# Letzte/Diese Vorlesung

**Lambda-Funktionen** — Letzte Woche

**Mapping:** `mapcar, maplist`

**Iteration:** `dolist, dotimes`

**Gleichheit:** `eq, eql, equal` — Diese Woche

**Iteration:** `do*, do`

**Iteration:** `loop`



# Manches ist gleicher als anderes: `eq, eql, equal`

**Neben `equal` kennen wir schon `=` und `string-equal`.**

- `eq` ist nur erfüllt, wenn die Objekte *identisch* sind. Bei einer Liste müssen es *dieselben* Conszellen sein und nicht nur *gleiche* Conszellen.
- `eql` ist wie `eq`, ist aber auch für alle Zahlen und Character erfüllt.
- `equal` prüft nur, ob die Struktur *gleich* ist. *Identität* ist nicht notwendig.

# Gleichheit mit `eq` und `equal`

```
? (eq 'a 'a)  -->  T
? (eq 1.0 1.0)  -->  NIL
? (eql 1.0 1.0)  -->  T
? (eq "zwei" "zwei")  -->  NIL
? (equal "42" "42")  -->  T
? (equal '(1 2 3) '(1 2 3)  -->  T
? (let* ((a "zwei")
         (b "zwei"))
     (eq a b))  -->  ???
? (let* ((a "zwei")
         (b a))
     (eq a b))  -->  ???
```

```
? (equal '(((42)) 42) (list '((42)) (1+ 41)))
? (let* ((a '(1 (2)))
         (b a)) (eq a b)) -->  T
? (let* ((a '(1 (2)))
         (b (rest a)))
     (eq b (last a))) -->  ???
```



# Allgemeine Iteration: `do`, `do*`

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

# Allgemeine Iteration: `do`, `do*`

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



# 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))))
```

# 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))))))
```



# Prädikate: `every()`, `some()` et al.

```
(every ⟨predicate⟩ ⟨list⟩+)
(some ⟨predicate⟩ ⟨list⟩+)
(notany ⟨predicate⟩ ⟨list⟩+)
(notevery ⟨predicate⟩ ⟨list⟩+)
```

```
(every #'oddp '(17 33 1 3 5)) → t
(every #'eq '(1 2 3) '(1 2 6)) → nil
(some #'oddp '(2 4 5)) → t
(some #'eq '(1 2 3) '(1 2 6)) → t
(notevery #'oddp '(17 33 1 3 5)) → nil
```

# Monster-Iteration: `loop`

Mit `loop` kann man alle bisher bekannten Iterationen ausführen, allerdings ist die Syntax nicht besonders LISPisch

```
(loop
   initially (print "Jetzt geht’s los!")            ;; Erste Klausel
   for i from 1 to (compute-top-value)              ;; Zweite Klausel
   while (not (unacceptable i))                     ;; Dritte Klausel
   collect (square i)                               ;; Vierte Klausel
   do (format t "Arbeite an ~d~%" i)                ;; Fuenfte Klausel
   when (evenp i)                                   ;; Sechste Klausel
     do collect i into odds
   finally (setq *odds* odds))                      ;; Siebte Klausel
```



# `loop` Beispiele

- `loop` statt `dotimes`:

```
(loop for ⟨var⟩ from 0 to (1- ⟨Initialisierung⟩) by 1
    do ⟨Body⟩
    finally (return ⟨Resultat-Form⟩)
    )
```

- `loop` statt `mapcar`:

```
(loop for ⟨var⟩ in ⟨Liste⟩
    collect ⟨form⟩
    )
```

oder `mapcan`:

```
(loop for ⟨var⟩ on ⟨Liste⟩
    collect ⟨form⟩
   )
```

- `loop` mal ganz anders:

```
(loop for i = 2  then (* i i)
    while (< i 42)
    sum i
    )
```

# Die verschiedenen `loop` Klauseln

**Variablen setzen und iterieren:** `for, as, with, repeat`

**Abbruchkriterien:** `while, until, always, never, thereis`

**Werteberechnung:** `collect, append, sum, count, maximize, ...`

**Unbedingte Ausführung:** `do`

**Bedingte Ausführung:** `if, else, when, unless`

**Sonstige:** `initially, finally`

Nur `do`, `initially` und `finally` dürfen mehrere Ausdrücke folgen.



# `loop`: Prolog, Body und Epilog

Die Verarbeitung von `loop` besteht aus drei Phasen:

**Prolog** Alle Variablen (eingeführt durch `for` oder `with`) werden in der angegebenen Reihenfolge initialisiert, dann werden nacheinander alle `initially` Formen ausgeführt

**Body** Hier werden die Werte iteriert (dabei wird eigener Iterationscode ausgewertet) und eventuelle Abbruchbedingungen überprüft

**Epilog** Zunächst werden alle `finally` Formen nacheinander ausgeführt, dann werden Rückgabewerte aus z.B. `collect` oder `always` Formen evtl. berechnet und zurückgegeben.

# `loop`: Vier verschiedene `for` Iterationen

- Arithmetisches Iterieren
  `for` ⟨*var*⟩ [ { `from` | `downfrom` | `upfrom` } ⟨*start*⟩ ]
  [ { `to` | `downto` | `upto` } ⟨*end*⟩ ]
  [ `by` ⟨*end*⟩ ]

- Iterieren über Listenelemente
  `for` ⟨*var*⟩ `in` ⟨*list-expr*⟩ [ `by` ⟨*step-fun*⟩ ]

- Iterieren über Listen
  `for` ⟨*var*⟩ `on` ⟨*list-expr*⟩ [ `by` ⟨*step-fun*⟩ ]

- "Allgemeines" Iterieren
  `for` ⟨*var*⟩ `=` ⟨*start*⟩ [ `then` ⟨*step-fun*⟩ ]



# `loop`: Abbruchtests

`while` ⟨*expr*⟩ Abbruch, wenn der Ausdruck `NIL` wird *(Solange* ⟨expr⟩ *gilt ...)*

`until` ⟨*expr*⟩ Abbruch, wenn der Ausdruck `T` wird *(Bis* ⟨expr⟩ *gilt ...)*.

`always` ⟨*expr*⟩ Abbruch, wenn der Ausdruck `NIL` wird (wie `while`) und gibt dann `NIL` zurück. Wird die Schleife beendet ohne daß der Ausdruck zu `NIL` evaluiert, gibt das `loop` `T` zurück.

`never` ⟨*expr*⟩ Analog zu `always`, nur daß NIL zurückgegeben wird, wenn der Ausdruck irgendwann zu `T` evaluiert und `NIL` sonst

`thereis` ⟨*expr*⟩ Evaluiert der Ausdruck mindestens einmal in der Schleife zu einem nicht-`NIL` Wert, wird dieser Wert zurückgegeben, `NIL` sonst

# `loop`: Werteberechnung

`collect` ⟨*expr*⟩ [ `into` ⟨*var*⟩ ] Sammele die Ergebnisse der Auswertungen des Ausdrucks in einer Liste auf

`append` ⟨*expr*⟩ [ `into` ⟨*var*⟩ ] Hänge die Ergebnisse des Ausdrucks wie mit `append` zusammen und gib das Resultat bei Beendigung von `loop` zurück. Die Ergebnisse müssen jeweils Listen sein!

`sum` ⟨*expr*⟩ [ `into` ⟨*var*⟩ ]

`maximize` ⟨*expr*⟩ [ `into` ⟨*var*⟩ ]

`minimize` ⟨*expr*⟩ [ `into` ⟨*var*⟩ ] Bilde resp. die Summe, das Maximum oder Minimum der numerischen Ergebnisse des Ausdrucks und gib sie bei Beendigung von `loop` zurück.

`count` ⟨*expr*⟩ [ `into` ⟨*var*⟩ ] Zähle, wie oft der Ausdruck einen Wert hatte, der nicht `NIL` war



# `loop`: Bedingte Ausführung

`if` ⟨*expr*⟩ ⟨*clause*⟩ { `and` ⟨*clause*⟩ }* 
[ `else` ⟨*clause*⟩ { `and` ⟨*clause*⟩ }* ] [ `end` ]
Führe die entsprechenden `loop` Klauseln im ersten Teil nur aus, wenn die Bedingung nicht `NIL` ergibt, die Klauseln im `else` Teil nur im umgekehrten Fall

`when` ⟨*expr*⟩ ⟨*clause*⟩ { `and` ⟨*clause*⟩ }* [ `end` ]

`unless` ⟨*expr*⟩ ⟨*clause*⟩ { `and` ⟨*clause*⟩ }* [ `end` ]

Warum braucht man diese Klauseln in `loop`? Man hat doch `(if ...)`

# loop ist (mindestens) so mächtig wie do*

```
(loop for i from 2 to (* 2 (length list)) by 2
    for element in list
    when (> i 0)
    collect i)

(loop for i in number-list
    if (oddp i)
      collect i into odds
    else
      collect i into evens
    end              ;; das ist optional und dient nur der Lesbarkeit
    finally (return (append odds evens)))
```



# Mehr loop Beispiele

```
(defun get-good-pizza (pizza-stand)
  (loop for pizza in (get-pizza-list pizza-stand)
      thereis (when (not (mit-anchovis-p pizza))
                 pizza)))

(loop for pizza-stand in (get-pizza-staende 'Saarbruecken)
    for gute-pizza = (get-good-pizza pizza-stand)
    then (get-good-pizza pizza-stand)
    when gute-pizza
      if (preis-ok-p gute-pizza)
        collect (list pizza-stand gute-pizza)
      else
        do (format T "Unverschaemt teuer: ~a" pizza-stand)
      end
    end
    )
```

# Pathologischere `loop` Beispiele

Sehr nett: Destructuring

```
(loop for (name adresse) in '((BERT SESAMSTRASSE) (KERMIT MUPPET-SHOW))
    do (format T "~a wohnt in der ~a~%"))
```

Eher gefährlich: Parallele Bindung wie in do (ohne *)

```
(loop for x from 1 to 3
    and y = nil then x
    collect (list x y))
```

⟶ `((1 NIL) (2 1) (3 2))`