

## 8 Prozeduren als Daten

- Prozeduren als Parameter
  - Prozeduren als Ergebnisse
- ⇒ *Prozeduren höherer Ordnung (higher-order procedures)*
- ⇒ *Programmierung höherer Ordnung*
- Verwendung als Abstraktionsmittel

## 8.1 Prozeduren als Parameter

### 8.1.1 Beispiel: Gästeliste

```
; Ein Gast ist ein Wert
; (make-guest name sex veggie)
; wobei name      : string
;       sex       : boolean (#t für männlich, #f für weiblich)
;       veggie    : boolean
(define-record-procedures guest
  make-guest guest?
  (guest-name guest-male? guest-vegetarian?))
(: make-guest (string boolean boolean -> guest))
```

## 8.1.2 Vegetarier suchen

; aus einer Gästeliste die Liste der Vegetarier extrahieren

```
(: guests-vegetarian ((list guest) -> (list guest)))
```

```
(define guests-vegetarian
```

```
  (lambda (guests)
```

```
    (cond
```

```
      ((empty? guests)
```

```
        empty)
```

```
      ((pair? guests)
```

```
        (let* ((guest (first guests))
```

```
               (result (guests-vegetarian (rest guests))))
```

```
          (if (guest-vegetarian? guest)
```

```
              (make-pair guest result)
```

```
              result))))))
```

### 8.1.3 Männer suchen

; aus einer Gästeliste die Liste der Männer extrahieren

```
(: guests-male ((list guest) -> (list guest)))  
(define guests-male  
  (lambda (guests)  
    (cond  
      ((empty? guests)  
       empty)  
      ((pair? guests)  
       (let* ((guest (first guests))  
              (result (guests-male (rest guests))))  
         (if (guest-male? guest)  
             (make-pair guest result)  
             result))))))
```

### 8.1.4 Nach einem Prädikat suchen

```
; aus einer Gästeliste die Liste der Gäste extrahieren,  
; die ein Prädikat erfüllen  
(: guests-filter ((guest -> boolean) (list guest) -> (list guest)))  
(define guests-filter  
  (lambda (pred? guests)  
    (cond  
      ((empty? guests)  
       empty)  
      ((pair? guests)  
       (let* ((guest (first guests))  
              (result (guests-filter (rest guests))))  
         (if (pred? guest)  
             (make-pair guest result)  
             result))))))
```

### 8.1.5 Verwendung von guests-filter

; aus einer Gästeliste die Liste der Vegetarier extrahieren

```
(: guests-vegetarian ((list guest) -> (list guest)))
```

```
(define guests-vegetarian
```

```
  (lambda (guests)
```

```
    (guests-filter guest-vegetarian? guests)))
```

; aus einer Gästeliste die Liste der Männer extrahieren

```
(: guests-male ((list guest) -> (list guest)))
```

```
(define guests-male
```

```
  (lambda (guests)
```

```
    (guests-filter guest-male? guests)))
```

### 8.1.6 filter als polymorphe Prozedur

```
; aus einer Liste die Elemente extrahieren,  
; die ein Prädikat erfüllen  
(: filter ((%a -> boolean) (list %a) -> (list %a)))  
(define filter  
  (lambda (pred? xs)  
    (cond  
      ((empty? xs)  
       empty)  
      ((pair? xs)  
       (let* ((x (first xs))  
              (result (filter pred? (rest xs))))  
         (if (pred? x)  
             (make-pair x result)  
             result))))))
```

### 8.1.7 `filter` ist Prozedur höherer Ordnung

- `filter` *abstrahiert* vom Muster der Prozeduren `guests-vegetarian` und `guests-male`

#### Vorteile von solcherart abstrahierten Prozeduren höherer Ordnung

- Verkürzung des Programms
- Verbesserung der Lesbarkeit des Programms
- Verbesserung der Wartbarkeit des Programms
  - Ein Fehler in `filter` muss nur einmal korrigiert werden.
  - Wenn `filter` korrekt ist, dann sind auch alle Anwendungen von `filter` korrekt.

# MANTRA

## Mantra #11 (Abstraktion aus Mustern)

Wenn mehrere Prozeduren bis auf wenige Stellen gleich aussehen, dann schreibe eine allgemeinere Prozedur, die über diese Stellen abstrahiert. Ersetze die ursprünglichen Prozeduren durch Anwendungen der neuen, allgemeineren Prozedur.

## 8.2 Listen falten

Vielen Listenoperationen liegt eine einzige gemeinsame Abstraktion zugrunde:  
Das Falten einer Liste.

## 8.2.1 Beispiel: Elemente einer Liste aufsummieren

```
; Elemente einer Liste aufsummieren
(: list-sum ((list number) -> number))
(define list-sum
  (lambda (l)
    (cond
      ((empty? l)
       0)
      ((pair? l)
       (+ (first l) (list-sum (rest l)))))))
```

## 8.2.2 Beispiel: Elemente einer Liste aufmultiplizieren

```
; Elemente einer Liste aufmultiplizieren
(: list-product ((list number) -> number))
(define list-product
  (lambda (l)
    (cond
      ((empty? l)
       1)
      ((pair? l)
       (* (first l) (list-product (rest l)))))))
```

### 8.2.3 Abstraktion des Musters liefert list-fold

; Elemente einer Liste auffalten

```
(: list-fold (number (number number -> number) (list number) -> number))
```

```
(define list-fold
```

```
  (lambda (e f l)
```

```
    (cond
```

```
      ((empty? l)
```

```
        e)
```

```
      ((pair? l)
```

```
        (f (first l) (list-fold e f (rest l))))))
```

e : Startwert für leere Liste

f : Kombinationsfunktion für erstes Listenelement und rekursiven Aufruf auf der Restliste

## 8.2.4 Verwendung von list-fold

; Elemente einer Liste aufsummieren

```
(: list-sum ((list number) -> number))
```

```
(define list-sum
```

```
  (lambda (xs)
```

```
    (list-fold 0 + xs)))
```

; Elemente einer Liste aufmultiplizieren

```
(: list-product ((list number) -> number))
```

```
(define list-product
```

```
  (lambda (xs)
```

```
    (list-fold 1 * xs)))
```

## 8.2.5 list-fold ersetzt die Listenkonstruktoren

`(list-fold e f l)` ersetzt

- jedes Vorkommen von `empty` in `l` durch `e` und
- jedes Vorkommen von `make-pair` durch `f`

und wertet den entstehenden Ausdruck aus:

```
(list-fold e f
  (make-pair x1 (make-pair x2 ... (make-pair xn empty))))
===
(f      x1 (f      x2 ... (f      xn e)))
```

## 8.2.6 Polymorpher Vertrag für list-fold

```
; Elemente einer Liste auffalten  
(: list-fold (%a (%b %a -> %a) (list %b) -> %a))  
(define list-fold ...)
```

Zum Beispiel kann cat (Listenkonkatenation) mit list-fold implementiert werden:

```
; Listen verketteten  
(: cat ((list %a) (list %a) -> (list %a)))  
(define cat  
  (lambda (xs ys)  
    (list-fold ys make-pair xs)))
```

## 8.3 Anonyme Prozeduren

### 8.3.1 Listenlänge

```
; Länge einer Liste  
(: list-length ((list %a) -> natural))  
(define list-length  
  (lambda (l)  
    (cond  
      ((empty? l) 0)  
      ((pair? l) (+ 1 (list-length (rest l)))))))
```

- Aufgabe: Schreibe `list-length` mit Hilfe von `fold`

### 8.3.2 Listenlänge als Faltung

Passende Kombinationsfunktion:

```
; Hilfsfunktion zur Definition von length mit Hilfe von list-fold
```

```
(: add-1-for-length (%a natural -> natural))
```

```
(define add-1-for-length
```

```
  (lambda (ignore n)
```

```
    (+ 1 n)))
```

```
; Länge einer Liste
```

```
(: my-length ((list %a) -> natural))
```

```
(define my-length
```

```
  (lambda (xs)
```

```
    (list-fold 0 add-1-for-length xs)))
```

- add-1-for-length hat nur eine Verwendung im Programm

⇒ es lohnt nicht, sie zu benennen!

### 8.3.3 Listenlänge als Faltung mit anonymer Prozedur

```
; Länge einer Liste  
(: my-length ((list A) -> natural))  
(define my-length  
  (lambda (xs)  
    (list-fold 0 (lambda (ignore n) (+ 1 n)) xs)))
```

### 8.3.4 Filtern als Faltung

```
(define filter
  (lambda (pred? xs)
    (list-fold empty
              (lambda (elem result)
                (if (pred? elem)
                    (make-pair elem result)
                    result))
              xs)))
```

### 8.3.5 Gültigkeit als Faltung

; prüfen, ob ein Prädikat auf alle Elemente einer Liste zutrifft

```
(: every? ((%a -> boolean) (list %a) -> boolean))
```

```
(define every?
```

```
  (lambda (pred? xs)
```

```
    (list-fold #t
```

```
              (lambda (elem result)
```

```
                (and (pred? elem) result))
```

```
    xs)))
```

## 8.4 Prozedurfabriken

### 8.4.1 Komposition von Prozeduren

```
; zwei Prozeduren komponieren (zusammensetzen, hintereinander ausführen)
(: compose ((%b -> %c) (%a -> %b) -> (%a -> %c)))
(define compose
  (lambda (f g)
    (lambda (x) (f (g x)))))
```

**Erklärung:** `(compose f g)` liefert die *Komposition* der Funktionen `f` und `g`, d.h., eine Funktion, die zuerst `g` auf ihr Argument anwendet und dann `f` auf das Ergebnis anwendet.

## 8.4.2 Beispiele für Prozedurkomposition

```
(define add-5  
  (lambda (x) (+ 5 x)))
```

- Eine Funktion, die erst quadriert und dann noch fünf addiert:

```
(compose add-5 square)      ; == (lambda (x) (+ 5 (* x x)))
```

```
(define f (compose add-5 square))
```

```
(f 0)      ; == 5
```

```
(f 1)      ; == 6
```

```
(f 2)      ; == 9
```

- Eine Funktion, die das *zweite* Element einer Liste liefert

```
(define second (compose first rest))
```

```
(second (list 1 2 3))      ; == 2
```

### 8.4.3 Beispiel für die Verwendung

**Beispielrechnung:** Bestimme Wert von `second`, dann von `(second (list 1 2 3))`

```
(compose first rest)
=> ((lambda (f g)
      (lambda (x) (f (g x))))
    first
    rest)
=> [ f := first, g := rest ]
    (lambda (x) (first (rest x)))
;-----
    (second (list 1 2 3))
=> ((lambda (x) (first (rest x))) (list 1 2 3))
=> [ x := (list 1 2 3) ]
    (first (rest (list 1 2 3)))
=> 2
```

#### 8.4.4 Prozedur wiederholt anwenden

```
; Prozedur mit sich selbst komponieren  
(: repeat (natural (%a -> %a) -> (%a -> %a)))  
(define repeat  
  (lambda (n f)  
    (if (= n 0)  
        (lambda (x) x)  
        (compose f (repeat (- n 1) f)))))
```

- Was macht

```
((repeat 6 (lambda (n) (* 2 n))) 1)
```

## 8.5 Der Schönfinkel-Isomorphismus

- Manchmal wird eine Spezialversion einer mehrstelligen Prozedur benötigt.
- Bsp: Hilfsfunktionen wie `add-5` als Spezialversion der zweistelligen Addition.
- Alternative: Definiere die *geschönfinkelte* (bzw. *curryfizierte*) Version der Addition

```
(define add
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

- `add` erwartet zwei Argumente *nacheinander*
- Verwendung

```
((add 17) 4)
=> ((lambda (y) (+ 17 y)) 4)
=> (+ 17 4)
```

### 8.5.1 Zwei weitere Prozedurfabriken

; Prozedur erzeugen, die mit einer Konstante multipliziert

```
(: make-mult (number -> (number -> number)))
```

```
(define make-mult
```

```
  (lambda (x)
```

```
    (lambda (y)
```

```
      (* x y))))
```

; Prozedur erzeugen, die ein Element an eine Liste anfügt

```
(: make-prepend (%a -> ((list %a) -> (list %a))))
```

```
(define make-prepend
```

```
  (lambda (x)
```

```
    (lambda (y)
```

```
      (make-pair x y))))
```

- Gleiches Muster: abstrahieren!

## 8.5.2 Die curry-Prozedur

```
; Prozedur mit zwei Parametern staffeln  
(: curry ((%a %b -> %c) -> (%a -> (%b -> %c))))  
(define curry  
  (lambda (f)  
    (lambda (x)  
      (lambda (y)  
        (f x y))))))
```

Damit ergeben sich

```
(define make-add      (curry +))  
(define make-mult    (curry *))  
(define make-prepend (curry make-pair))
```

### 8.5.3 Entstaffeln von Prozeduren

```
; Prozedur mit zwei Parametern entstaffeln  
(: uncurry ((%a -> (%b -> %c)) -> (%a %b -> %c)))  
(define uncurry  
  (lambda (f)  
    (lambda (x y)  
      ((f x) y))))
```

Es gilt für alle  $p$  mit passendem Typ:

$$(\text{uncurry } (\text{curry } p)) \equiv p$$

## 8.6 Zusammenfassung

- Prozeduren abstrahieren von wiederkehrenden Programmstücken.
- Prozeduren höherer Ordnung erlauben
  - die Abstraktion von Programmiermustern.
  - das Zusammensetzen von Programmiermustern.
- Anonyme Prozeduren
- Schönfinkel-Isomorphismus