Projet de Programmation 3

Cours : Irène Durand
TD : Irène Durand, Richard Moussa,
Kaninda Musumbu (2 groupes)

	Semaines	
Cours mercredi 9h30-10h50,	2-7 9-10	Amphi A29
TD 12 séances de 1h20	5-7 9 10 12	
1 Devoir surveillé mercredi 9h30-11h	11	Amphi A29
1 Projet surveillé	9 à 13	

Le devoir surveillé ET le projet surveillé sont *obligatoires*.

Support de cours : transparents et exemples disponibles sur page Web

http://dept-info.labri.u-bordeaux.fr/~idurand/enseignement/PP3/

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

Robert Strandh
et Irène Durand — Traité de programmation en Common Lisp
MétaModulaire

Sonya Keene : — Object-Oriented Programming in Common Lisp
A programmer's guide to CLOS
Addison Wesley

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)

Common Lisp Interface Manager (CLIM)
http://bauhh.dyndns.org:8000/clim-spec/index.html

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

David Lamkins : Successful Lisp (Tutorial en-ligne)

Objectifs et contenu

Passage à l'échelle

- Appréhender les problèmes relatifs à la réalisation d'un vrai projet
- Installation et utilisation de bibliothèques existantes
- Bibliothèque graphique (`mcclim`)
- Création de bibliothèques réutilisables
- Modularité, API (Application Programming Interface) ou Interface de Programmation
- Utilisation de la Programmation Objet
- Entrées/Sorties (flots, accès au système de fichiers)
- Gestion des exceptions

Paquetages (Packages)

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

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

Par contre, on veut utiliser certaines fonctions, variables, classes, macros proposées dans l'interface de programmation (API) du système.

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

Pour que ce soit possible, il faut que le système se serve de la notion de paquetage (package).

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 doubler (n) (* 2 n))
DOUBLER
CL-USER> (symbol-package 'doubler)
#<PACKAGE "COMMON-LISP-USER">
CL-USER> (symbol-package ':test)
#<PACKAGE "KEYWORDS">
```

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> (doubler 3)
The function DOUBLER is undefined.
   [Condition of type UNDEFINED-FUNCTION]
SUDOKU> (common-lisp-user::doubler 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 'doubler)
T
CL-USER> (in-package :sudoku)
#<PACKAGE "SUDOKU">
SUDOKU> (common-lisp-user:doubler 2)
4
SUDOKU> (import 'doubler :common-lisp-user)
T
SUDOKU> (doubler 2)
4
```

Paquetages (suite)

Quand on crée un paquetage correspondant à un système, 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 :gui-sudoku
  (:use :common-lisp :clim)
  (:import-from :sudoku
    #:make-grid
    #:show
    ...))
```

`:use` permet d'importer tous les symboles exportés par un ou plusieurs paquetages.

Remarque : si on n'importe pas les symboles de `:common-lisp`, on n'a aucun opérateur (fonction ou macro) ou variable prédéfinis. On n'a donc que les constantes du langage.

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 package COMMON-LISP-USER:
  1. SUDOKU:FOO
  2. FOO

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

Pathnames

Un objet de type `pathname` contient une représentation structurée d'un nom de fichier (au sens large).

```
CL-USER> *default-pathname-defaults*
#P"/nfs4/home4/idurand/"
CL-USER> (namestring *default-pathname-defaults*)
"/nfs4/home4/idurand/"
```

La plupart des fonctions qui s'appliquent à un `pathname` s'appliquent aussi à la chaîne de caractères.

```
CL-USER> (pathname-directory *default-pathname-defaults*)
(:ABSOLUTE "nfs4" "home4" "idurand")
CL-USER> (pathname-directory "Enseignement/PP3")
(:RELATIVE "Enseignement")
```

On peut passer de la représentation sous forme de chaîne de caractères à un pathname et réciproquement.

```
CL-USER> (setf *p*
           (make-pathname
             :directory (pathname-directory
                          *default-pathname-defaults*)
             :name "fichier"))
#P"/nfs4/home4/idurand/fichier"
CL-USER> (namestring *p*)
"/nfs4/home4/idurand/fichier"
CL-USER> (file-namestring *p*)
"fichier"
CL-USER> (directory-namestring *p*)
"/nfs4/home4/idurand/"
CL-USER> (make-pathname :directory "usr/labri/idurand")
#P"/usr/labri/idurand/"
```

ASDF : Another System Definition Facility

ASDF est utilisé par SLIME donc chargé automatiquement quand on travaille sous SLIME.
Sinon on peut le charger manuellement : `* (require :asdf)`
ASDF recherche les systèmes (définis pas leur fichier `.asd`)

```
CL-USER> (asdf:COMPUTE-SOURCE-REGISTRY)

...
 #P"/opt/local/lib/sbcl/site-systems/"
 #P"/Users/idurand/.sbcl/systems/" #P"/opt/local/lib/sbcl/")
```

On peut modifier ce comportement par défaut.

ASDF : Charger un système

Pour pouvoir charger un système, il faut que sa définition (fichier `.asd`) se trouve dans l'un des répertoires connus par ASDF.
Ces répertoires contiennent en général, des liens symboliques vers le fichier `.asd` fourni avec les sources.

```
$ ln -s ../site/flexichain/flexichain.asd .
```

```
CL-USER> (asdf:operate 'asdf:load-op 'flexichain)
; compiling file "/usr/labri/idurand/.sbcl/site/flexichain/flexichain-package.lisp" (
; compiling (DEFPACKAGE :FLEXICHAIN ...)

; /usr/labri/idurand/.sbcl/site/flexichain/flexichain-package.fasl written
; compilation finished in 0:00:00
; compiling file "/usr/labri/idurand/.sbcl/site/flexichain/utilities.lisp" (written 1
; compiling (IN-PACKAGE :FLEXICHAIN)
; compiling (DEFUN SQUARE ...)

...
NIL
```

Le système est compilé, si nécessaire.

```
CL-USER> (asdf:operate 'asdf:load-op 'flexichain)

NIL
```

Sous SBCL, on peut aussi utiliser `require` (pas dans la norme, implémentation diffère avec les LISP).

```
CL-USER> (require :mcclim)

; loading system definition from
; /usr/labri/idurand/.sbcl/systems/spatial-trees.asd into
; #<PACKAGE "ASDF0">
; registering #<SYSTEM :SPATIAL-TREES A8DE101> as SPATIAL-TREES
; loading system definition from /usr/labri/idurand/.sbcl/systems/clx.asd
; into #<PACKAGE "ASDF0">
; registering #<SYSTEM CLX BC52081> as CLX
NIL
```

Au démarrage, LISP exécute automatiquement un fichier d'initialisation (s'il existe).

SBCL	`~/.sbclrc`
OpenMCL	`~/openmcl-init.lisp`
CMUCL	`~/cmucl-initlisp.lisp`

ASDF : Définir un système

```
$ pwd
/usr/labri/idurand/.sbcl/site/Compta
$ ls *.{asd,lisp}
compta.asd  gui.lisp  model.lisp  packages.lisp
$ cat compta.asd
(in-package #:cl-user)

(asdf:defsystem :compta
  :depends-on (:mcclim)
  :components
  ((:file "packages" :depends-on ())
   (:file "model" :depends-on ("packages"))
   (:file "io" :depends-on ("packages" "model"))
   (:file "gui" :depends-on ("packages" "model"))
   ))
```

Seul le nom du système est obligatoire. Tous les autres arguments sont passés par mot-clé.

Création d'une image mémoire

La plupart des LISP offrent la possibilité de sauvegarder l'état du système pour reprise de l'exécution plus tard. Cet état est sauvegardé dans un fichier core (image mémoire).

Sous SBCL

```
Function:
sb-ext:save-lisp-and-die
  core-file-name
  &key toplevel purify root-structures environment-name executable
```

Quand on lance SBCL, par défaut l'image mémoire `.../lib/sbcl/sbcl.core` est chargée.
Elle contient toutes les fonctions, macros, variables qui constituent le Lisp de base (défini par la HyperSpec).

Si on prévoit d'utiliser régulièrement certaines bibliothèques, on peut lancer Lisp, charger les bibliothèques, sauver l'image mémoire et par la suite lancer Lisp avec cette nouvelle image mémoire. On obtient un Lisp "enrichi".

Principe :
Sous `SBCL` lancé depuis un terminal (pas depuis `Emacs` et `Slime`),

1. charger toutes les bibliothèques dont on veut disposer en permanence
2. sauver l'image mémoire : `(save-lisp-and-die "clim.core")`
3. par la suite lancer `SBCL` avec ce nouveau core : `sbcl --core <chemin vers clim.core>`

Pour que ce Lisp enrichi soit chargé automatiquement dans `SLIME`, modifier le `.emacs` :

```
(setq inferior-lisp-program "sbcl --core <chemin vers fichier .core>")
```

Remarque : comme on est un peu perdu sans les facilités d'édition de Emacs et SLIME, on peut mettre toutes les commandes à taper dans un fichier et le charger.

```
[idurand@chataigne.labri.fr 318]cat build-core.lisp
(require :asdf)
(require :mcclim)
(save-lisp-and-die "clim.core")
[idurand@chataigne.labri.fr 317]sbcl
This is SBCL 1.0.9, an implementation of ANSI Common Lisp.
...
* (load "build-core.lisp")
...
[saving current Lisp image into /nfs4/home4/idurand/clim.core:
writing 1912 bytes from the read-only space at 0x01000000
...
done]
[idurand@chataigne.labri.fr 318]
```

On peut aussi tout intégrer dans un seul shell script : `build-core.sh`

```
#!/bin/sh
sbcl --no-userinit <<EOF
(require :asdf)
(require :mcclim)
(save-lisp-and-die "clim.core")
EOF
```

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
```

Méthodes et fonctions génériques

Le prototype et la documentation d'une opération sont donnés par une fonction générique (`defgeneric`).

Une méthode est une implémentation partielle d'une opération pour certains types d'arguments (`defmethod`).

Par défaut, au moment de l'appel d'une opération, une seule méthode est choisie (mécanisme de sélection de méthode).

L'ensemble des méthodes applicables est trié suivant un ordre total.
La première méthode dans cet ordre sera appelée.

Il existe un ordonnancement par défaut : méthode la plus spécifique en suivant l'ordre lexicographique.

Dans le corps d'une méthode, l'appel `(call-next-method)` déclenche l'exécution de la méthode suivante.

Méthodes primaires

Exemple avec `print-object` dans `protocole-et-premiere-implementation`.

```
(defgeneric print-object (object stream)
  (:documentation "..."))
```

Si on n'avait pas défini `print-objet` pour `polygon`

```
CL-USER> (setf *triangle* (make-polygon #c(0 0) #c(2 0) #c(0 1)))
#<POLYGON D2C1811>
```

```
(defmethod print-object ((polygon polygon) stream)
  (call-next-method)
  (format stream "(corners: ~A)" (corners polygon)))
```

```
CL-USER> (setf *triangle* (make-polygon #c(0 0) #c(2 0) #c(0 1)))
#<POLYGON D2C1811>(corners: (0 2 #C(0 1)))
```

Méthodes primaires

Essayer `C-c C-d C-d` (`slime-describe-symbol`) sur `print-object`.

```
PRINT-OBJECT is an external symbol in #<PACKAGE "COMMON-LISP">.
#<STANDARD-GENERIC-FUNCTION PRINT-OBJECT (208)> is a generic function.
Its lambda-list is:
  (SB-PCL::OBJECT STREAM)
Its method-combination is:
  #<SB-PCL::STANDARD-METHOD-COMBINATION STANDARD NIL {917CF51}>
Its methods are:
  (PRINT-OBJECT (CIRCLE T))
  (PRINT-OBJECT (POLYGON T))
...
```

Remarquer `method-combination is: ... STANDARD ....`

Le mécanisme de sélection de méthodes est paramétrable.
Voir Section 7.6.6 Method Selection and Combination.

Il existe des mécanismes prédéfinis.
Voir Section 7.6.6.4 Built-in Method Combination Types

On peut aussi définir son propre mécanisme !

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

Méthodes secondaires after et before

Utilisation de méthodes :after, :before

Mécanisme par défaut :
Avant appel de la méthode primaire, toutes les méthodes before sont appelées, de la plus spécifique à la moins spécifique.

Après appel de la méthode primaire, toutes les méthodes after sont appelées, de la moins spécifique à la plus spécifique.

L'exemple vu prédédemment avec `print-object`, peut s'écrire plus élégamment avec une méthode after, sans redéfinir la méthode primaire pour `polygon`.

```
(defmethod print-object :after ((polygon polygon) stream)
  (format stream "(corners: ~A)" (corners polygon)))
```

plus besoin du `call-next-method` qui sera par contre utile dans les méthodes around vues prochainement.

Méthodes secondaires, exemples

```
(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))
```

Possibilité d'étendre un comportement sans avoir accès au source

Souplesse dans l'implémentation

Méthodes secondaires

Utilisation de méthodes :around (application possible : mémoïsation)

```
(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
```

Application à la Mémoïsation

Principe : sauvegarder le résultat d'un calcul pour ne pas avoir à le refaire plus tard.

Illustration avec les programmes

`protocole-et-premiere-implementation.lisp`

et

`memo.lisp`

Fonctions sur les méthodes

```
CL-USER> (find-class 'polygon)
#<STANDARD-CLASS POLYGON>
CL-USER> (find-method #'area '() (mapcar #'find-class '(region)))
#<STANDARD-METHOD AREA (REGION) BE06D71>
CL-USER> (find-method #'area '(:around) (mapcar #'find-class
                        '(area-mixin)))
#<STANDARD-METHOD AREA :AROUND (AREA-MIXIN) D2C4C71>
PACMAN> (find-method #'collision '()
                                (mapcar #'find-class '(pacman tablet)))
#<STANDARD-METHOD COLLISION (PACMAN TABLET) ADE7091>
CL-USER> (remove-method
           #'area
          (find-method #'area '(:around)
                       (mapcar #'find-class '(area-mixin))))
#<STANDARD-GENERIC-FUNCTION AREA (1)>
```

Combinaisons non standard

9 combinaisons de méthodes prédéfinies :

`+`, `and`, `or`, `list`, `append`, `nconc`, `min`, `max`, `progn`.

Mécanisme :

1. toutes les méthodes sont appelées, par défaut, de la plus spécifique vers la moins spécifique.
2. la valeur retournée est : - calculée en appliquant l'opérateur (`+`, `and`, ...) aux valeurs retournées par chacune des méthodes appelées dans les 8 premiers cas, - la valeur de la dernière méthode appelée pour `progn`.

La combinaison est choisie au niveau du `defgeneric`.

```
(defgeneric m (object)
  (:method-combination list))
```

pour inverser l'ordre :

```
(defgeneric m (object)
  (:method-combination list :most-specific-last))
```

Exemple avec append

```
(defclass c () ())
(defclass c1 (c) ())
(defclass c11 (c1) ())

(defgeneric m (object)
  (:documentation "test std)"))

(defmethod m ((o c))
  (format t "method for c"))

(defmethod m ((o c1))
  (format t "method for c1"))

(defmethod m ((o c11))
  (format t "method for c11"))
CL-USER> (defparameter
           *o*
           (make-instance 'c11))
*O*
CL-USER> (m *o*)
method for c11
NIL
```

```
(defgeneric m-append (object)
  (:method-combination append
   :most-specific-last)
  (:documentation "test append)"))

(defmethod m-append append ((o c))
  (format t "method for c~%")
  '(c))

(defmethod m-append append ((o c1))
  (format t "method for c1~%")
  '(c1))

(defmethod m-append append ((o c11))
  (format t "method for c11~%")
  '(c11))
CL-USER> (m *o*)
method for c
method for c1
method for c11
(C C1 C11)
```

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)
  ...
  ...)

(with-input-from-string
```

ASDF-INSTALL

Système (bibliothèque) permettant de télécharger et d'installer des systèmes depuis Internet.

ASDF-INSTALL est inclus dans SBCL et doit être chargé avant utilisation.

```
CL-USER> (require :asdf-install)
("ASDF-INSTALL")
```

Par défaut les systèmes installables sont recherchés à l'url http://www.cliki.net/.
Exemples :

http://www.cliki.net/Flexichain

http://www.cliki.net/McClim

http://www.cliki.net/Gsharp

ASDF-INSTALL (2)

et installés au choix dans un des répertoires indiqués par `asdf-install:*locations*`.

```
CL-USER> asdf-install:*locations*
((#P"/usr/lib/sbcl/site/" #P"/usr/lib/sbcl/site-systems/"
  "System-wide install")
 (#P"/usr/labri/idurand/.sbcl/site/" #P"/usr/labri/idurand/.sbcl/syste
  "Personal installation"))
CL-USER> (asdf-install:install 'flexichain)
Install where?
1) System-wide install:
   System in /usr/lib/sbcl/site-systems/
   Files in /usr/lib/sbcl/site/
2) Personal installation:
   System in /usr/labri/idurand/.sbcl/systems/
   Files in /usr/labri/idurand/.sbcl/site/
 -->
```

ASDF-INSTALL (3)

On peut aussi donner en argument à `install` l'`url` d'une archive :

```
CL-USER> (asdf-install:install
           "http://weitz.de/files/cl-ppcre.tar.gz")
...
NIL
```

ou encore l'adresse d'une archive `tar.gz` locale :

```
CL-USER> (asdf-install:install "/tmp/flexichain_1.2.tgz")
...
NIL
```

L'étape suivante serait d'appendre comment rendre sa propre application asdf-installable pour la communauté des utilisateurs.

ASDF-INSTALL (4)

Pendant l'installation d'un système depuis Internet, la présence de sa clé publique est vérifiée. En général, on trouve cette clé sur la page Web de l'auteur du système.

```
No key found for key id 0x#1=595FF045057958C6.  Try some command like
  gpg  --recv-keys 0x#1#
   [Condition of type ASDF-INSTALL::KEY-NOT-FOUND]

Restarts:
  0: [SKIP-GPG-CHECK] Don't check GPG signature for this package
  1: [ABORT-REQUEST] Abort handling SLIME request.
  2: [TERMINATE-THREAD] Terminate this thread (#<THREAD "new-repl-thread" DF92281>)

Backtrace:
  0: (ASDF-INSTALL::VERIFY-GPG-SIGNATURE/STRING "-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.5 (Cygwin)
```

Il est **risqué** d'installer depuis Internet sans récupérer les clés publiques (GP) des auteurs des systèmes car les systèmes se trouvent sur un serveur de type wiki.

La fonction `read`

`read &optional input-stream eof-error-p eof-value recursive-p => object`
Fonction `read` de la REPL.

```
CL-USER> (read)
(+ 1 2)
(+ 1 2)
CL-USER> (with-input-from-string (stream "12") (read stream))
12
12
CL-USER> (with-input-from-string (stream "") (read stream))
end of file on #<SB-IMPL::STRING-INPUT-STREAM B12D731>
   [Condition of type END-OF-FILE]
CL-USER> (with-input-from-string (stream "") (read stream nil))
NIL
CL-USER> (with-input-from-string (stream "") (read stream nil 'toto))
TOTO
```

Le dernier argument `recursive-p` est par défaut à faux. On le met quand on appelle `read` récursivement à l'intérieur d'un appel à `read` ; ce sera le cas dans les read-macros.

Read macros

Rappel : dans la boucle d'interaction, le reader (grâce à la fonction `read`), lit un texte en entrée qu'il transforme en une structure interne (liste ou atome) laquelle est ensuite évaluée par `eval`.

Le comportement du `reader` est paramétré par des variables. (`*read`*complétion* pour voir le nom de ces variables) En changeant, ces variables, on peut changer le comportement du reader.

La table accessible à partir de la variable `*readtable*` contient des correpondances entre un caractère et une fonction de lecture à utiliser par le reader quand ce caractère se présente.

On peut gérer plusieurs readtables, `*readtable*` est celle utilisée par défaut par le reader.

get-macro-character

`get-macro-character`

`char &optional readtable => function, non-terminating-p`

retourne la fonction de lecture associée au caractère `char`, par défaut dans la `*readtable*` courante.

```
CL-USER> (get-macro-character #\( )
#<FUNCTION SB-IMPL::READ-LIST>
NIL
CL-USER> (get-macro-character #\) )
#<FUNCTION SB-IMPL::READ-RIGHT-PAREN>
NIL
CL-USER> (get-macro-character #\' )
#<FUNCTION SB-IMPL::READ-QUOTE>
NIL
CL-USER> (get-macro-character #\# )
#<FUNCTION SB-IMPL::READ-DISPATCH-CHAR>
NIL
CL-USER> (get-macro-character #\a )
NIL
NIL
```

set-macro-character (1)

`set-macro-character`

`char new-function &optional non-terminating-p readtable => t`

Cette fonction associe la fonction `new-function` au caractère `char`.
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))))
```

Exemple :

```
(set-macro-character
  #\$
  (lambda (stream char)
            (declare (ignore char))
    (let ((n (read stream t nil t)))
      (make-array n))))
```

```
CL-USER> $3
#(0 0 0)
```

set-macro-character (2)

```
(set-macro-character
  #\$
  (lambda (stream char)
            (declare (ignore char))
    (let ((n (read stream t nil t)))
      (list 'quote (make-list n)))))

CL-USER> $3
(NIL NIL NIL)
```

Dans le cas de `make-array`, `quote` pas obligatoire puisque `#(0 0 0)` s'évalue en `#(0 0 0)`.

Par contre, pour `make-list (NIL NIL NIL)` donnerait une erreur. On écrirait `'(nil nil nil)` ou encore `(quote (nil nil nil))`.

Le reader retourne `(quote (nil nil nil))` qui s'évalue en `(NIL NIL NIL)`.

read-delimited-list

```
read-delimited-list char &optional input-stream recursive-p => list
```

```
CL-USER> (read-delimited-list #\))
1 2 3)
(1 2 3)
CL-USER> (read-delimited-list #\])
1 2 3 ]
(1 2 3)
CL-USER> (read-delimited-list #\])
1 2 3]
]
(1 2 3])
```

set-syntax-from-char

On associe au caractère], la même fonction que celle qui est associée au caractère).

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

ou plus simplement utiliser la fonction `set-syntax-from-char`

```
CL-USER> (set-syntax-from-char #\] #\))
T
```

Le reader va maintenant bien considérer le caractère] comme un délimiteur :

```
CL-USER> (read-delimited-list #\])
1 2 3]
(1 2 3)
```

Exemple

On veut pouvoir utiliser expressions binaires en notation infixe : {e1 op e2} doit être transformé en (op e1 e2).

```
CL-USER> (set-syntax-from-char #\} #\))
T
CL-USER> (set-macro-character
  #\{
  (lambda (stream char)
    (declare (ignore char))
    (let ((l (read-delimited-list #\} stream t)))
      (list (second l) (first l) (third l))))
  nil)
T
CL-USER> (read)
{(1+ 3) - 4}
(- (1+ 3) 4)
CL-USER> {(1+ 3) - 4}
0
```

Dispatching character

Par défaut, le caractère # est un caractère de "dispatch" (dispatching character).

Le caractère qui suit le caractère # servira à déclencher la bonne fonction de lecture.

La plupart des types de données (sauf les listes) sont introduits par #.

```
CL-USER>  #\c
#\c  ;; un caractère
CL-USER>  #(1 2)
#(1 2)   ;; un tableau
CL-USER>  #C(1 2)
#C(1 2)  ;; un complexe
```

Possibilité de définir d'autres caractères de dispatch que # (mais pas souvent nécessaire)

set-dispatch-macro-character

Possibilité de spécifier un caractère associé à un caractère de dispatch (# le plus souvent) avec `set-dispatch-macro-character`

```
set-dispatch-macro-character
     disp-char sub-char new-function &optional readtable => t
```

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

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

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

read-macros (suite)

Exemple : création de listes

```
CL-USER> (set-dispatch-macro-character #\# #\{
           (lambda (stream char1 char2)
             (declare (ignore 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)))))
#<FUNCTION (LAMBDA (STREAM CHAR1 CHAR2)) BEF1DE5>
CL-USER> (read)
#{2 7}
'(2 3 4 5 6 7)
CL-USER> #{2 7}
(2 3 4 5 6 7)
```

Système IO

Petit système permettant de sauver et charger facilement les données d'un modèle.

Le format choisi pour représenter les données au format externe doit être facile à relire avec des read-macros.

Ce système est indépendant de tout modèle, mais pour chaque modèle, il faut définir pour chaque objet du modèle, les créneaux à sauvegarder grâce à la macro `define-save-info`.

```
(define-save-info arc
  (:origin origin)
  (:extremity extremity))

(define-save-info node
  (:out-arcs node-out-arcs))

(define-save-info graph
  (:nodes graph-nodes)
  (:arcs graph-arcs))
```

```
[GRAPH-MODEL:GRAPH :ORIENTED-P COMMON-LISP:NIL
 :NODES
  (#1=[GRAPH-MODEL:NODE :LABEL "Bordeaux" :NUM 5
       :OUT-ARCS
        (#2=[GRAPH-MODEL:ARC :ORIGIN #1#
             :EXTREMITY
              #3=[GRAPH-MODEL:NODE :LABEL "Paris" :NUM 1
                  :OUT-ARCS
                   (#4=[GRAPH-MODEL:ARC :ORIGIN #3# :EXTREMITY #1# ]
                    #5=[GRAPH-MODEL:ARC :ORIGIN #3#
                        :EXTREMITY
                         #6=[GRAPH-MODEL:NODE :LABEL "Lyon" :NUM 2
                             :OUT-ARCS
                              (#7=[GRAPH-MODEL:ARC :ORIGIN #6#
                                   :EXTREMITY
                                    #8=[GRAPH-MODEL:NODE :LABEL "Marseille" :NUM 3
                                        :OUT-ARCS
                                         (#9=[GRAPH-MODEL:ARC :ORIGIN #8#
                                              :EXTREMITY
                                               #10=[GRAPH-MODEL:NODE :LABEL "Toulouse" :NUM 4
                                                    :OUT-ARCS
                                                     (#11=[GRAPH-MODEL:ARC
                                                           :ORIGIN #10# :EXTREMITY #8# ]
                                                      #12=[GRAPH-MODEL:ARC
                                                           :ORIGIN #10# :EXTREMITY #1# ]) ] ]
                                          #13=[GRAPH-MODEL:ARC :ORIGIN #8# :EXTREMITY #6# ]) ] ]
                                          #14=[GRAPH-MODEL:ARC :ORIGIN #6# :EXTREMITY #3# ]) ] ]) ] ]
         #15=[GRAPH-MODEL:ARC :ORIGIN #1# :EXTREMITY #10# ]) ]
   #10# #8# #6# #3# [GRAPH-MODEL:NODE :LABEL "Ajaccio" :NUM 0 :OUT-ARCS COMMON-LISP:NIL ])
 :ARCS (#11# #9# #2# #4# #15# #12# #13# #7# #14# #5#) ]
```

Readmacros pour charger un objet

```
;;; fichier io.lisp
(defparameter *io-readtable* (copy-readtable))

(defun read-model-object (stream char)
  (declare (ignore char))
  (apply #'make-instance (read-delimited-list #\] stream t)))

(set-macro-character #\[ #'read-model-object nil *io-readtable*)
(set-syntax-from-char #\] #\) *io-readtable*)

GRAPH-IO> (let ((*readtable* *io-readtable*))
            (read))
[GRAPH-MODEL:NODE :LABEL "Ajaccio" :NUM 0 :OUT-ARCS COMMON-LISP:NIL]
#<NODE 1003E34781>
```

Chargement d'un objet du modèle

```
(defun read-graph-from-stream (stream)
  (read-model-from-stream stream *graph-allowed-version-names*))

(defun read-model-from-stream (stream allowed-version-names)
  (let ((version (read-line stream)))
    (assert (member version allowed-version-names :test #'string=))
    (if (member version allowed-version-names :test #'string=)
        (let ((*read-eval* nil)
              (*readtable* *io-readtable*))
          (read stream))
        'unknown-file-version)))
```

Combinaison append dans IO

Fichier io.lisp.

```
(defgeneric save-info (object)
  (:method-combination append :most-specific-last))
```

À l'implémentation de save-info pour des objets de la classe a-class, il faudra utiliser une méthode append :

```
(defmethod save-info append ((object a-class)) ...)
```

Exemples

```
(defmethod save-info append ((obj organization))
  '((:NAME NAME) (:ACCOUNTS ACCOUNTS) (:TRANSACTIONS TRANSACTIONS)))

(defmethod save-info append ((obj date))
  '((:YEAR YEAR) (:MONTH MONTH) (:DAY DAY)
    (:HOUR HOUR) (:MINUTE MINUTE)))
```

Sauvegarde de données IO

On sauve les informations essentielles (celles qui ne peuvent pas être calculées à partir des autres) et sous un format qui permettra de les relire facilement.

Pour chaque type d'objet de l'application, on redéfinit `print-object`.

```
(defmethod print-object ((obj organization) stream)
  (if *print-for-file-io*
      (print-model-object obj stream)
      (call-next-method)))

(defmethod print-object ((obj date) stream)
  (if *print-for-file-io*
      (print-model-object obj stream)
      (call-next-method)))
...
```

Sauvegarde de données (2)

```
GRAPH-MODEL> (let ((io:*print-for-file-io* t))
                (print-object (make-instance 'node :num 10 :label "Par
[NODE :WEIGHT NIL :LABEL "Paris" :NUM 10 :OUT-ARCS NIL ]
NIL
```

Comme on doit définir save-info et print-object pour chaque type d'objet de `Graph`.

C'est une macro de `io` qui fait ce travail :

```
(defmacro define-save-info (type &body save-info)
  `(progn
     (defmethod print-object :around ((obj ,type) stream)
       (if *print-for-file-io*
           (print-model-object obj stream)
           (call-next-method)))
     (defmethod save-info append ((obj ,type))
       ',save-info)))
```

Sauvegarde des données (3)

```
(macroexpand-1 '(define-save-info arc
                                (:origin origin) (:extremity extremity)))

(PROGN
 (DEFMETHOD PRINT-OBJECT :AROUND ((IO::OBJ ARC) STREAM)
            (IF *PRINT-FOR-FILE-IO* (IO::PRINT-MODEL-OBJECT IO::OBJ STREAM)
                (CALL-NEXT-METHOD)))
 (DEFMETHOD IO::SAVE-INFO APPEND ((IO::OBJ ARC))
            '((:ORIGIN ORIGIN) (:EXTREMITY EXTREMITY))))
T


GRAPH-MODEL> (setf *node* (make-instance node))
10<Paris>
GRAPH-MODEL> (io::save-info *node*)
((:WEIGHT WEIGHT) (:LABEL LABEL) (:NUM NUM) (:OUT-ARCS NODE-OUT-ARCS))
```

```
(defun print-model-object (obj stream)
  (pprint-logical-block (stream nil :prefix "[" :suffix "]")
    (format stream "~s ~2i" (class-name (class-of obj)))
    (loop for info in (save-info obj)
          do (format stream
                     "~_~s ~W "
                     (car info)
                     (funcall (cadr info) obj)))))

(defun write-model-to-stream (stream version-name object)
  (let ((*print-circle* t)
        (*print-for-file-io* t)
        (*package* (find-package :keyword)))
    (format stream "~A~%" version-name )
    (pprint object stream) ;; appelle :around print-object du define-save-info
    (terpri stream)
    (finish-output stream)))

(defun write-model (filename version-name object)
  (with-open-file (stream filename
                          :direction :output
                          :if-exists :supersede
                          :if-does-not-exist :create)
    (when stream
      (write-model-to-stream stream version-name object))))
```

Mécanisme `unwind-protect`

Pour écrire des programmes robustes, il faut un mécanisme qui assure que certaines parties cruciales du code seront exécutées quoi qu'il arrive (erreur, interruption de l'utilisateur, ...).

`unwind-protect protected-form cleanup-form* => result*`

```
(let (resource stream)
  (unwind-protect
       (progn
         (setf resource (allocate-resource)
               stream (open-file))
         (process stream resource))
    (when stream (close stream))
    (when resource (deallocate-resource resource))))
```

Sauts non locaux

Un saut non local est une façon d'abandonner l'exécution normale (en séquence, ou dans une boucle) 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

Sortie d'un bloc lexical : return-from

Saut non local similaire au `break` du langage C

```
CL-USER> (block hello
           'debut
           (when (> *print-base* 20)
             (return-from hello 'saut))
           'fin)
FIN
```

```
CL-USER> (block hello
           'debut
           (when (< *print-base* 20)
             (return-from hello 'saut))
           'fin)
SAUT
```

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]
```

Typage avec assert

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 en temps d'exécution).

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 qu'au 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.

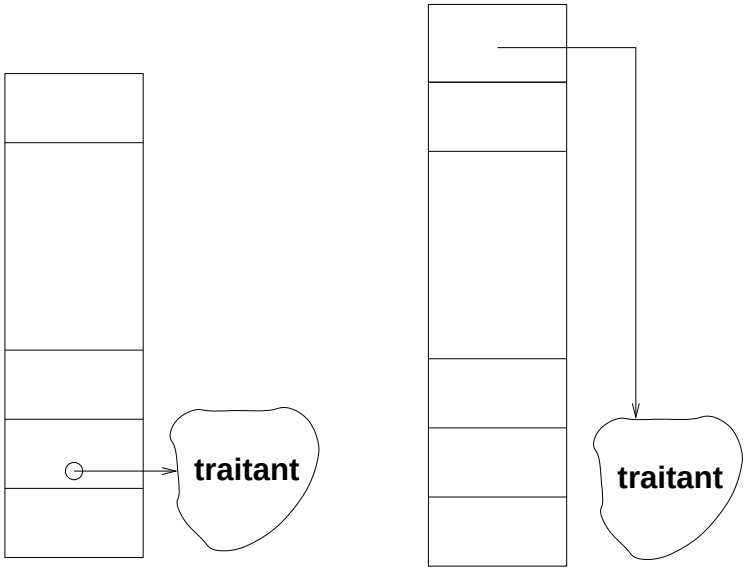
Le traitant est associé à un type de condition, et le traitant est appelé seulement si la condition est du bon type.

Principes de base

1. positionnement d'un traitant de condition (`handler-bind`)
2. signalisation d'une condition (`signal`,`warn`, `error`, `assert`)

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.

Macro handler-case

permet de gérer les erreurs à la manière de Python ou de Java (saut non local (et dépilement) à l'endroit où le traitant avait été positionné)

```
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
```

handler-case (suite)

Un handler-case se traduit en un handler-bind pouvant faire un saut non local (return-from).

```
CL-USER> (macroexpand-1 '(handler-case
            (/ *n* 2)
          (division-by-zero ()
            (prog1 nil (print "error")))))

(BLOCK #:G3894
  (LET ((#:G3895 NIL))
    (DECLARE (IGNORABLE #:G3895))
    (TAGBODY
      (HANDLER-BIND
       ((DIVISION-BY-ZERO
         (LAMBDA (SB-IMPL::TEMP)
           (DECLARE (IGNORE SB-IMPL::TEMP))
           (GO #:G3896))))
       (RETURN-FROM #:G3894
         (MULTIPLE-VALUE-PROG1 (/ *N* 2) (SB-KERNEL:FLOAT-WAIT))))
     #:G3896
      (RETURN-FROM #:G3894 (LOCALLY (PROG1 NIL (PRINT "error")))))))
T
```

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 de 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`,...

Définition d'une condition : exemples

```
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)
```

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.

Exemple complet