

Letzte/Diese Vorlesung

Letzte Woche:

- Listen als Mengen – `adjoin`, `member`, ...
- Eigene Funktionen definieren – (`defun` ($\langle param \rangle^*$) $\langle form \rangle^*$)
- Kontrollstrukturen
 - Sequenzen
 - Bedingungen – `cond`
 - Rekursion



Letzte/Diese Vorlesung

Heute:

Blöcke/Seiteneffekte

Rekursion II

Programmierstil

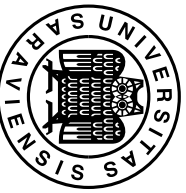


Blöcke / Seiteneffekte

Beispiel für Blöcke: Funktionskörper und *cond* Klauseln

```
(defun foo (x) (cond (if (= i 0)
                        (bar x)
                        ((= i 0)
                         (progn
                          (warn "i ist Null!")
                          i)
                         (cond ((> i 0) (- i 1)))
                         )
                        )
                    )
  )
```

Diese Blöcke sind implizite **progn** Formen. Syntax: (**progn** *<form>**). Der Rückgabewert ist jeweils der Wert der letzten ausgewerteten Form. Die anderen Formen in einem Block können nur *Seiteneffekte* bewirken.

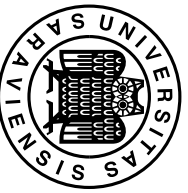


Blöcke / Seiteneffekte II

```
(defun deep-thought (number)
  (cond ((>= number 0) (* number number))
        ((< number 0) (- (* number number))))
        ((= number 42)
         (setq *THE-ANSWER* 42) (print "Hurra!"))))

(defun deep-thought (number)
  (cond ((= number 42)
         (setq *THE-ANSWER* 42) (print "Hurra!")))
        ((>= number 0) (* number number))
        ((< number 0) (- (* number number)))))
```

Ohne die Seiteneffekte sind zwei aufeinanderfolgende **cond** Formen in einem Block wirkungslos.



Restrekursion: Vergleich

nicht restrekursiv

```
(defun ! (i)
  (cond ((= i 0) 1)
        (t (* i (! (1- i))))))
```

restrekursiv

```
(defun ! (i)
  (help i 1))
(defun help (i n)
  (cond ((= i 0) n)
        (t (help (1- i)
                  (* i n)))))
```

- das Ergebnis des rekursiven Aufrufs geht in weitere Berechnung(en) ein;

- der rekursive Aufruf ist gleichzeitig der Rückgabewert der Funktion.



Rekursion – Beispiel (Restrekursion)

Linear

```
? (! 3)
(* 3 (! 2))
(* 3 (* 2 (! 1)))
(* 3 (* 2 (* 1 (! 0))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
6
```

Iterativ

```
? (! 3)
(!-help 3 1)
(!-help 2 1)
(!-help 1 2)
(!-help 0 6)
6
```

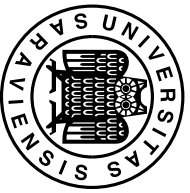


Mehrfache Rekursion

- **Aufgabe:** Sortiere eine Liste von Zahlen in aufsteigender Reihenfolge
- **Idee:** Füge das erste Element in den sortierten Rest der Liste ein.
- **Teilproblem:** Einfügen eines Elements in eine sortierte Liste

```
(defun insert-sort (list)
  (cond ((> (length list) 1)
        (insert (first list) (insert-sort (rest list))))
        (t list)))
```

```
(defun insert (element sortierte-liste)
  (cond ((null sortierte-liste) (list element))
        ((< element (first sortierte-liste))
         (cons element sortierte-liste))
        (T
         (cons (first sortierte-liste)
               (insert element (rest sortierte-liste))))))
```



Einführungskurs Common-Lisp (65)

Common-Lisp — 11 Nov 99 (janal/becker@dfki.de)

Mehrfache Rekursion

- **Aufgabe:** Sortiere eine Liste von Zahlen in aufsteigender Reihenfolge
- **Idee:** Teile die Liste in einen Teil mit kleinen Zahlen und einen Teil mit großen Zahlen, sortiere die Teillisten und hänge sie wieder zusammen

```
(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)))
```



Verschachtelte (doppelte) Rekursion

- **Aufgabe:** Erweitere die Funktion `spiegel()` (aka `reverse()`) so, daß auch eingebettete Listen gespiegelt werden.

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



Gute vs. Schlechte Programmierstile – Einrückung

- Wozu Einrückung? Einrückung macht Programme für Menschen lesbarer
 - Anfang und Ende eines Blocks/einer Funktion sind unmittelbar zu erkennen
 - Zusammengehörende Teile sind auf einer Ebene beieinander
 - für die meisten Lisp-Konstrukte gibt es eine oder mehrere allgemein akzeptierte Einrückungen, aber es gibt nicht *die* Einrückung
 - Manche Editoren (z.B. Emacs) erleichtern das Einrücken mit automatischen Funktionen



Gute vs. Schlechte Programmierstile – Einrückung

- defun – ! zweimal schlecht eingerückt:

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

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

- defun – ! besser eingerückt:

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



Gute vs. Schlechte Programmierstile – Namen

- Der Name einer Funktion oder einer Variablen sollte den Inhalt und Zweck dieser Variablen möglichst gut beschreiben
- Man sollte bei Programmen von kryptischen Namen wie `fumble`, `bar`, `foo` absehen
- Eine Liste heißt `l`, `list`, `liste` aber nie `x`, `y`, `...`
- Ein Integer heißt `i`, `j`, `k`, `n`, `m`, `...` aber nie `list`, `x` (!?)
- Eine Fließkommazahl heißt z.B. `x`, `y`, `z` aber nie `n`, `liste`, `...`
- Ein Atom heißt z.B. `atom`
- Eine beliebige Struktur heißt z.B. `obj`, `object`, `objekt`, `ding`, `thing`, `...` aber nie `l`, `n`, `...` (möglicherweise `x`).
- Eine temporäre Variable heißt z.B. `tmp`, `temp`, `...`



Gute vs. Schlechte Programmierstile – Kommentare

- Kommentare sind, vor allem bei komplexeren Programmen, unerlässlich.
- Je trickreicher die Programmierung, desto besser und klarer muß der Kommentar sein
- Faustregel: 30% Kommentare sind gut, fifty-fifty ist besser
- Nicht vergessen: Kommentare sind auch (schon nach kurzer Zeit) auch für das eigene Verstaendnis des Programms wichtig

