Letzte/Diese Vorlesung

Bindungsumgebungen	Letzte Woche
Lambda-Funktionen	

Verwendung von Funktionsobjekten: `lambda`	Diese Woche
Programmierstil: Indentierung, Variablennamen	

Gute vs. Schlechte Programmierstile – Indentierung

Indentierung ist ein tolles Konzept, um Programme lesbar(er) für uns Menchen zu machen.

- Was ist Indentierung?
  - Einrückung und Formatierung von Programmen, um die Lesbarkeit zu steigern.
- Es gibt für fast jede Form in Common-Lisp eine allgemein akzeptierte Art, den Ausdrücken ein "schönes" Aussehen zu geben.
- Es gibt nicht die Art ein Programm zu indentieren — es gibt aber ein paar allgemeine Regeln.

Gute vs. Schlechte Programmierstile – Beispiele:

- `defun` – `!` zweimal schlecht indentiert:

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

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

- `defun` – `!` besser indentiert:

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


Gute vs. Schlechte Programmierstile – Beispiele:

Ein paar Beispiele aus alten Übungen ...

- `pair-up` (Lösung von NN)

```
(defun pair-up (L) (cond  ((or (null L)  (not (rest L)))  nil)
(T  (cons (list (first L) (second L))  (pair-up (rest (rest L)))))))
```

- `pair-up` (Lösung von NN)

```
(defun pair-up (liste)
      (cond ((> (length liste) 1)
            (append (list(list (first liste) (second liste))) (pair-up (rest (rest l
```

Gute vs. schlechte Programmierstile – Beispiele:

- `pair-up` besser?

```
(DEFUN PAIR-UP (L)
  (COND ((OR (NULL L) (NOT (REST L))) NIL)
        (T (CONS (LIST (FIRST L) (SECOND L)) (PAIR-UP (REST (REST L)))))))
```

- `pair-up` Noch besser?

```
(defun pair-up (liste)
  (cond ((or (null liste)
             (not (rest liste)))
          nil)
        (t
          (cons (list (first liste)
                      (second liste))
                (pair-up (rest (rest liste)))))))
```


Gute vs. schlechte Programmierstile – Beispiele:

Namen für formale Parameter – ein paar Tips:

- Eine Liste heißt: `l, list, liste` aber nie `x, y, ....`
- Ein Integer heißt `i, j, k, n, m, ...` aber nie `list, x` (l?)
- 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, ...`

Letzte Woche

Funktionen als Argumente:

- Wenn die Zahl der Argumente bekannt ist:

```
? (funcall #'(lambda (liste) (1+ (length liste))) '(1 2 3))
--> 4
```

- Wenn die Zahl der Argumente unklar ist:

```
? (apply #'(lambda (a b c d) (cons (list a b) (list c d)))
         '(17 42 4711 8x4))
--> ((17 42) 4711 8x4)
```

- `apply` erwartet eine *Funktion*, dann beliebig viele *Argumente* und dann eine *Liste mit restlichen Argumenten*:

```
? (apply #'(lambda (a b c d) (list a b c d))
         17 42 '(4711 8x4))
--> (17 42 4711 8x4)
```


Verwendung von Funktionsobjekten:

Sortieren verschiedener Datentypen mit verschiedenen *Vergleichsfunktionen*.

- Zahlen werden mit `<` verglichen.
- Strings werden mit `string<` oder `string-lessp` verglichen.
- ... oder der Länge nach sortiert mit (geschickter Verwendung von) `length`
- solche Vergleichsfunktionen werden auch (zweistellige) *Prädikate* genannt.

Verwendung von Funktionsobjekten:

#'<*symbol*> ist ein *Reader-Macro* und liefert die *Funktion*, die an das <*symbol*> gebunden ist.

- Sortieren von Zahlen:
```
? (sort '(42 17 4711) #'(lambda (n m) (< n m))) --> (17 42 4711)
```
- ...oder einfacher:
```
? (sort '(42 17 4711) #'<) --> (17 42 4711)
```
- Sortieren von Strings:
```
? (sort '("jabba" "the" "hutt") #'string< )
--> ("hutt" "jabba" "the")
```


Weitersortieren:

- Sortieren mit Groß- und Kleinschreibung:
```
? (sort '("Jabba" "the" "fat" "Hutt") #'string< )
--> ("Hutt" "Jabba" "fat" "the")
? (sort '("Jabba" "the" "fat" "Hutt") #'string-lessp )
--> ("fat" "Hutt" "Jabba" "the")
```
- Sortieren der Länge nach:
```
? (sort '("Jabba" "the" "fat" "Hutt") #'length )
--> ("Hutt" "fat" "the" "Jabba")
```

  Wieso denn das?

Verwendung von Lambda-Ausdrücken:

Mit `lambda` werden namenlose Funktionen definiert, ähnlich wie mit `defun`.

- Nochmal: Sortieren der Länge nach:

```
? (sort '("Jabba" "the" "fat" "Hutt")
        #'(lambda (a b)
            (< (length a)
               (length b))))

--> ("the" "fat" "Hutt" "Jabba")
```


Prädikate für Strings und Zahlen:

Alles auf einen Blick:

Zahlen	Strings mit und	ohne Groß/Klein-Unterscheidung
`<`	`string<`	`string-lessp`
`>`	`string>`	`string-greapterp`
`<=`	`string<=`	`string-not-greaterp`
`>=`	`string>=`	`string-not-lessp`
`=`	`string=`	`string-equal`
`/=`	`string/=`	`string-not-equal`

...und dann noch `eq, eql, equal`.

Destruktive Operationen

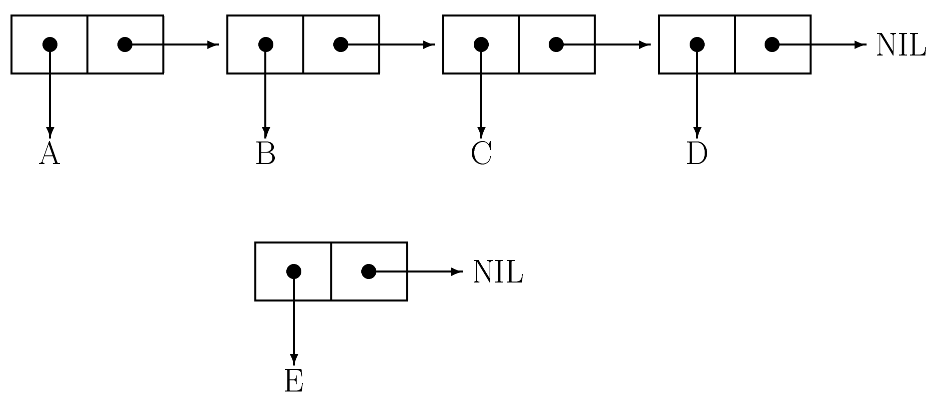
Eine **destruktive Operation** verändert den Wert einer Datenstruktur oder eines Teils der Datenstruktur auf Dauer. Die grundlegende destruktive Operation in Lisp heißt `setf`. Syntax:

`(setf` $\{ \langle \textit{Zugriffsfunktion} \rangle \; \langle \textit{Wert} \rangle \}^*$ `)`

```
(setf (first '(a b c d)) 'e) ⟶ E  ...  (E B C D)
(setf (rest '(a b c d)) '(e)) ⟶ (E)  ...  (A E)
(setf (symbol-function 'foo) #'< ⟶ #'<
(setf foo 42)) ⟶ 42
(foo 2 3) ⟶ ?
```


Wirkung von `setf`

Was passiert bei `(setf (rest '(a b c d)) '(e))` ?



Aus `(a . (b c d))` wird `(a . (e))`, respektive `(a e)`

Andere destruktive Funktionen

Es gibt von vielen Funktionen in Lisp auch destruktive Versionen, die man (auch) benutzen kann, um den Speicherplatzverbrauch zu reduzieren.
Bei Datenstrukturen wie Suchbäumen ist aber die destruktive Funktionalität explizit gewünscht.

- `nconc` arbeitet wie `append`, benutzt aber die Ausgangslisten.
  `(setf (rest (last l1)) l2)` entspricht `(nconc l1 l2)`
- `delete` arbeitet wie `remove`, benutzt aber die `cons` Zellen aus der Originalliste
- `push` und `pop` sind destruktive Funktionen, mit denen man eine Liste als Stack benutzen kann.
- `sort` ist auch eine destruktive Funktion!

Weitere Destruktive Funktionen

Es gibt einige Funktionen, die in einer destruktiven und einer nicht destruktiven Version existieren.
Konvention: Die Namen der destruktive Funktionen fangen alle (?) mit “`n`” an.

Beispiele:

```
reverse -- nreverse
substitute -- nsubstitute
append -- nconc
union -- nunion
...
```

Destruktive Funktionen

Aufpassen: Sort ist eine destruktive Funktion...

```
[1] USER(81): (setq l
               (let ((res nil))
                 (dotimes (x 10)
                    (push (random 100) res)) res))
(89 3 71 0 22 43 17 90 14 14)
[1] USER(82): (sort l #'>)
(90 89 71 43 22 17 14 14 3 0)
[1] USER(83): l
(89 71 43 22 17 14 14 3 0)
```