Letzte/Diese Vorlesung

Letzte Woche:

- Kontrollstrukturen: `do do* dotimes dolist ...` (iteration)
- Glögg + “Schwedische” Kuche

Letzte/Diese Vorlesung

Heute:

Proschd Neijohr!

Kontrollstrukturen:

- Funktionsobjekte
- Higher order functions
- Mapping

Funktionsobjekte

Die Funktion **defun** erzeugt ein Funktionsobjekt und legt diese in der Funktionszelle des durch den Funktionsnamen bezeichneten Symbols ab. Über den Namen des Symbols kann dann die durch die Definition spezifizierte Funktion angesprochen werden.

Naives Modell eines LISP-Symbols:

```
foo  ──────┬─► value:    NIL
           ├─► function: ???
           ├─► plist:    (:USED NIL :DOC "by JR")
           ├─► name:     "FOO"
           └─► package:  #:<"USER" package>
```

Frage: Wie sieht ein solches Funktionsobjekt aus ?

Funktionsobjekte Die Repräsentation von Funktionsobjekten in LISP basiert auf der *Lambda-Notation* (vgl. Church41).

Lambda-Abstraktion – Ein *Lambda-Ausdruck*, wie z.B. `(lambda (x) (+ x 1))` steht für die Funktion `f` mit `f(x) = x + 1`. Wichtig hierbei ist, daß kein Name für die Funktion benötigt wird, d.h. Lambda-Ausdrücke beschreiben *anonyme* Funktionen.

Lambda-Applikation – Ein Lambda-Ausdruck beschreibt eine Funktion, ist aber selbst kein direkt auswertbarer Ausdruck. Erst durch die zusätzliche Angabe von Argumenten entsteht ein auswertbarer Funktionsaufruf. Bei einer solchen *Applikation* eines Lambda-Ausdrucks werden die formalen Parameter durch die konkreten Argumente, d.h. die Aufrufparameter, ersetzt und die angegebene Berechnungsvorschrift angewendet.

Funktionsobjekte – Syntax der Lambda-Ausdrücke

Die Syntax eines Lambda-Ausdruck entspricht weitgehend der schon bekannten Syntax von **defun**.

(lambda *Parameterliste Funktionskörper***)**

Die Parameterliste spezifiziert dabei die formalen Parameter, die dann im Funktionskörper auftreten können. Der Funktionskörper besteht wiederum aus einer Folge symbolischer Ausdrücken, wobei der Wert der Funktion durch den zuletzt ausgewerteten Ausdruck bestimmt wird.

Beispiel:

```
(lambda (x y) (+ x y))

(lambda (l)
  (sum l))
```


Funktionsobjekte – Funktionsaufruf

Um einen lambda-Ausdrück verwenden zu können, muß man ihn in ein *Funktionsobjekt* umwandeln. Das wird mit Hilfe der *special form* `function` gemacht

Beispiel:

```
(function (lambda (x) (* x x)))
-->  #<Interpreted Function (unnamed) @ #x1c8237a>
```

Für `function` gibt es (analog wie für `quote`) ein Readermakro `#'`. Eine äquivalente Schreibweise für das Beispiel oben ist also: `#'(lambda (x) (* x x))`

Funktionsobjekte – Funktionsaufruf

Mit Hilfe der Funktionen **funcall** und **apply** können die durch Lambda-Ausdrücke definierten Funktionen auch explizit angewendet werden. Dabei wendet **funcall** die durch den Wert des ersten Argumentes bezeichnete Funktion auf die Werte der restlichen Argumente an. Im Gegensatz dazu erwartet **apply** als letztes Argument immer eine Liste mit den (restlichen) Funktionsargumenten.

Beispiel:

```
(setq fu #'(lambda (x y) (list x y x)))

(funcall fu 'a 'h)  -->  (A H A)
(apply fu (list 'a 'h))  -->  (A H A)
(apply fu 'a '(h))  -->  (A H A)
(apply fu 'a 'h '()))  -->  (A H A)
```


Funktionsobjekte – Anwendung anonymer Funktionen

Die direkte Verwendung von Lambda-Ausdrücken, d.h. der Gebrauch anonymer Funktionen, bietet sich oft in solchen Fällen an, in denen eine bestimmte Berechnungsvorschrift nur lokal an einer Stelle benötigt wird.

```
(let ((zahlen '(33 6 27 44 79 72 94 51 91 93)))
   (sort zahlen #'(lambda (x y) (cond ((oddp x) t)
                                      ((oddp y) nil)
                                      (t t)))))
--> (93 91 51 79 27 33 94 72 44 6)
```

Mapping

Eine besondere Art von Iteration stellt die sukzessive Anwendung einer Funktion auf Teile einer oder mehrerer Listen dar. In LISP stehen zur Realisierung dieses sogenannten *Mapping* besondere Konstrukte zur Verfügung. Sie erwarten jeweils als erstes Argument eine Funktion und als weitere Argumente mindestens eine abzuarbeitende Liste.

MAP-Funktionen im Überblick

	Jedes Element	Restliste
Ergebnis **nil**	**mapc**	**mapl**
Ergebnis ist Liste der Resultate	**mapcar**	**maplist**
Resultate werden mit **nconc** aneinandergehängt	**mapcan**	**mapcon**

Mapping – Beispiele

Die allgemeine Syntax für die map-Funktionen lautet:

(map{c|l|...|con} *function list** **)**

```
(mapcar #'reverse '((a) (b c) (d)))
   --> ((A) (C B) (D))
(mapcan #'reverse  '((a) (b c) (d)))
   --> (A C B D)
(maplist #'reverse '((a) (b c) (d)))
   --> (((D) (B C) (A)) ((D) (B C)) ((D)))
(mapcon #'reverse '((a) (b c)  (d)))
   --> ((D) (B C) (A) (D) (B C) (D))
```

Mapping – Beispiele

Man kan auch beliebig viele Listen als Argument geben:

```
(mapcar #'list '(1 2 3) '(a b c))
   --> ((1 A) (2 B) (3 C))
(mapcan #'list '(1 2 3) '(a b c))
   --> (1 A 2 B 3 C)
(maplist #'list '(1 2 3) '(a b c))
   --> (((1 2 3) (A B C)) ((2 3) (B C)) ((3) (C)))
(mapcon #'list '(1 2 3) '(a b c))
   --> ((1 2 3) (A B C) (2 3) (B C) (3) (C))
```


Mapping – Beispiele

`mapc` und `mapl` underscheidet sich von `mapcar` und `maplist` dadürch, daß sie die Ergebnisse ihre Funktionen nicht akkumulieren.

Mapping und Rekursion:

```
(defun reverse-all (x)
  (cond ((atom x) x)
        (t (reverse (mapcar #'reverse-all x)))))

(defun sumlist (list)
  (cond ((null list)
         0)
        (t (+ (first list) (sumlist (rest list))))))

(defun sumls (list-of-integers)
  (cond ((atom list-of-integers)
         list-of-integers)
        (t (sumlist (mapcar #'sumls list-of-integers)))))
```


Kontrollstrukturen – Iteration

```
(defun count-atoms (l)
  (do* ((stack l)
        (head (first stack) (first stack))
        (n 0))
       ((null stack) (format t "~%; ~R atoms in ~A" n l) n)
    (print stack)
    (cond ((null head) (pop stack))
          ((atom head) (incf n) (pop stack))
          (t (pop stack)
             (push (rest head) stack)
             (push (first head) stack)))))

(count-atoms '(a (b) (c d)))
 -->> (A (B) (C D)) -->> ((B) (C D)) -->> (B NIL (C D)) -->> (NIL (C D))
 -->> ((C D))  -->> (C (D))  -->> ((D))  -->> (D NIL)  -->>  (NIL)
 -->>  ; four atoms in (A (B) ((C D)))
 -->  4
```

Kontrollstrukturen – Iteration

```
;; Besser?

(defun count-atoms (l)
  (cond ((null l) 0)
        ((atom l) 1)
        (t (+ (count-atoms (first l))
              (count-atoms (rest l))))))

;; Noch besser?

(defun count-atoms (l)
   (cond ((null l) 0)
         ((atom l) 1)
         (t (apply #'+ (mapcar #'count-atoms l)))))
```


Kontrollstrukturen – Iteration

```
(defun flatten (l)
  (cond ((null l) nil)
        ((atom l) (list l))
        (t (append (flatten (first l))
                   (flatten (rest l))))))

;; Besser?

(defun flatten2 (l)
  (cond ((null l) nil)
        ((atom l) (list l))
        (t (mapcan #'flatten2 l))))
```