
CL-USER> *x*
```

```
3
```

## Backquote

```
CL-USER> (setf *a* 1 *b* 2)
```

```
2
```

```
CL-USER> `(a is ,*a* and b is ,*b*)
```

```
(A IS 1 AND B IS 2)
```

```
CL-USER> (defmacro pi! (var)  
           `(setf ,var pi))
```

```
PI!
```

```
CL-USER> (pi! *x*)
```

```
3.141592653589793d0
```

```
CL-USER> *x*
```

```
3.141592653589793d0
```

```
CL-USER> (setf *l* '(1 2 3))
```

```
(1 2 3)
```

```
CL-USER> `(the list is ,*l* I think)
```

```
(THE LIST IS (1 2 3) I THINK)
```

## Suppression des parenthèses avec @

```
CL-USER> '(the elements are ,@*l* I think)
(THIS ELEMENTS ARE 1 2 3 I THINK)
```

```
CL-USER> (defmacro while (test &rest body)
           '(do ()
               ((not ,test))
               ,@body))
```

WHILE

```
CL-USER> (macroexpand-1 '(while (plussp *n*)
                                (prin1 *n*) (decf *n*)))
(DO () ((NOT (PLUSP *N*))) (PRIN1 *N*) (DECF *N*)))
```

T

```
CL-USER> (setf *n* 4)
```

4

```
CL-USER> (while (plussp *n*) (prin1 *n*) (decf *n*)))
```

4321

NIL

## Exemples

Remarque : quand le paramètre `&rest` correspond à une liste d'instructions correspondant au corps d'une instruction, on utilise de préférence `&body` au lieu de `&rest`.

```
CL-USER> (defmacro for (init test update &body body)
           '(progn
              ,init
              (while ,test
                  ,@body
                  ,update)))
```

FOR

```
CL-USER> (for (setf *i* 4) (>= *i* 0) (decf *i*) (prin1 *i*))
```

43210

NIL

## Difficultés : capture de variables

```
;;; incorrect
(defmacro ntimes (n &body body)
  '(do ((x 0 (1+ x)))
        ((>= x ,n))
        ,@body))
```

```
CL-USER> (let ((x 10))
           (ntimes 5
                 (incf x))
           x)
```

10

```
CL-USER> (gensym)
```

#:G4551

```
CL-USER> (gensym)
```

#:G4552



## Difficultés : évaluations multiples

```
;;; incorrect malgré le gensym
(defmacro ntimes (n &body body)
  (let ((g (gensym)))
    `(do ((,g 0 (1+ ,g)))
        ((>= ,g ,n)
         ,@body))))
```

```
CL-USER> (let ((v 10))
           (ntimes
            (decf v)
            (format t "*"))))
```

\*\*\*\*\*

NIL

## Difficultés

```
;;; enfin correct
(defmacro ntimes (n &body body)
  (let ((g (gensym))
        (h (gensym)))
    `(let ((,h ,n))
      (do ((,g 0 (1+ ,g)))
          ((>= ,g ,h))
        ,@body))))
```

## Quand écrire une macro

- création de nouvelles instructions (cf while)
- contrôle de l'évaluation des arguments (none, once)
- efficacité (inlining)
- ...

Inconvénients :

- difficulté de relecture du code
- se rappeler comment les arguments sont évalués
- ...

## Exemple : macro `defcommand`

```
(defvar *commands* (make-hash-table :test #'eq) "table of commands")
```

Nous souhaitons pouvoir faire quelque chose comme ceci :

```
(defcommand com-case ((ligne entier) (colonne entier))  
  (format t "aller à la case (~A,~A) ~%" ligne colonne)  
  (values ligne colonne)))
```

qui se traduise en quelque chose qui ressemble à :

```
(PROGN  
  (DEFUN COM-CASE (LIGNE COLONNE)  
    (FORMAT T "aller à la case (~A,~A) ~%" LIGNE COLONNE)  
    (VALUES LIGNE COLONNE))  
  (SETF (GETHASH 'COM-CASE *COMMANDS*)  
    (LAMBDA NIL  
      (COM-CASE  
        (PROGN (PRINC "ligne (entier): ") (READ))  
        (PROGN (PRINC "colonne (entier): ") (READ)))))))
```

## Invocation d'une commande

```
(defun invoke-command (name)
  (funcall (gethash name *commands*)))
```

```
CL-USER> (invoke-command 'com-case)
```

```
ligne (entier): 4
```

```
colonne (entier): 5
```

```
aller à la case (4,5)
```

```
4
```

```
5
```

```
CL-USER>
```

## Vers une solution pour `defcommand` :

```
CL-USER> (setf *arguments* '((ligne entier) (colonne entier)))
((LIGNE ENTIER) (COLONNE ENTIER))
CL-USER> (mapcar #'car *arguments*)
(LIGNE COLONNE)
CL-USER> (mapcar (lambda (arg)
                    (let ((name (car arg))
                          (type (cadr arg)))
                      '(progn
                        (princ
                          ,(format nil "~a (~a): "
                                   (string-downcase name)
                                   (string-downcase type)))
                        (read))))
              *arguments*)
((PROGN (PRINC "ligne (entier): ") (READ))
 (PROGN (PRINC "colonne (entier): ") (READ)))
```

## Une solution possible pour `defcommand` :

```
(defmacro defcommand (name arguments &body body)
  '(progn
    (defun ,name ,(mapcar #'car arguments) ,@body)
    (setf (gethash ',name *commands*)
          (lambda ()
            (,name
             ,@(mapcar
                  (lambda (arg)
                    (let ((name (car arg))
                        (type (cadr arg)))
                      '(progn
                        (princ
                          ,(format nil "~a (~a): "
                                (string-downcase name)
                                (string-downcase type)))
                        (read))))
                  arguments))))))
```