`&args` und `&optional`

Um eine Funktion, die beliebig viele Argumente nimmt, zu definieren, benutzt man das Lambda-List-Schlüsselwort `&rest`. Um eine Funktion mit einem oder mehreren optionalen Parametern auszustatten, das Schlüsselwort `&optional`. Den optionalen Parametern kann man einen Wert zuweisen, falls die nicht angegeben wurden.

Beispiel:

```
(defun my-plus (&rest l-of-numbers)
  (apply #'+ l-of-numbers))

(defun ! (n &optional (res 1))
  (cond ((= n 0) res)
        (t (! (1- n) (* n res)))))
```


Letzte/Diese Vorlesung

Wiederholung Was – Wie — Letzte Woche

Bindungsumgebungen

Lambda-Funktionen

Destruktive Funktionen

Verwendung von Funktionsobjekten: `lambda` — Diese Woche

Mapping: `mapcar, maplist`

Iteration: `dolist, dotimes`

Mapping: `mapcar`

- Erhöhe jeden Wert der Liste um 1:

```
? (mapcar #'1+ '(3 4 5 42))
--> (4 5 6 43)
```

- Spiegele jede Teilliste:

```
? (mapcar #'reverse '((a b c d e) (1 2) (foo bar baz)))
--> ((E D C B A) (2 1) (BAZ BAR FOO))
```

- Berechne das Maximum jeder Teilliste:

```
? (mapcar #'(lambda (liste) (apply #'max liste))
          '((4 6 3) (3 4 6 3) (2 4 6 7) (3 42)))
--> (6 6 7 42)
```


Mapping

Mapping **bedeutet, eine** *Funktion* **mehrfach auf Elemente eines** *Objekts* **anzuwenden.**

- Wende die *Funktion* auf jedes Element einer *Liste* an.
  `mapcar`
- Wende die *Funktion* auf sukzessive `cdr` einer *Liste* an.
  `maplist`
- Wende die *Funktion* auf jedes Element einer *Hashtabelle* etc. an (später!)
  `maphash`

Mapping: `maplist`

- Einfaches Beispiel:
```
? (maplist #'print '(1 2 3 4)) -->
(1 2 3 4)
(2 3 4)
(3 4)
(4)
((1 2 3 4) (2 3 4) (3 4) (4))
```
- Berechne für eine Liste von täglichen Börsenkursen (aktuellster Wert *zuerst*), den jeweils bis dahin maximalen Kurs für jeden Zeitraum
```
? (setq kurse '(120.5 120 118.25 116 117.5 117))
? (maplist #'(lambda (list) (apply #'max list)) kurse)
--> (120.5 120 118.25 117.5 117.5 117)
```


Mapping: `mapcar`

`mapcar` nimmt aber beliebig viele Listen. Das Mapping hört auf wenn eine Liste leer ist.

- Mache neue Listen aus den Elementen der jeweiligen Teillisten:
```
? (mapcar #'list '(a b c d e) '(1 2) '(foo bar baz))
--> ((A 1 FOO) (B 2 BAR))
```
- Berechne das Maximum der jeweiligen Elemente jeder Teilliste:
```
? (mapcar #'max '(4 6 3) '(3 4 6 3) '(2 4 6 7) '(3 42))
--> (4 42)
```
- Konstruiere eine Liste mit Cons-Zellen aus den jeweiligen Elementen:
```
? (mapcar #'cons '(einz zwei drei vier) '(one two three four))
--> ((EINZ . ONE) (ZWEI . TWO) (DREI . THREE) (VIER . FOUR))
```

Even more mapping: `mapcan`

Falls man bestimmte elemente aus einer Liste, oder Bestimmte Ergebnisse von Funktionsapplikationen an den jeweiligen Elemente einer Liste sucht, kan man **mapcan** benutzen. Mapcan erwartet eine (möglicherweise leere) Liste als Rückgabewert von der Fuinktion, und die Listen werden (destruktiv) Konkateniert (mittels nconc).

Beispiel:

```
? (mapcan #'(lambda (x)
              (cond ((do-I-know-this-type? x)
                     (list (cdr x)))
                    (t nil)))
    '((unknown . #<...>) (number . 42) (cons . (nil . nil)) (atom . "atom")))
 --> (42 (nil . nil) "atom")
```


More mapping: `mapc, mapl`

- Wenn die Einzelergebnisse irrelevant sind:
- Neben `mapcar` gibt es `mapc`
- Neben `maplist` gibt es `mapl`
- `mapc` und `mapl` liefern das Argument unverändert zurück, interessant sind nur die „Seiteneffekte“. (Besser wäre es, „Nebeneffekte“ zu sagen!)

Nicht ganz Mapping: `reduce`

Eine mögliche Implementation

```
(defun my-reduce (f l)
  (cond ((not (rest l)) (first l))
        (t (my-reduce f
                      (cons (funcall f (first l) (second l))
                            (rest (rest l)))))))
```


Nicht ganz Mapping: `reduce`

Wir wollen eine Liste mit einer zwestelligen Funktion reduzieren – zB alle Zahlen in einer Liste addieren. Die Funktion dafür heisst `reduce`

Beispiel:

```
(reduce #'(lambda (x y) (+ x y)) '(1 2 3 4 5))
```

berechnet

```
(+ (+ (+ (+ (1 2)) 3) 4) 5)
-->
```

Nicht ganz Mapping: `dotimes`

`dotimes` führt eine Aktion wiederholt durch.
Die Zahl der Wiederholungen steht am Anfang fest.

- `(dotimes (` ⟨*Zähler*⟩ ⟨*Wiederholungen*⟩ [⟨*Rückgabewert*⟩] `)` ⟨*body*⟩ `)`
- `? (dotimes (i 4 42) (print i)) -->`

```
0  ;; Zählen beginnt bei Null!!
1
2
3
42 ;; Rückgabewert
```


Nicht ganz Mapping: `reduce`

Was berechnet ,,??''?

```
(defun ? (n)
  (cond ((= n 0) (list 1))
        (t (cons n (? (1- n))))))

(defun ?? (n)
  (reduce #'(lambda (x y)
              (* x y))
          (? n)))
```

Schon fast Mapping: `dolist`

`dolist` führt eine Aktion für jedes Element einer Liste durch.

- `(dolist (` ⟨*Element*⟩ ⟨*Liste*⟩ [⟨*Rückgabewert*⟩] `)` ⟨*body*⟩ `)`
- Berechne das Maximum jeder Teilliste:

```
(setq liste '((4 6 3) (3 4 6 3) (2 4 6 7) (3 42)))
(dolist (element liste)
  (print (apply #'max element)))
```

- Aber auch so geht's:

```
(mapcar #'(lambda (liste) (print (apply #'max liste))) liste)
```

- Manchmal ist `mapcar` besser, manchmal `dolist`.

Nicht ganz Mapping: `dotimes`

Die Zahl der Wiederholungen steht am Anfang fest!

-

```
(defun my-fakultaet (n)
  (let ((ergebnis 1))
    (dotimes (i n ergebnis)
      (setq ergebnis (* ergebnis (1+ i))))))
```

- Achtung: `(eql (my-fakultaet 7.01) (my-fakultaet 8))` !!
- Wie oft wird hier wiederholt?

```
(setq foo 42)
(dotimes (i foo) (setq foo 100) (print "Was ist die Antwort?"))
```

Quicksort mit selbstdefiniertem Split:

- Zur Erinnerung:

```
(defun qsort (list)
  (cond ((> (length list) 1)
         (append (qsort (split<= (first list)
                                 (rest list)))
                 (list (first list))
                 (qsort (split> (first list)
                                (rest list)))))
        (t list)))
```


Schon fast Mapping: `dolist`

- Manchmal ist `mapcar` besser, manchmal `dolist`.
- `mapcar` sammelt die Einzelergebnisse gleich in einer Liste auf, mit `dolist` ist dies umständlicher:

```
(setq liste '((4 6 3) (3 4 6 3) (2 4 6 7) (3 42)))
(let ((ergebnis))
  (dolist (element liste ergebnis)
    (setq ergebnis
          (append ergebnis (list (apply #'max element))))))
```

- Zur Erinnerung:

```
(mapcar #'(lambda (liste) (apply #'max liste)) liste)
```

Allgemeine Iteration: `do`, `do*`

Neben `do` und `do*` gibt es auch noch `loop`, das aber viele, viele Optionen hat. Zur Einführung also erstmal `do`:

- Ein Aufruf von `do` hat drei Komponenten:
  1. lokale Variablen und ihre Iteration
  2. Abbruchbedingung und Rückgabewert
  3. globale Aktion bei jeder Iteration

Splitfunktionen für Quicksort

- ...und nun die Splitfunktionen:

```
(defun split<= (pivot list)
  (let ((ergebnis nil))
    (dolist (element list ergebnis)
       (cond ((<= element pivot)
               (setq ergebnis (cons element ergebnis)))
             (T nil)))))
```

- ...aber in Common-Lisp ist so vieles schon vordefiniert:

```
(defun split<= (pivot list)
  (remove-if #'(lambda (n) (not (<= n pivot)))
             list))
```

Ein einfaches Beispiel: `do*`

- Gib jeden Wert einer Liste um 1 erhöht aus:

```
(defun alles-plus-eins (liste)
  (do* ((der-rest liste          (cdr der-rest))
        (element  (car der-rest) (car der-rest)))
     ((null der-rest))
    (print (1+ element))))
```

- Bei so einfachen Aufgaben reicht natürlich `dolist`:

```
(defun alles-plus-eins (liste)
  (dolist (element liste) (print (1+ element))))
```


Allgemeine Iteration: `do`, `do*`

- Formal ist `do` so definiert:
  `(do (` ⟨*lokale Variable mit Iteration*⟩* `)`
  `(` ⟨*Abbruchbedingung*⟩ ⟨*Rückgabe-sexpr*⟩* `)`
  ⟨*globale-sexpr*⟩* `)`
- Dabei ist:
  ⟨*lokale Variable mit Iteration*⟩ ::= `(`⟨*Variable*⟩ [⟨*Anfangswert*⟩ [⟨*Iterations-sexpr*⟩]] `)`
- Bei `do` werden die Variablen parallel gesetzt, bei `do*` sequentiell.
  Genau wie bei `let` und `let*`.

Ein komplexeres Beispiel: `do*`

- Simuliere `mapcar` (annähernd):

```
(defun my-mapcar (funktion liste)
  (do* ((der-rest  liste          (cdr der-rest))
        (das-erste (car der-rest) (car der-rest))
        (ergebnis  nil                           ))
      ((null der-rest)
       ergebnis)
     (setq ergebnis (append ergebnis
                            (list (funcall funktion das-erste))))))
```