Programmation Fonctionnelle et projet

Responsable et cours :	Robert Strandh
TD :	Robert Strandh David Sherman Vincent Boudet

Planning :	3 semaines (6, 7, 9) 10,5h de cours 14h de TD 14h de TD (projet) 25h de travail individuel

Web : http://dept-info.labri.fr/~strandh/
(suivre le lien Programmation Fonctionnelle)

Support de cours

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

Transparents

Bibliographie

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.

Autres documents

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

David Lamkins : Successful Lisp (Tutorial en-ligne)

CMUCL user's guide

CLX reference manual

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

Objectifs et contenu

Objectif : Maîtriser un certain nombre de méthodes et techniques de la programmation fonctionnelle et symbolique.

Contenu : Langage Common Lisp
Types : symbole, liste, ...
Fonctions de base
Techniques élémentaires de programmation

But de l'enseignement de programmation

Méthodes et techniques pour l'écriture d'applications maintenables, réutilisables, lisibles, modulaires, générales, élégantes.

Langages, mécanismes et outils adaptés à ces méthodes et techniques

Aspects de la programmation non enseignés :

- Efficacité extrême
- Temps réel
- Applications particulières (jeux, image, numériques, ...)
- Détails de langages particuliers
- Implémentations particulières

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.

Pourquoi le langage Common Lisp?

- Langage très riche
- Syntaxe simple et uniforme
- Sémantique simple et uniforme
- Langage programmable (macros, reader macros)
- Représentation de programmes sous la forme de données
- Normalisé par ANSI, y compris CLOS
- Programmation par objets plus puissante qu'avec d'autres langages

Définition : langage interprété (deux définitions possibles)

1. Un langage pouvant être implémenté par un interprète
2. Un langage forcement implémenté par un interprète

Si 1, alors tous les langages sont interprétés. Si 2, alors aucun langage n' est interprété.

Alors, "langage interprété" n'a pas de sens.

Historique de Common Lisp

Langage inventé à MIT par John McCarthy en 1959 (donc avec Fortran l'un des plus vieux langages toujours utilisés)

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 1987

Processus de standardisation de Common Lisp

1. Introduction d'un concept nouveau
2. Distribution sous la forme de fonctions+macros
3. Utilisation par les programmeurs pendant plusieurs années
4. Proposition de normalisation du concept
5. Intégration du concept dans la norme

Tour guidé de Common Lisp

- Langage dynamique
- Sémantique par référence uniforme
- Gestion automatique de la mémoire
- Mots clés (passage de paramètres)
- Compilation
- Macros
- Programmation orientée objets

Tour guidé de Common Lisp

Types de données

- entiers à précision arbitraire
- rationnels, flottants
- complexes rationnels et complexes flottants
- la plupart des opérations arithmétiques marchent sur les complexes
- symboles
- caractères, chaînes de caractères
- séquences, listes, tableaux
- tableaux de bits, tableaux d'hachage
- fonctions
- structures, classes, métaclasses
- séries, générateurs/consommateurs
- flots

Tour guidé de Common Lisp (suite)

Structures de contrôle

- conditionnels (*if*, *when*, *unless*, *cond*, *case*)
- récursivité
- itération (*while*, *do*, *dolist*, *loop*)
- blocs (*prog1*, *progn*, *tagbody*, *let*, *let**)
- fermetures
- fonctions génériques
- valeurs multiples
- exceptions

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.

Il faut pouvoir comprendre le programme sans regarder les ()

Utiliser lisp-mode de GNU Emacs

L'indentation n'est donc pas une question de goût personnel

Programmation fonctionnelle

Common Lisp est un langage mixte (fonctionnel, impératif, orienté objets)

Pour un bon résultat, utiliser la programmation fonctionnelle le plus possible

Par programmation fonctionnelle, il faut comprendre la programmation sans effets de bord

Les programmes sont plus faciles à tester

Programmation ascendante (bottom-up)

Paramètres sont souvent des fonctions (fermetures)

Sémantique par référence uniforme

Manipulation d'objets par des références (pointeurs)

Le compilateur peut remplacer une référence par l'objet correspondant si cela ne change pas la sémantique

Implémentation d'objets typés par le marquage (tags)

Les petits objets (fixnums, caractères, ...) sont représentés directement dans le pointeur.

La sémantique par référence uniforme fait que les pointeurs explicites ne sont pas nécessaires ("pas de pointeurs")

Les déclarations peuvent aider le compilateur à remplacer les références par les objets correspondants

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 expression, mais une expression peut aussi avoir des effets de bord.

En particulier un effet de bord peut être de modifier la valeur d'une variable ou de créer une fonction.

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 exécute la commande Lisp une seule fois par séance.

Common Lisp est interactif (suite)

À l'ENSEIRB, une séance est un TD ou une demie 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.

Il faut écrire les applications pour faciliter l'interactivité aussi. Souvent cela nécessite la programmation sans effets de bord.

Expressions

Une expression (ou une forme) Common Lisp peut être :

- un objet auto-évaluant,
- un symbole, ou
- une expression composée.

Expressions, analyse syntaxique

Une expression tapée à la boucle d'interaction est d'abord analysée syntaxiquement. Le résultat de cette analyse est une représentation interne de l'expression. La fonction responsable de cette analyse s'appelle *read*.
Cette fonction est disponible à l'utilisateur.

Expressions, évaluation

La représentation interne de l'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 d'expression s'appelle *eval*.
Cette fonction est disponible à l'utilisateur.

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.

Lancer et quitter le système Lisp

```
napperon: lisp
CMU Common Lisp release x86-linux...
...
* 1234

1234
*
```

Lancer et quitter le système Lisp

```
* hello

Error in ...:  the variable HELLO is unbound.

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)
...
0] 0
* (quit)
napperon:
```

Objets auto-évaluants

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

```
* 1234

1234
* 6/8

3/4
* #\c

#\c
*
```

Objets auto-évaluants

```
* "bonjour"

"bonjour"
* #(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.

```
* *standard-output*

#<Synonym Stream to SYSTEM:*STDOUT*>
* nil

NIL
* t

T
*
```

Symboles

* `*features*`

```
(:MK-DEFSYSTEM :CLX-MIT-R5 ...)
```

* `+`

```
*FEATURES*
```

*

Expressions composées

Une expression composée est une liste de sous-expressions entourées de parenthèses. 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.

Expressions composées

```
* (+ 3 4)

7
* (length "hello")

5
* (+ (* 3 4 2) (- 5 4) (/ 5 3))

80/3
*
```

Expressions composées

```
* (if (> 5 4) "oui" "non")

"oui"
* (length *features*)

37
*
```

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

Expressions avec effets de bord

Certains 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 REPL)

```
* (setf x 3)
Warning:  Declaring X special.

3
* x

3
* (+ (print 3) 4)

3
7
*
```

Expressions avec effets de bord

Ici, *setf* est le nom d'une macro et *(setf x 3)* est une expression macro (anglais : macro form); *print* est le nom d'une fonction avec un effet de bord et *(print 3)* est une expression fonction (anglais : function form).

Définition de fonction

L'utilisateur peut définier des fonctions en utilisant la macro *defun* :

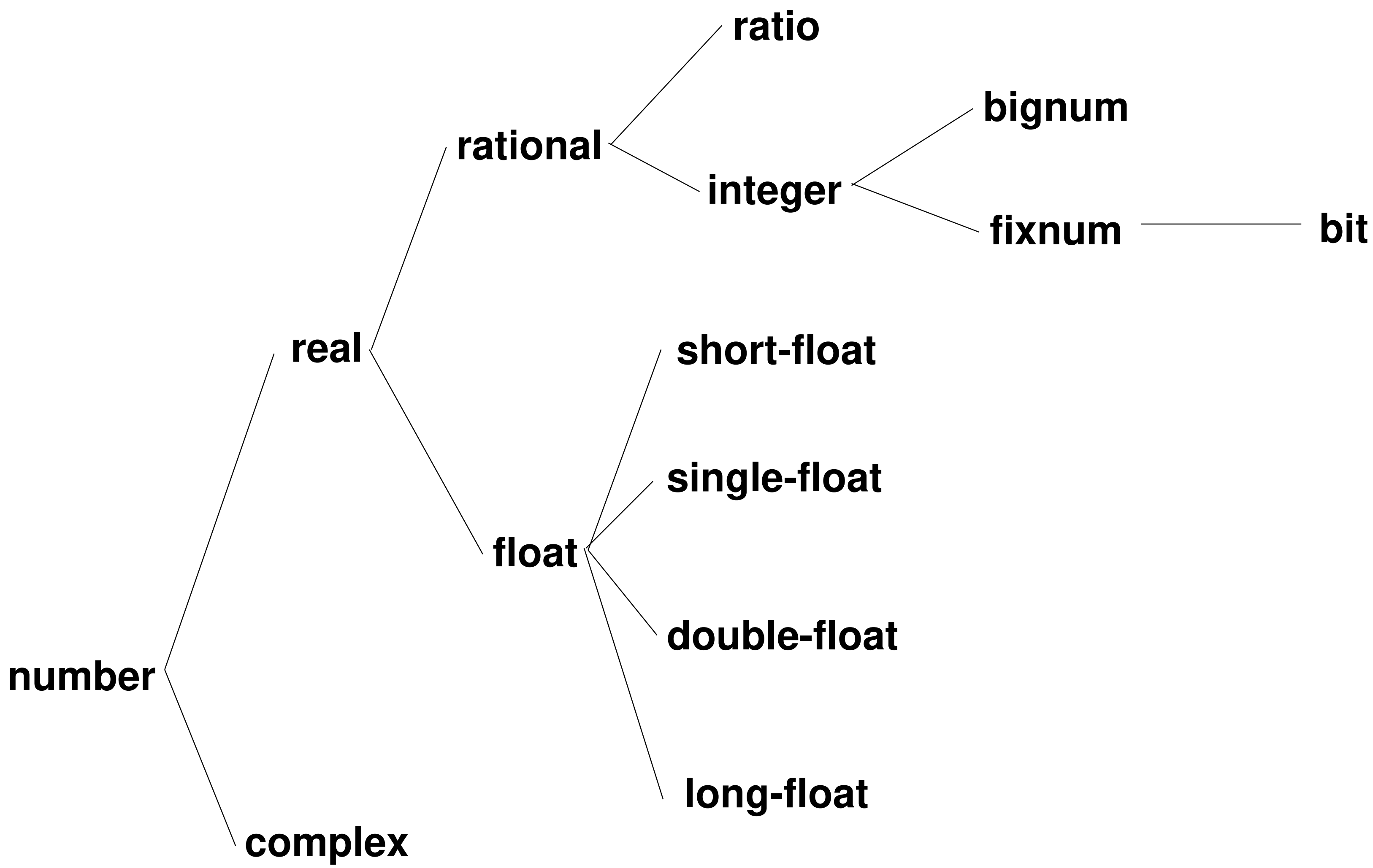
```
* (defun double (x)
    (* 2 x))

DOUBLE
* (double 10)

20
* (double 3/2)

3
* (double 1.5)

3.0
*
```

Définition de fonction

```
* (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.

Nombres


ratio
bignum
rational
integer
fixnum
bit
real
short-float
single-float
float
double-float
number
long-float
complex

Nombres: Entiers

Précision arbitraire (bignums)

```
* (defun fact (n)
    (if (= n 0)
        1
        (* n (fact (1- n)))))

FACT
* (fact 5)

120
* (fact 40)

815915283247897734345611269596115894272000000000
*
```

Nombres: Entiers

Opérations logiques sur les entiers: vecteurs infinis de bits ayant soit un nombre fini de "1", soit un nombre fini de "0".

logior, *logxor*, *logand*, *logeqv*, *lognand*, *lognor*, *logandc1* *logandc2*, *logorc1*, *logorc2*, *boole*, *lognot*, *logtest*, *logbitp* *ash*, *logcount*

```
* (logior 14 3)

15
* (ash 200000000000000000000 2)

800000000000000000000
*
```

Nombres: Entiers

Les fixnums sont possible à représenter dans le pointeur

La manipulation de "bytes" d'un nombre de bits arbitraire

Avec les declarations, possible de représenter de façon compacte (par exemple dans les tableaux)

En particulier des tableaux de bits

```
* (fixnump (ash 1 28))

T
* (fixnump (ash 1 29))

NIL
* (make-array '(3 5) :element-type 'bit)

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

Nombres: ratio

Permets de représenter le résultat d'une grande classe de calculs de façon exacte. Exemples:

- La programmation linéaire avec la méthode simplex
- Élasticité de widgets

Constantes : *1/3*, *2/10*, . . .

```
* (+ 1/3 4/5)

17/15
* (/ 4/3 2/3)

2
*
```

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*

```
* (sin (/ pi 6))

0.49999999999999994d0
* (sin #c(3 4))

#C(3.8537378 -27.016813)
*
```

Exemples d'utilité : transformée de Fourier

Booléens

Constantes : *nil*, *t*

nil = *()*

n'importe quel objet sauf *nil* est considéré équivalent à *t* pour les tests

nil et *t* sont des symboles

la valeur de la variable *nil* est le symbole *nil* la valeur de la variable *t* est le symbole *t*

```
* (if nil 2 3)

3
*
```

Booléens

```
* (if 0 2 3)

2
* (< 2 4 7 123)

T
*
```

Symboles

Constantes : *abc*, *234hello*, |*ceci n'est pas une pipe*|

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

```
* (defvar abc 22)

ABC
* (defvar 234abc 11)

234ABC
* (defvar |ceci n'est pas une pipe| 8)

|ceci n'est pas une pipe|
* (+ |ceci n'est pas une pipe| 3)

11
*
```

Quote

Comment est-ce que la valeur d'un symbole peut-être un autre symbole? (la valeur de *nil* est *nil*, et la valeur de *t* est *t*)

La solution est un opérateur spécial appelé *quote*

Quote

```
* (defvar *a* (quote hello))

*A*
* *a*

HELLO
* (defvar hello (quote |ceci n'est pas une pipe|))

HELLO
* hello

|ceci n'est pas une pipe|
*
```

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

Symboles

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

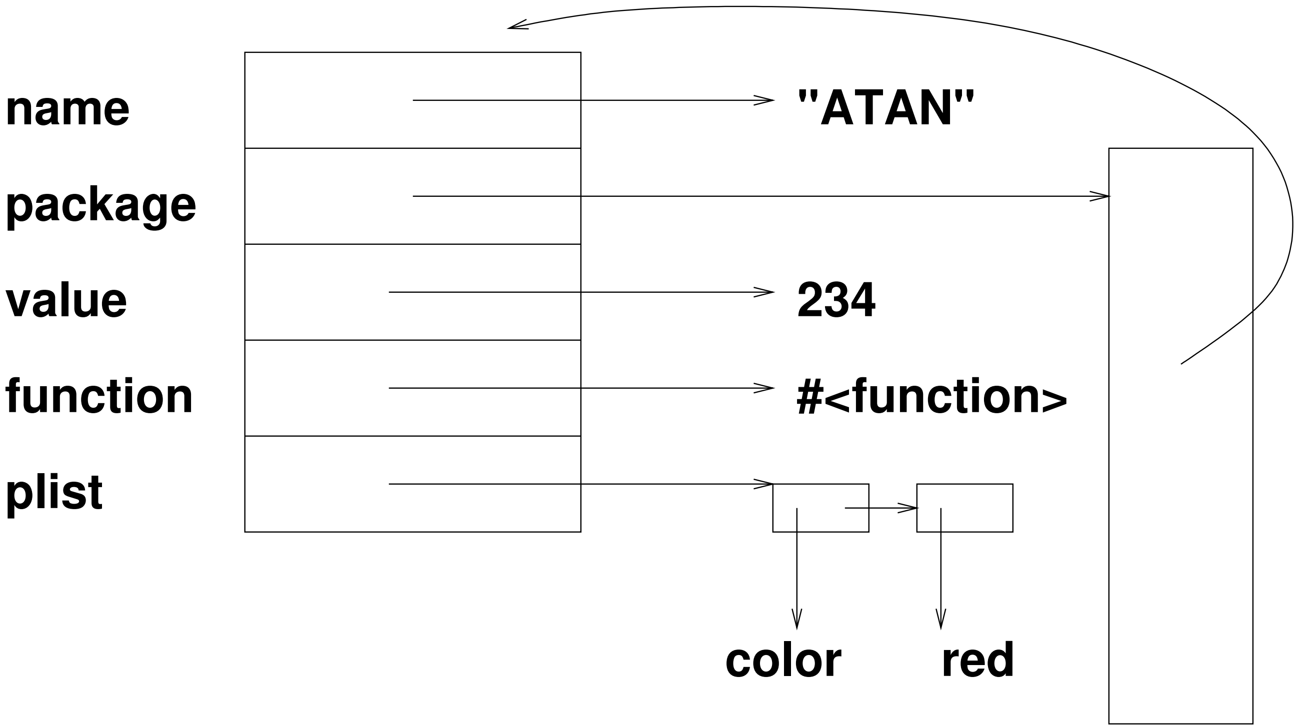
Tableau d'hachage dans le paquetage (package) courant.

C'est la fonction *read* qui cherche le tableau d'hachage et crée le symbole.

```
* (defvar *c* '|ceci n'est pas une pipe|)

*C*
* (eq *c* '|ceci n'est pas une pipe|)

T
* (eq *c* 'hello)

NIL
*
```

Symboles

Représentation:



Symboles: property list (plist)

Une base de données simple Exemple:

```
* (setf (get 'jean 'pere) 'paul)

PAUL
* (setf (get 'paul 'profession) 'plombier)

PLOMBIER
* (get (get 'jean 'pere) 'profession)

PLOMBIER
*
```

Caractères

Constantes : *#\a*, *#\b*, *#\Space*, *#\Newline*, ...

Fonctions :

alpha-char-p, *upper-case-p*, *lower-case-p*, *both-case-p*, *digit-char-p*, *alphanumericp*, *char=*, *char/=*, *char<*, *char>*, *char<=*, *char>=*, ...

Chaînes de caractères

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

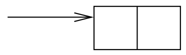
Une chaîne de caractères est un vecteur, et un vecteur est un tableaux mais aussi une séquence.

```
* (aref "abc def" 3)

#\Space
* (lower-case-p (aref "abc def" 4))

T
* (char< (aref "abc def" 1) #\a)

NIL
*
```

Listes

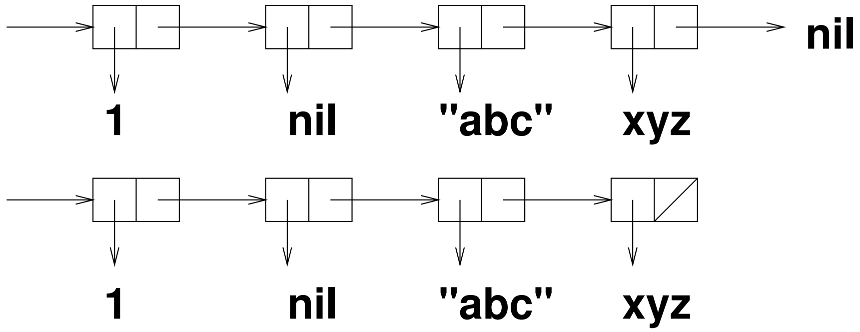
Une list est une séquence

Le type de base est la paire ou le *cons*



Listes

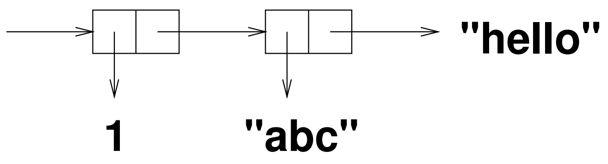
Une liste (anglais: proper list) est une suite de paires terminée par *nil*.



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

Listes

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*.

Listes (suite)

```
* (+ 3 4)

7
* '(+ 3 4)

(+ 3 4)
* (defvar *my-list* '(+ 3 4))

*MY-LIST*
* *my-list*

(+ 3 4)
*
```

Listes (suite)

Operations de base: *cons*, *car*, *cdr*

```
* (cons 1 (cons 2 (cons 3 nil)))

(1 2 3)
* (cons 'hello '(how are you))

(HELLO HOW ARE YOU)
*
```

Listes (suite)

```
* (setf *my-list* '(how are you))

(HOW ARE YOU)
* (cons 'hello *my-list*)

(HELLO HOW ARE YOU)
* *my-list*

(HOW ARE YOU)
* (car *my-list*)

HOW
* (cdr *my-list*)

(ARE YOU)
*
```

Listes (suite)

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

```
* (setf *a* 'hello)

HELLO
* (setf *my-list* '(*a* 3 *standard-output*))

(*A* 3 *STANDARD-OUTPUT*)
* (setf *my-list* (list *a* 3 *standard-output*))

(HELLO 3 #<Synonym Stream to SYSTEM:*STDOUT*>)
* (append *my-list* '(a b c))

(HELLO 3 #<Synonym Stream to SYSTEM:*STDOUT*> A B C)
* *my-list*
```

Listes (suite)

```
(HELLO 3 #<Synonym Stream to SYSTEM:*STDOUT*>)
* (reverse *my-list*)

(#<Synonym Stream to SYSTEM:*STDOUT*> 3 HELLO)
* *my-list*

(HELLO 3 #<Synonym Stream to SYSTEM:*STDOUT*>)
*
```

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))))

GREATER-THAN
* (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.

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

(5 6 5 7)
*
```

Listes (suite)

Attention:

```
* (car nil)

NIL
* (cdr nil)

NIL
*
```

Atomes et listes

```
(defun my-atom (x)
  (not (consp x)))

(defun my-listp (x)
  (or (null x) (consp x)))
```

Égalité

```
* (defparameter *x* (list 'a))

*X*
* (eq *x* *x*)

T
* (eql *x* *x*)

T
* (equal *x* *x*)

T
*
```

Égalité

```
* (eq *x* (list 'a))

NIL
* (eql *x* (list 'a))

NIL
* (equal *x* (list 'a))

T
*
```

Listes (suite)

Construction: *list**, *append*, *copy-list*, *copy-alist*, *copy-tree*, *revappend*, *nconc*, *nreconc*, *push*, *pushnew*, *pop*, *butlast*, *nbutlast*, *ldiff*, *rplaca*, *rplacd*, *subst*, *subst-if*, *subst-if-not*, *nsubst*, *nsubst-if*, *nsubst-if-not*, *sublis*, *member*, *tailp*, *adjoin*,*union*, *intersection*, *set-difference*, *nset-difference*, *set-exclusive-or*, *nset-exclusive-or*, *subsetp*, *acons*, *pairlis*, *assoc*, *assoc-if*, *assoc-if-not*, *find*, *rassoc*, *rassoc-if*, *rassoc-if-not*

Listes (suite)

Accès: *cons*, *list car*, *cdr*, *nth*, *nthcdr*, *last*, *elt*, *cadadr*, *cdaaar*, . . . *first*, *second*, . . ., *tenth*

Autres: *length*, *subseq*, *copy-seq*, *fill*, *replace*, *count*, *reverse*, *nreverse*, *concatenate*, *position*, *find*, *sort*, *merge*, *map*, *some*, *every*, *notany*, *notevery*, *reduce*, *search*, *remove*, *remove-dupicates*, *delete*, *delete-duplicates*, *substitute*, *nsubstitute*, *mismatch*

Tableaux

```
* (make-array '(4 3))

#2A((0 0 0) (0 0 0) (0 0 0) (0 0 0))
* (make-array '(4 3) :initial-element 5)

#2A((5 5 5) (5 5 5) (5 5 5) (5 5 5))
* (make-array 4 :initial-contents (list (+ 3 4) "hello" 'hi t))

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

Tableaux

```
* (setf *x* (make-array 6 :fill-pointer 4))

#(0 0 0 0)
* (length *x*)

4
* (setf (fill-pointer *x*) 6)

6
* (length *x*)

6
* (adjust-array *x* '(30) :fill-pointer 20)

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

Structures

```
* (defstruct person name address phone-number)

PERSON
* (make-person)

#S(PERSON :NAME NIL :ADDRESS NIL :PHONE-NUMBER NIL)
* (make-person :name "Bobby Beach")

#S(PERSON :NAME "Bobby Beach" :ADDRESS NIL :PHONE-NUMBER NIL)
* (defstruct person (name "unknown") address (phone-number 0))

PERSON
* (make-person :name "Bobby Beach")

#S(PERSON :NAME "Bobby Beach" :ADDRESS NIL :PHONE-NUMBER 0)
*
```

Tables d'hachage

```
* (setf *x* (make-hash-table))

#<EQL hash table, 0 entries 480AA9B5>
* (gethash 'weight *x*)

NIL
NIL
* (setf (gethash 'weight *x*) 234)

234
* (gethash 'weight *x*)

234
T
*
```

Tables d'hachage

```
* (setf (gethash 11 *x*) nil)

NIL
* (gethash 11 *x*)

NIL
T
*
```

Fonctions de première classe

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

```
* #'1+

#<Function 1+ 103744B9>
* (mapcar #'1+ '(4 3 4 8))

(5 4 5 9)
* (reduce #'max '(4 3 5 8))

8
* (funcall (complement #'<) 3 5)

NIL
*
```

Fonctions anonymes

```
* (mapcar (lambda (x) (+ x 2)) '(3 4 5 6))

(5 6 7 8)
* (find-if (lambda (x) (> x 5)) '(5 8 3 9 4 2) :from-end t)

9
* (defparameter *l* (list "how" "hi" "are" "you"))

*L*
* (sort *l* (lambda (x y) (< (length x) (length y))))

("hi" "how" "are" "you")
* *l*

("how" "are" "you")
*
```

Structures de contrôle

- Blocs
- Contextes
- Conditionnels
- Itération
- Valeurs multiples
- Sauts non locaux

Blocs

Trois types de bloc: *progn*, *block*, *tagbody*

progn permet d'évaluer des expression en séquence. La valeur de la dernière expression est la valeur de l'expression *progn*

```
* (progn
    (setf *l* '(5 2 4 3))
    (setf *ll* (sort *l* #'<))
    (car *ll*))

2
*
```

Blocs

block est comme un *progn* avec un nom.

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

"hello"
23
*
```

tagbody est la construction de base pour les blocs, mais n'est jamais utilisé directement par un utilisateur

Contexte

Un contexte est un bloc plus des variables locales

```
* (let ((len (length *features*))
        (fun (complement #'<)))
    (funcall fun len 10))

T
* (let* ((len (length *features*))
         (min (min len 5)))
    (make-array (list min)))

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

Contexte

la dernière expression est équivalente à :

```
* (let ((len (length *features*)))
    (let ((min (min len 5)))
      (make-array (list min))))

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

Conditionnels

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

(when (f ...)
  (g ...)
  (setf x 23)
  (+ x y))

(unless (f ...)
  (g ...)
  (setf x 23)
  (+ x y))
```

Conditionnels

```
(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))

(typecase ...)
```

Itération

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

(do* ...)

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

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

(mapc ...)
```

Itération

```
(loop for i from 1 to (compute-top-value)
      while (not (unacceptable i))
      collect (square i)
      do (format t "Working on ~D now~%" i)
      when (evenp i)
        do (format t "~D is a non-odd number~%" i)
      finally (format t "About to exit!"))
```

Valeurs multiples

```
* (multiple-value-bind (x y) (floor 75 4)
    (list x y))

(18 3)
* (values (cons 1 2) 'hello (+ 3 4))

(1 . 2)
HELLO
7
* (multiple-value-call #'+ (floor 75 4))

21
* (multiple-value-list (floor 75 4))

(18 3)
*
```

Loop

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (let* ((wins (car lst))
             (max (funcal fn wins)))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (when (> score max)
              (setf wins obj)
                    max score))))
        (values wins max))))
```

Loop

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (loop with wins = (car lst)
            with max = (funcall fn wins)
            for obj in (cdr lst)
            for score = (funcall fn obj)
            when (> score max)
              do (setf wins obj
                       max score)
            finally (return (values wins max)))))
```