Techniques et Fondements de la Programmation (partie Techniques)

Responsable et cours:	**Robert Strandh**
TD:	**Pierre Castéran** **Philippe Narbel** **Oliver Ly**
Planning:	**13 semaines** **(12 semaines de cours, 12 semaines de TD)** **2h de cours + 2h de TD par semaine** **4h de travail individuel par semaine**
Support de cours:	**Paul Graham, ANSI Common Lisp, ou** **David Touretzky, Common Lisp: A Gentle introcution to Symbolic Computation** **Transparents**

Web: http://dept–info.labri.u–bordeaux.fr/~strandh/Teaching/TFP/2000–2001/Dir.html

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

Autres documents

Guy Steele:

Common Lisp, the Language, second edition

Digital Press

(disponible sur WWW en HTML)

CMUCL user's guide

CLX reference manual

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

David Lamkins:

Successful Lisp (Tutorial en–ligne)

Objectifs et contenu

Objectif: **Maîtriser un certain nombre de méthodes et techniques de programmation symbolique, impérative et fonctionnelle.**

Contenu:

Langage Common Lisp

Types: symbole, liste, ...

Fonctions de base

Techniques élémentaires de programmation

Macros

Exceptions

Programmation par objets avec CLOS

But de l'enseignement de programmation

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

Langages, mécanismes et outils adaptés à ces méthodes et techniques

Aspects de la programmation non enseignés

Efficacité extrème

Temps réel

Applications particulières (jeux, image & son, numérique, ...)

Détails de langages particuliers

Impémentations particulières

Comment faire pour apprendre à programmer?

Il faut surtout lire beacoup de code écrit par des experts

Il faut lire la litterature sur la programmation. Il n'y en a pas beacoup (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 uniformizer 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. Intregration du concept dans la norme

Tour guidé de Common Lisp

Langage dynamique

Sémantique par référence uniforme

Gestion automatique de la mémoire

Types de données

- **Entiers à précision arbitraire**
- **rationnels**
- **flottants**
- **complexes rationnels** — **la plupart des opérations arithmetiques**
- **complexes flottants** — **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
- iteration (while, do, dolist, loop)
- blocs (prog1, progn, tagbody, let, let*)
- fermétures
- fonctions génériques
- valeurs multiples
- exceptions

Mots clés (passage de paramètres)

Compilation

Macros

Programmation orientée objets

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 (fonctionnelle, 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 ascendente (bottom–up)

Paramètres sont souvent des fonctions (fermétures)

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

Implementation 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 nécéssaires ("pas de pointeurs")

Les declarations peuvent aider le compilateur à rempacer les références par les objets correspondants

Common Lisp est interactif

Common Lisp est presque toujouts 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.
En EMI, 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 symbol, ou

une expression composée.

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 responsble de cette analyse s'appelle read. Cette fonction est disponible à l'utilisateur.

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.

Les objets resultant de l'évaluation sont ensuite affichés (ou imprimés) en représentation externe. La fonction responsable de l'affichage s'appele print. Cette fonction est disponible à l'utilisateur.

Lancer et quitter le système Lisp

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

1234
* 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
* "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
* *features*

(:MK-DEFSYSTEM :CL-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.

```
* (+ 3 4)

7
* (length "hello")

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

80/3
* (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
*
```

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).

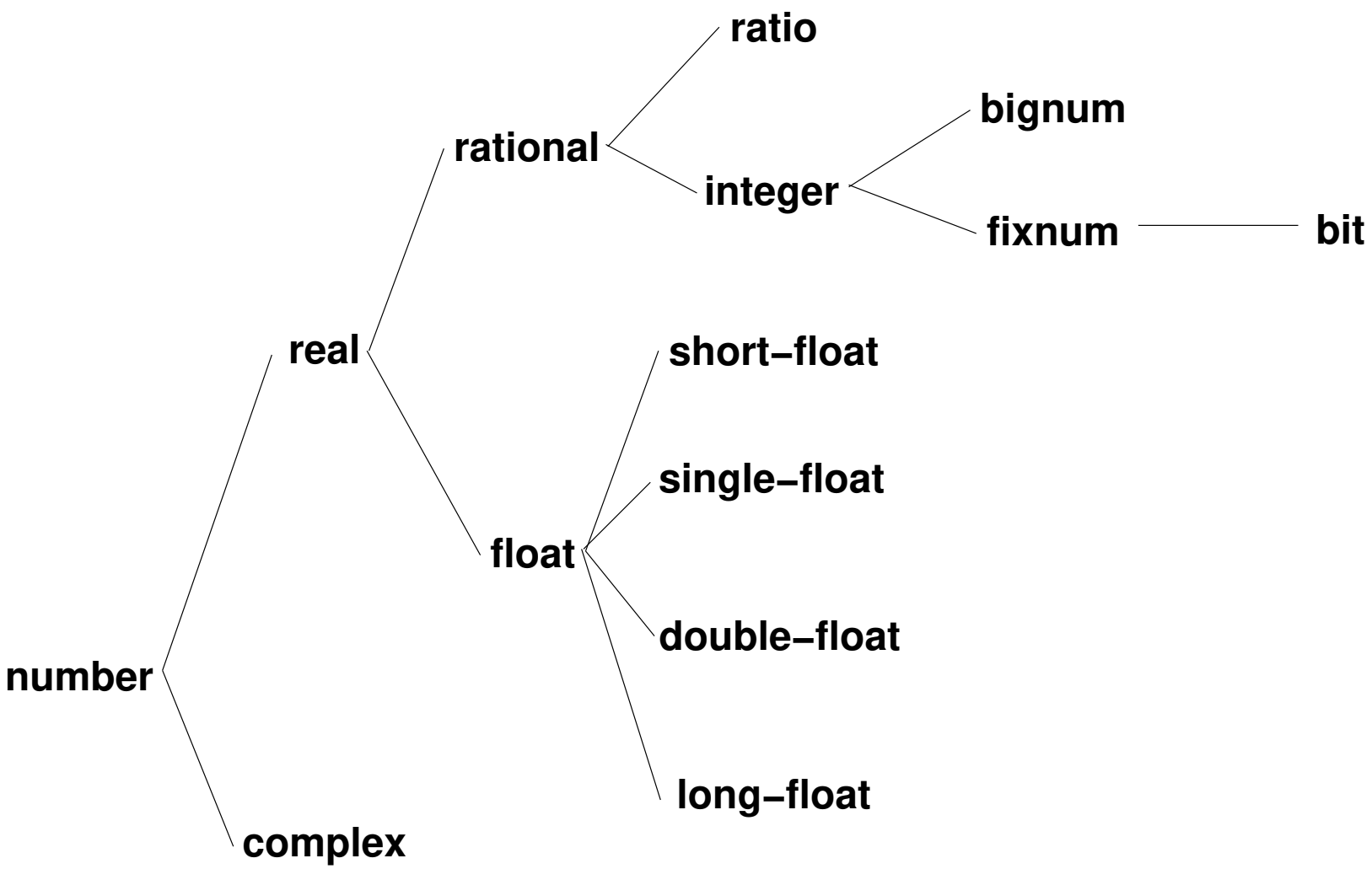
Définition de fonction

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
* (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


number
real
rational
ratio
integer
bignum
fixnum
bit
float
short–float
single–float
double–float
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)

815915283247897773434561126959611589427200000000

*
```

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 20000000000000000000 2)

80000000000000000000
*
```

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
```

Exemples d'utilité : transformée de Fourier

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

0.4999999999999994d0

* (sin #c(3 4))

#C(3.8537378 -27.016813)

*
```

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
* (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

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

*A*
* *a*

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

HELLO
* *a*

HELLO
* hello

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

Au lieu de taper (quote machin) on peut taper 'machin

Symboles

Tableau d'hachage dans le paquetage (package) courant

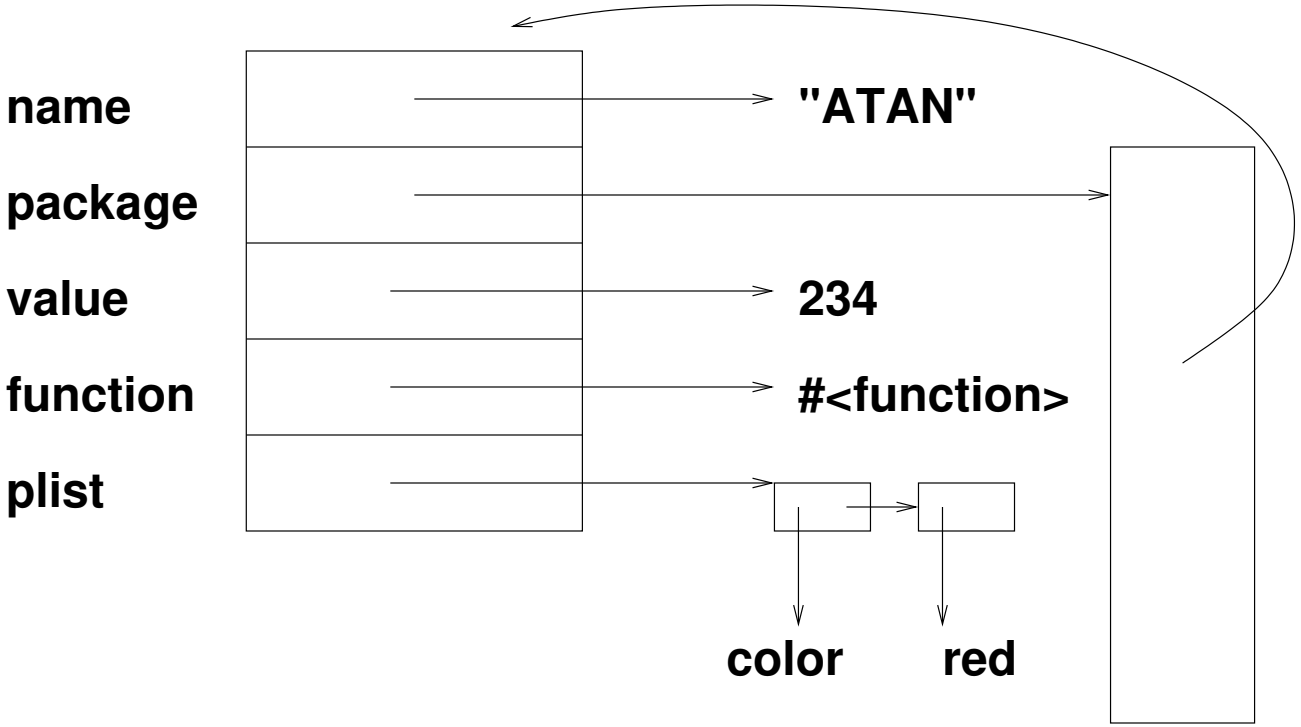
C'est la fonction read qui cherche le tableau d'hachage et crée le symbole

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

```
* (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 et chaînes de caractères

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", ...

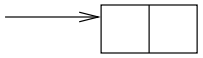
Une chaîne de caractères est un vecteur est un tableaux

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

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

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

NIL
*
```

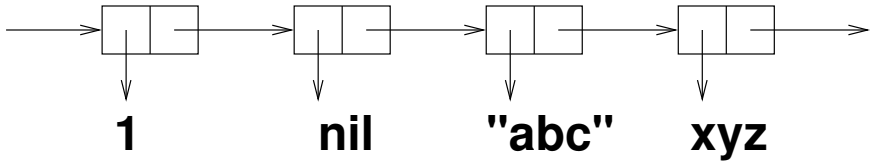
Listes

Une list est une séquence

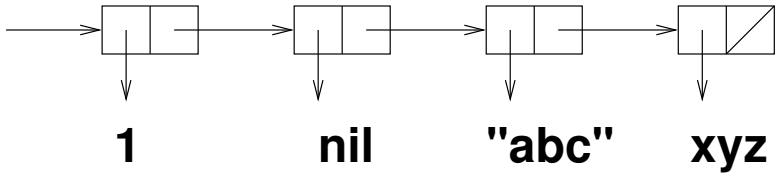
Le type de base est la paire ou le cons



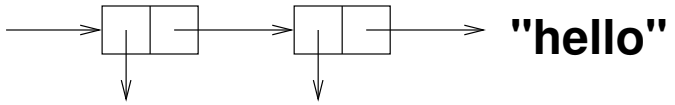
Une liste (anglais: proper list) est une suite de paires terminée par nil




```
(1 nil "abc" xyz)
```



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




```
(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.

```
* (+ 3 4)

5
* '(+ 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)
* (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 Stram to SYSTEM:*STDOUT*>)
* (append *my-list* '(a b c))

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

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

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

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

Listes (suite)

Structure de contrôle de base: la récursivite

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)
* (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
*
```

Listes (suite)

Atomes et listes

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

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

Égalité

```
* (setf x (list 'a))
(A)
* (eq x x)
T
* (eql x x)
T
* (equal x x)
T
* (eq x (list 'a))
NIL
```

```
* (eql x (list 'a))
NIL
* (equal x (list 'a))
T
```

Listes (suite)

Construction: `cons, list`

Accès: `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`

`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`

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)
* (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* :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 (480558ED)>
* (gethash 'weight *x*)

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

234
* (gethash 'weight *x*)

234
T
* (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 5 8))

(5 4 6 9)
* (reduce #'max '(4 3 5 8))

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

T
*
```

```
(defun mappend (fn &rest lsts)
  (apply #'append (apply #'mapcar fn lsts)))
```

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

("how" "hi" "are" "you")
* (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 a (+ x y))
  (myfun a)
  (* a 5))
```

Block est comme un progn avec un nom

```
(block found
  (do-stuff ...)
  (when (detected) (return-from found 23))
  (do-some-more ...))
```

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

Contexte

Un bloc + 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)
*
```

la dernière expression est équivalente à :

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

```
(defun bin-search (obj vec)
  (let ((len (length vec)))
    (and (not (zerop len))
         (finder obj vec 0 (- len 1)))))

(defun finder (obj vec start end)
  (let ((range (- end start)))
    (if (zerop range)
        (if (eql obj (aref vec start))
            obj
            nil)
        (let ((mid (+ start (round (/ range 2)))))
          (let ((obj2 (aref vec mid)))
            (if (< obj obj2)
                (finder obj vec start (- mid 1))
                (if (> obj obj2)
                    (finder obj vec (+ mid 1) end)
                    obj)))))))
```

Conditionnels

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

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

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

(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 ...)
```

```
(defun insert-before (before after lst)
  (cond ((null lst) nil)
        ((eql (car lst) after)
         (cons before lst))
        (t (cons (car lst) (insert-before before after (cdr lst))))))

(defun insert-after (before after lst)
  (cond ((null lst) nil)
        ((eql (car lst) before)
         (cons before (cons after (cdr lst))))
        (t (cons (car lst) (insert-after before after (cdr lst))))))
```

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 ...)

(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!"))
```

```
(defun map-int (fn n)
  (let ((acc nil))
    (dotimes (i n)
      (push (funcall fn i) acc))
    (nreverse acc)))

(defun filter (fn lst)
  (let ((acc nil))
    (dolist (x lst)
      (let ((val (funcall fn x)))
        (if val (push val acc))))
    (nreverse acc)))
```

Valeurs multiples

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

=> (18 3)

(values (f ...) 'hello (+ 3 4))

(multiple-value-call #'+ (floor 75 4))

(multiple-value-list (floor 75 4))
```

```
(defun make-queue () (cons nil nil))

(defun enqueue (obj q)
  (if (null (car q))
      (setf (cdr q) (setf (car q) (list obj)))
      (setf (cdr (cdr q)) (list obj)
            (cdr q) (cdr (cdr q))))
  (car q))

(defun dequeue (q)
  (pop (car q)))
```

```
;;; -*- Mode: Lisp; Syntax: Common-Lisp; -*-
;;; Code from Paradigms of Artificial Intelligence Programming
;;; Copyright (c) 1991 Peter Norvig

(defstruct op "An operation"
  (action nil) (preconds nil) (add-list nil) (del-list nil))

(defparameter *school-ops*
  (list
    (make-op :action 'drive-son-to-school
         :preconds '(son-at-home car-works)
         :add-list '(son-at-school)
         :del-list '(son-at-home))
    (make-op :action 'shop-installs-battery
         :preconds '(car-needs-battery shop-knows-problem shop-has-money)
         :add-list '(car-works))
    (make-op :action 'tell-shop-problem
         :preconds '(in-communication-with-shop)
         :add-list '(shop-knows-problem))
    (make-op :action 'telephone-shop
         :preconds '(know-phone-number)
         :add-list '(in-communication-with-shop))
    (make-op :action 'look-up-number
         :preconds '(have-phone-book)
         :add-list '(know-phone-number))
    (make-op :action 'give-shop-money
         :preconds '(have-money)
         :add-list '(shop-has-money)
         :del-list '(have-money))))
```

```
(defvar *state* nil "The current state: a list of conditions.")

(defvar *ops* nil "A list of available operators.")

(defun GPS (*state* goals *ops*)
  "General Problem Solver: achieve all goals using *ops*."
  (if (every #'achieve goals) 'solved))

(defun achieve (goal)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (or (member goal *state*)
      (some #'apply-op
            (find-all goal *ops* :test #'appropriate-p))))

(defun appropriate-p (goal op)
  "An op is appropriate to a goal if it is in its add list."
  (member goal (op-add-list op)))

(defun apply-op (op)
  "Print a message and update *state* if op is applicable."
  (when (every #'achieve (op-preconds op))
    (print (list 'executing (op-action op)))
    (setf *state* (set-difference *state* (op-del-list op)))
    (setf *state* (union *state* (op-add-list op)))
    t))
```

```
* (gps '(son-at-home car-needs-battery have-money have-phone-book)
       '(son-at-school)
       *school-ops*)
(EXECUTING LOOK-UP-NUMBER)
(EXECUTING TELEPHONE-SHOP)
(EXECUTING TELL-SHOP-PROBLEM)
(EXECUTING GIVE-SHOP-MONEY)
(EXECUTING SHOP-INSTALLS-BATTERY)
(EXECUTING DRIVE-SON-TO-SCHOOL)
SOLVED

* (gps '(son-at-home car-needs-battery)
       '(son-at-school)
       *school-ops*)
NIL

* (gps '(son-at-home car-works)
       '(son-at-school)
       *school-ops*)
(EXECUTING DRIVE-SON-TO-SCHOOL)
SOLVED
```

Macros

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

Evaluation en deux phases: macro–expansion puis évaluation

Important: la macro–expansion est faite par le compilateur

```
(nil! x)
```

devient

```
(setf x nil)
```

qui sera compilé à la place de l'expression initiale

Les macros permettent donc de programmer le compilateur.

Backquote

```
* (setf a 1 b 2)

2
* `(a is ,a and b is ,b)

(A IS 1 AND B IS 2)
* (defmacro nil! (var)
    `(setf ,var nil))

NIL!
* (nil! *x*)

NIL
* (setf lst '(1 2 3))

(1 2 3)
* `(the list is ,lst I think)

(THE LIST IS (1 2 3) I THINK)
* `(the elements are ,@lst I think)

(THE ELEMENTS ARE 1 2 3 I THINK)
*
```

Exemples

```
(defmacro while (test &rest body)
  `(do ()
       ((not ,test))
     ,@body))
```

```
(defmacro for (init test update &rest body)
  `(progn
     ,init
     (while ,test
       ,@body
       ,update)))
```

Difficultés

Capture de variables

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

incorrect

```
* (let ((x 10))
    (ntimes 5
      (setf x (1+ x)))
    x)

10
*
```

Difficultés

Multiple évaluation

```
(defmacro ntimes (n &rest body)
  (let ((g (gensym)))
    `(do ((,g 0 (1+ ,g)))
         ((>= ,g ,n))
       ,@body)))
```

incorrect

```
(let ((v 10))
  (ntimes (setf v (1- v))
    (format t "*")))

*****
```

Difficultés

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

correct

```
(defmacro while (test &rest body)
  `(do ()
       ((not ,test))
     ,@body))


(defun quicksort (vec l r)
  (let ((i l)
        (j r)
        (p (svref vec (round (+ l r) 2))))    ; 1
    (while (<= i j)                           ; 2
      (while (< (svref vec i) p) (incf i))
      (while (> (svref vec j) p) (decf j))
      (when (<= i j)
        (rotatef (svref vec i) (svref vec j))
        (incf i)
        (decf j)))
    (if (> (- j l) 1) (quicksort vec l j))    ; 3
    (if (> (- r i) 1) (quicksort vec i r)))
  vec)
```

Exemple: condlet

```
> (condlet (((= 1 2) (x (princ 'a)) (y (princ 'b)))
            ((= 1 1) (y (princ 'c)) (x (princ 'd)))
            (t (x (princ 'e)) (z (princ 'f))))
    (list x y z)

CD
(D C NIL)
```

Exemple: condlet (suite)

```
(defmacro condlet (clauses &body body)
  (let ((bodfn (gensym))
        (vars (mapcar #'(lambda (v) (cons v (gensym)))
                      (remove-duplicates
                        (mapcar #'car
                                (mappend #'cdr clauses))))))
    `(labels ((,bodfn ,(mapcar #'car vars)
                ,@body))
       (cond ,@(mapcar #'(lambda (cl)
                           (condlet-clause vars cl bodfn))
                       clauses)))))

(defun condlet-clause (vars cl bodfn)
  `(,(car cl) (let ,(mapcar #'cdr vars)
                (let ,(condlet-binds vars cl)
                  (,bodfun ,@(mapcar #'cdr vars))))))

(defun condlet-binds (vars cl)
  (mapcar #'(lambda (bindform)
              (if (consp bindform)
                  (cons (cdr (assoc (car bindform) vars))
                        (cdr bindform))))
          (cdr cl)))
```

Exemple: with–db

```
(let ((temp *db*))
  (setq *db* db)
  (lock *db*)
  (prog1 (eval-query q)
         (release *db*)
         (setq *db* temp)))
```

```
(with-db db
  (eval-query q))
```

Exemple: with–db (suite)

```
(defmacro with-db (db &body body)
  (let ((temp (gensym)))
    `(let ((,temp *db*))
       (unwind-protect
         (progn
           (setq *db* ,db)
           (lock *db*)
           ,@body)
         (progn
           (release *db*)
           (setq *db* ,temp))))))
```

Exemple: with–db (suite)

```
(defmacro with-db (db &body body)
  (let ((gbod (gensym)))
    `(let ((,gbod #'(lambda () ,@body)))
       (declare (dynamic-extent ,gbod))
       (with-db-fn *db* ,db ,gbod))))

(defun with-db-fn (old-db new-db body)
  (unwind-protect
    (progn
      (setq *db* new-db)
      (lock *db*)
      (funcall body))
    (progn
      (release *db*)
      (setq *db* old-db))))
```

Exemple: macros anaphoriques

```
(let ((it (complicated-calculation)))
  (if it
      (do-something-with it)
      (do-something-else-with it)))


(aif (complicated-calculation)
     (do-something-with it)
     (do-something-else-with it))
```

Exemple: macros anaphoriques (suite)

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))

(defmacro awhen (test-form &body body)
  `(aif ,test-form
        (progn ,@body)))

(defmacro awhile (expr &body body)
  `(do ((it ,expr ,expr))
       ((not it))
     ,@body))

(defmacro aand (&rest args)
  (cond ((null args) t)
        ((null (cdr args)) (car args))
        (t `(aif ,(car args) (aand ,@(cdr args))))))

(defmacro acond (&rest clauses)
  (if (null clauses)
      nil
      (let ((cl1 (car clauses))
            (sym (gensym)))
        `(let ((,sym ,(car cl1)))
           (if ,sym
               (let ((it ,sym)) ,@(cdr cl1))
               (acond ,@(cdr clauses)))))))
```

Introduction à la programmation par objets avec CLOS (motivation)

La programmation ascendente est plus facile si les modules sont organisés par types de données

Pour améliorer la maintenabilité du code, il faut éviter la duplication de code. Il faut donc factoriser le code

Le langages de programmation traditionnels ne peuvent pas représenter cette organisation du code

Exemple: verrou (lock)

Utilisé pour éviter un acces multiple à une ressource (imprimante, CD–ROM, etc)

Plusieurs types sont possibles

Utilisation uniforme (polymorphisme)

Chaque verrou a un nom

Pour simplifier: deux types de verrou: simple et null

Opérations: verrouiller, déverrouiller, récupérer le nom, récupérer le propriétaire (celui qui l'a verrouillé)

Exemple: verrou (suite)
(version orientée opérations)

```
(defstruct null-lock name)

(defstruct simple-lock name owner)

(defun lock-name (lock)
  (typecase lock
    (null-lock (null-lock-name lock))
    (simple-lock (simple-lock-name lock))))

(defun lock-seize (lock)
  (typecase lock
    (null-lock nil)
    (simple-lock (setf (null-lock-owner lock) ...))))

(defun lock-release (lock)
  (typecase lock
    (null-lock nil)
    (simple-lock (setf (null-lock-owner lock) nil))))
```

Version souhaitée (mais pas possible sans OO)

```
(defstruct null-lock name)

(defun lock-seize (lock)
  nil)

(defun lock-release (lock)
  nil)
```

```
(defstruct simple-lock
  name owner)

(defun lock-seize (lock)
  (setf (simple-lock-owner lock) ...))

(defun lock-release (lock)
  (setf (simple-lock-owner lock) nil))
```

```
(defun lock-name (lock)
  (???-name lock))
```

Introduction à la programmation orientée objets

La programmation orientée objets à été conçue pour résoudre certains problèmes de modularitè et de réutilisabilité dus à l'utilisation de langages traditionnels (comme le C) pour l'écriture d'applications.

Par conséquent, l'utilité de la programmation orientée objets est difficile à comprendre pour des programmeurs non expérimentés.

Pour éviter l'influence de la terminologie d'un langage particulier, nous allons commencer par une discussion générale des notions de la conception et de la programmation orientées objets.

Puis, nos montrerons la correspondance entre ces notions et l'utilisation de CLOS pour la programmation orientée objets.

Définition: objet

Un objet est une valeur qui peut être affectée à une variable, fournie en tant qu'argument à une fonction, ou renvoyée par l'appel à une fonction.

Un objet a une identité. Deux objets peuvent être comparés concernant leurs identités.

Si la sémantique par références uniforme est utilisée, alors cette identité est préservée par l'affectation.

Souvent, un object est implémenté par un pointeur sur de l'espace en mémoire, mais des objets simples peuvent ne pas avoir de la mémoire allouée. L'identité et la sémantique peuvent quand–même être préservées, à condition que des effets de bord ne soient pas possibles sur ces objets.

Définition: type (abstrait)

Un type abstrait (ou simplement type) est un ensemble d'objets.

Les types ne sont pas forcement mutuellement exclusifs.

Un objet peut donc être de plusieurs types.

En particulier, un type peut être un sous type d'un autre type.

Par exemple, un objet de type voiture est aussi de type véhicule.

Opération

Un opération est une fonction ou une procédure dont les arguments sont des objets.

Chaque argument est d'un type particulier.

Par exemple, monter–moteur peut être une opération de deux arguments, un objet de type véhicule à moteur et un objet de type moteur. L'effet de bord de cette opération est de changer le moteur du véhicule.

Définition: protocole

Un protocole est une collection d'opérations utilisées ensemble.

Typiquement, au moins un type est partagé entre toutes les opérations du protocole.

Exemple: Dessiner des objets graphiques. Un tel protocole peut par exemple contenir les opérations dessiner, effacer et déplacer, utilisant des objets de type window, pixmap, cercle, rectangle, etc

Classe

Une classe est utilisée pour décrire les détails d'implémentation d'objets, tels que le nombre et les noms des champs (ou créneaux).

Un type est typiquement constitué par un ensemble de classe qui peuvent être relatées par l'héritage (mais qui ne le sont pas forcement)

Un objet est l'instance directe d'exactement une classe. C'est la classe de l'objet.

Remarque: on peut donc parler de la classe d'un objet mais pas du type d'une objet

Définition: Héritage

Les classes sont organisées dans un graphe sans cycles (héritage multiple) ou dans un arbre (héritage simple)

Un objet est instance de la classe C si et seulement si il est soit l'instance directe de C ou instance d'une classe qui hérite de C (ou équivalent: qui est dérivée de C).

Si un type contient la classe C, il contient aussi toutes les classe dérivées de C.

Programmation orientée objets avec CLOS
Définition: Méthode

Une méthode est une implémentation partielle d'une opération.

L'implémentation est partielle, car la méthode l'implémente uniquement pour des arguments instances de certaines classes.

Pour que l'opération soit complète, il faut que les méthodes couvrent l'ensemble des classes des types des arguments.

Les méthodes d'une opération peuvent être physiquement réparties.

Toutes les méthodes d'une seule opération ont le même nom.

Fonctions génériques

Une fonction générique est une implémentation complète d'une opération.

Elle est utilisée comme une fonction ordinaire.

Une implémentation partielle (une méthode) est choisie selon la classe des arguments.

Sélection

Le mécanisme de sélection est responsable du choix d'une méthode particulière d'une fonction générique selon la classe des arguments

En CLOS, la sélection est multiple, à savoir, plusieurs arguments sont utilisés pour déterminer la méthode choisie

Héritage multiple

C’est la possibilité en CLOS pour une classe d’être dérivée de plusieurs autres classes.

Comparaison avec Java (et C++, Smalltalk, Eiffel, etc)
Typage statique

Ces langages sont statiquement typés, ce qui veut dire que l'utilisateur est obligé de déclarer la classe des variables et des arguments à une fonction ou procédure.

Par conséquent, un type abstrait doit aussi être une classe.

On introduit donc souvent une classe racine (parfois sans instances) dont le nom est le nom du type abstrait

Restrictions sur les opérations

Tous ces langages ont la sélection simple, à savoir qu'une la méthode d'une opération peut être déterminée uniquement par un seul argument (le premier).

C'est pourquoi ces langages traitent le premier argument différemment des autres.

Le premier argument est tellement différent que la syntaxe est spéciale:

CLOS: **(mount−engine my−car the−engine)**

Autres: **my_car.mount_engine(the_engine) ou the_engine.mount_engine(my_car)**

selon si la sélection est souhaité par le type de voiture ou le type de moteur

Cette syntaxe spéciale est responsable de l'impossibilité de passer une opération en tant qu'argument à une autre opération.

Relations entre classe et méthodes

Avec ces langages, la relation entre une classe et un unsemble de méthodes est tellement forte que les méthodes sont physiquement à l'intérieur de la définition de la classe.

Héritage simple ou multiple

Certains de ces langages ont l'héritage simple uniquement, par exemple Smalltalk.

Certains autres ont l'héritage multiple, comme C++.

Le langage Java à l'héritage simple plus la notion d'interface presque équivalent à l'héritage multiple.

Terminologie CLOS

Un ensemble d'objets est représenté par un objet de type classe

Un objet dans l'ensemble est une instance directe de la classe

Chaque instance directe d'une classe a le même comportement, la même structure et le même type

La structure d'une instance est définie par les créneaux (slots) (similaires aux champs d'une structure)

Chaque créneau a un nom et une valeur

Un créneau est soit local (une valeur différente dans chaque instance) ou partagé (une valeur commune pour l'ensemble des instances)

Une classe peut être définie comme une extension d'une autre classe ou de plusieurs autres classes, appelées superclasses. La nouvelle classe est une sous–classe de ses superclasses

Fonctions Génériques

Une fonction générique est une définition d'interface et non d'implémentation

L'implementation d'une fonction générique est faite par zero, une ou plusieurs méthodes

Le choix de la méthode à utiliser est déterminé par le types des arguments (mécanisme de sélection (dispatch))

En CLOS, la sélection est déterminé par l'ensemble des argument (multiple dispatch, multi methods), alors que d'autres langages n'utilisent que le premier argument pour la sélection

Exemple: verrou encore

```
(defclass lock ()
  ((name :initarg :name :reader lock-name))
  (:documentation "The foundation of all locks."))
```

Nom de la classe: `lock`

Spécification de créneau: `(name :initarg :name :reader lock-name)`

Nom du créneau: `name`

Option de créneau: `:initarg :name`

argument clé pour la création: `(make-instance ... :name "the name")`

Option de créneau: `:reader lock-name`

nom d'une fonction générique à utiliser pour la lecture du créneau

`(lock-name lock)`

Option de classe: `(:documentation "...")`

```
(defclass null-lock (lock)
  ()
  (:documentation "A lock that is always free"))
```

Superclasse: `lock`

Exemple (suite)

```
(defclass simple-lock (lock)
  ((owner :initform nil :accessor lock-owner))
  (:documentation "A lock that is either free or busy."))
```

Créneau supplementaire: `owner`

(par rapport aux créneaux de la superclasse lock)

Option de créneau: `:initform nil`

valeur initiale par défaut du créneau

Option de créneau: `:accessor lock-owner`

nom d'une fonction générique à utiliser pour la lecture et de l'écriture du créneau:

```
(lock-owner lock)

(setf (lock-owner lock) *current-process*)
```

Création d'instances

```
(setf *null-lock*
  (make-instance 'null-lock :name "Null lock"))

(setf *simple-lock*
  (make-instance 'simple-lock :name "Simple lock"))
```

Ne pas utiliser make–instance dans un module client.
Écrire une fonction spécifique pour la création

```
(defun make-null-lock (name)
  (make-instance 'null-lock :name name))

(defun make-simple-lock (name)
  (make-instance 'simple-lock :name name))
```

Fonctions génériques

```
(defgeneric seize (lock)
  (:documentation
"Seize the lock.
Return the lock when the operation succeeds.
Some locks simply wait until they can succeed, while
other locks return NIL if they fail."))


(defgeneric release (lock &optional failure-mode)
  (:documentation
"Release the lock if it is currently owned by this process.
Return T if the operation succeeds.
If unsuccessful and failure-mode is :no-error, return NIL.
If unsuccessful and failure-mode is :error, signal an error
The default for failure-mode is :no-error."))
```

Les méthodes

```
(defmethod seize ((l null-lock))
  l)  ; return lock, no waiting

(defmethod release ((l null-lock) &optional failure-mode)
  (declare (ignore failure-mode))  ; never fails for null locks
  t)
```

La liste des argument (lambda list) contient des paramètres spécialisés par des classes

Une méthode est applicable uniquement si la restriction sur les classes est vérifiée.

Les méthodes (suite)

On suppose l'existance de primitives:

- **without–process–preemption (pour l'exécution atomique)**
- **process–wait (attendre la valeur vrai d'une fonction)**
- ***current–process* (le processus courant)**

```
;; If value of place is old-value, set it to new-value
;; Return T if the setf worked, nil otherwise
(defmacro setf-if (place old-value new-value)
  `(without-process-preemption
     (cond ((eql ,place ,old-value)
            (setf ,place ,new-value)
            t)
           (t nil))))
```

```
(defmethod check-for-mylock ((l simple-lock) process)
  (when (eql (lock-owner l) process)
    (error "Can't seize ~A because you already own it." l)))
```

Sauts non locaux

Un façon d'abandonner l'exécution actuelle et de continuer à un autre point du programme

Trois types: go, return–from, throw

Le contenu de la pile entre le sommet et la cible du transfert et abandonné (y compris les liens dynamiques de variables spéciales, tags "catch", et traitants de conditions)

Les formes protégées par "unwind–protect" sont exécutées

Contrôle est transferé à la cible

catch/throw

catch est utilisé pour établir une étiquette à utiliser par throw

l'étiquette peut être n'importe quel objet (mais c'est souvent un symbole constant)

puis une suite de formes est évaluée, comme avec progn

**si pendant l'évaluation des formes, il y a un throw à l'étiquette,
alors l'évaluation est abandonnée et catch renvoie les valeurs envoyées par throw**

```
* (defun f (x)
    (throw 'hello 345))

F
* (defun g (a)
    (catch 'hello
      (print 'hi)
      (f a)
      (print 'bonjour)))

G
```

```
* (g 234)

A
345
*
```

return–from

Similaire à break du langage C

```
* (block hello
    (list 'a 'b 'c)
    (return-from hello 234)
    (error "an error"))

234
*
```

Situations exceptionnelles

Il y en a trois types:

Les échecs

Ce sont des situations dues aux défauts de programmation.

L'épuisement de ressources

C'est quand il n'y a plus de mémoire ou quand le disque est plein.

Les autres

Ce sont des situations normales, prévues par le programmeur, mais dont le traitement nécessite une structure de contôle particulière

Les échecs

Il n'y rien à faire

Il faut arrêter l'exécution le plus vite possible

Avec la macro "assert", on peut parfois tester les échecs et arrêter l'exécution avec un message standard

Il ne sert à rien de faire mieux, car on ne sait pas dans quel état se trouve le programme

La macro "assert" est aussi une bonne méthode pour documenter l'interface d'un type abstrait

À l'aide du compilateur, on peut supprimer les tests quand on est "sûr" qu'il n'y a plus de défauts

Il est souvent meiux de les laisser (sauf ceux qui coûtent cher)

L'épuisement de ressources

**Dans un programme non interactif, facile à relancer,
arrêter l'exécution avec un message destiné à l'utilisateur
(et non comme assert destiné au programmeur)**

**Dans un programme interactif, c'est un peu trop brutal.
Relancer la boucle d'intercation après un message indiquant
comment faire pour libérer des ressources**

**La programmation dans le cas interactif est très délicate. Il ne faut pas
que le message ou les commandes pour la libération de ressources
nécessitent de la ressource épuisée (l'affichage peut nécessiter
de la mémoire, par exemple)**

Les autres

Ouverture d’un fichier qui n’existe pas, mais c’est normal

Parcour d’un tableau à la recherche d’un élément particulier, et on l’a trouvé

Tentative de reculer d’un caractère, alors que le point est au début du tampon

Multiplication d’une suite de nombres, et l’un des nombres est zéro

Etc...

Le problème des situations exceptionnelles

La situation est souvent détectée par du code de bas niveau

Mais c'est du code haut niveau qui possède la connaissance pour son traitement

Exemple (ouverture de fichier non existant):

> **Le problème est détecté par open (très bas niveau)**
>
> **Si c'est un fichier d'installation c'est un défaut. Si c'est un fichier utilisateur c'est une situation de type "autre".**

Il faut donc communiquer le problème du bas niveau au haut niveau

Conditions

Une condition est une instance (direct ou non) de la classe "condition"

C’est donc un objet avec des créneaux

Une condition est signalée à un traitant éventuel

Le traitant est une fonction (ordinaire ou générique) d’un seul argument, la condition

Le traitant est associé à un type de conditions, et le traitant est appelé seulement si la condition est du bon type

Pour définir des sous classes de la classe "condition" utiliser define−condition et non defclass

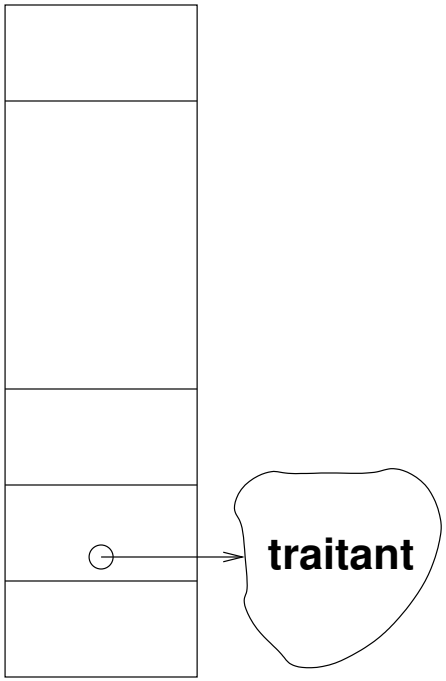
Pour créer une instance de la classe "condition", utiliser make−condition et non make−instance

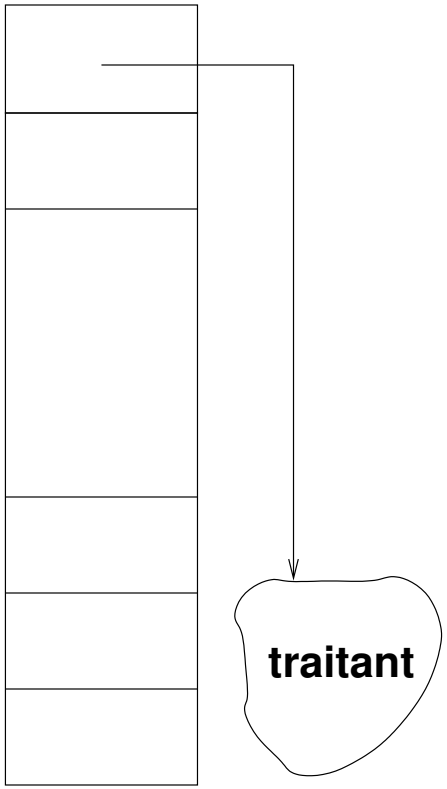
Plusieurs types de conditions existent: arithmetic−error, end−of−file, . . .

Conditions (suite)

La signalisation d'une condition se fait en deux étapes:

1. **La recherche d'un traitant actif associé au type de la condition à signaler**
2. **Le traitant trouvé est appelé**





Conditions (suite)

Les traitant est désactivé avant d'être appelé

La pile est intacte pendant l'exécution du traitant

Ceci permet d'examiner la pile par le debugger et de continuer l'exécution du programme dans certains cas

Le traitant peut:

1. **refuser (decline). Pour ça, il retourne normalement**

2. **traiter. Pour ça, il exécute un saut non local**

3. **reporter la décision. Il peut par exemple appeler le debugger, ou signaler une autre condition.**

Conditions (suite)

Les traitants sont établis avec handler–bind ou handler–case

Pour signaler, on utilise signal, warn, error, cerror, assert, etc

```
* (define-condition zero-found ()
    ())

ZERO-FOUND
* (defun multiply-leaves (l)
    (cond ((consp l) (reduce #'* (mapcar #'multiply-leaves l)))
          ((zerop l) (signal 'zero-found))
          (t l)))

MULTIPLY-LEAVES
* (multiply-leaves '((1 2 (3 4)) (((5 6))) 7))

5040
* (handler-case (multiply-leaves '((1 2 (3 0)) (((5 6))) "hello"))
    (zero-found () 0))

0
*
```

Spécifications de types

Utilité:

Vérification de type

Optimisation de la compilation

Représentation de données
(tableaux de bits, flots d'octets, ...)

Les types simples sont indiqués par des symbols.

Les types simples forment une arborescence (ou plutôt un DAG)

Le type simple le plus général est t (un objet et toujours de type t)

Le type simple le mois général est nil (aucun objet n'est de type nil)

Types non simples

On peut construire une spécification de type pour n'importe quel ensemble d'objets

Exemple:

```
(or vector (and list (not (satisfies circular?))))

(integer 1 100)
```

Possibilité d'indiquer le type contenu dans un tableau:

```
(simple-array fixnum)

(make-array (list height width) :element-type 'bit)
```

Certains éléments du type peuvent être omis:

```
(simple-array fixnum (* *))

(simple-array fixnum *)
```

Définition de nouveaux noms de types

Utilisation de la macro deftype (similaire à defmacro)

```
(deftype proseq ()
  '(or vector (and list (not (satisfies circular?)))))

> (typep #(1 2) 'proseq)
T
```

Paramétragre avec des arguments:

```
(deftype multiple-of (n)
  `(and integer (satisfies (lambda (x)
                             (zerop (mod x ,n))))))

> (typep 12 '(multiple-of 4))
T
```

Flots binaires

```
(with-open-file
  (input filename :direction input
                  :element-type 'unsigned-byte)
  ...
  (read-byte input)
  ...
  ...)
```

Read macros

Possibilité de programmer la fonction de lecture de code

Association entre un caractère est une fonction arbitraire

Les caractères ’ (quote) et ‘ (anti quote) sont implémentés en utilisant ce mécanisme

```
(set-macro-character #\'
  #'(lambda (stream char)
      (list (quote quote) (read stream t nil t))))
```

Read macros (suite)

Possibilité de définier des caractères similaires à #
(mais pas souvent nécessaire)

Possibilité de rajouter un deuxième caractère à # (ou similaire)
avec set–dispatch–macro–character

```
(set-dispatch-macro-character #\# #\?
  #'(lambda (stream char1 char2)
      (list 'quote
            (let ((lst nil))
              (dotimes (i (+ (read stream t nil t) 1))
                (push i lst))
              (nreverse lst)))))

> #?7
(0 1 2 3 4 5 6 7)
```

caractères réservés au programmeur: #!, #?, #[, #], #{, #}

Read macros (suite)

Création de listes:

```
(set-macro-character #\} (get-macro-character #\)))

(set-dispatch-macro-character #\# #\{
  #'(lambda (stream char1 char2)
      (let ((accum nil)
            (pair (read-delimited-list #\} stream t)))
        (do ((i (car pair) (+ i 1)))
            ((> i (cadr pair))
             (list 'quote (nreverse accum)))
          (push i accum)))))

> #{2 7}
(2 3 4 5 6 7)
```

Packages

Pour traduire un nom à un symbole

Package courant dans *package*

Le package courant est utilisé par read

Initalement, le package courant est common–lisp–user

Fonctions package–name et find–package:

```
> (package-name *package*)
"COMMON-LISP-USER"
> (find-package "COMMON-LISP-USER")
#<The COMMON-LISP-USER package, ...>
```

Fonction symbol–package pour obtenir le package d'un symbole

```
> (symbol-package 'sym)
#<The COMMON-LISP-USER package, ...>
```

Packages (suite)

Création et changement de package:

```
> (setf sym 99)
99
> (setf *package* (make-package 'mine
                                :use '(common-lisp)))
#<The MINE package, ...>
> sym
Error ... The variable SYM is unbound
> common-lisp-user::sym
99
```

Éviter l'utilisation de ::

Packages (suite)

Utiliser plutôt le concept d'exportation

```
> (in-package common-lisp-user)
#<The COMMON-LISP-USER package, ...>
> (export 'bar)
T
> (setf bar 5)
5
> (in-package mine)
#<The MINE package, ...>
> common-lisp-user:bar
5
> (import 'common-lisp-user:bar)
T
> bar
5
```

Importation de l'ensemble des symboles exportés:

```
> (use-package 'common-lisp-user)
T
```

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))))

(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)))))
```