5 Itérations

En général, chaque forme d'itération classique (*for – to*, *while*) peut être exprimée en une récursion terminale.

LISP contient quand-même des commandes pour faire des itérations:

- dotimes cause une itération qui dépend d'un compteur, et
- dolist cause une itération qui dépend d'une liste dont chaque élement est inspecté.

Syntaxe de dotimes

```
(dotimes (<compteur> <forme valeur limite> <resultat>)
         <corps>)
```

Quand dotimes est appelé, l'expression <forme valeur limite> est évaluée et la valeur retournée est la limite supérieure du paramètre compteur. Si cette limite est n, les nombres 0 à $n-1$ sont attribués à <compteur> un après l'autre. Chaque fois, <corps> est évalué. A la fin, la liaison entre <compteur> est sa derniere valeur est supprimée et ensuite, <resultat> est évalué et sa valeur est retournée. <resultat> est optionnel, si dotimes n'a pas d'argument <resultat>, sa valeur est nil.

Exemple: fonction liste-n qui prend en nombre n comme argument et qui retourne la liste (1 2 ... n):

```
> (defun liste-n (n)
    (let ((resultat ()))
         (dotimes (count n resultat)
                  (setq resultat (append resultat (list (+ 1 count)))))))
LISTE-N
> (liste-n 4)
(1 2 3 4)
```

Récursion terminale correspondante:

```
(defun rec-liste-n (n &optional (count 0) (resultat ()))
    (if (eql n count)
        resultat
        (rec-liste-n n (+ count 1) (append resultat (list (+ 1 count))))))
```

dolist fait une itération en prenant tous les éléments d'une liste.

Syntaxe:

```
(dolist (<element> <forme liste> <resultat>)
        <corps>)
```

Quand dolist est appelé, l'expression <forme liste> est évaluée et la valeur retournée est la liste des éléments pour l'itération. Ensuite, ces éléments sont attribués à <element> un après l'autre. Chaque fois, <corps> est évalué. A la fin, la liaison entre <element> est sa derniere valeur est supprimée et ensuite, <resultat> est évalué et sa valeur est retournée. <resultat> est optionnel, si dolist n'a pas d'argument <resultat>, sa valeur est nil.

Exemple: Fonction atoms qui extrait tous les atomes d'une liste:

```
> (defun atoms (l)
```

```
    (let ((resultat ()))
         (dolist (el l resultat)
                 (if (atom el)
                     (setq resultat (append resultat (list el)))))))
ATOMS
> (atoms '(a (b) c))
(A C)
```

Récursion terminale correspondante:

```
(defun rec-atoms (l &optional (resultat ()))
       (cond ((null l) resultat)
             ((atom (car l))
              (rec-atoms (cdr l) (append resultat (list (car l)))))
             (T (rec-atoms (cdr l) resultat))))
```

Une commande plus générale que dotimes ou dolist est do. Syntaxe:

```
(do ((<par 1> <val 1>)
     ...
     (<par n> <val n>))
    (<teste d'arret> <action resultat>)
    <corps>)
```

Dans un appel de do, d'abord des liaisons temporaires entre <par 1> et <val 1>, ... <par n> et <val n> sont établies. Ensuite, tant que <teste d'arret> est nil, le corps est réitéré. Avant chaque évaluation du corps, la condition d'arrêt est testée. Quand elle n'est plus nil, la fonction s'arrête et retourne la valeur de <action resultat>.

Exemple:

```
(defun do-liste-n (n)
    (do ((resultat ())
         (count 0))
        ((= n count) resultat)
        (setq count (+ count 1))
        (setq resultat (append resultat (list count)))))

(defun do-atoms (l)
    (do ((liste l)
         (resultat ()))
        ((null liste) resultat)
        (if (atom (car liste))
            (setq resultat (append resultat (list (car liste)))))
        (setq liste (cdr liste))))
```

Une itération sans condition d'arrêt est loop. loop prend un seul argument, son corps. Ce corps est réitéré jusqu'à l'occurrence d'une commande return. Si loop s'arrete avec un return, la valeur de l'argument de ce return est rendu par loop.

Exemple:

```
(defun loop-atoms (l)
    (let ((liste l)
         (resultat ()))
```

```
    (loop
       (if (null liste) (return resultat))
       (if (atom (car liste))
          (setq resultat (append resultat (list (car liste)))))
       (setq liste (cdr liste)))))
```

Pour évaluer une suite d'expressions une après l'autre, on peut utiliser `prog1` ou `progn`. La première fonction retourne la valeur de la première expression, la deuxième retourne celle de la dernière. Souvent des expressions sont déjà évaluées une après l'autre (p.e. dans le corps de `defun`, `let`, `labels`, `do` etc.) et dans ce cas on n'a pas besoin de `progn` ou `prog1`. Mais par exemple chaque argument d'un `if` n'est qu'une seule expression. Ici `prog1` ou `progn` peuvent être utiles:

```
>(if (eq 'a (car '(a b)))
     (progn (setq new-list '(a b))
            (setq element (car new-list)))
     (cdr new-list))
A
>(if (eq 'a (car '(a b)))
     (prog1 (setq new-list '(a b))
            (setq element (car new-list)))
     (cdr new-list))
(A B)
```