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
*
```

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

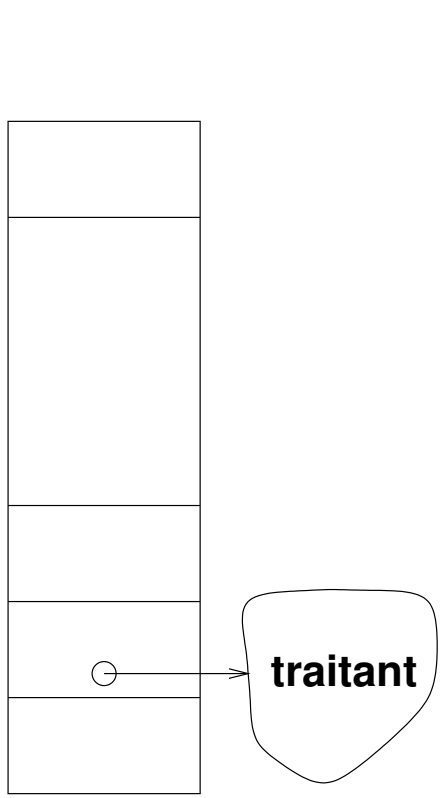
Pour créer une instance de la classe "condition", utiliser make-condition et non make-instance

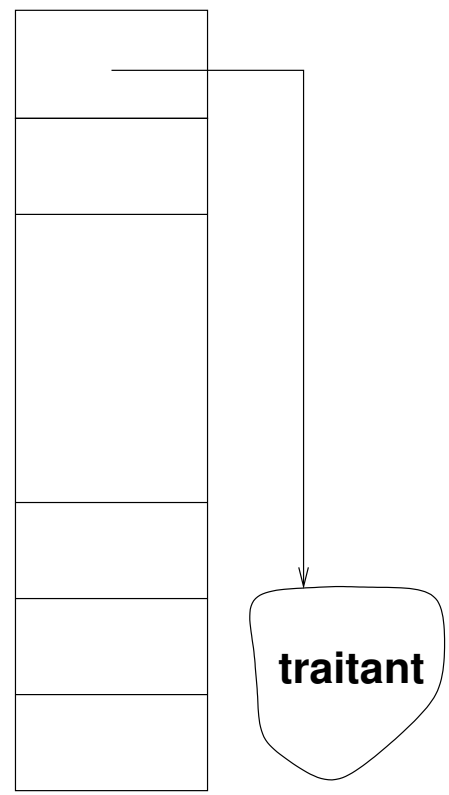
Plusieurs types de conditions existent: arithmetic-error, end-of-file, . . .

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

```
* (defun handle-division-conditions (condition)
    (format t "Considering condition for division~%")
    (when (and (typep condition 'arithmetic-error)
               (eq '/ (arithmetic-error-operation condition)))
      (invoke-debugger condition)))

HANDLE-DIVISION-CONDITIONS
* (defun handle-other-arithmetic-errors (condition)
    (format t "Considering condition for other arithmetic~%")
    (when (typep condition 'arithmetic-error)
      (abort)))

HANDLE-OTHER-ARITHMETIC-ERRORS
* (define-condition a-condition-with-no-handler (condition) ())

A-CONDITION-WITH-NO-HANDLER
* (signal 'a-condition-with-no-handler)

NIL
*
```

Conditions (suite)

```
* (handler-bind ((condition #'handle-division-conditions)
                 (condition #'handle-other-arithmetic-errors))
    (signal 'a-condition-with-no-handler))

Considering condition for division
Considering condition for other arithmetic
NIL
* (handler-bind ((arithmetic-error #'handle-division-conditions)
                 (arithmetic-error #'handle-other-arithmetic-errors))
    (signal 'arithmetic-error :operation '* :operands '(1.2. b)))

Considering condition for division
Considering condition for other arithmetic
*
```