Projet de Programmation 3

Cours : Robert Strandh
TD : Damien Cassou

Support de cours

Robert Strandh et Irène Durand :

Traité de programmation en Common Lisp

Transparents

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

Bibliographie

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

Peter Norvig : Paradigms of Artificial Intelligence Programming
Case Studies in Common Lisp
Morgan Kaufmann

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

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)

Pourquoi le langage 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

Features

interactivity (dynamic redefinitions), first-class symbols, arbitrary-precision integers, exact rational arithmetic, well-integrated complex numbers, generalized references, multiple values, first-class functions, anonymous functions, macros, multiple inheritance, multiple dispatch, generic functions, method combination, (first-class) classes and meta classes, (first-class) packages, built-in programmable parser (read), built-in programmable unparser, reader macros, compiler macros, optional argument, keyword arguments, meta-object protocol, special (dynamically scoped) variables, named blocks, nonlocal goto (catch/throw), conditions, restarts, the `loop` macro, the `format` function, type declarations, compiler available at run-time, extensive list processing features.

Paquetages (Packages)

Supposons qu'on veuille utiliser un module (une bibliothèque) (la bibliothèque graphique `McCLIM` par exemple).

On ne veut pas devoir connaître toutes les fonctions, variables, classes, macros utilisées pour implémenter ce module.

Par contre, on veut utiliser certaines fonctions, variables, classes, macros proposées dans l'interface (API pour Application Program Interface) du module.

On ne veut pas de conflit entre les noms qu'on définit et ceux utilisés dans l'implémentation du module.

Pour que ce soit possible, il faut que le module se serve de la notion de paquetage.

Paquetages (suite)

Un paquetage détermine la correspondance entre une suite de caractères (le nom d'un symbole) et un symbole.

Le paquetage par défaut dans lequel est placé l'utilisateur est le paquetage `COMMON-LISP-USER` (petit nom `CL-USER`).

Le paquetage courant est mémorisé dans la variable globale `*package*`.

```
CL-USER> *package*
#<PACKAGE "COMMON-LISP-USER">
```

Fonctions `package-name` et `find-package` :

```
CL-USER> (package-name *package*)
"COMMON-LISP-USER"
CL-USER> (find-package "COMMON-LISP-USER")
#<PACKAGE "COMMON-LISP-USER">
```

Paquetages (suite)

La variable `*package*` est utilisée par la fonction `read` pour traduire un nom en un symbole.

Fonction `symbol-package` pour obtenir le paquetage d'un symbole :

```
CL-USER> (symbol-package 'car)
#<PACKAGE "COMMON-LISP">
CL-USER> (defun double (n) (* 2 n))
DOUBLE
CL-USER> (symbol-package 'double)
#<PACKAGE "COMMON-LISP-USER">
```

Création et changement de paquetage

Création

```
CL-USER> (defpackage  :sudoku (:use :common-lisp))
#<PACKAGE "SUDOKU">
CL-USER>
```

Changement

```
CL-USER> (in-package :sudoku)  ou raccourci SLIME C-c M-p
#<PACKAGE "SUDOKU">
SUDOKU> (double 3)
The function DOUBLE is undefined.
   [Condition of type UNDEFINED-FUNCTION]
SUDOKU> (common-lisp-user::double 3)
6
```

Éviter l'utilisation de `::` qui casse les barrières d'abstraction

Paquetages (suite)

Utiliser plutôt le concept d'exportation

```
CL-USER> (in-package :cl-user)     ou  :common-lisp-user
#<PACKAGE "COMMON-LISP-USER">
CL-USER> (export 'double)
T
CL-USER> (in-package :sudoku)
#<PACKAGE "SUDOKU">
SUDOKU> (common-lisp-user:double 2)
4
SUDOKU> (import 'double :common-lisp-user)
T
SUDOKU> (double 2)
4
```

Paquetages (suite)

Quand on crée un paquetage correspondant à un module, il est normal d'exporter tous les symboles qui sont dans l'API

```
(defpackage :sudoku
  (:use :common-lisp)
  (:export
    #:make-grid
    #:show
    #:automatic-filling
    #:look-at-deterministic
    #:look-at-squares
    ...
    #:*automatic*
    #:which-at
    #:where-in
    ...))
```

Importation de symboles

Si dans un paquetage, on souhaite utiliser des symboles exportés par un autre paquetage, il faut les importer (`import`, `use-package`).

Importation de symboles :

```
CL-USER> (import 'make-grid :sudoku)   ou (import 'sudoku:make-grid)
T
CL-USER> (defparameter *g* (make-grid))
#<SUDOKU::GRID B70E281>
```

Importation de l'ensemble des symboles exportés :

```
CL-USER> (use-package :sudoku)
T
CL-USER> (show *g*)
```

Paquetages (suite)

Définition d'un paquetage important des symboles d'autres paquetages

Supposons qu'on souhaite faire un interface graphique pour le Sudoku :

```
(defpackage :visual-sudoku
  (:use :common-lisp :clim)
  (:import-from :sudoku
    #:make-grid
    #:show
    ...))
```

Paquetages : un ennui classique

```
SUDOKU> (defun foo () 'foo)
FOO
SUDOKU> (export 'foo)
T
```

Dans `cl-user`, on oublie d'importer `foo` :

```
CL-USER> (foo)
The function FOO is undefined.
   [Condition of type UNDEFINED-FUNCTION]
CL-USER> (import 'sudoku:foo)

 IMPORT SUDOKU:FOO causes name-conflicts in #<PACKAGE "COMMON-LISP-USER">

between the following symbols:

  SUDOKU:FOO, FOO

   [Condition of type SB-INT:NAME-CONFLICT]
```

```
See also:
  Common Lisp Hyperspec, 11.1.1.2.5 [section]

Restarts:
  0: [RESOLVE-CONFLICT] Resolve conflict.
  1: [ABORT-REQUEST] Abort handling SLIME request.
  2: [TERMINATE-THREAD] Terminate this thread (#<THREAD "repl-thread" B237591>)

Select a symbol to be made accessible in paquetage COMMON-LISP-USER:
  1. SUDOKU:FOO
  2. FOO

Enter an integer (between 1 and 2): 1
T
CL-USER> (foo)
SUDOKU:FOO
```

Vers la Programmation par Objets

Imaginons un jeu de Pacman.

Il y a des participants : pacman, des pastilles et des fantômes.

Comment programmer la collision entre deux objets ?

Comment programmer l'affichage ?

Programmation impérative traditionnelle

```
(defun collision (p1 p2)
  (typecase p1
    (tablet (typecase p2
              (phantom ...)
              (pacman ...)))
    (phantom (typecase p2
               (tablet ...)
               (phantom ...)
               (pacman ...)))
    (pacman (typecase p2
              (tablet ...)
              (phantom ...)))))
```

```
(defun display (p)
  (typecase p
    (tablet ...)
    (phantom ...)
    (pacman ...)))
```

Problèmes avec l'approche traditionnelle

– code pour un type d'objets dispersé
– difficile de rajouter un type d'objet
– il faut avoir accès au code source

Supposons qu'on veuille rajouter un participant supplémentaire : une super pastille dont la propriété est de rendre Pacman invincible.

Il faut modifier `collision`, `display` et ....

Il faut rajouter des cas pour la collision entre les fantômes et la super pastille, même s'il se passe la même chose qu'entre les fantômes et une pastille ordinaire.

Vers les langages à objets

De manière générale pour améliorer la maintenabilité du code, il faut éviter la duplication de code et donc le factoriser.

La programmation est facilitée si les modules sont organisés par types de données (programmation ascendante).

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

Apports de la POO

Première amélioration : l'héritage

```
(defclass participant ()
  (...))

(defclass tablet (participant)
  (...))

(defclass super-tablet (tablet)
  (...))
```

Deuxième amélioration : les opérations génériques

```
(defgeneric display (pane participant)
  (:documentation "display a participant on the board"))

(defgeneric collision (participant1 participant2)
  (:documentation "handle collision between two participants"))
```

Introduction à la programmation orientée objets

Programmation orientée objets a été conçue pour résoudre certains problèmes de modularité et de réutilisabilité dans les langages traditionnels (comme C)

L'utilité de la programmation orientée objets est difficile à comprendre pour des programmeurs non expérimentés.

Discussion générale sur les concepts de la POO
Mise en oeuvre en CL

Notion d'objet

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

Un objet a une identité.
On peut comparer l'identité de deux objets.

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

CL : sémantique par référence uniforme.

Notion de type

Un type est un ensemble d'objets.

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

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

Un objet peut donc être de plusieurs types.

Les types ne sont pas forcément mutuellement exclusifs.

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` est une opération à 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.

Protocole

Un protocole est une collection d'opérations utilisées ensemble.

Le plus souvent, 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 constitué par un ensemble de classes qui peuvent être liées par héritage (mais qui ne le sont pas forcément).

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

Un objet peut avoir plusieurs types.

Héritage

Une classe peut être définie comme une extension d'une ou plusieurs classes appelées super-classes. La nouvelle classe est une sous-classe de ses super-classes.

Les classes sont organisées dans un graphe sans cycle (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.

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.

Sélection

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

En CL, la sélection est multiple, à savoir, plusieurs arguments sont utilisés pour déterminer la méthode choisie.
On écrit : `(collision pacman phantom)`

Java, C++, Smalltalk, Eiffel, ... ont la sélection simple : la méthode est déterminée uniquement par un seul argument (le premier).

La syntaxe met en avant le premier argument :
`the-pacman.collision(a-phantom)` ou
`a-phantom.collision(the-pacman)`

selon si la sélection est souhaitée sur le type du premier ou du deuxième participant.

Relations entre classe et méthodes

Sélection simple : la relation entre une classe et un ensemble de méthodes est tellement forte que les méthodes sont physiquement à l'intérieur de la définition de la classe.

```
class  pacman: public mobile-participant {
  int force = 0;
  int invincible = 0;
  ...
  public collision (phantom p) { ...  }
}
class  phantom: public mobile-participant {
  public collision (pacman p) { ... }
}

  pacman the_pacman;
  phantom a_phantom;

  the_pacman.collision(phantom);
  a_phantom.collision(the_pacman);
```

Relations entre classe et méthodes

Sélection multiple : les méthodes sont en dehors des classes.

```
(defclass  pacman (mobile-participant)
  ((force :initform 0 ...)
   (invincible :initform 0 ...))
   (:documentation "the hero of the game"))

(defclass phantom (mobile-participant) ())

(defgeneric collision (participant1 participant2)
  (:documentation "collision between two game participants"))

(defmethod collision ((pacman pacman) (phantom phantom))
  ...)

...
  (collision the-pacman a-phantom)
```

Héritage simple ou multiple

Héritage multiple : quand une classe dérive de plusieurs autres classes.

Certains langages ont l'héritage simple uniquement (Small-talk).

Certains autres ont l'héritage multiple (Common Lisp, C++)

Java a l'héritage simple plus la notion d'interface presque équivalente à celle d'héritage multiple.

Objets et Classes

Un ensemble d'objets similaires est représenté par une classe.

Un objet de cet ensemble est une instance directe de la classe.

Chaque instance directe d'une classe a la même structure précisée par la définition de la classe.

Définition de classes

Macro `defclass` : quatre parties

- nom de la classe
- liste des classes héritées
- liste des créneaux (slots)
- documentation

```
(defclass participant ()
  ((x :initarg :x :initform 0 :accessor participant-x)
   (y :initarg :y :accessor participant-y))
  (:documentation "game participant"))
```

Créneau : nom du créneau : `x`
ou
liste contenant au minimum le nom du créneau puis des précisions supplémentaires données à l'aide de paramètres mot-clés : `(x :initform 0 ...)`

Création d'une instance d'une classe

```
CL-USER> (setf *p* (make-instance 'participant :x 3))
#<PARTICIPANT 9F4AC49>
CL-USER> (describe *p*)
#<PARTICIPANT 9F59C71> is an instance of class
#<STANDARD-CLASS PARTICIPANT>.
The following slots have :INSTANCE allocation:
 X    3
 Y    "unbound"
```

À la création de l'object est appelée automatiquement la méthode `initialize-instance` qui effectue les initialisations par défaut demandées par `initform` puis par `initarg`.

Accès aux créneaux

On peut toujours accéder à un créneau avec `slot-value` :

```
CL-USER> (slot-value *p* 'x)
3
CL-USER> (setf (slot-value *p* 'y) 2)
2
```

Mais on n'utilise `slot-value` que dans l'implémentation de la classe.
Les utilisateurs de la classe ne sont pas censés connaître le nom des créneaux et doivent utiliser les accesseurs qui leur sont fournis sous forme de fonctions génériques.

Fonctions Génériques

Chaque opération est décrite par une fonction générique qui décrit les paramètres et la sémantique de l'opération.

```
(defgeneric collision (participant1 participant2)
  (:documentation "handle collision between two participants"))
```

Les méthodes sont des implémentations partielles d'une opération

```
(defmethod collision ((pacman pacman) (tablet tablet))
  (incf (pacman-force pacman))
  (kill tablet))
```

Une opération est implémentée par un ensemble de méthodes.
Même utilisation qu'une fonction ordinaire :
`(collision pacman tablet)`
La méthode à appliquer est choisie selon la classe des arguments. Une fonction générique est une définition d'interface et non d'implémentation.

Fonction d'impression

Il existe une fonction générique

```
(defgeneric print-object (object stream)
  (:documentation "write object to stream"))
```

On peux écrire une méthode `print-object` spécialisée pour les objets de la classe `participant`.

```
(defmethod print-object ((p participant) stream)
  (format stream "#Participant(~A,~A)" (slot-value p 'x)
                                       (slot-value p 'y)))
```

```
CL-USER> *p*
#Participant(3,2)
```

Les objets sont dynamiques

Les objets créés sont dynamiques : si on rajoute un créneau à la classe ils seront mis à jour à leur prochaine utilisation :

```
(defclass participant ()
  ((x :initarg :x :initform 0 :accessor participant-x)
   (y :initarg :y :initform 0 :accessor participant-y)
   (name :initarg :name :initform "unknown" :reader participant-name))
  (:documentation "game participant"))

(defmethod print-object ((p participant) stream)
  (format stream "#Participant-~A(~A,~A)" (slot-value p 'name)
                                          (slot-value p 'x)
                                          (slot-value p 'y)))
```

```
CL-USER> *p*
#Participant-unknown(3,2)
```

Définition de créneaux

Mot-clés possibles : `:initarg` `:initform` `:type` `:documentation`.
`:reader` `:writer` `:accessor` `:allocation`

Un créneau est
- un créneau d'instance s´il est propre à chaque objet instance de la classe
- un créneau de classe s'il est commun à toutes les instances

`:allocation :class` -> créneau de classe
par défaut : `:allocation :instance` -> créneau d'instance

```
(defclass tablet (participant)
  ((color :allocation :class :initform +red+ :accessor tablet-color)))
```

Classe abstraite

Classe abstraite : classe dont on ne souhaite pas dériver d'instance mais qui servira à dériver d'autres classes.

```
(defclass mobile-participant (participant)
  ()
  (:documentation "base class for all mobile participants"))
```

Les classes abstraites n'ont pas de créneaux d'instance mais peuvent avoir des créneaux de classe.

```
(defclass triangle (polygon)
  ((number-of-sides :initform 3
                    :allocation :class
                    :reader number-of-sides)))
```

Définition automatique d'accesseurs

`:reader`

```
(defclass triangle (polygon)
  ((nb-sides :initform 3
                    :allocation :class
                    :reader number-of-sides)))
```

Définit automatiquement la méthode

```
(defmethod number-of-sides ((p triangle))
  (slot-value p 'nb-sides))
```

qui est une implémentation de la fonction générique

```
(defgeneric number-of-sides (p)
  (:documentation "number of sides of a polygon P"))
```

On pourra écrire `(number-of-sides p)` pour un polygone `p` de type `triangle`.

Définition automatique d'accesseurs

`:accessor`

```
(defclass pacman (mobile-participant)
  ((force :initform 0 :accessor pacman-force)
   (invincible :initform nil :accessor pacman-invincible))
  (:documentation "the hero of the game"))
```

Définit automatiquement les méthodes (et de même pour `pacman-invincible`) :

```
(defmethod pacman-force ((pacman pacman))
  (slot-value pacman 'force))
(defmethod (setf pacman-force) (value (pacman pacman))
  (setf (slot-value pacman 'force) value))
```

On pourra écrire `(pacman-force *pacman*)` ou

`(setf (pacman-force *pacman*) (1+ (pacman-force *pacman*))`

`*pacman*)`

ou `(incf (pacman-force *pacman*))` pour un object `*pacman*`

de type `pacman`.

Quelques classes du Pacman

```
(defclass participant ()
  ((x :initform 0 :initarg :x :accessor participant-x)
   (y :initform 0 :initarg :y :accessor participant-y)))

(defclass tablet (participant)
  ((color :allocation :class :initform +red+ :accessor tablet-color)))

(defclass mobile-participant (participant) ())

(defclass phantom (mobile-participant) ())

(defclass pacman (mobile-participant directed-mixin)
  ((force :initform 0 :accessor pacman-force)
   (invincible :initform nil :accessor pacman-invincible)))

(defclass labyrinth ()
  ((size :reader size :initarg :size)
   (participants :initform nil :initarg :participants :accessor participants)
   (obstacles :initform nil :initarg :obstacles :accessor obstacles)
   (pacman :initform nil :accessor pacman)))
```

Quelques fonctions génériques du Pacman

```
(defgeneric display (pane participant)
  (:documentation "display a participant on the board"))

(defgeneric kill (participant)
  (:documentation "kill a participant"))

(defgeneric collision (participant1 participant2)
  (:documentation "handle collision between two participants"))

(defgeneric tic (participant)
  (:documentation "handle a participant at each step of the game"))
```

Coeur du Pacman

```
(defun display-labyrinth (labyrinth pane)
  (display-board labyrinth pane)
  (mapc #'(lambda (obj) (display pane obj))
        (participants labyrinth)))

(loop
  ...
  (display-labyrinth ...)
  (mapc #'tic (participants *labyrinth*))
  ...
  ))
```

Quelques méthodes du Pacman

```
(defmethod kill ((participant participant))
  (setf (participants *labyrinth*)
  (delete participant (participants *labyrinth*))))

(defmethod collision ((p1 participant) (p2 participant))
  (declare (ignore p1 p2))
  nil)

(defmethod tic ((participant participant))
  (declare (ignore participant))
  nil)

(defmethod tic ((phantom phantom))
  (random-move phantom))

(defmethod collision ((pacman pacman) (phantom phantom))
  (kill (if (pacman-invincible pacman) phantom pacman)))

(defmethod collision ((pacman pacman) (tablet tablet))
  (incf (pacman-force pacman))
  (kill tablet))
```

Sélection de la méthode

Quand CL trouve plusieurs méthodes possible, il applique la plus spécifique.

Ordre lexicographique.

Exemple :

```
(defmethod m ((p1 c1) (p2 c2)) ...)
(defmethod m ((p1 c11) (p2 c2)) ...)
(defmethod m ((p1 c1) (p2 c21)) ...)
```

Appel `(m pc11 pc21)`

Le deuxième méthode est choisie.

Méthodes pour des objects particuliers

Sélection sur un l'objet particulier.

```
(defmethod collision ((p1 participant) (p2 (eql nil)))
  (declare (ignore p2))
  p1)

(defmethod collision ((p1 (eql nil)) (p2 participant))
  (declare (ignore p1))
  p2)

(defmethod division ((dividende number) (diviseur number))
  (/ dividende diviseur))

(defmethod division ((dividende number) (diviseur (eql 0)))
  (error "Impossible de diviser par zero"))
```

Un object est plus spécifique que sa classe.

Méthodes secondaires

Utilisation de méthodes :after, :before

```
(defmethod kill :after ((pacman pacman))
  (setf (pacman *labyrinth*) nil))

(defmethod initialize-instance :after ((participant phantom)
                                       &rest initargs &key &allow-other-keys)
  (declare (ignore initargs))
  (setf (participant-x participant) (random *labyrinth-size*))
  (setf (participant-y participant) (random *labyrinth-size*)))

(defmethod collision :before ((pacman pacman) (tablet super-tablet))
  (setf (pacman-invincible pacman) t))
```

Méthodes secondaires

Utilisation de méthodes :around

```
(defmethod test ()
  (print "primary" t) 0)

(defmethod test :after ()
  (print "after" t))

(defmethod test :before ()
  (print "before" t))

CL-USER> (test)

"before"
"primary"
"after"
0

(defmethod test :around ()
  (print "around" t) 1)

CL-USER> (test)

"around"
1

(defmethod test :around ()
  (print "debut around" t)
  (call-next-method)
  (print "fin around" t) 2)

CL-USER> (test)

"debut around"
"before"
"primary"
"after"
"fin around"
2
```

Héritage multiple

```
(defclass value-gadget (standard-gadget)
  ((value :initarg :value
          :reader gadget-value)
   (value-changed-callback :initarg :value-changed-callback
                           :initform nil
                           :reader gadget-value-changed-callback)))

(defclass action-gadget (standard-gadget)
  ((activate-callback :initarg :activate-callback
                      :initform nil
                      :reader gadget-activate-callback)))

(defclass text-field (value-gadget action-gadget)
  ((editable-p :accessor editable-p :initarg editable-p :initform t))
  (:documentation "The value is a string"))
```

L'utilisation de l'héritage multiple avec plusieurs classes concrètes (ou ayant des classes descendantes concrètes) est relativement rare.

Classes mixin

Une classe mixin est une classe abstraite destinée à être utilisée uniquement dans le cadre d'un héritage multiple.

Un classe mixin permet de rajouter du comportement à des objets d'autres classes.

Solution sans classe mixin

```
(defclass directed-participant (mobile-participant)
  ((direction :initform nil :initarg :direction :accessor :direction)
   (directions :initform nil :initarg :directions :accessor :directions
  (:documentation "a mobile participant that can be directed"))

(defclass pacman (directed-participant)
  ((force :initform 0 ...)
   (invincible :initform 0 ...))
   (:documentation "the hero of the game"))
```

Classes mixin (suite)

Avec classe mixin : la classe directed-mixin apporte la possibilité qu'un objet soit dirigé :

```
(defclass directed-mixin ()
  ((direction :initform nil :accessor direction)
   (directions :initform () :accessor directions)))

(defclass pacman (mobile-participant directed-mixin)
  ((force :initform 0 :accessor pacman-force)
   (invincible :initform nil :accessor pacman-invincible)))
```

Avantage : le comportement peut-être réutilisé avec une autre classe que mobile-participant

Macro with-slots

Quand on veut accéder à plusieurs créneaux d'un même objet, au lieu d'écrire

```
CL-USER> (format t " abscisse = ~A , ordonnée = ~A~%"
                 (slot-value *p* 'x)
                 (slot-value *p* 'y))
abscisse = 3 , ordonnée = 2
NIL
```

on peut utiliser la macro with-slots :

- avec le noms des créneaux auxquels on veut accéder :

```
CL-USER> (with-slots (x y) *p*
           (format t " abscisse = ~A , ordonnée = ~A~%" x y))
```

- en leur donnant un "petit nom" :

```
CL-USER> (with-slots ((c color) (n name)) *t*
           (format t " couleur = ~A , nom = ~A~%" c n))
couleur = #<NAMED-COLOR "red">, nom = unknown
NIL
```

Sauts non locaux

Un saut non local est une façon d'abandonner l'exécution normale d'un programme pour la reprendre à un autre point du programme.

Trois types de saut : `return-from`, `throw`, `go`

Avant un saut non local, les formes protégées par `unwind-protect` sont exécutées

Le contrôle est transféré à la cible du saut

return-from

Saut non local similaire au `break` du langage C

```
CL-USER> (block hello
           (list 'a 'b 'c)
           (when (< *print-base* 20)
             (return-from hello 234))
           (error "an error"))
234
```

Une fonction est un bloc implicite :

```
CL-USER> (defun f (l)
           (dolist (e l)
             (unless e
               (return-from f 'ok))))
F
CL-USER> (f '(1 2 nil 3))
OK
CL-USER> (f '(1 2 3))
NIL
```

Ce mécanisme est lexical : l'étiquette du bloc n'est accessible qu'aux expressions du bloc

Mécanisme catch/throw

cf setjmp, longjmp de C

```
(catch etiquette
  e1
  ...
  en)
```

`catch` est utilisé pour établir une étiquette (qui pourra être utilisée par `throw`). La suite `e1 e2 ... en` est un `progn` implicite.

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

Si pendant l'évaluation des expressions, un `throw` avec l'étiquette est effectué, alors l'évaluation est abandonnée et `catch` renvoie les valeurs envoyées par `throw`.

Exemple : catch/throw

```
CL-USER> (defun f (x)
           (throw 'hello 345))
F
CL-USER> (defun g (a)
           (catch 'hello
             (print 'hi)
             (f a)
             (print 'bonjour)))
G
CL-USER> (g 234)
HI
345
```

Situations exceptionnelles

Il y en a trois types :

1. les échecs : ce sont des situations dues aux erreurs de programmation.
2. l'épuisement de ressources : c'est quand il n'y a plus de mémoire, quand le disque est plein...
3. les autres : ce sont des situations normales, prévues par le programmeur, mais dont le traitement nécessite une structure de contrôle particulière.

Les échecs

Il n'y rien à faire à part arrêter l'exécution au plus vite.

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

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

```
(defmethod real-move ((p mobile-participant) direction)
  (assert (participant-can-move-p p direction))
  ...)

(setf *m* (make-instance 'mobile-participant :x 0 :y 0))
(real-move *m* 'u)

The assertion (PARTICIPANT-CAN-MOVE-P P DIRECTION) failed.
   [Condition of type SIMPLE-ERROR]
```

La macro `assert` est aussi une bonne méthode pour typer les opérations.

```
(defun fact (n)
  (assert (and (integerp n) (not (minusp n))
  ...)
```

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 mieux 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 au programmeur comme avec `assert`).

Dans un programme interactif, c'est un peu trop brutal.

Relancer la boucle d'interaction après un message indiquant comment faire pour libérer des ressources.

Dans le cas interactif, la programmation peut s'avérer très délicate si le message ou les commandes de 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

Parcours 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 de 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 faire remonter le problème du bas niveau vers un plus haut niveau

Le système de conditions

Les conditions CL correspondent aux exceptions d'autres langages (Java, Python, C++).

Mais les conditions CL peuvent servir à autre chose que le traitement des erreurs.

Une condition est une instance (directe ou non) de la classe `condition`.

C'est donc un objet avec des créneaux.

Une condition est signalée à un traitant (handler) éventuel.

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

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

Conditions

Pour définir des sous-classes de la classe "condition",
utiliser `define-condition` et non `defclass`.

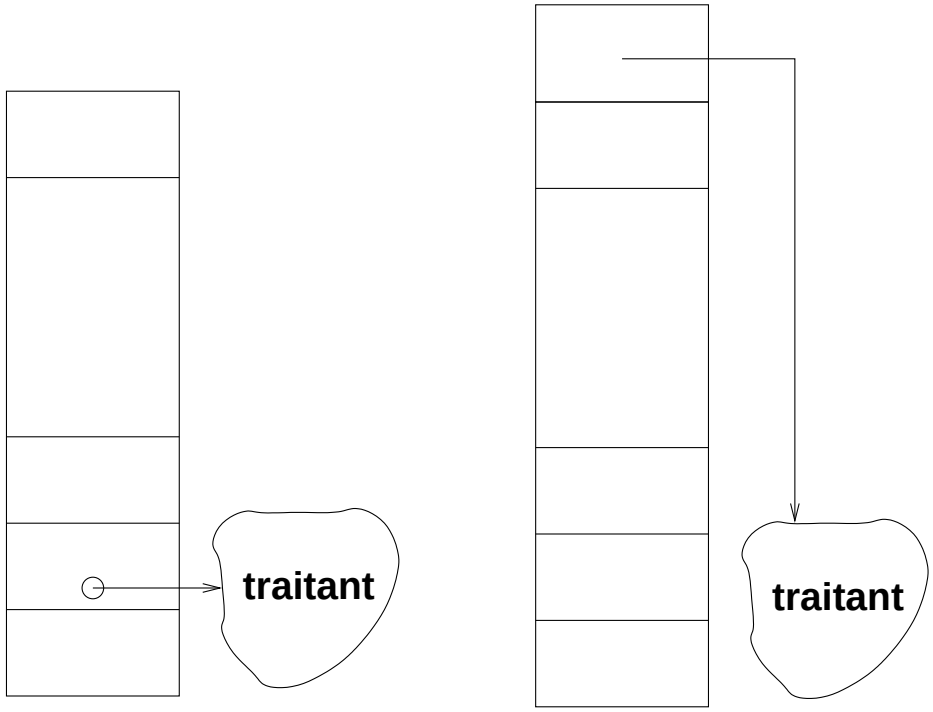
Pour créer une instance de la classe `condition`,
utiliser `make-condition` et non `make-instance`.

Plusieurs classes de conditions sont prédéfinies : `arithmetic-error`,
`end-of-file`, `division-by-zero`,...

Conditions (suite)

La signalisation d'une condition déclenche

1. la recherche d'un traitant actif associé au type de la condition signalée,
2. l'appel du traitant trouvé.



Conditions (suite)

Le 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.

Positionnement d'un traitant de condition

Les traitants sont établis avec `handler-bind` ou `handler-case`

```
CL-USER> (defparameter *n* 10)
*N*
CL-USER> (handler-case
             (/ *n* 2)
           (division-by-zero ()
             (prog1 nil (print "error"))))

5
CL-USER> (handler-case
             (/ *n* 0)
           (division-by-zero ()
             (prog1 nil (print "error"))))

NIL
```

Signalement d'une condition

Pour signaler, on utilise `signal`, `warn`, `error`, `assert`, etc en fonction du type de la condition.

```
(defun fact (n)
  (when (minusp n)
    (error "~S negatif" n))
  (if (zerop n) 1 (* n (fact (1- n)))))
```

Si aucun traitant n'est positionné le debugger est invoqué.

Positionnement d'un traitant

Grâce à `handler-case` ou `handler-bind` on positionne un (ou plusieurs) traitant d'interruption chargé de récupérer les conditions du type correspondant susceptibles d'être émises.

```
(defun intermediaire (n)
  (fact (- n)))
```

```
CL-USER> (handler-case
           (intermediaire 9)
           (error ()
             (prog1 nil
               (format *error-output* "erreur factorielle récupérée"))))
erreur factorielle récupérée

NIL
```

Définition d'une condition

```
CL-USER> (define-condition zero-found ()
           ())
ZERO-FOUND
```

`(define-condition c () ...)` ≡ `(define-condition c (condition) ...)`

```
CL-USER> (defun multiply-leaves (l)
           (cond
             ((consp l) (reduce #'* (mapcar #'multiply-leaves l)))
             ((zerop l) (signal 'zero-found))
             (t l)))
MULTIPLY-LEAVES
CL-USER> (multiply-leaves '((1 2 (3 4)) (((5 6))) 7))
5040

CL-USER> (handler-case
           (multiply-leaves '((1 2 (3 0)) (((5 6))) "hello"))
           (zero-found () 0))
0
```

Définition d'une condition avec créneau

```
(define-condition bad-entry-error (error)
  ((contents :initarg :contents :reader contents)))

(defun parse-entry (contents)
  (let ((ok (integerp contents)))
    (if ok
      contents
      (error 'bad-entry-error :contents contents))))

(defun parse-log (stream)
  (loop
   for text = (read stream nil nil)
   while text
   for entry = (handler-case (parse-entry text)
                 (bad-entry-error (c)
                   (format *error-output* "bad-entry ~A~%" (contents c))))
   when entry collect entry))

CL-USER> (with-input-from-string (foo "12 23 ee 1 rr 45")
           (parse-log foo))
bad-entry EE
bad-entry RR
(12 23 1 45)
```

Conditions : exemple tiré du Sudoku

```
(define-condition impossible-assignment (error)
  ((digit :initarg :digit :reader digit)
   (coord :initarg :coord :reader coord))
  (:report
    (lambda (c stream)
      (format stream  "impossible to put ~d on location ~s"
        (digit c)
        (coord c)))))

(define-condition impossible-prohibition (error)
  ...)
```

:report introduit une "print-function" pour les conditions de ce type.

Conditions : exemple tiré du Sudoku (suite)

```
(defun put (grid n coord)
  "tries to put n on cell (coord) ; returns NIL if not possible"
  (handler-case (set-digit grid n coord)
    ((or impossible-assignment impossible-prohibition) ()
      (format t "probably inconsistent grid~%"))))


(defun set-digit (grid coord n)
...
  (unless (member n (possibilities cell) :test #'=)
    (error 'impossible-assignment :digit n :coord coord))
...)
```

Documentation des conditions

Pour les fonctions prédéfinies, se reporter aux rubriques `Exceptional Situations` de la `Hyperspec`.

```
Function OPEN
...
Exceptional Situations:
...
An error of type file-error is signaled if (wild-pathname-p filespec)
returns true.

An error of type error is signaled if the external-format is not
understood by the implementation.
...
```

Le programmeur doit décrire les conditions qui peuvent être signalées par les fonctions qu'il propose.

Flots (Streams)

```
CL-USER> (open "/usr/labri/idurand/Data"
               :direction :input)
error opening #P"/usr/labri/idurand/Data": No such file or directory
   [Condition of type SB-INT:SIMPLE-FILE-ERROR]
CL-USER> (open "/usr/labri/idurand/Data"
               :direction :input
               :if-does-not-exist nil)
NIL
CL-USER> (open "/usr/labri/idurand/Data"
               :direction :input
               :if-does-not-exist nil)
#<SB-SYS:FD-STREAM for "file /usr/labri/idurand/Data" A800A79>
```

Flots

```
CL-USER> (defparameter *flot*
                       (open "/usr/labri/idurand/Data"
                             :direction :input
                             :if-does-not-exist nil))
*FLOT*
CL-USER> (handler-case
           (loop
             for line = (read-line *flot*)
             while line
             do (print line))
           (end-of-file () 'fin-de-fichier))
"Bonjour,"
""
"ceci est un test."
""
FIN-DE-FICHIER
```

Flots

```
CL-USER> (close *flot*)
T
```

Macros `with-open-file`, `with-input-from-string`, `with-output-to-string`, `with-open-stream`

```
(with-open-file
  (flot filename :direction :output
                 :element-type 'unsigned-byte)
  ...
  (write-byte b flot)
  ...
  ...)
```

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 symboles.

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 moins 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)
```

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`)

```
CL-USER> (deftype proseq ()
           '(or vector (and list (not (satisfies circular?)))))
PROSEQ
CL-USER> (typep #(1 2) 'proseq)
T
```

Définition de nouveaux noms de types

Paramétrage avec des arguments :

```
CL-USER> (deftype multiple-of (n)
           `(and integer (satisfies (lambda (x)
                                      (zerop (mod x ,n))))))
MULTIPLE-OF
CL-USER> (typep 12 '(multiple-of 4))
T
CL-USER> (typep 12 '(multiple-of 5))
NIL
```

Read macros

Possibilité de programmer la fonction de lecture de code

Association entre un caractère et 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 définir des caractères similaires à # (mais pas souvent nécessaire)

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

Read macros (suite)

```
CL-USER> (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)))))
...
CL-USER> #?7
(0 1 2 3 4 5 6 7)
```

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

Read macros (suite)

Création de listes :

```
CL-USER> (set-macro-character #\} (get-macro-character #\)))
T
```

Read macros (suite)

```
CL-USER> (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)))))
...
CL-USER> #2 7
(2 3 4 5 6 7)
```