# 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 transféré à la cible

# catch/throw

*catch* est utilisé pour établir une étiquette à utiliser par *throw* puis une suite de formes est évaluée, comme avec *progn*.

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

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*.

# catch/throw

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

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

G
* (g 234)

HI
345
*
```

# return-from

Similaire à `break` du langage C

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

234
*
```

# Situations exceptionnelles

Il y en a trois types: Les échecs
Ce sont des situations dues aux défauts de programmation.

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

Les autres
Ce sont des situations normales, prévues par le programmeur, mais dont le traitement nécessite une structure de contrôle particulière.

# Les échecs

Il n'y rien à faire.

Il faut arrêter l'exécution le plus vite possible.

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.

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

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

Il est souvent 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 comme *assert* destiné au programmeur).

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.

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

# Les autres

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

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 haut niveau qui possède la connaissance pour son traitement

Exemple (ouverture de fichier non existant):

Le problème est détecté par open (très bas niveau) fichier utilisateur c'est une situation de type "autre".

Si c'est un fichier d'installation c'est un défaut. Si c'est un

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

# Conditions

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

C'est donc un objet avec des créneaux.

Une condition est signalée à un *traitant* éventuel.

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

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

# 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*.

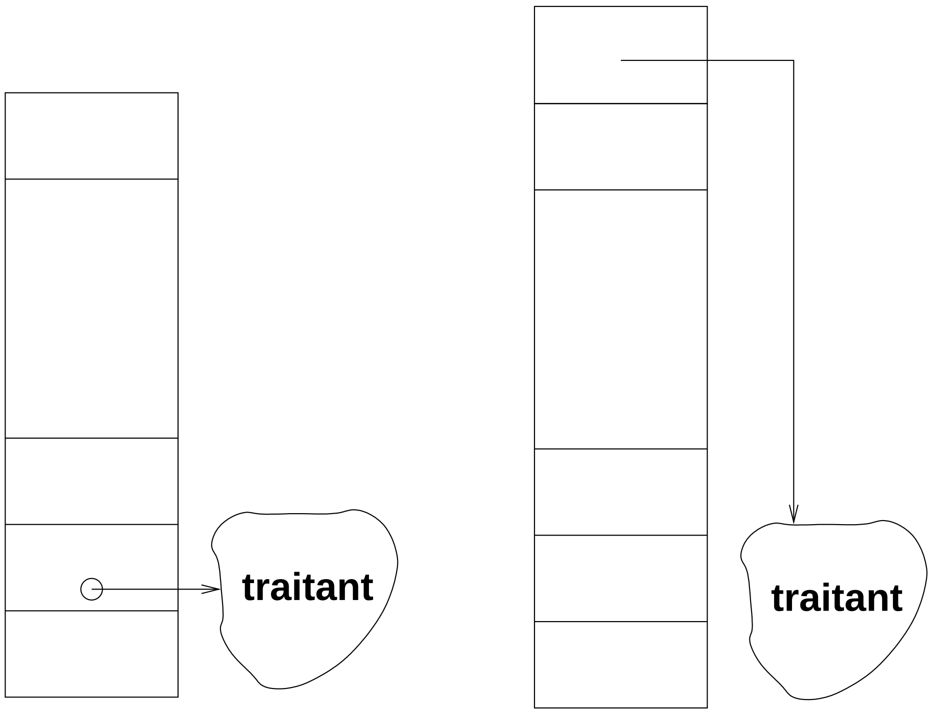
Plusieurs types de conditions existent: *arithmetic-error*, *end-of-file*, ...

# 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

# Conditions (suite)

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

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

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

5040
*
```

# Conditions (suite)

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

0
*
```