

# Informatik II Skript Sommersemester 2015

Finn Ickler

24. Juli 2015

## Inhaltsverzeichnis

<b>14.4.2015</b>	<b>5</b>
<b>16.4.2015</b>	<b>6</b>
<b>21.4.2015</b>	<b>8</b>
<b>23.4.2015</b>	<b>11</b>
<b>28.4.2015</b>	<b>14</b>
<b>30.4.2015</b>	<b>17</b>
<b>5.5.2015</b>	<b>21</b>
<b>7.5.2015</b>	<b>23</b>
<b>12.5.2015</b>	<b>27</b>
<b>19.5.2015</b>	<b>31</b>
<b>21.5.2015</b>	<b>36</b>
<b>9.6.2015</b>	<b>41</b>
<b>11.6.2015</b>	<b>44</b>
<b>16.6.2015</b>	<b>49</b>

<b>18.6.2015</b>	<b>51</b>
<b>23.6.2015</b>	<b>55</b>
<b>25.6.2015</b>	<b>60</b>
<b>30.6.2015</b>	<b>64</b>
<b>2.7.2015</b>	<b>69</b>
<b>7.7.2015</b>	<b>74</b>
<b>9.7.2015</b>	<b>78</b>
<b>14.7.2015</b>	<b>83</b>
<b>16.7.2015</b>	<b>90</b>
<b>21.7.2015</b>	<b>98</b>
<b>23.7.2015</b>	<b>103</b>

## Codebeispiele

1	Arithmetik mit Fließkommazahlen . . . . .	6
2	Schlüsselwort define . . . . .	7
3	Lambda Abstraktion . . . . .	8
	Prozeduren.rkt . . . . .	8
4	Bilderzusammenstellung am Beispiel einer Uhr . . . . .	10
5	Das Substitutionsmodell . . . . .	13
6	Die one-of Signatur . . . . .	14
7	Konstruktion eines eigenen Ifs? . . . . .	14
8	Absolutbetrag durch cond . . . . .	16
9	Boolsche Ausdrücke mit and und or . . . . .	17
	Record-Definitionen.rkt . . . . .	17
10	Record Definitionen . . . . .	17
11	Check-property . . . . .	19

12	Übersetzung mathematischer Aussagen in check-property . . . . .	19
13	Konstruktoren und Selektoren . . . . .	20
14	predicate Signaturen am Beispiel von Längen- und Breitengrade . . . . .	23
15	Ersetzung one-of durch predicate Signaturen . . . . .	23
16	Geocoding . . . . .	25
17	cond mit gemischten Daten . . . . .	26
18	Wrapper und Worker . . . . .	27
19	make-pair, ein polymorpher Datentyp . . . . .	30
20	Listen mit Signatur list-of . . . . .	31
21	Geschachtelte Listen . . . . .	33
22	Rekursion auf Listen: Länge einer Liste . . . . .	34
23	Rekursion: Zusammenfügen zweier Listen . . . . .	35
24	Bildmanipulation mit Listen aus Pixeln . . . . .	36
25	Check-property mit Einschränkungen . . . . .	39
26	Rekursion auf natürlichen Zahlen: Fakultät . . . . .	39
27	Fehlerhafte Rekursionen . . . . .	40
	Endrekursion.rkt . . . . .	41
28	Umdrehen einer Liste durch lambda Rekursion . . . . .	42
29	Letrec und endrekursives Umdrehen einer Liste . . . . .	43
	HigherOrderProcedures.rkt . . . . .	51
30	Anwendungsbeispiele foldr . . . . .	53
	Animationen-und-HOP-Typ2.rkt . . . . .	55
31	Animation 1: Ein Zähler . . . . .	55
32	Animation 2: Ein Raumschiff . . . . .	55
33	Anwendungen von Combined . . . . .	57
34	+ als Higher Order Funktion . . . . .	58
	CurryUndMengen.rkt . . . . .	60
35	Einfache Curry Beispiele . . . . .	60
36	Ableitungen berechnen mit Curry . . . . .	61
37	Mengenoperationen Teil 1 . . . . .	62
38	Mengenoperationen Teil 2 . . . . .	63
	StreamsUndMengen.rkt . . . . .	64
39	Listen zu Mengen Konvertierung . . . . .	64

40	Mengenoperationen . . . . .	65
41	Implementation von Streams . . . . .	67
42	Rekursiv definierter Stream . . . . .	69
	Baeume.rkt . . . . .	70
43	Implementation von Bäumen . . . . .	72
44	Berechnung der Größe eines Baumes . . . . .	74
45	Entwicklung einer Pretty Print Methode für Bäume . . . . .	75
46	Fold über Bäume . . . . .	78
47	Isomorphie von Listen und rechtstiefe Bäumen . . . . .	79
48	Breitendurchlauf eines Baumes . . . . .	81
	Huffman.rkt . . . . .	83
49	Huffman Codes . . . . .	84
50	Decoding von Huffman Codes . . . . .	87
51	Huffman-Code Encode . . . . .	88
52	Erzeugen eines Huffman-Trees . . . . .	90
53	Ausdruck Arithmetischer Ausdrücke mittels Quote . . . . .	95
	lambdacalc.rkt . . . . .	98
54	Implementation des lambda Kalküls . . . . .	99
55	Freie und gebundene Variablen im lambda Kalkül . . . . .	102
56	$\beta$ -Reduktion . . . . .	105

**14.4.2015****Scheme**

Ausdrücke , Auswertung und Abstraktion

**Dr Racket**

Definitonsfenster

Willkommen bei [DrRacket](#), Version 6.1.1 [3m].Sprache: **Die Macht der Abstraktion**; memory limit: **128 MB**.> [Interaktionsfenster](#)

Die Anwendung von Funktionen wird in Scheme ausschliesslich in Präfixnotation durchgeführt

Mathematik	Scheme
$44 - 2$	<code>(- 44 2)</code>
$f(x, y)$	<code>(f x y)</code>
$\sqrt{81}$	<code>(sqrt 81)</code>
$9^2$	<code>(! 3)</code>

Allgemein: `(<funktion><argument1><argument2> ...)`

`(+ 40 2)` und `(odd? 42)` sind Beispiele für *Ausdrücke*, die bei *Auswertung* einen Wert liefern.


(Notation:  $\rightsquigarrow$ )

$(+ \ 40 \ 2) \rightsquigarrow 42$   
*Reduktion*  
 $(\text{odd? } 42) \rightsquigarrow \#f$

Interaktionsfenster:

$\underbrace{Read \rightarrow Eval \rightarrow Print \rightarrow Loop}_{REPL}$

*Literale* stehen für einen konstanten Wert (auch: *Konstante*) und sind nicht weiter reduzierbar.

Literal		Sorte, Typ
$\#f, \#t$	(true, false, Wahrheitswert)	boolean
"x"	(Zeichenketten)	String
0 1904 42 -2	(ganze Zahl)	Integer
0.42 3.14159	(FlieSS kommazahl)	real
1/2, 3/4, -1/10	(rationale Zahlen)	rational
	(Bilder)	image

## 16.4.2015

Auswertung *zusammengesetzter Ausdrücke* in mehreren Schritten (Steps), von "innen nach außen", bis keine Reduktion mehr möglich ist.

$(+ \ ( \ (+ \ 20 \ 20) \ (+ \ 1 \ 1)) \rightsquigarrow (+ \ 40 \ (+ \ 1 \ 1) \rightsquigarrow (+ \ 40 \ 2) \rightsquigarrow 42$

Codebeispiel 1: **Achtung:** Scheme rundet bei Arithmetik mit Fließkommazahlen (interne Darstellung ist binär

```

; Achtung: Arithmetik mit Fließkommazahlen (real)
  unterliegt Rundung!
(+ 0.7
  (- (/ 1/2 0.25)
    (/ 0.6 0.3)))

```

```

10 (- (+ 0.7
      (/ 1/2 0.25))
   (/ 0.6 0.3))

; Arithmetik mit rationalen Zahlen (rational) ist exakt
(- (+ 7/10
      (/ 1/2 1/4))
   (/ 6/10 3/10))

```

Ein Wert kann an einen *Namen* (auch *Identifizier*) gebunden werden, durch

```
(define <id> <e>)      <id>Identifizier <e>Ausdruck
```

Erlaubte konsistente Wiederverwendung, dient der Selbstdokumentation von Programmen

**Achtung:** Dies ist eine sogenannte Spezialform und kein Ausdruck. Insbesondere besitzt diese Spezialform *keinen* Wert, sondern einen Effekt Name  $\langle id \rangle$  wird an den Wert von  $\langle e \rangle$  gebunden.

Namen können in Scheme beliebig gewählt werden, solange

- (1) die Zeichen `()[]{}“‘;#|` nicht vorkommen
- (2) dieser nicht einem numerischen Literal gleicht.
- (3) kein Whitespace (Leerzeichen, Tabulator, Return) enthalten ist.

Beispiel: euro→US\$

**Achtung:** Groß- \ Kleinschreibung ist irrelevant.

### Codebeispiel 2: Bindung von Werten an Namen

```

5 (define absoluter-nullpunkt -273.15)
  (define pi 3.141592653)
  (define Gruendungsjahr-SC-Freiburg 1904)
  (define top-level-domain-germany "de")
  (define minutes-in-a-day (* 24 60))
  (define vorwahl-tuebingen (sqrt 1/2))

```

Eine *lambda-Abstraktion* (auch Funktion, Prozedur) erlaubt die Formatierung von Ausdrücken, in denen mittels *Parametern* von konkreten Werten abstrahiert wird.

```
(lambda (<p1><p2>...) <e>)
```

$\langle e \rangle$  Rumpf: enthält Vorkommen der Parameter  $\langle p_n \rangle$

(lambda(...)) ist eine Spezialform. Wert der lambda-Abstraktion ist # $\langle$ procedure $\rangle$

. *Anwendung* (auch Application) des lambda-Aufrufs führt zur Ersetzung aller Vorkommen der Parameter im Rumpf durch die angegebenen *Argumente*.

### Codebeispiel 3: Lambda-Abstraktion

```
; Abstraktion: Ausdruck mit "Loch" ☉
(lambda (☉) (* ☉ (* 155 minutes-in-a-day)))

5 ; Zuwachs der Weltbevölkerung innerhalb von days Tagen
(define population-growth-in-days
  (lambda (days) (* days (* 155 minutes-in-a-day))))

(population-growth-in-days 7)

(lambda (days) (* days (* 155 minutes-in-a-day))) 365) ~~~>
(* 365 (* 155 minutes-in-a-day)) ~~~> 81468000
```

In Scheme leitet ein Semikolon einen Kommentar ein, der bis zum Zeilenende reicht und vom System bei der Auswertung ignoriert wird.

Prozeduren sollten im Programm ein- bis zweizeilige *Kurzbeschreibungen* direkt vorangestellt werden.

## 21.4.2015

Eine Signatur prüft, ob ein Name an einen Wert einer angegebenen Sorte (Typ) gebunden wird. Signaturverletzungen werden protokolliert.

```
(: <id> <signatur>)
```

Bereits eingebaute Signaturen



natural	$\mathbb{N}$	boolean
integer	$\mathbb{Z}$	string
rational	$\mathbb{Q}$	image
real	$\mathbb{R}$	...
number	$\mathbb{C}$	

(: ...) ist eine Spezialform und hat keinen Wert, aber einen Effekt: Signaturprüfung

*Prozedur Signatur* spezifizieren sowohl Signaturen für die Parameter  $P_1, P_2, \dots, P_n$  als auch den Ergebniswert der Prozedur,

(: <Signatur  $P_1$ > ... <Signatur  $P_n$ > -> <Signatur Ergebnis>)

Prozedur Signaturen werden *bei jeder Anwendung* einer Prozedur auf Verletzung geprüft. *Testfälle* dokumentieren das erwartete Ergebnis einer Prozedur für ausgewählte Argumente:

(check-expect <e1> <e2>)

Werte Ausdruck  $\langle e_1 \rangle$  aus und teste, ob der erhaltene Wert der Erwartung  $\langle e_2 \rangle$  entspricht (= der Wert von  $\langle e_2 \rangle$ ) Einer Prozedur sollte Testfälle direkt vorangestellt werden.

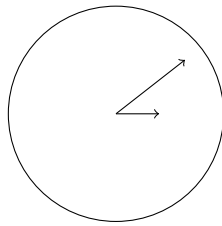
Spezialform: kein Wert, sondern Effekt: Testverletzung protokollieren

*Konstruktionsanleitung für Prozeduren:*

- (1) Kurzbeschreibung (ein- bis zweizeiliger Kommentar mit Bezug auf Parametername)
- (2) Signaturen
- (3) Testfälle
- (4) Prozedurrumpf

*Top-Down-Entwurf* (Programmieren durch "Wunschdenken")

Beispiel: Zeichne Ziffernblatt (Stunden- und Minutenzeiger) zu Uhrzeit h:m auf einer analogen 24h-Uhr



Minutenzeiger legt  $\frac{360^\circ}{60}$  Grad pro Minute zurück (also  $\frac{360}{60} \cdot m$ )

Stundenzeiger legt  $\frac{360}{12}$  pro Stunde zurück ( $\frac{360}{12} \cdot h + \frac{360}{12} \cdot \frac{m}{60}$ )

#### Codebeispiel 4: Bauen der Uhr durch Top Down Entwurf

```

; Grad, die Minutenzeiger pro Minute zurücklegt
(define degrees-per-minute 360/60)

5 ; Grad, die Stundenzeiger pro voller Stunde zurücklegt
  (define degrees-per-hour 360/12)

; Zeichne Ziffernblatt zur Stunde h und Minute m
(: draw-clock (natural natural -> image))
10 (check-expect (draw-clock 4 15) (draw-clock 16 15))
(define draw-clock
  (lambda (h m)
    (clock-face (position-hour-hand h m)
                 (position-minute-hand m))))

15 ; Winkel (in Grad), den Minutenzeiger zur Minute m einnimmt
    (: position-minute-hand (natural -> rational))
    (check-expect (position-minute-hand 15) 90)
    (check-expect (position-minute-hand 45) 270)
    (define position-minute-hand
20   (lambda (m)
      (* degrees-per-minute m)))

; Winkel (in Grad), den Stundenzeiger zur Stunde h einnimmt
    (: position-hour-hand (natural natural -> rational))
25 (check-expect (position-hour-hand 3 0) 90)
    (check-expect (position-hour-hand 18 30) 195)

```

```

(define position-hour-hand
  (lambda (h m)
    (+ (* degrees-per-hour (modulo h 12))      ; h mod 12
       {0,1,...,11}
       (* degrees-per-hour (/ m 60)))))

; Zeichne Ziffernblatt mit Minutenzeiger um dm und
; Stundenzeiger um dh Grad gedreht
(: clock-face (rational rational -> image))
(define clock-face
  (lambda (dh dm)
    (clear-pinhole
     (overlay/pinhole
      (circle 50 "outline" "black")
      (rotate (* -1 dh) (put-pinhole 0 35 (line 0 35 "red")))
      (rotate (* -1 dm) (put-pinhole 0 45 (line 0 45
        "blue"))))))))

; Zeichne zwei Ziffernblätter um 23:50 und 23:12 Uhr
(draw-clock 23 50)
(draw-clock 23 12)

```

## 23.4.2015

### Substitutionsmodell

*Reduktionsregeln* für Scheme (Fallunterscheidung je nach Ausdrücken) wiederhole, bis keine Reduktion mehr möglich

- literal (1, "abc", #t, ...)  $\rightsquigarrow$  [eval<sub>lit</sub>]
- Identifier id(pi, clock-face,...)  $\rightsquigarrow$  gebundene Wert [eval<sub>id</sub>]
- lambda Abstraktion  $(\text{lambda } (...) ...) \rightsquigarrow (\text{lambda } (...) ...) \quad [\text{eval}_\lambda]$
- Applikationen (f e<sub>1</sub> e<sub>2</sub>...)

(1)  $f, e_1, e_2$  reduzieren erhalte:  $f', e_1', e_2'$

(2)  $\begin{cases} \text{Operation } f' \text{ auf } e_1' \text{ und } e_2' [\text{apply}_{\text{prim}}] & \text{falls } f' \text{ primitiv ist} \\ \text{Argumentenwerte in den Rumpf von } f' \text{ einsetzen, dann reduzieren} & \text{falls } f' \text{ lambda Abstraktion} \end{cases}$

Beispiel:

`(+ 40 2)`  $\xrightarrow[\text{eval id}]{} (\#<\text{procedure+}> 40 2)$   $\xrightarrow{} 42$

`(position-minute-hand 30)`  $\xrightarrow[\text{eval id}]{} ((\text{lambda } (m) (* \text{degrees-per-minute } m)) 30)$   
 $\xrightarrow[\text{eval lambda}]{} (* \text{degrees-per-minute } 30)$   
 $\xrightarrow[\text{eval id}]{} (\#<\text{procedure } *> 360/60 30)$   
 $\xrightarrow[\text{apply prim}]{} 180$

Bezeichnen `(lambda (x) (* x x))` und `lambda (r) (* r r)` die gleiche Prozedur?  $\Rightarrow$  JA!

Achtung: Das hat Einfluß auf das Korrekte Einsetzen von Argumenten für Prozeduren (siehe apply)

## Prinzip der Lexikalischen Bindung

Das *bindende Vorkommen* eines Identifiers `id` kann im Programmtext systematisch bestimmt werden: Suche strikt von innen nach außen, bis zum ersten

(1) `(lambda (r) <Rumpf>`

(2) `(define <e>)`

Übliche Notation in der Mathematik: *Fallunterscheidung*

$$\max(x_1, x_2) = \begin{cases} x_1 & \text{falls } x_1 \geq x_2 \\ x_2 & \text{sonst} \end{cases}$$

*Tests* (auch Prädikate) sind Funktionen, die einen Wert der Signatur `boolean` liefern.

Typische primitive Tests.

`(: = (number number -> boolean))`

`(: < (real real -> boolean))`

auch `>`, `<=`, `>=`

`(: String=? (string string -> boolean))`

auch `string>?`, `string<=?`

`(: zero? (number -> boolean))`

auch `odd?`, `even?`, `positive?`, `negative?`

Binäre Fallunterscheidung *if*

*if*

$\langle e_1 \rangle$  Mathematik:

$$\langle e_2 \rangle \begin{cases} e_1 & \text{falls } t_1 \\ e_2 & \text{sonst} \end{cases}$$

$\langle e_2 \rangle$

### Codebeispiel 5: Das Substitutionsmodell

```

; SUBSTITUTIONSMODELL

; Markiere und reduziere mittels Stepper
5 ; (beobachte Reduktionsregeln eval_id, apply_, apply_prim)
  (position-minute-hand 30)

10

; Markiere und reduziere mittels Stepper:
; unendliche Reduktion, Ergebnis ist ("bottom")
(define ping
  (lambda (x)
15   (pong x)))

(define pong
  (lambda (x)
20   (ping x)))

; (ping 42)

; Markiere und reduziere mittels Stepper:
25 ; unendliche Reduktion, Ergebnis ist ("bottom")
(define forever
  (lambda (x)
    (forever x)))

30 ; (forever "SCF")

; Syntaxprüfung zeigt die bindenden Vorkommen für

```

```

; die Identifier (Parameter) r
35 ((lambda (r)
    (* pi
      ((lambda (r)
        (* r r))
       r)))
40 10)

```

## 28.4.2015

Die Signatur *one of* lässt genau einen der ausgewählten Werte zu.

`(one of <e1> <e2> ... <en>)`

### Codebeispiel 6: one-of am Beispiel des Fußballpunktesystems

```

; Punkte der Heimmannschaft bei Ergebnis h:a
(: heim-punkte (natural natural -> (one-of 3 0 1)))
(check-expect (heim-punkte 2 0) 3)
5 (check-expect (heim-punkte 1 4) 0)
  (check-expect (heim-punkte 3 3) 1)
  (define heim-punkte
    (lambda (h a)
      (cond ((> h a) 3)
10          ((< h a) 0)
            (else 1))))

```

Reduktion von if:

`(if t1 <e1> <e2>)`

① Reduziere  $t_1$ , erhalte  $t'_1 \rightsquigarrow$   $\begin{cases} \langle e_1 \rangle & \text{falls } t'_1 = \#t, \langle e_2 \rangle \text{ niemals ausgewertet} \\ \langle e_2 \rangle & \text{falls } t'_1 = \#f, \langle e_1 \rangle \text{ niemals ausgewertet} \end{cases}$   
 ②

### Codebeispiel 7: Koennen wir unser eigenes 'if' aus 'cond' konstruieren? (Nein!)

```

; Bedingte Auswertung von e1 oder e2 (abhängig von t1)
(check-expect (my-if (= 42 42) "Yes!" "No!") "Yes!")
(check-expect (my-if (odd? 42) "Yes!" "No!") "No!")
5 (define my-if
  (lambda (t1 e1 e2)

```

```

      (cond (t1 e1)
            (else e2)))

10
; Sichere Division x/y, auch für y = 0
(: safe-/ (real real -> real))
(define safe-/
15  (lambda (x y)
      (my-if (= y 0)      ; <-- Funktion my-if wertet ihre
        Argumente
        x                ; vor der Applikation aus: (/ x y)
        wird
        (/ x y)))      ; in *jedem* Fall reduziert. :-(

20
(safe-/ 42 0)          ; Führt zu Fehlermeldung "division by
  zero"
                        ; (Reduktion mit Stepper durchführen)

```

Spezifikation Fallunterscheidung (conditional expression):

(cond	Mathematik:
(< $t_1$ > < $e_1$ >)	$\left\{ \begin{array}{l} e_1 \text{ falls } t_1 \\ e_2 \text{ falls } t_2 \\ \dots \\ e_n \text{ falls } t_n \\ e_{n+1} \text{ sonst} \end{array} \right.$
(< $t_2$ > < $e_2$ >)	
...	
(< $t_n$ > < $e_n$ >)	
(else < $e_{n+1}$ >))	

Werte die Tests in den Reihenfolge  $t_1, t_2, t_3, \dots, t_n$  aus.

Sobald  $t_i \neq t$  ergibt, werte Zweig  $e_i$  aus.  $e_i$  ist Ergebnis der Fallunterscheidung. Wenn  $t_n \neq t$  liefert, dann liefert

{	Fehlermeldung „cond: alle Tests ergaben false“	falls kein else Zweig
	< $e_{n+1}$ >	sonst

## Codebeispiel 8: Absolutwert von x

```

; Absolutwert von x
(: my-abs (real -> real))
(check-within (my-abs -4.2) 4.2 0.001) ; Wichtig:
5 (check-within (my-abs 4.2) 4.2 0.001) ; Tesfälle decken
    alle Zweige
(check-within (my-abs 0) 0 0.001) ; der conditional
    expression an
(define my-abs
  (lambda (x)

```

Reduktion von cond [eval<sub>cond</sub>]

```

(cond (<t1> <e1>) (<t2> <e2>) ... (<tn> <en>))

```

① Reduziere  $t_1$  erhalte  $t'_1 \rightsquigarrow$   $\begin{cases} <e_1> & \text{falls } t'_1 = \#t \\ (\text{cond } <t_2> <e_2>) & \text{sonst} \end{cases}$

②  $(\text{cond}) \rightsquigarrow$  „Fehlermeldung: alle Test ergaben false“

$(\text{cond } (\text{else } <e_{n+1}>)) \rightsquigarrow e_{n+1}$

cond ist syntaktisches Zucker (auch abgeleitete Form) für eine verbundene Anwendung von if

```

(cond (<t1><e1>) (<t2><e2>) ... (<tn><en>) (else <en+1>))
5
      if (<t1> <e1>
         if <t2> <e2>
         ...
         if <tn> <en>
         <en+1>))

```

Spezialform 'and' und 'or'

```

(or <t1> <t2> ... <tn>)  $\rightsquigarrow$  (if <t1> (or <t2> ... <tn>) #t)
(or)  $\rightsquigarrow$  #f
(and <t1> <t2> ... <tn>)  $\rightsquigarrow$  (if <t1> (and <t2> ... <tn>) #f)
(and)  $\rightsquigarrow$  #t

```



## Codebeispiel 9: Konstruktion komplexer Prädikate mittels 'and' und 'or'

```

; Konstruktion komplexer Prädikate mittels `and' und `or':

(and #t #f) ; #f (Mathematik: , Konjunktion)
5 (or #t #f) ; #t (Mathematik: , Disjunktion)

; Kennzeichen am/pm für Stunde h
(: am/pm (natural -> (one-of "am" "pm" "???")))
10 (check-expect (am/pm 10) "am")
    (check-expect (am/pm 13) "pm")
    (check-expect (am/pm 25) "???" )
(define am/pm
  (lambda (h)
15    (cond ((and (>= h 0) (< h 12)) "am")
          ((and (>= h 12) (< h 24)) "pm")
          (else "???" )))

```

**30.4.2015***Zusammengesetzte Daten*Ein Charakter *besteht* aus drei *Komponenten*

- Name des Charakters (name)
  - Handelt es sich um einen Jedi? (jedi?)
  - Stärke der Macht (force)
- } Datendefinition für zusammengesetzte Daten

Konkrete Charakter:

name	„Luke Skywalker“
jedi?	#f
force	25

## Codebeispiel 10: Starwars Charakter als Racket Records

```

; Ein Charakter (character) besteht aus
; - Name (name)
; - Jedi-Status (jedi?)
; - Stärke der Macht (force)
5 (: make-character (string boolean real -> character))
   (: character? (any -> boolean))
   (: character-name (character -> string))

```

```

10 (: character-jedi? (character -> boolean))
   (: character-force (character -> real))
   (define-record-procedures character
     make-character
     character?
     (character-name
      character-jedi?
15      character-force))

; Definiere verschiedene Charaktere des Star Wars Universums
(define luke
  (make-character "Luke_Skywalker" #f 25))
20 (define r2d2
  (make-character "R2D2" #f 0))
  (define dooku
    (make-character "Count_Dooku" #f 80))
  (define yoda
25 (make-character "Yoda" #t 85))

```

Zusammengesetzte Daten = *Records* in Scheme Record-Definition legt fest:

- Record-Signatur
- *Konstruktor* (baut aus Komponenten einen Record)
- Prädikat (liegt ein Record vor?)
- Liste von *Selektoren* (lesen jeweils eine Komponente des Records)

```

(define-record-procedure <t>
  make-<t>
  <t>?
  (<t>-<comp1> ... <t>-<comp2>))
5 ;Liste der n Selektoren

```

Verträge des Konstruktors der Selektoren für Record- Signatur

$\langle t \rangle$  mit Komponenten namens  $\langle \text{comp}_1 \rangle \dots \langle \text{comp}_n \rangle$

```

(: make-<t> (<t1>...<t2>) -> <t>)
(: <t>-<comp1> (<t> -> <t1>))
(: <t>-<compn> (<t> -> <tn>))

```

Es gilt für alle Strings  $n$ , Booleans  $j$  und Integer  $f$ :

```

(character-name (make-character n j f) n)
(character-jedi? (make-character n j f) j)
(character-force (make-character n j f) f)

```

Spezialform check-property:

```
(check-property
  (for-all ((<id1> <sig1>) ...
            (<idn> <sign>))
    <e>))
5      ↓
;Bezieht sich auf <id1> ... <idn>
```

Test erfolgreich, falls  $\langle e \rangle$  für beliebig gewählte Bedeutungen für  $\langle id_1 \rangle \dots \langle id_n \rangle$  immer #t ergibt

### Codebeispiel 11: Interaktion von Selektoren und Konstruktor:

```
; Stärker: teste Interaktion von Konstruktor und Selektoren
; *für alle* (beliebige) Konstruktorargumente
; (algebraische Eigenschaft)
5 (check-property
  (for-all ((n string)
            (j boolean)
            (f real))
    (expect (character-name (make-character n j f)) n)))

10 (check-property
  (for-all ((n string)
            (j boolean)
            (f real))
    (expect (character-jedi? (make-character n j f)) j)))

15 (check-property
  (for-all ((n string)
            (j boolean)
            (f real))
20 (expect-within (character-force (make-character n j f)) f 0
    .001)))
```

*Beispiel:* Die Summe von zwei natürlichen Zahlen ist mindestens so groß wie jeder dieser Zahlen:  $\forall x_1 \in \mathbb{N}, x_2 \in \mathbb{N} : x_1 + x_2 \geq \max\{x_1, x_2\}$

### Codebeispiel 12: Mathematische $\forall$ -Aussage in Racket

```
; Für alle natürlichen Zahlen x1, x2 gilt:  $x_1 + x_2 \geq \max(x_1, x_2)$ 
(check-property
```

```

(for-all ((x1 natural)
          (x2 natural))
  (>= (+ x1 x2) (max x1 x2)))

```

Konstruktion von Funktionen, die bestimmte gesetzte Daten *konsumiert*.

- Welche Record-Componenten sind relevant für Funktionen?

→ Schablone:

```

(: sith? (character -> boolean))
(define sith?
  (lambda (c)
    ... (character-jedi? c)
    ... (character-force c) ...))

```

Konstruktion von Funktionen, die zusammengesetzte Daten *konstruieren*

- Der konstruktor *muss* aufgerufen werden

→ Schablone:

```

(define
  lambda (...)
    ... (make-<t>) ...)

```

- Konkrete Beispiele:

#### Codebeispiel 13: Abfragen der Eigenschaften von character Records

```

; Ist Charakter c ein Jedi oder sein Sith?
(: jedi-or-sith (character -> (one-of "Jedi" "Sith")))

5 (check-expect (jedi-or-sith yoda) "Jedi")
  (check-expect (jedi-or-sith dooku) "Sith")
  (check-error  (jedi-or-sith r2d2) "Not_a_Jedi_or_a_Sith")

(define jedi-or-sith
10 (lambda (c)
    (cond ((character-jedi? c) "Jedi")
          ((sith? c) "Sith")
          (else (violation "Not_a_Jedi_or_a_Sith")))))

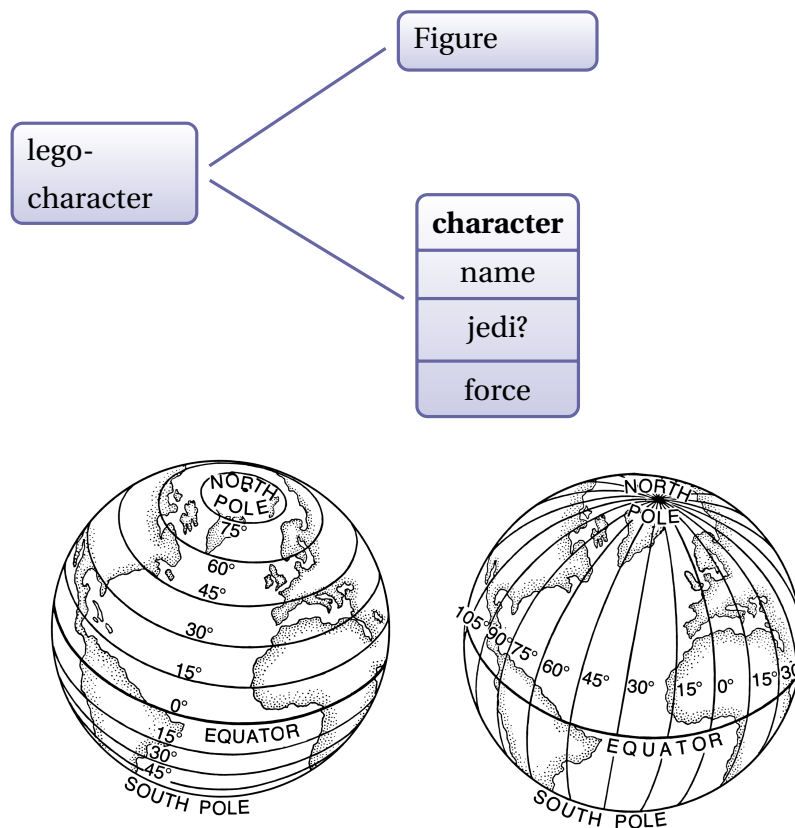
```

```

15 (define train-jedi
    (lambda (c)
      (make-character (character-name c)
                      (> (character-force c) 0) ; #t, falls
                        c zuvor überhaupt Die Macht besitzt
                        (* 2 (character-force c))))))

```

## 5.5.2015



Position Nord/Südwest vom Äquator Position west/östlich vom Nullmeridian

Sei  $\langle p \rangle$  ein Prädikat mit Signatur  $\langle t \rangle \rightarrow \text{boolean}$ .

Eine Signatur der Form  $\langle \text{predicate } \langle p \rangle \rangle$  gilt für jeden Wert der Signatur  $\langle t \rangle$  sofern  $\langle p \rangle \rightsquigarrow \#t$

Signaturen des Typs  $\text{predicate } \langle p \rangle$  sind damit *spezifischer* (restriktiver) als die Signatur  $\langle t \rangle$  selbst.

```
(define <newt> (signature <t>))
```

*Beispiele:*

```
(define farbe  
  (signature (one-of "Blatt" "Herz" "Blatt" "Eichel"  
    "Schell")))
```

## Codebeispiel 14: Restriktive Signaturen mit predicate

```

; Ist x ein gültiger Breitengrad
; zwischen Südpol (-90°) und Nordpol (90°)?
(: latitude? (real -> boolean))
(check-expect (latitude? 78) #t)
5 (check-expect (latitude? -92) #f)
(define latitude?
  (lambda (x)
    (within? -90 x 90)))
; Ist x ein gültiger Längengrad westlich (bis -180°)
10 ; bzw. östlich (bis 180°) des Meridians?
(: longitude? (real -> boolean))
(check-expect (longitude? 0) #t)
(check-expect (longitude? 200) #f)
15 (define longitude?
  (lambda (x)
    (within? -180 x 180)))
; Signaturen für Breiten-/Längengrade basierend auf
; den obigen Prädikaten
20 (define latitude
  (signature (predicate latitude?)))
(define longitude
  (signature (predicate longitude?)))

```

## 7.5.2015

Man kann jedes `one-of` durch ein `predicate` ersetzen.

## Codebeispiel 15: Das "große One-of Sterben des Jahres 2015"

```

(: f ((one-of 0 1 2) -> natural))
(define f
  (lambda (x)
    x))
5 ; And then the "The Great one-of Extinction" of 2015 occurred

```



```

(: g (predicate
  (lambda (x) (or (= x 0) (= x 1) (= x 2)))) -> natural))
(define g

```

10

```
(lambda (x)
  x)
```

Geocoding: Übersetze eine Ortsangabe mittels des Google Maps Geocoding API (Application Programm Interface) in eine Position auf der Erdkugel.

```
(: geocoder (string -> (mixed geocode geocode-error)))
```

Ein geocode besteht aus:

*Signatur*

- Adresse (address) string
- Ortsangabe (loc) location
- Nordostecke (northeast) location
- Südwestecke (southwest) location
- Typ (type) string
- Genauigkeit (accuracy) string

Ein geocode-error besteht aus:

```
(: geocode-address (geocode -> string))
(: geocode-loc (geocode -> location))
(: geocode-... (geocode -> ...))
```

*Signatur*

- Fehlerart (level) (one-of "TCP" "HTTP" "JSON" "API")
- Fehlermeldung (message) string

*Gemischte Daten*

Die Signatur

```
(mixed <t1> ... <tn>)
```

ist gültig für jeden Wert, der mindestens eine der Signaturen  $\langle t_1 \rangle \dots \langle t_n \rangle$  erfüllt.

*Beispiel:* Data-Definition

Eine Antwort des Geocoders ist *entweder*

- ein Geocode (geocode) *oder*
- eine Fehlermeldung (geocode-error)

Beispiel (eingebaute Funktion `string->number`)



```
(: string->number (string -> (mixed number (one-of #f))))
(string->number "42") ~> 42
(string-> number "foo") ~> #f
```

### Codebeispiel 16: Die Google Geocode API

```
(define geocoder-response
  (signature (mixed geocode geocode-error)))

(: sand13 geocoder-response)
5 (define sand13
   (geocoder "Sand_13,_Tübingen"))

(geocode-address sand13)
(geocode-type sand13)
10 (location-lat (geocode-loc sand13))
(location-lng (geocode-loc sand13))
(geocode-accuracy sand13)

15 (: lady-liberty geocoder-response)
(define lady-liberty
  (geocoder "Statue_of_Liberty"))

(: alb geocoder-response)
20 (define alb
   (geocoder "Schwäbische_Alb"))

(: A81 geocoder-response)
25 (define A81
   (geocoder "A81,_Germany"))
```

#### Erinnerung:

Das Prädikat  $\langle t \rangle?$  einer Signatur  $\langle t \rangle$  unterscheidet Werte der Signatur  $\langle t \rangle$  von allen anderen Werten:

```
(:  $\langle t \rangle?$  (any -> boolean))
```

Auch: Prädikat für eingebaute Signaturen

```
number?
complex?
real?
rational?
5 integer?
natural?
```

```
string?
boolean?
```

Prozeduren, die gemischte Daten der Signaturen  $\langle t_1 \rangle \dots \langle t_n \rangle$  konsumieren:

*Konstruktionsanleitung:*

```
(: <t> ((mixed <t1> ... <tn>) -> ...))
(define <t>
  (lambda (x)
    (cond
      5      ((<t1>? x) ...)
              ...
              ((<tn>? x) ...))))
```

Mittels *let* lassen sich Werte an *lokale Namen* binden,

```
(let (
  (<id1> <e1>)
  (...)
  (<idn> <en>))
5 <e>
)
```

Die Ausdrücke  $\langle e_1 \rangle \dots \langle e_n \rangle$  werden *parallel* ausgewertet.  $\Rightarrow \langle id_1 \rangle \dots \langle id_n \rangle$  können in  $\langle e \rangle$  (und nur hier) verwendet werden. Der Wert des *let* Ausdrucks ist der Wert von  $\langle e \rangle$ .

#### Codebeispiel 17: Liegt der Geocode r auf der südlichen Erdhalbkugel?

```
; (Breitengrad < 0°?)
(: southern-hemisphere? (string -> boolean))

(check-expect (southern-hemisphere? "Cape_Town") #t)
5 (check-expect (southern-hemisphere? "Tübingen") #f)
(check-error (southern-hemisphere? "Mos_Eisley") "Unknown_
  location")

(define southern-hemisphere?
  (lambda (r)
    10 (let ((gc (geocoder r)))
        (cond ((geocode? gc)
                (< (location-lat (geocode-loc gc)) 0))
              ((geocode-error? gc)
               (violation "Unknown_location"))))))
```

*ACHTUNG:*

'let' ist verfügbar auf ab der Sprachebene "Macht der Abstraktion".

'let' ist syntaktisches Zucker.

```

(let (
  (( $\langle id_1 \rangle \langle e_1 \rangle$ )
  (...)
  ( $\langle id_n \rangle \langle e_n \rangle$ ))
  5   $\langle e \rangle$ 
)

(lambda ( $\langle id_1 \rangle \dots \langle id_n \rangle$ )
  ( $\langle e \rangle$ )
   $\equiv$ 
  ( $\langle e_1 \rangle$ 
  ( $\langle e_2 \rangle \dots$ 
  ( $\langle e_n \rangle$ )))

```

## 12.5.2015

Abstand zweier geographischer Positionen  $b_1, b_2$  auf der Erdkugel in km (lat, lng jeweils in Radian).

### Codebeispiel 18: Abstand zweier geographischer Positionen

```

; Abstand zweier geographischer Positionen l1, l2 auf der
; Erdkugel in km (lat, lng jeweils in Radian):
; dist(l1, l2) =
;   Erdradius in km *
;   acos(cos(l1.lat) * cos(l1.lng) * cos(l2.lat) * cos(l2.lng)
5   +
;   cos(l1.lat) * sin(l1.lng) * cos(l2.lat) * sin(l2.lng)
;   +
;   sin(l1.lat) * sin(l2.lat))
;   pi
(define pi 3.141592653589793)

10 ; Konvertiere Grad d in Radian ( $\pi = 180^\circ$ )
; (: radians (real -> real))
; (check-within (radians 180) pi 0.001)
; (check-within (radians -90) (* -1/2 pi) 0.001)
(define radians
15   (lambda (d)
     (* d (/ pi 180))))

; Abstand zweier Orte o1, o2 auf Erdkugel (in km)
20 ; [Wrapper]

```

```

(: distance (string string -> real))
(check-within (distance "Tübingen" "Freiburg") (distance
  "Freiburg" "Tübingen") 0.001)
(define distance
  (lambda (o1 o2)
    25      (let ((dist (lambda (l1 l2) ; Abstand zweier
                          Positionen l1, l2 (in km) [Worker]
                          (let ((earth-radius 6378) ; Erdradius (in km)
                              (lat1 (radians (location-lat l1)))
                              (lng1 (radians (location-lng l1)))
                              (lat2 (radians (location-lat l2)))
                              (lng2 (radians (location-lng l2))))
                            (* earth-radius
                              (acos (+ (* (cos lat1) (cos lng1) (cos
                                30      lat2) (cos lng2))
                                      (* (cos lat1) (sin lng1) (cos
                                          lat2) (sin lng2))
                                      (* (sin lat1) (sin lat2))))))))))
              (gc1 (geocoder o1))
              (gc2 (geocoder o2)))
    35      (if (and (geocode? gc1)
                    (geocode? gc2))
                (dist (geocode-loc gc1) (geocode-loc gc2))
                (violation "Unknown_location(s)"))))
  40
; ... einmal quer durch die schöne Republik
(distance "Konstanz" "Rostock")

```

### PARAMETRISCH POLYMORPHE PROZEDUREN

Beobachtung: Manche Prozeduren arbeiten unabhängig von den Signaturen ihrer Argumente : *parametrisch polymorphe Funktion* (griechisch : vielgestaltig).

Nutze *Signaturvariablen* %a , %b,...

Beispiel:

```

; die Identität
(: id (%a -> %a))
(define id
  (lambda (x) x))
5
; die konstante Funktion
(: const (%a %b -> %a))
(define const

```

```

    (lambda (x y) x))
10 ; die Projektion
    (: proj ((one-of 1 2) %a %b -> (mixed %a %b)))
    (define proj
      (lambda (i x y)
15         (cond ((= i 1) x)
                  ((= i 2) y))))

```

Eine polymorphe Signatur steht für alle Signaturen, in denen die Signaturvariablen durch konkrete Signaturen ersetzt werden.

Beispiel: Wenn eine Prozedur `(: number %a %b -> %a)` erfüllt, dann auch:

```

(: number string boolean -> string)
(: number boolean natural -> boolean)
(: number number number -> number)

```

"x"	23
-----	----

2	#f
---	----

```

; Ein polymorphes Paar (pair-of %a %b) besteht aus
; - einer ersten Komponente (first)
; - einer zweiten Komponente (rest)
(: make-pair (%a %b -> (pair-of %a %b)))
5 (: pair? (any -> boolean))
  (: first ((pair-of %a %b) -> %a))
  (: rest  ((pair-of %a %b) -> %b))
(define-record-procedures-parametric pair pair-of
  make-pair
10 pair?
  (first
   rest))

```

`(pair-of <t1> <t2>)` ist eine Signatur für Paare deren erster bzw. zweiter Komponente die Signaturen  $\langle t_1 \rangle$  bzw.  $\langle t_2 \rangle$  erfüllen.

```

;→ pair-of Signatur mit (zwei) Parametern
(: make-pair (%a %b -> (pair-of %a %b)))
(: pair? (any -> boolean))
(: first ((pair-of %a %b) -> %a))
5 (: rest ((pair-of %a %b) -> %b))

```

## Codebeispiel 19: Paare aus verschiedenen Datentypen

```

; Ein paar aus natürlichen Zahlen
; FIFA WM 2014
(: deutschland-vs-brasilien (pair-of natural natural))
(define deutschland-vs-brasilien
5   (make-pair 7 1))

; Ein Paar aus einer reellen Zahl (Messwert)
; und einer Zeichenkette (Einheit)
(: measurement (pair-of real string))
10 (define measurement
    (make-pair 36.9 "°C"))

; "Liste" der Zahlen 1,2,3,4
15 (define nested
    (make-pair 1
               (make-pair 2
                           (make-pair 3
                                       4))))
20

; Extrahiere das dritte Element der Liste (hier: 3)
(first (rest (rest nested)))

```

Eine *Liste* von Werten der Signatur  $\langle t_t \rangle$  ist entweder

- leer (Signatur `empty-list`) oder:
- ein Paar (Signatur `pair-of`) aus einem Wert der Signatur  $\langle t \rangle$  und einer Liste von Werten der Signatur  $\langle t \rangle$ .

```

(define list-of
  (lambda (t)
    (signature (mixed empty-list
                      (pair-of t (list-of t))))))

```

Signatur `empty-list` bereits in Racket vordefiniert.

Ebenfalls vordefiniert:

```

(:empty empty-list)
(: empty? (any -\zu boolean))

```

*Operatoren auf Listen*

Konstruktoren	<code>(: empty-list)</code>	leere liste
	<code>(: make-pair (% a (list-of % a))</code>	Konstruiert Liste aus Kopf und Rest
Predikate:	<code>(: empty (any -&gt; boolean)</code>	liegt leere Liste vor?
	<code>(: pair? (any -&gt; boolean))</code>	Nicht leere Liste?
Selektoren:	<code>(: first (list-of %a)-&gt; %a)</code>	Kopf-Element
	<code>(: rest (list-of %a)-&gt; (list-of %a))</code>	Rest Liste

### Codebeispiel 20: Listen aus einem oder verschiedenen Datentypen

```

; Noch einmal (jetzt mit Signatur): Liste der natürlichen
; Zahlen 1,2,3,4
(: one-to-four (list-of natural))
(define one-to-four
  (make-pair 1
    (make-pair 2
      (make-pair 3
        (make-pair 4
          empty))))))

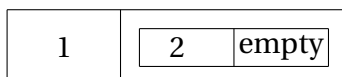
; Eine Liste, deren Elemente natürliche Zahlen oder Strings
; sind
(: abstiegskampf (list-of (mixed number string)))
(define abstiegskampf
  (make-pair "SCF"
    (make-pair 96
      (make-pair "SCP"
        (make-pair "VfB" empty))))))

```

## 19.5.2015

```
(make-pair 1 (make-pair 2 empty))
```

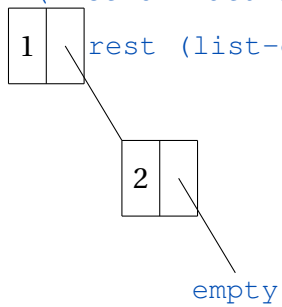
*Visualisierung Listen*





### Spine (Rückgrat)

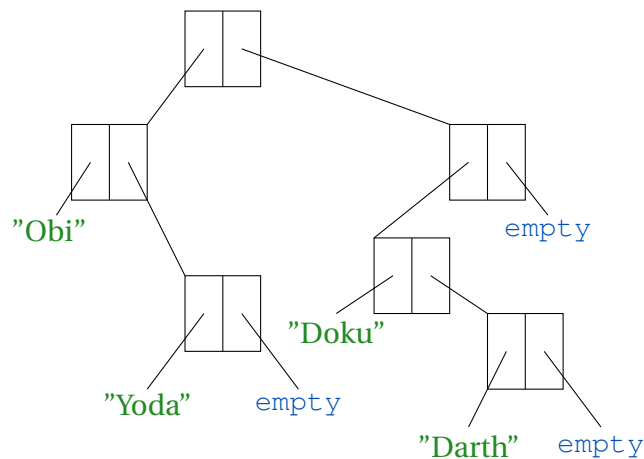
```
(pair-of natural (list-of natural))
(natural first) 1 rest (list-of natural)
```



```
(: one-to-four (list-of natural))
(define one-two
  (make-pair 1
             (make-pair 2
                        empty)))
```

```
(: jedis-and-siths (list-of (list-of string)))
```





### Codebeispiel 21: Jedis und Siths in einer geschachtelten Liste

```

; Geschachtelte Listen
(: jedis-and-siths (list-of (list-of string)))
(define jedis-and-siths
  (MAKE-PAIR (make-pair "Yoda"
                        (make-pair "Obi-Wan" empty))
             (MAKE-PAIR (make-pair "Dooku"
                                    (make-pair "Vader" empty))
                        empty)))

; Navigation in geschachtelten Listen
(check-expect (first (first jedis-and-siths)) "Yoda")
(check-expect (first (rest (first (rest jedis-and-siths))))
  "Vader")

```

### Prozeduren, die Liste konsumieren

#### Konstruktionsanleitung:

#### Beispiel:

```

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

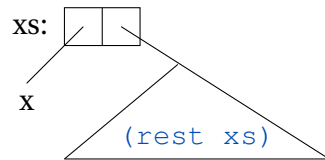
(check-expect (list-sum empty) 0)
(check-expect (list-sum (make-pair 40
                                   (make-pair 2
                                               empty)))) 42)

(check-expect (list-sum one-to-four) 10)

(define list-sum
  (lambda (xs)

```

```
(cond ((empty? xs) 0)
      ((pair? xs) (+ (first xs)
                      (list-sum (rest xs))))))
```



(rest xs) mit Signatur  
(list-of number)  
ist selbst wieder eine  
*kürzere Liste* von Zahlen.  
(list sum (rest  
xs)) erzielt Fortschritt

Konstruktionsanleitung für Prozeduren:

```
(: <f> ((list-of <t1>) -> <t2>))
(define <f>
  (lambda (xs)
    (cond
      5      ((empty? xs) ...)
              ((pair? xs) ...  $\overbrace{(\text{first } xs)}^{<t_1>}$  ...)
              (<f>  $\underbrace{(\text{rest } xs)}_{<t_1>}$  ) ...)))
```

Neue Sprachebene "Macht der Abstraktion"

- Signatur (list-of % a) eingebaut

```
(list <e1> <e2> ... <en>)
≡
(make-pair (<e1>)
  (make-pair <e2>)
  5  ... (make-pair <en>) empty) ...)
```

- Ausgabeformat für nicht leere Listen:

```
{#<list x1x2... xn>
```

### Codebeispiel 22: Länge einer Liste

```
; Länge der Liste xs
(: list-length ((list-of %a) -> natural))

(check-expect (list-length empty) 0)
5 (check-expect (list-length (list 1 1 3 8)) 4)
  (check-expect (list-length jedis-and-siths) 2) ; nicht 4!
```

```

10 (define list-length
    (lambda (xs)
      (cond ((empty? xs) 0)
            ((pair? xs) (+ 1
                           (list-length (rest xs)))))))

```

Füge Listen  $xs$ ,  $ys$  zusammen (concatination)

Zwei Fälle ( $xs$  leer oder nicht leer)

$$\begin{aligned}
 \textcircled{1} \quad & \overbrace{\text{empty}}^{xs} \quad \overbrace{y_1 y_2 \dots y_m}^{ys} \quad \overbrace{y_1 y_2 \dots y_m}^{(cat \ xs \ ys)} \\
 \textcircled{2} \quad & x_1 \quad \overbrace{x_2 \dots x_n}^{(rest \ xs)} \quad y_1 y_2 \dots y_m \quad x_1 \quad \overbrace{x_2 \dots x_n y_1 y_2 \dots y_m}^{(cat \ rest \ xs)}
 \end{aligned}$$

Beobachtung:

- Die Längen von  $xs$  bestimmt die Anzahl der rekursiven Aufrufe von `cat`
- Auf  $xs$  werden *Selektoren* angewendet

### Codebeispiel 23: Zusammenfügen zweier Listen

```

; Füge Listen xs, ys (in dieser Reihenfolge) zusammen
(: cat ((list-of %a) (list-of %a) -> (list-of %a)))

(check-expect (cat (list 1 2) (list 3 4)) (list 1 2 3 4))
5 (check-expect (cat one-to-four empty) one-to-four)
  (check-expect (cat empty one-to-four) one-to-four)

10 (define cat
    (lambda (xs ys)
      (cond ((empty? xs)
              ys)
            ((pair? xs)
              (make-pair (first xs) ; <- cat dennoch param.
                          polymorph
                          (cat (rest xs) ys))))))

15 ; Hinweis: Verfügbar als eingebaute Funktion `append'

```

**21.5.2015**

Codebeispiel 24: Ausflug: Bluescreen Berechnung wie in Starwars mit Listen:



```
(define yoda
```



```
(define dagobah
```

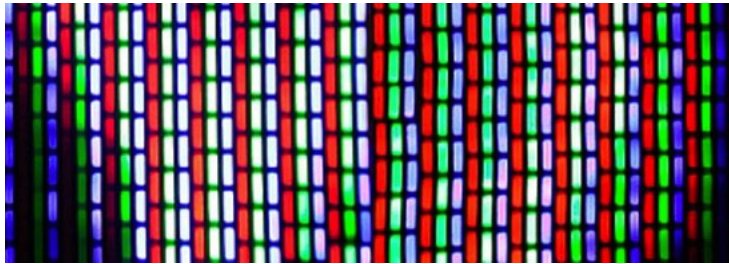
```

;
;Zugriff auf die Liste der Bildpunkte (Pixel) eines Bildes:
5  (: image->color-list (image -> (list-of rgb-color)))
   (: color-list->bitmap ((list-of rgb-color) natural natural ->
   image))

;Breite/Höhe eines Bildes in Pixeln:
10 (: image-width (image -> natural))
   (: image-height (image -> natural))

; Eine Farbe (rgb-color) besteht aus ihrem
15 ; - Rot-Anteil 0..255 (red)
   ; - Grün-Anteil 0..255 (green)
   ; - Blau-Anteil 0..255 (blue)

```



```

20 ; (define-record-procedures rgb-color
;   make-color
;   color?
;   (color-red color-green color-blue))
25 ; _____

; Signatur für color-Records nicht in image2.rkt eingebaut.
; Roll our own...
(define rgb-color
  (signature (predicate color?)))
30

; Ist Farbe c bläulich?
(: bluish? (rgb-color -> boolean))
(define bluish?
35   (lambda (c)
     (< (/ (+ (color-red c) (color-green c) (color-blue c))
            3)
         (color-blue c))))

40 ; Worker:
; Pixel aus Hintergrund bg scheint durch, wenn der
; entsprechende Pixel im Vordergrund fg bläulich ist.
; Arbeite die Pixellisten von fg und bg synchron ab
; Annahme: fg und bg haben identische Länge!
45 (: bluescreen ((list-of rgb-color) (list-of rgb-color) ->
  (list-of rgb-color)))
(define bluescreen
  (lambda (fg bg)
    (cond ((empty? fg)
           empty)
50         ((pair? fg)
           (make-pair
             (if (bluish? (first fg))
                 (first bg)
                 (first fg))
             (first fg)))))

```

```

55         (bluescreen (rest fg) (rest bg))))))

; Wrapper:
; Mische Vordergrund fg und Hintergrund bg nach
; Bluescreen-Verfahren
(: mix (image image -> image))
60 (define mix
  (lambda (fg bg)
    (let ((fg-h (image-height fg))
          (fg-w (image-width fg))
          (bg-h (image-height bg))
          (bg-w (image-width bg)))
      65 (if (and (= fg-h bg-h)
                 (= fg-w bg-w))
              (color-list->bitmap
                (bluescreen (image->color-list fg)
                           (image->color-list bg))
                fg-w
                fg-h)
              (violation "Dimensionen_von_Vorder-/Hintergrund_
                          verschieden")))))
70
75 ; Yoda vor seine Hütte auf Dagobah setzen

```



```
(mix yoda dagobah) ~~~>
```

Generierung aller natürlichen Zahlen (vgl. gemischte Daten)

Eine natürliche Zahl (natural) ist entweder

- die 0 (zero)
- der Nachfolge (succ) einer natürlichen Zahl

$$\mathbb{N} = \{0, (\text{succ}(0)), (\text{succ}(\text{succ}(0))), \dots\}$$

*Konstruktoren*

```
(: zero natural)
(define zero 0)
(: succ (natural -> natural))
(define succ (lambda (n) (+ n 1)))
```

Vorgänger (pred), definiert für  $n > 0$

```
(: pred (natural -> natural))
(define pred
  (lambda (n) (- n 1)))
```

Bedingte algebraische Eigenschaft (für check-property):

```
(==> <p> <t>)
```

Nur wenn  $\langle p \rangle \rightsquigarrow\# t$  ist, wird Ausdruck  $\langle t \rangle$  ausgewertet und getestet  $\langle t \rangle \rightsquigarrow\# t$

#### Codebeispiel 25: ==> als Einschränkungoperator

```
; Eigenschaft nur auswerten, wenn n > 0 (==>)
(check-property
  (for-all ((n natural))
    (==> (> n 0)
      (= (succ (pred n)) n))))
```

Beispiel für Rekursion auf natürlichen Zahlen: Fakultät

$$0! = 1$$

$$n! = n \cdot (n-1)!$$

$$3! = 3 \cdot 2!$$

$$= 3 \cdot 2 \cdot 1!$$

$$= 3 \cdot 2 \cdot 1 \cdot 0!$$

$$= 3 \cdot 2 \cdot 1 \cdot 1$$

$$= 6$$

$$10 = 3628800$$

#### Codebeispiel 26: Fakultät rekursiv

```
; Berechne n!
(: factorial (natural -> natural))
(check-expect (factorial 0) 1)
(check-expect (factorial 3) 6)
```

```

5 (check-expect (factorial 10) 3628800)

(define factorial
  (lambda (n)
    (cond ((= n 0) 1)
          ((> n 0) (* n (factorial (- n 1)))))))
10

```

Konstruktionsanleitung für Prozeduren über natürlichen Zahlen:

```

(:<f> (natural -> <t>))
  (define <f>
    (lambda (n)
      (cond ((= n 0) ...)
            ((> n 0) ... (<f> (- n 1)) ...))))
5

```

Beobachtung:

- Im letzten Zweig ist  $n > 0 \rightarrow$  pred angewandt
- $(\langle f \rangle \ (- \ n \ 1))$  hat die Signatur  $\langle t \rangle$

Satz:

Eine Prozedur, die nach der Konstruktionsanleitung für Listen oder natürliche Zahlen konstruiert wurde *terminiert immer* (= liefert immer ein Ergebnis).

(Beweis in Kürze)

### Codebeispiel 27: Fehlerhafte Rekursionen

```

; Fehlerhaft: kein Fortschritt im rekursiven Aufruf
; => potentiell "unendliche" Reduktion
(define unfactorial
  (lambda (n)
    (cond ((= n 0) 1)
          ((> n 0) (* n (unfactorial n))))))
5

; Fehlerhaft: kein definierter Abbruch der Rekursion
; => Abbruch der Reduktion bei n = 0 ("cond: alle Tests
    ergaben #f")
10 (define not-factorial
    (lambda (n)

```



```
(cond ((> n 0) (* n (not-factorial (- n 1)))))
```

merken  
 $(3 \cdot (2 \cdot (1 \cdot 0!)))$

Die Größe eines Ausdrucks ist proportional zum Platzverbrauch des Reduktionsprozesses im Rechner

⇒ Wenn möglich Reduktionsprozesse, die *konstanten* Platzverbrauch - unabhängig von Eingabeparametern - benötigen

## 9.6.2015

Beobachtung: `(factorial 10)`.

```
(* 10(* 9(* 8(* 7(* 6(factorial 5)))))
= (*(*(*(*(* 109)8)7)6)(factorial 5)) ~> (* 30240(factorial 5))
```

Assoziativität von ·

→ Multiplikationen können vorgezogen werden :-)

Idee: Führe Multiplikation sofort aus. Schleife des Zwischenergebnis (*akkumulieren-des Argument*) durch die ganze Berechnung. Am Ende erhält der Akkumulator das Endergebnis.

Beispiel: Berechne 5!

```
(: fac-worker (natural natural -> natural))
```

n	acc	
-1 ↗ 5	1 ↘ · 5	neutrales Element
-1 ↗ 4	5 ↘ · 4	
-1 ↗ 3	20 ↘ · 3	
-1 ↗ 2	60 ↘ · 2	
-1 ↗ 1	120 ↘ · 1	
-1 ↗ 0	120	

```
; Berechne n!
; Wrapper
5 (: fac (natural -> natural))
  (check-expect (fac 0) 1)
```

```

(check-expect (fac 3) 6)
(define fac
  (lambda (n)
10    (fac-worker n 1)))

; Berechne n! (mit Zwischenergebnis/Akkumulator acc),
  endrekursiv
; Worker
(define fac-worker
15  (lambda (n acc)
    (cond ((= n 0) acc)
          ((> n 0) (fac-worker (- n 1) (* n acc))))))

```

Ein Berechnungsprozess ist *iterativ*, falls seine Größe konstant bleibt.

Damit:

`factorial` nicht iterativ

`fac-worker` iterativ

Wieso ist `fac-worker` iterativ?

Der Rekursive Aufruf ersetzt den aktuell reduzierten Aufruf *vollständig*. Es gibt keinen *Kontext* (umgebenden Ausdruck), der auf das Ergebnis des rekursiven Aufrufs "wartet"

Kontext des rekursiven Aufrufs in:

```

- factorial:                                     (* n □)
- fac-worker:                                   keiner

```

Eine Prozedur ist *endrekursiv* (tail call), wenn sie keinen Kontext besitzt. Prozeduren, die nur endrekursive Prozeduren beinhalten, heißen selber endrekursiv. Endrekursive Prozeduren generieren *iterative* Berechnungsprozesse

```
(: rev ((list-of %a)) -> (list-of %a))
```

#### Codebeispiel 28: Liste xs umdrehen

```

; Aufwand:  $\frac{1}{2} \times n \times (n + 1)$  Aufrufe von make-pair wenn xs die
  Länge n hat
(: rev ((list-of %a) -> (list-of %a)))

```

```

5 (check-expect (rev empty) empty)
  (check-expect (rev (list 1 2 3 4)) (list 4 3 2 1))

(define rev
  (lambda (xs)
10    (cond ((empty? xs) empty)
          ((pair? xs)
           (cat (rev (rest xs)) (list (first xs)))))))

```

Beobachtung: von `(rev (from-to 11000))`

1000 · make-pair

$(\text{cat } (\text{list } 1000 \dots 2) (\text{list } 1))$   
 $(\text{cat } (\text{list } 1000 \dots 3) (\text{list } 2))$

→ Aufrufe von `make-pair`:  $1000+999+998+\dots+1$

$\sum_{i=1}^n i = \frac{n \cdot (n+1)}{2}$  Quadratische Aufrufe :-)

Konstruiere iterative Listenumkehrfunktion `backwards`:

```
(: backwards-worker ((list-of %a) (list-of %a) -> (list-of %a)))
```

n	acc
rest ✓ (list 123)	(list) ↘ make-pair
rest ✓ (list 23)	(list 1) ↘ make-pair
rest ✓ (list 3)	(list 21) ↘ make-pair
empty	(list 321)

Mittels `letrec` lassen sich Werte an lokale Namen binden.

```
(letrec
  ((⟨id1⟩ ⟨e1⟩) ...
   (⟨idn⟩ ⟨en⟩)) ⟨e⟩)
```

Die Ausdrücke  $\langle e_1 \rangle, \dots, \langle e_n \rangle$  und  $\langle e \rangle$  dürfen sich auf die Namen  $\langle id_1 \rangle \dots \langle id_n \rangle$  beziehen

### Codebeispiel 29: Effizientere Variante eine Liste umzudrehen

```

; Wrapper
(: backwards ((list-of %a) -> (list-of %a)))

5 (check-expect (backwards empty) empty)
  (check-expect (backwards (list 1 2 3 4)) (list 4 3 2 1))

(define backwards
  (lambda (xs)
10    (letrec
        ((loop (lambda (acc)
                  (if (empty? xs)
                      acc
                      (loop (cat (rev (rest xs)) (list (first xs)))))))
          (loop empty))))

```

```

; Liste xs umdrehen (mit Akkumulator acc, endrekursiv)
; Worker
; Aufwand: n Aufrufe von make-pair, wenn xs die Länge n hat
15 (letrec ((backwards-worker
          (lambda (xs acc)
            (cond ((empty? xs) acc)
                  ((pair? xs)
                   (backwards-worker (rest xs) (make-pair
                                           (first xs) acc)))))))
20 (backwards-worker xs empty)))

```

## 11.6.2015

### *Induktive Definition*

Konstante Definition der natürlichen Zahlen  $\mathbb{N}$ .

Definition: (Peano Axiome)

- (P1)  $0 \in \mathbb{N}$
- (P2)  $\forall n \in \mathbb{N} : \text{succ}(n) \in \mathbb{N}$
- (P3)  $\forall n \in \mathbb{N} : \text{succ}(n) \neq 0$
- (P4)  $\forall n, m \in \mathbb{N} : \text{succ}(n) = \text{succ}(m) \Leftrightarrow n = m$

TODO: "Plot" mit Punkten und Pfeilen

(P5) Für jede Menge  $M \subset \mathbb{N}$  mit  $0 \in M$

und  $\forall n : (n \in M \Rightarrow \text{succ}(n) \in M)$ , gilt  $M = \mathbb{N}$

" $\mathbb{N}$  enthält nicht mehr als die 0 und die durch  $\text{succ}()$  generierten Elemente

"Nicht ist sonst in  $\mathbb{N}$ ,

TODO: Plot von zwei Kreisen ineinander Beweisschema der *vollständigen Induktion*

Sei  $P(n)$  eine Eigenschaft einer Zahl  $n \in \mathbb{N}$

(: P (natural -> boolean))

Ziel:  $\forall n \in \mathbb{N} : P(n)$

Definiere  $M = \{n \in \mathbb{N} \mid P(n)\} \subset \mathbb{N}$

M enthält die Zahlen n für die  $P(n)$  gilt

*Induktionsaxiom*

Falls

$$0 \in M$$

und

$$\forall n : (n \in M \Rightarrow \text{succ}(n) \in M)$$

dann

$$M \in \mathbb{N}$$

Induktionsstart

Falls

$$P(0)$$

und

$$\forall (P(n) \Rightarrow P(\text{succ}(n)))$$

Induktionsschritt

dann

$$\forall n \in \mathbb{N} P(n)$$

*Beispiel:*

$$1 = 1$$

$$1 + 3 = 4$$

$$1 + 3 + 5 = 9$$

$$1 + 3 + 5 + 7 = 16$$

...

$$P(n) = \underbrace{\sum_{i=0}^n (2i+1)}_{\substack{\text{Summe der} \\ \text{ersten } n \\ \text{ungeraden Zahlen}}} \stackrel{!}{=} (n+1)^2$$

*Induktionsschluss  $P(0)$*

$$\sum_{i=0}^0 (2i+1) = 2 \cdot 0 + 1 = (0+1)^2 \checkmark$$

Induktionsschritt  $\forall n (P(n) \Rightarrow P(n+1))$

$$\begin{aligned} \sum_{i=0}^{n+1} (2i+1) &\stackrel{!}{=} \sum_{i=0}^n (2i+1) + (2(n+1)+1) \\ &\stackrel{i.v.}{=} (n+1)^2 + 2n+3 \\ &= n^2 + 4n + 4 \\ &= ((n+1)+1)^2 \checkmark \end{aligned}$$

*Beispiel:*

```

(define factorial
  (lambda (k)
    (if
      5
      (= k 0) 1
      (* k (factorial (- k 1))))))

```

$P(x) \equiv (\text{factorial } n) = \boxed{n!}$

$\boxed{x}$ : (Racket Repräsentation für  $x \in \mathbb{N}$ )

Zeige:  $\forall n \in \mathbb{N} : P(n)$

Induktionsbasis  $P(0)$

$(\text{factorial } (\boxed{0}))$

$\rightsquigarrow^* ((\text{lambda } (k) \dots) \boxed{0})$

$\rightsquigarrow (\text{if } (= \boxed{0} 0) 1 \dots)$

$\rightsquigarrow (\text{if } \#t 1 \dots)$

$\rightsquigarrow 1 = \boxed{0}! \checkmark$

Induktionsschritt:  $\forall n : (P(n) \rightarrow P(n+1))$

$(\text{factorial } \boxed{n+1})$

$\rightsquigarrow^* ((\text{lambda } (n) \dots) \boxed{n+1})$

$\rightsquigarrow (\text{if } (= \boxed{n+1} 0) 1 \dots (\dots))$

$\rightsquigarrow (\text{if } \#f 1 \dots (\dots))$

$\rightsquigarrow (* \boxed{n+1} (\text{factorial } (- \boxed{n+1} 1)))$

$\rightsquigarrow (* \boxed{n+1} (\text{factorial } \boxed{n}))$

$\stackrel{iv}{=} (* \boxed{n+1} n!)$

$= (n+1)! \checkmark$

Unter der  
Annahme, dass  
tatsächlich  
Subtraktion  
implementiert ist

*Beispiel:*

Jede durch die Konstruktionsanleitung für Funktionen über natürliche Zahlen konstruierte Funktion liefert ein Ergebnis (*terminiert immer*)

```
(define f
  (lambda (n)
    (if
      (= n 0) base
      (step (f (n-1)) n))))

(: base natural)
(: step (natural natural -> natural)) Bsp: step → (lambda (x y) (* x
y))
```

Dann gilt  $P(n) = (f\ n)$  terminiert (Mit Ergebnis der Signatur *natural*)

Zeige  $\forall n \in \mathbb{N} : P(n)$

*Induktionsbasis*  $P(0)$ :

(f 0)

$\rightsquigarrow$  (if (= 0 0) base ...)

$\rightsquigarrow$  (if #t base

$\rightsquigarrow$  base ✓

*Induktionsschritt*  $\forall n : (P(n) \rightarrow P(n+1))$

(f n+1)

$\rightsquigarrow$  (if (= n+1 0) base ... (step ...))

$\rightsquigarrow$  (if #f base ... (step ...))

$\rightsquigarrow$  (step (f (- n+1 1)) n+1)

$\rightsquigarrow$  (step  $\underbrace{(f\ n)}_{\text{terminiert}}$  n+1)

$\Rightarrow$  (step (f n) n+1) terminiert

*Definition:* (Listen. endliche Folge)

Die Menge  $M^*$  (= Listen mit Elementen aus  $M$  + `list-of M` ist *induktiv* definiert

- Nicht leere Liste (L1)  $\text{empty} \in M^*$
- $\in M^*$  (L2)  $\forall x \in M, xs \in M^*$
- (L3) Nichts sonst in  $M^*$

`(make-pair x  
xs)`

Beweisschema *Listeninduktion*

So  $P(xs)$  eine Eigenschaft von Listen über  $M$ .

`(: P ((list-of M) -> boolean))`

Induktionsanfang

Falls  $P(\text{empty})$   
und  
 $\forall x \in M, xs : P(xs) \Rightarrow (P(xs) \Rightarrow (P(\text{make-pair } x \text{ } xs)))$   
dann

Induktionsschritt

$\forall xs \in M^* : P(xs)$



# 16.6.2015

Beispiel:

```
(define cat
  (lambda (xs ys)
    (cond
      ((empty? xs) ys)
      ((pair? xs) (make-pair (first xs) (cat
                               (rest xs) ys))))))
```

$$\left\{ \begin{array}{l} (1) \text{ cat empty ys} = \text{ys} \\ (2) \text{ (cat xs empty)} = \text{xs} \\ (3) \text{ (cat (cat xs ys) ys)} = \text{(cat xs (cat ys zs))} \end{array} \right.$$

Beweise:

$(M^*, \text{cat}, \text{empty})$   
ist ein Monoid

$$(1) \text{ (cat empty ys)} \xrightarrow{\star} \text{ys} \checkmark$$

$$(2) P(xs) = \text{(cat xs empty)} = \text{xs}$$

Induktionsanfang  $P(\text{empty})$

$$\text{(cat empty empty)} \stackrel{(1)}{=} \text{empty} \checkmark$$

Induktionsschritt  $\forall x \in M : P(xs) \Rightarrow P(\text{(make-pair x xs)})$

```
(define make-pair mp)
```

```
(cat (mp x xs) empty)
```

$$\xrightarrow{\star} \text{(mp (first (mp x xs)) (cat (rest (mp x xs)) empty))}$$

$$\xrightarrow{\sim} \text{(mp x (cat xs empty))}$$

$$\stackrel{iv.}{=} \text{(mp x xs)} \checkmark$$

(3) Listeninduktion über xs (ys, zs  $\in M^*$  beliebig)

$$P(xs) \equiv \text{(cat (cat xs ys) zs)} = \text{(cat xs (cat ys zs))}$$

Induktionsanfang  $P(\text{empty})$

```
(cat (cat empty ys) zs)
```

$$\xrightarrow{\sim} \stackrel{(1)}{=} \text{(cat ys zs)}$$

$$\rightsquigarrow \stackrel{(1)}{=} (\text{cat empty } (\text{cat } ys \ zs)) \checkmark$$

*Induktionsschritt*  $\forall x \in M : P(xs) \Rightarrow P((\text{make-pair } x \ xs))$   
 $(\text{cat } (\text{cat } (\text{mp } x \ xs) \ ys) \ zs))$

$$\rightsquigarrow^* (\text{cat } (\text{mp } x \ (\text{cat } xs \ ys)) \ zs)$$

$$\rightsquigarrow^* (\text{mp } (\text{cat } (\text{cat } xs \ ys)) \ zs)$$

$$\stackrel{iv.}{=} (\text{mp } (\text{cat } (\text{cat } xs \ ys) \ zs))$$

$$\rightsquigarrow (\text{cat } (\text{mp } x \ xs) \ (\text{cat } ys \ zs)) \checkmark$$

*Beispiel:* Interaktion von `length` und `cat` (Distributivität)

```
(define length
  (lambda (xs)
    (cond
      ((empty? xs) 0)
      ((pair? xs) (+ 1
                     (length (rest xs)))))))
```

$P(xs): (\text{length } (\text{cat } xs \ ys)) = (+ (\text{length } xs) (\text{length } ys)),$   
 $ys \in M^*$  beliebig.

*Induktionsbasis:*

$(\text{length } (\text{cat empty } ys))$

$$\stackrel{(1)}{=} (\text{length } ys)$$

$$\stackrel{+}{=} (+ \ 0 \ (\text{length } ys))$$

$$\rightsquigarrow (+ \ (\text{length empty}) \ (\text{length } ys)) \checkmark$$

*Induktionsschritt*

$(\text{length } (\text{mp } x \ xs) \ ys)$

$$\text{cat } \rightsquigarrow^* (\text{length } (\text{mp } x \ (\text{cat } xs \ ys)))$$

$$\text{length } \rightsquigarrow^* (+ \ 1 \ (\text{length } (\text{rest } (\text{mp } x \ (\text{cat } xs \ ys)))))$$

$$\text{rest } \rightsquigarrow^* (+ \ 1 \ (\text{length } (\text{cat } xs \ ys)))$$

*iv.*  $\equiv (+\ 1(+\ (\text{length}\ xs)\ (\text{length}\ ys)))$

ass.  $\stackrel{(+)}{=} (+\ (+\ 1(\text{length}\ xs)\ (\text{length}\ ys)))$

$\text{length} \xrightarrow{\star} (+\ (\text{length}\ (\text{mp}\ x\ xs)\ (\text{length}\ ys))) \checkmark$

### Prozeduren höherer Ordnung

(higher-order procedures)

```
; Filtere Liste xs nach Elementen, die Prädikat p? erfüllen
; (Prozedur höherer Ordnung: Parameter p? ist selbst eine
  Funktion)
(: filter ((%a -> boolean) (list %a) -> (list %a)))
(define filter
5   (lambda (p? xs)
      (cond
        ((empty? xs) empty)
        ((pair? xs)
10         (if (p? (first xs))
                (make-pair (first xs)
                           (filter p? (rest xs)))
                (filter p? (rest xs)))))))
```

Wert des Parameters `p?` ist Prozedur  $\Rightarrow$  kann angewendet werden

## 18.6.2015

Zwei Arten von *Higher Order Prozeduren* (H.O.P)

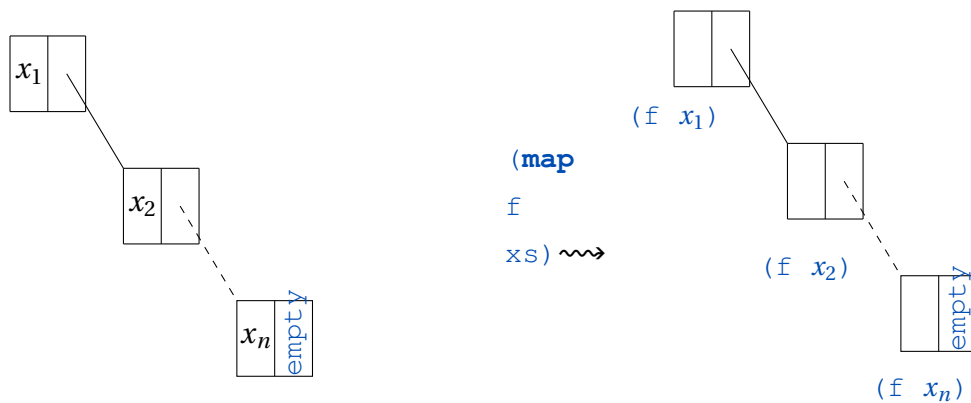
(1) akzeptieren, Prozeduren als Parameter oder/und

(2) liefern Prozeduren als Ergebnis

`filter` ist vom Typ (1).

H.O.P vermeiden Duplizierung von Code und führen zu kompakteren Programmen, verbesserte Lesbarkeit und verbesserte Wartbarkeit.

Beispiel: `(map f xs)`



```

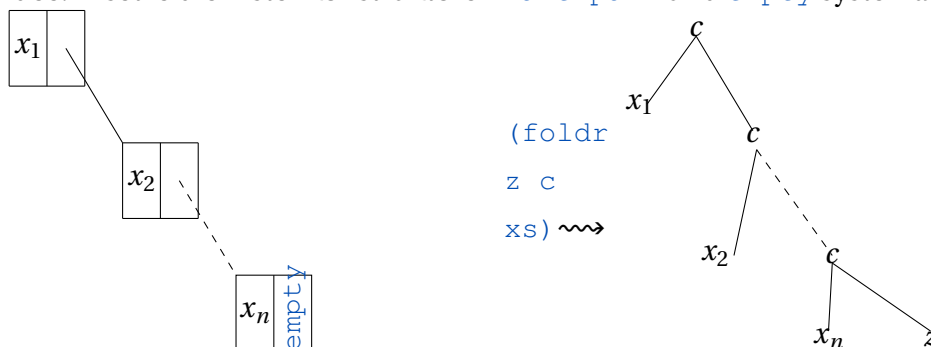
;Wende f auf Elemente von Liste xs an
(: map ((%a -> %b) (list-of %a) -> (list-of %a)))
(define map
  (lambda (f xs)
    (cond
      ((empty? xs) empty)
      ((pair? xs) (make-pair (f (first xs)
                               (map f (rest xs)))))))

```

5

Allgemeine Transformation von Listen *Listenfaltung* (list folding)

Idee: Ersetze die Listenkonstruktoren `make-pair` und `empty` systematisch.



`(foldr z c xs)` wirkt als Spinetransformer

- `empty`  $\rightsquigarrow$  `z`
- `make-pair`  $\rightsquigarrow$  `c`
- Eingabe : Liste `(list-of %a)`
- Ausgabe : im Allgemeinen *keine* Liste mehr: `%b`

```

;Falte Liste xs bzgl. c und z
(: foldr (%b (%a %b -> %b) (list-of %a) -> %b))
(define foldr
  (lambda (z c xs)
    (cond
      ((empty? xs) z)
      ((pair? xs)
       (c (first xs)
           (foldr z c (rest xs)))))))

```

Beispiele: Listenreduktion mit `foldr`

```

(: sum ((list-of number) -> number))
(define sum (lambda (xs) (foldr 0 + xs)))

```

Beispiel: Länge einer Liste durch Listenreduktion TODO: Bild Plotten

```

; Listenreduktion via foldr: Länge der Liste xs
(: my-length ((list-of %a) -> natural))
(define my-length
  (lambda (xs)
    (foldr 0 (lambda (x l) (+ 1 l)) xs)))

```

### Codebeispiel 30: Fold und seine Anwendungen

```

; Listenreduktion via foldr: Summe der Liste xs
(: my-sum ((list-of number) -> number))
(define my-sum
  (lambda (xs)
    (foldr 0 + xs)))

; Listenreduktion via foldr: Produkt der Liste xs
(: my-product ((list-of number) -> number))
(define my-product
  (lambda (xs)
    (foldr 1 * xs)))

; Listenreduktion via foldr: Maximum der Liste xs
(: my-maximum ((list-of number) -> number))
(define my-maximum
  (lambda (xs)
    (foldr -inf.0 max xs)))

; Identität (auf Listen), implementiert via foldr
(: my-id ((list-of %a) -> (list-of %a)))

```

```
(define my-id
  (lambda (xs)
    (foldr empty make-pair xs)))

25 ; Reimplementation von append via foldr
(: my-append ((list-of %a) (list-of %a) -> (list-of %a)))
(define my-append
  (lambda (xs ys)
    (foldr ys make-pair xs)))

30 ; Reimplementation von map via foldr
(: my-map ((%a -> %b) (list-of %a) -> (list-of %b)))
(define my-map
  (lambda (f xs)
35    (foldr empty
              (lambda (y ys) (make-pair (f y) ys))
              xs)))

; Reimplementation von reverse via foldr
40 (: my-reverse ((list-of %a) -> (list-of %a)))
(define my-reverse
  (lambda (xs)
    (foldr empty
              (lambda (y ys) (append ys (list y)))
45              xs)))

; Listenreduktion via foldr: Länge der Liste xs
(: my-length ((list-of %a) -> natural))
(define my-length
50 (lambda (xs)
    (foldr 0 (lambda (x l) (+ 1 l)) xs)))

; Reimplementation von filter mittels foldr
(: my-filter ((%a -> boolean) (list-of %a) -> (list-of %a)))
55 (define my-filter
  (lambda (p? xs)
    (foldr empty
              (lambda (y ys) (if (p? y)
                                (make-pair y ys)
60                                ys))
              xs)))
```

## 23.6.2015

Teachpack 'universe' nutzt H.O.P Animationen (Sequenzen von Bildern/Szenen) zu definieren.

```
(big bang
  (<init>)
  (ontick <tock>)
  (todraw <render><w><h>))
```

- (<init> %a) Startzustand
- (: <tock> (%a -> %a)) Funktion, die einen neuen Zustand aus alten Zustand berechnet
- (: <render> (%a -> image)) Funktion, die aus dem aktuellen eine Szene berechnet (wird in Fenster mit Dimension  $\langle w \rangle \cdot \langle h \rangle$  Pixel angezeigt)
- Beim Schließen der Animation wird der letzte Zustand zurückgegeben

### Codebeispiel 31: Ein animierter Zähler

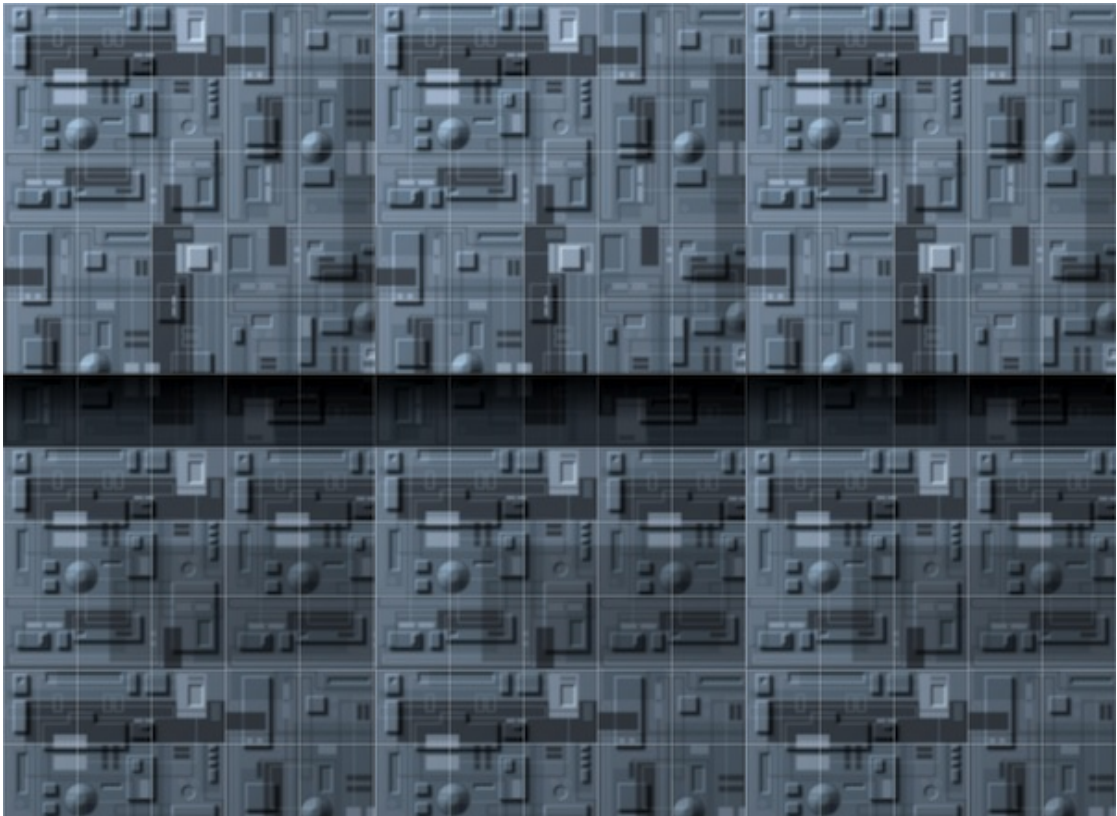
```
; Erstellung von Animationen mit Teachpack "universe"
; (1) Zähler

5 (define scene (natural -> image))
  (define scene
    (lambda (t)
      (text (number->string t) 100 "red")))
(big-bang 0
10   (on-tick (lambda (t) (+ t 1)))
      (to-draw scene 200 100))
```

### Codebeispiel 32: Ein animiertes Raumschiff

```
; Erstellung von Animationen mit Teachpack "universe"
; (2) X-Wing Fighter + Scrolling Death Star

(define death-star
```



5



```

(define x-wing                                     )

; Erhalte einfachen Scrolling-Effekt durch Herausschneiden von
  Teilbildern
10 ; aus dem Bild der Todessternoberfläche
; (zu crop und overlay: siehe Dokumentation des Teachpack
  "image2")
(: scroll-death-star (natural -> image))
(define scroll-death-star
  (lambda (t)
15   (overlay x-wing

```



```

(crop (modulo (* 8 t) 200) 0 400 440
      death-star)))
(big-bang 0
  (on-tick (lambda (t) (+ t 1))))

```

Ausgabe der römischen Episoden nummern für Film  $f$ : `(roman (film-episode f))`

Gesuchte Funktion ist *Komposition* von zwei existierenden Funktionen:

- (1) Erst `film-episode` anwenden, *dann*
- (2) Wende `roman` auf das Ergebnis von (1) an

Komposition von Prozeduren allgemein:

```

( ( compose f g ) x)  $\equiv$  (f (g x))

```

neue Prozedur realisiert  
Komposition von f und g

[ Mathematisch  $(\text{compose } f \ g) \equiv f \circ g$  ]

```

(: compose (%b -> %c) ($a -> %b) -> (%a -> %c))
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x))))))

```

### Codebeispiel 33: Zweites und Drittes Element durch Combined

```

; Greife auf das zweite Element der Liste xs zu
(: second ((list-of %a) -> %a))
(check-expect (second (list 1 2 3)) 2)
5 (check-expect (second (string->strings-list "SCF")) "C")
(define second
  (lambda (xs)
    ((compose first rest) xs)))

10 ; Greife auf das dritte Element der Liste xs zu
(: third ((list-of %a) -> %a))
(check-expect (third (list 1 2 3)) 3)
(check-expect (third (string->strings-list "SCF")) "F")
15 (define third
  (lambda (xs)
    ((compose first (compose rest rest)) xs)))

```

`repeat`: n-fache Komposition von `f` auf sich selbst  
(n-fache Anwendung von `f`, Exponentiation)

$$f^0 = \text{id} \qquad (\text{id} \equiv (\text{lambda } (x) x))$$

$$f^n = f \circ f^{n-1}$$

```
(: repeat (natural (%a -> %a) -> (%a -> %a)))
(define repeat
  (lambda (n f)
    (cond
      5      ((= n 0) (lambda (x) x)
              (> n 0) (compose f (repeat (- n 1) f))))))
;Greife auf das n-te Element der Liste xs zu
(: nth (natural (list-of %a) -> %a))
(define nth
10  (lambda (n xs)
      ((compose first (repeat (- n 1) rest))xs)))
```

### Codebeispiel 34: Gibt die Funktion + zurück

```
; Funktionen, die ihre Argument schrittweise konsumieren
; Konsumiert Argumente x,y in einem Schritt (eine Reduktion
  von apply_)
5 (: plus (number number -> number))
(define plus
  (lambda (x y)
    (+ x y)))

10 ; Konsumiert Argumente x,y in zwei Schritten (zwei Reduktionen
    von apply_).
; Nach dem ersten Schritt ist nur Argument x festgelegt,
  Ergebnis ist eine
; Funktion, die das zweite Argument y erwartet.
(: add (number -> (number -> number)))
(define add
15  (lambda (x)
      (lambda (y)
        (+ x y))))
```

```

20 (map (add 1) (list 1 2 3 4 5 6 7 8 9 10)); ~> (list 2 3 4 5 6 7
      8 9 10 11)
    (map (add 10) (list 1 2 3 4 5 6 7 8 9 10)); ~> (list 11 12 13 14
      15 16 17 18 19 20)

```

Reduktion: ((add 1) 41)

~>  
eval<sub>id</sub> ((lambda (x) (lambda (y) (+ x y)) 1) 41)

~>  
apply<sub>λ</sub> [lambda(x)] ((lambda (y) (+ 1 y)) 41)  
Funktion die 1 auf  
ihr Argument anwenden

~>  
apply<sub>λ</sub> [lambda(y)] (+ 1 41)

**25.6.2015**

$(\%a \%b \rightarrow \%c) \longrightarrow \text{Applikation auf zwei Argumente (Signaturen } \%a, \%b) \longrightarrow \%c$   
 $\text{Curry} \downarrow \uparrow \text{uncurry} \quad = \quad \uparrow \downarrow$   
 $(\%a \rightarrow (\%b \rightarrow \%c)) \rightarrow \text{App. auf Arg. (Sig. } \%a) \rightarrow (\%b \%c) \text{ App. auf Arg. (Sig. } \%b) \rightarrow \%c$

*Currying* (Haskell B. Curry, Moses Schönfinkel)

Anwendung einer Prozedur auf ihr erstes Argument liefert Prozedur der restlichen Argumente.

Jede n-stellige Prozedur lässt sich in eine alternative curried Prozedur transformieren, die in n Schritte jeweils ein Argument konsumiert. Uncurry ist die umgekehrte Transformation.

```

5  (: curry ((%a %b -> %c) -> (%a -> (%b -> %c))))
   (define curry
     (lambda (f)
       (lambda (x)
         (lambda (y)
           (f x y)))))
   (: uncurry (%a -> (%b -> %c) -> (%a %b -> %c)))
   (define uncurry
10  (lambda (f)
     (lambda (x y)
       ((f x) y))))

```

Es gilt für jeder Prozedur p:

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

„Schönfinkel Isomorphismus“

### Codebeispiel 35: Einfache Anwendung von Curry

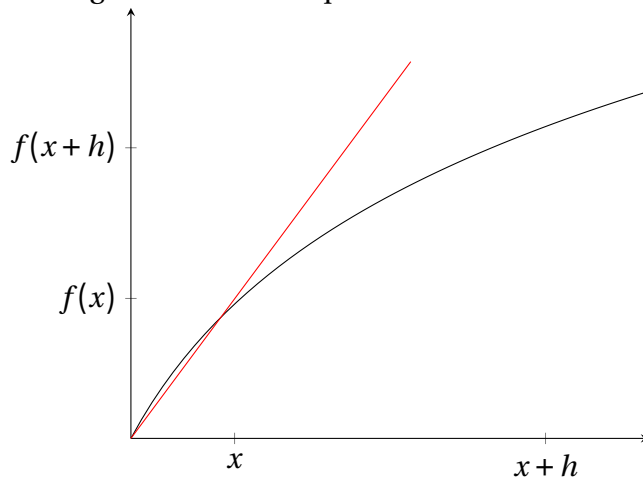
```

5  (map ((curry +) 1) (list 1 2 3 4 5 6 7 8 9 10))
   ; ~~~~ (list 2 3 4 5 6 7 8 9 10 11)
   (map ((curry +) 10) (list 1 2 3 4 5 6 7 8 9 10))
   ; ~~~~ (list 11 12 13 14 15 16 17 18 19 20)
   (filter ((curry =) 2) (list 1 2 3 4 5 4 3 2 1))
   ; ~~~~ (list 2 2)

```

*Erinnerung:* Bestimmung der ersten Ableitung der reellen Funktion durch Bildung des Differentialquotienten

Bildung des Differentialquotienten:



$$\frac{f(x+h) - f(x)}{h}$$

Differenzenquotient

Steigung(↗)

$$\lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} = f'(x)$$

Operator ' (Ableitung konsumiert Funktionen und produziert Funktion) → ' ist höherer Order

### Codebeispiel 36: Ableitungen mit Curry

```

; Differenzenquotienten von f (mit Differenz h)
(: diffquot (real (real -> real) -> (real -> real)))
(define diffquot
5   (lambda (h f)
      (lambda (x)
        (/ (- (f (+ x h)) (f x))
           h))))

10 ; Berechne Differenzenquotienten mit Differenz h = 0.00001
; ((derive f) x) ≡ (f' x)
(: derive ((real -> real) -> (real -> real)))
(define derive
15   ((curry diffquot) 0.00001))

; Beispielfunktion: f1(x) = x² + 2x
(: f1 (real -> real))
(define f1
20   (lambda (x) (+ (* x x x)
                    (* 2 x))))

; Ableitung von f1(x)
; f1'(x) = 3x² + 2
25 (check-property

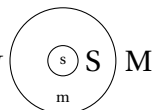
```

```

(for-all ((x real))
  (expect-within ((derive f1) x)
    (+ (* 3 x x) 2)
    0.01)))
30 ; Ableitung von f(x) = atan(x)
; f'(x) = 1 / (1 + x2)
(check-property
  (for-all ((x real))
    (expect-within ((derive atan) x)
35      (/ 1
        (+ 1 (* x x)))
        0.01)))

```

Charakteristische Funktion einer Menge  $S \subset M$



Charakteristische Funktion für S:  $(:\chi_s (M \rightarrow \text{Boolean}))$

$$\chi_s(x) = \begin{cases} \#t & x \in S \\ \#f & \text{sonst} \end{cases}$$

$$\chi_s(m) = \#f \quad \chi_s(s) = \#t$$

Idee Repräsentiere  $S \subseteq$  durch Prozedur  $(M \rightarrow \text{boolean})$  und Mengenoperation auf Prozeduren (H.O.P)

### Codebeispiel 37: Grundlagen Mengenimplementierung

```

; Charakteristische Funktion (M -> boolean) als Repräsentation
; für eine Menge  $S \subseteq M$ 
(define set-of
5   (lambda (t)
     (signature (t -> boolean))))

;  $S_{42} = \{ x \in \mathbb{Z} \mid x > 42 \}$ 
(: S42 (set-of integer))
10 (define S42
    (lambda (x)
      (> x 42)))

; Leere Menge  $\emptyset$ 
15 (: empty-set (set-of %a))
(define empty-set
  (lambda (x)
    #f))

```

```

20 ; Ist Element x in der Menge S (x ∈ S)?
  (: set-member? (%a (set-of %a) -> boolean))
  (define set-member?
    (lambda (x S)
      (S x)))

```

$\therefore$  Darstellung unendlicher Mengen ( $S_42 = \{x \in \mathbb{Z} \mid x > 42\}$ )

$\therefore$  Mengenoperationen ( $\cup, \cap, \setminus$ ) in *Konstanter Zeit*

Element  $x$  in Menge  $S$  einfügen:

$$\chi_{S \cup \{x\}}(y) = \begin{cases} \#f & x = y \\ \chi_S(y) & \text{sonst} \end{cases}$$

### Codebeispiel 38: Erweiterte Mengenoperationen

```

; Element x in Menge S hinzufügen: S ∪ {x}
(: set-insert (number (set-of number) -> (set-of number)))
5 (define set-insert
  (lambda (x S)
    (lambda (y)
      (or (= y x)
          (S y)))))

10 ; Test: die leere Menge enthält kein Element
  (check-property
    (for-all ((x integer))
      (boolean=? (set-member? x empty-set) #f)))

15 ; Test: die Menge ∅ ∪ {x} enthält x
  (check-property
    (for-all ((x integer))
      (set-member? x (set-insert x empty-set))))

20 ; Konstruiere {1,2,3,4,5} = (((∅ ∪ {1}) ∪ {2}) ∪ {3}) ∪ {4})
  ∪ {5})
  (: 1-to-5 (set-of integer))
  (define 1-to-5
    (set-insert
25 5
      (set-insert
        4

```

30

```

(set-insert
  3
  (set-insert
    2
    (set-insert
      1 empty-set))))))

```

## 30.6.2015

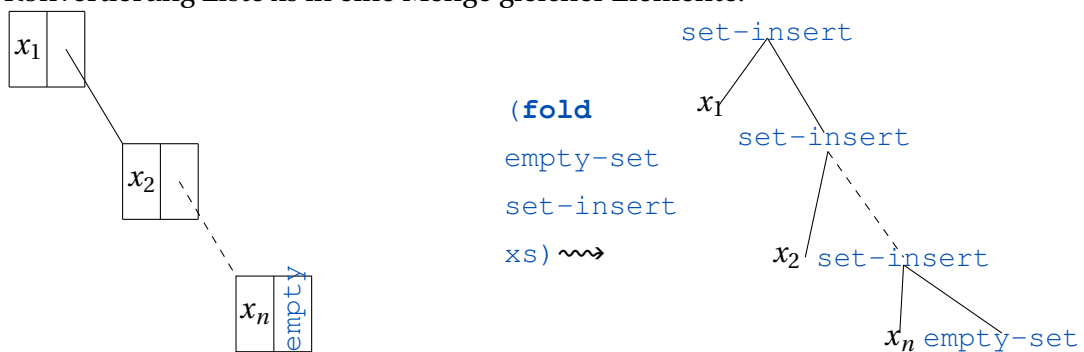
5

```

; Beispiel (Streams Stream):
; Erzeuge neuen Stream durch die Anwendung von f

```

Konvertierung Liste xs in eine Menge gleicher Elemente.



Codebeispiel 39: Konvertiert eine Liste zu einer Menge

```

(lambda (xs)
  (fold empty-set set-insert xs))

; Beispiel: Konstruiere {1,2,...,10}
(: 1-to-10 (set-of integer))
(define 1-to-10
  (list->set (list 1 2 3 4 5 6 7 8 9 10)))

; Element x aus Menge S löschen
(: set-delete (number (set-of number) -> (set-of number)))

```



Vereinigung:  $\chi_{S \cup T}(x) = \chi_S(x) \vee \chi_T(x)$ .

Weitere Mengenoperationen analog:

#### Codebeispiel 40: Mengenoperationen $\setminus, \cup, \cap, \Delta$

```

5      (lambda (x S)
        (lambda (y)
          (if (= y x)
              #f
              (S y))))))

; S U T
; x ∈ S U T ⇔ x ∈ S ∨ x ∈ T
10  (: set-union ((set-of %a) (set-of %a) -> (set-of %a)))
(define set-union
  (lambda (S T)
    (lambda (x)
      (or (S x) (T x))))))

15
; S ∩ T
; x ∈ S ∩ T ⇔ x ∈ S ∧ x ∈ T
(: set-intersect ((set-of %a) (set-of %a) -> (set-of %a)))
20  (define set-intersect
  (lambda (S T)
    (lambda (x)
      (and (S x) (T x))))))

25
; S \ T
; x ∈ S \ T ⇔ x ∈ S ∧ x ∉ T
(: set-difference ((set-of %a) (set-of %a) -> (set-of %a)))
30  (define set-difference
  (lambda (S T)
    (lambda (x)
      (and (S x) (not (T x))))))

;

```

Charakteristische Funktion zur Repräsentation Mengen:

- (1) Performance: `set-member` hat lineare Laufzeit bei mit `set-insert` konstruierte Mengen (wie Liste!)
- (2) Vorteile:

+ unendliche Mengen darstellbar

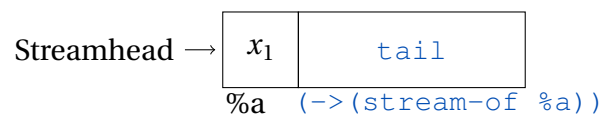
+ Mengenoperationen in konstanter Zeit durchführbar

(3) Nachteile

- Elemente sind nicht aufzählbar

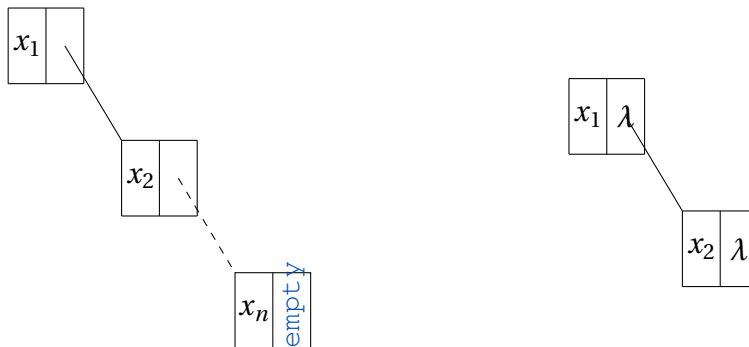
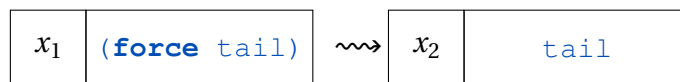
`Streams (stream-of %a)`: unendliche Ströme von Elementen `x`, mit Signatur `%a`. Ein Stream ist ein Paar:

- Erst eine Ausführung des Tails (**force**) erzeugt nächstes Stream-Element (faher



auch *lazylist*).

Vergleich:



Verzögerte Auswertung eines Ausdrucks (*delayed Evaluation*):

- (**delay** `e`): Verzögere die Auswertung des Ausdruckes `e` und liefere "Versprechen" (`promise`) `e` bei Bedarf später auswerten zu können.

`(delay e)  $\equiv$  (lambda ()` `e` `)`  
 $\uparrow$   
nicht  
ausgewertet

`(force p)` Erzwingt Auswertung des `promise`. `p` liefert Wert zurück

```
(: force ((-> %a)->%a))
(define force
  (lambda (p)
    (p)))
```

## Codebeispiel 41: Streams

```
(lambda (t)
  (signature (-> t))))

5 ; Verzögerte Auswertung (delay)
;
; Variante 1:
; (delay e) (lambda () e)
;
10 ; Variante 2 (nutzt selbstdefinierte Scheme-Syntax-Regel,
    verfügbar ab
; Sprachebene "DMdA - fortgeschritten"):
;
; (define-syntax delay
;   (syntax-rules ()
15 ;     (_ e)
;     (lambda () e))))

; Erzwungene Auswertung
(: force ((promise %a) -> %a))
20 (define force
  (lambda (p)
    (p)))

; Beispiel:
25 ; Promise (werde 41+1 berechnen, falls gefordert)
(: will-evaluate-to-42 (promise natural))
(define will-evaluate-to-42
  (lambda () ; oder äquivalent mit Variante 2: (delay (+ 1
    41))
    (+ 41 1)))

30 ; Verzögerte Ausführung...
will-evaluate-to-42
; ... und erzwungene Ausführung
(force will-evaluate-to-42)

35 ; Polymorphe Paare (isomorph zu `pair')
```

```

(: make-cons (%a %b -> (cons-of %a %b)))
(: head ((cons-of %a %b) -> %a))
(: tail ((cons-of %a %b) -> %b))
40 (define-record-procedures-parametric cons cons-of
    make-cons
    cons?
    (head
     tail))
45
; Ein Stream besteht aus
; - einem ersten Element (head)
; - einem Promise, den Rest des Streams generieren zu können
    (tail)
(define stream-of
50   (lambda (t)
      (signature (cons-of t (promise (stream-of t))))))

; Beispiel:
; Stream mit Zahlen ab n erzeugen
55 (: from (number -> (stream-of number)))
(define from
  (lambda (n)
    (make-cons n (lambda () (from (+ n 1))))))

60 ; Beispiel (Stream Liste):
; Erste n Elemente des Streams str in eine Liste extrahieren
(: stream-take (natural (stream-of %a) -> (list-of %a)))

(check-expect (stream-take 5 (from 1)) (list 1 2 3 4 5))
65 (check-expect (stream-take 0 (from 1)) empty)

(define stream-take
  (lambda (n str)
    (if (= n 0)
70       empty
        (make-pair (head str)
                    (stream-take (- n 1) (force (tail str))))))

; Beispiel (Stream Stream):
75 ; Filtere Stream str bzgl. Prädikat p?
(: stream-filter ((%a -> boolean) (stream-of %a) -> (stream-of
  %a)))

(check-expect (stream-take 10

```

```

80      (stream-filter (lambda (x) (=
                           (remainder x 2) 0))
                      (from 1)))
      (list 2 4 6 8 10 12 14 16 18 20))

85 (define stream-filter
    (lambda (p? str)
      (if (p? (head str))
          (make-cons (head str)
                     (lambda () (stream-filter p? (force (tail
                                                             str)))))
          (stream-filter p? (force (tail str))))))

```

## 2.7.2015

Generiere den unendlichen Strom der Fibonacci Zahlen.

$\text{fib}(0) = 1$

$\text{fib}(1) = 1$

$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

1, 1, 2, 3, 5, 8, 13, 21, ...

↑ ab hier jeweils Summe der beiden Vorgänger

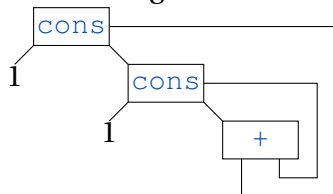
*Beobachtung:*

```

  1 1 2 3 5
+ 1 2 3 5
-----
  2 3 5 8

```

Stream-Diagramm zu fibs:



### Codebeispiel 42: Stream aller Fibonacci Zahlen

```

(: stream-zipWith ((%a %b -> %c) (stream-of %a) (stream-of %b)
  -> (stream-of %c)))
(define stream-zipWith
  (lambda (f str1 str2)

```

```

5      (make-cons (f (head str1) (head str2))
                  (lambda () (stream-zipWith f
                                              (force (tail str1))
                                              (force (tail
                                                    str2)))))))
10
; Die unendliche Folge der Fibonacci-Zahlen 1, 1, 2, 3, 5, ...
(: fibs (stream-of natural))
(check-expect (stream-take 10 fibs) (list 1 1 2 3 5 8 13 21 34
55))
(define fibs
15  (make-cons
    1
    (lambda ()
      (make-cons
20      1
      (lambda ()
        (stream-zipWith +
                        fibs
                        (force (tail fibs)))))))

```

Die Menge der Binärbäume  $T(M)$  ist induktiv definiert:

- (T1)  $\text{empty-tree} \in T(M)$
- (T2)  $\forall x \in M \text{ und } l, r \in T(M): (\text{make-node } l \ x \ r) \in T(M)$
- (T3) nichts sonst in  $T(M)$

*Hinweis:*

- Jeder Knoten  $(\text{make-node})$  in einem Binärbaum hat zwei Teilbäume sowie eine Markierung  $(\text{label})$ .
- Vergleiche:

$M^*$  und  $T(M)$

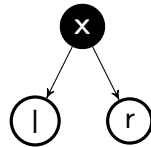
$\text{empty}$  und  $\text{empty-tree}$

$\text{make-pair}$  und  $\text{make-node}$

Visualisierung:

- `empty-tree` □

- `(make-node x l r)`



- Die Knoten mit Markierung  $x$  ist *Wurzel* (root) des Baumes
- Ein Knoten, der nur leere Teilbäume beinhaltet heißt *Blatt* (leaf). Alle anderen Knoten heißen *innere Knoten* (inner-nodes)

Beispiel für Binärbäume der Menge  $T(M)$

(Binär-) Bäume haben zahlreiche Anwendungen:

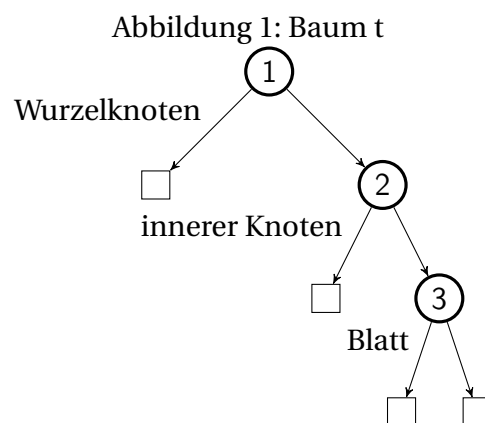
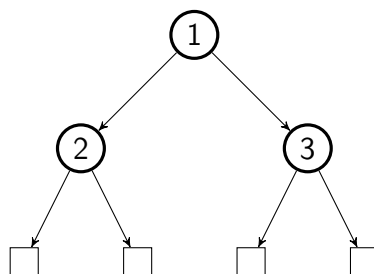


Abbildung 2: Baum  $t_2$  *balanciert*, alle Teilbäume auf einer Tiefe haben die selbe Anzahl an Knoten



- Suchbäume (z.B. Datenbanken)

- Datenkompression
- Darstellung von Termen (Ausdrücken)

Bäume sind *die* Induktiv definierte Datenstruktur

#### Codebeispiel 43: Verschiedene Bäume

```

; Ein Knoten (node) eines Binärbaums besitzt
; - einen linken Zweig (left-branch),
; - eine Markierung (label) und
; - einen rechten Zweig (right-branch)
5 (: make-node (%a %b %c -> (node-of %a %b %c)))
  (: node-left-branch ((node-of %a %b %c) -> %a))
  (: node-label       ((node-of %a %b %c) -> %b))
  (: node-right-branch ((node-of %a %b %c) -> %c))
(define-record-procedures-parametric node node-of
10  make-node
  node?
  (node-left-branch
   node-label
   node-right-branch))

15 ; Ein leerer Baum (empty-tree) besitzt
; keine weiteren Eigenschaften
  (: make-empty-tree (-> the-empty-tree))
(define-record-procedures the-empty-tree
20  make-empty-tree
  empty-tree?
  ())

; Der leere Baum (Abkürzung)
25 (: empty-tree the-empty-tree)
  (define empty-tree (make-empty-tree))

; Signatur für Binärbäume (btree-of t) mit Markierungen des
;   Signatur t
; (im linken/rechten Zweig jedes Knotens findet sich jeweils
;   wieder
30 ; ein Binärbaum)
  (define btree-of
    (lambda (t)
      (signature (mixed the-empty-tree
                        (node-of (btree-of t) t (btree-of t))))))
35 ;

```



```

;
;                                     zweifache Rekursion, s.
;   (list-of t)

40 ; Konstruiere Blatt mit Markierung x
   (: make-leaf (%a -> (btree-of %a)))
   (define make-leaf
     (lambda (x)
       (make-node empty-tree x empty-tree)))

45

; Beispiel: t1 (rechts-tief, listen-artig)
(: t1 (btree-of natural))
(define t1
50   (make-node empty-tree
               1
               (make-node empty-tree
                           2
                           (make-node empty-tree
                                       3
                                       empty-tree)))))

55

; Beispiel: t2 (balanciert)
(: t2 (btree-of natural))
60 (define t2
    (make-node (make-leaf 2)
               1
               (make-leaf 3)))

65

; Beispiel: Klassifikation von Star Wars Charakteren
; (left branch "no", right branch "yes")
(: classifier (btree-of string))
70 (define classifier
    (make-node (make-node (make-node (make-leaf "Han_Solo")
                                     "female?"
                                     (make-leaf "Padme_Amidala")))
               "droid?"
               (make-node (make-leaf "C-3PO")
                           "astromech?"
                           (make-leaf "R2D2"))))

75
    "force?"
    (make-node (make-node (make-leaf "Luke_Skywalker")

```

```

80                                     "prequel?"
                                       (make-leaf "Mace_Windu"))
    "dark_side?"
    (make-node (make-leaf "Emperor")
               "pilot?"
               (make-leaf "Darth_
                           Vader")))))

```

Die *Tiefe* (depth) eines Baumes ist die maximale Länge eines Weges von der Wurzel von  $t$  zu einem leeren Baum. Also:

```
(btree-depth empty-tree)  $\rightsquigarrow$  0
```

```
(btree-depth t1)  $\rightsquigarrow$  3
```

```
(btree-depth t2)  $\rightsquigarrow$  2
```

Schablone (gemischte Daten)

```

(: btree-depth ((btree-of %a) -> natural))
(define btree-depth
  (lambda (t)
    (cond ((empty-tree? t) 0)
          ((node? t) (+ 1
                        (max (btree-depth (node-left-branch t))
                            (btree-depth (node-right-branch t))))))
5

```

## 7.7.2015

### Codebeispiel 44: Die Größe eines Baumes

```

; Grösse (Anzahl Knoten) des Binärbaumes t
(: btree-size ((btree-of %a) -> natural))
(check-expect (btree-size empty-tree) 0)
5 (check-expect (btree-size t1) 3)
  (check-expect (btree-size t2) 3)
  (check-expect (btree-size classifier) 15)
(define btree-size
  (lambda (t)
10    (cond ((empty-tree? t)
            0)

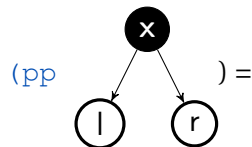
```

```
((node? t)
 (+ (btree-size (node-left-branch t))
```

Einschub: Pretty-Printing von Bäumen

Prozedur `(pp t)` erzeugt formatierten String für Binärbaum `t`.

```
(pp □) = "□"
```



Idee : Repräsentiere formatierten String als *Liste von Zeilen* (Strings).

- ⇒(1) Nutze `(string-append)` um Zeilen-String zu definieren (horizontale Konkatenation).
- (2) Nutze `(append)` um die einzelnen Zeilen zu einer Liste von Zeilen zusammenzusetzen (vertikale Konkatenation)

Erst direkt vor der Ausgabe werden die Zeilen-Strings zu einem auszugebenden String zusammengesetzt `(strings-list->string)`

#### Codebeispiel 45: Pretty Print eines Baumes

```
; Drucke Textrepräsentation des Baums t
(: print ((btree-of (mixed number string)) -> %void))
(define print
5   (lambda (t)
      (write-string (strings-list->string (pp t)))))

; Erzeuge Liste von Zeilen-Strings der Textrepräsentation des
  Baums t
(: pp ((btree-of (mixed number string)) -> (list-of string)))
10 (define pp
    (lambda (t)
      (cond
        ((empty-tree? t) (list "\n"))
        ((node? t)
15         (letrec ((lbl (node-label t))
```

```

(x (if (string? lbl) lbl (number->string
    lbl)))
(wx (string-length x))
(ppl (pp (node-left-branch t)))
(ppr (pp (node-right-branch t)))
20 (append (list (string-append x "--"
    (first ppr)))
    (map ((curry string-append)
        (string-append "" (replicate wx "_"))
        (rest ppr))
    (list (string-append "" (replicate wx "_")
        "\n"))
    (list (string-append "" (replicate (+ 1 wx)
        "-") (first ppl))
    25 (map ((curry string-append)
        (string-append "___" (replicate wx "_")
        "))) (rest ppl)))
    ))))

; Konkateniere String s genau n mal (Hilfsfunktion für pp)
30 (: replicate (natural string -> string))
(define replicate
  (lambda (n s)
    (cond ((= n 0) "")
          ((> n 0) (string-append s (replicate (- n 1) s)))))
  35

(check-property
  (for-all ((t (btree-of natural)))
    (==> (< (btree-size t) 10)
    40 (expect (print t) (write-string "\n")))))

```

### Induktion über Binärbäume

Sei  $P(t)$  eine Eigenschaft von Binärbäumen  $t \in T(M)$ , also  $(: P((btree-of M) \rightarrow \text{boolean}))$ .

Induktionsbasis

Falls `(empty-tree)` und

$$\forall x \in M, r, l \in T(M) : P(l) \wedge P(r) \Rightarrow P(\text{make-node } l \ x \ r)$$

Induktionsschritt

dann

$$\forall t \in T(M) : P(t)$$

Beispiel:

Zusammenhang zwischen Größe (`btree-size`) und Tiefe (`btree-depth`) eines Binärbaums  $t$ . („Ein Baum der Tiefe  $n$  enthält mindestens  $n$  und höchstens  $2^n - 1$  Knoten“).

$$P(t) \equiv (\text{btree-depth } t) \leq (\text{btree-size } t) \leq 2^{(\text{btree-depth } t)} - 1$$

Induktionsbasis  $P(\text{empty-tree})$

$(\text{size empty-tree})$

$$\rightsquigarrow 0$$

$$= 2^0 - 1 \checkmark$$

[depth]

Induktionsschritt  $(P(l) \wedge P(r) \Rightarrow P(\text{make-node } l \times r))$

$(\text{size } (\text{make-node } l \times r))$

$$\rightsquigarrow (\text{size } l) + 1 + (\text{size } r)$$

[size]

$$= 2^{(\text{depth } l)} - 1 + 1 + 2^{(\text{depth } r)} - 1$$

[i.v.]

$$= 2^{(\text{depth } l)} + 2^{(\text{depth } r)} - 1$$

$$\leq 2 \cdot \max(2^{(\text{depth } l)}, 2^{(\text{depth } r)}) - 1$$

$$= 2 \cdot 2^{\max(\text{depth } l, \text{depth } r)} - 1$$

$$\rightsquigarrow 2^{(\text{depth } (\text{make-node } l \times r))} - 1 \checkmark$$

[depth]

Wie müsste sich `btree-fold` eine fold-Operation für *Binärbäume* verhalten? Tree Transformer für Baum  $t$ : TODO: Bild

```
(: btree-fold (%a (%a %b %a -> %a) (tree-of %b )-> %b))
(define btreefold
  (lambda (z f t)
    (cond ((empty? t) z)
          ((node? t) (f (btree-fold z f (node-left-branch t))
                        (node-label t)
                        (btree-fold z f (node-right-branch t)))))))
```

**9.7.2015**Codebeispiel 46: Beispiele von `btree-fold`

```

; Tiefe des Baums t
(: new-btree-depth ((btree-of %a) -> natural))
(define new-btree-depth
5   (lambda (t)
      (btree-fold 0
                  (lambda (d1 x d2) (+ 1 (max d1 d2)))
                  t)))

10  ; Grösse des Baums t
(: new-btree-size ((btree-of %a) -> natural))
(define new-btree-size
15   (lambda (t)
      (btree-fold 0
                  (lambda (s1 x s2) (+ s1 1 s2))
                  t)))

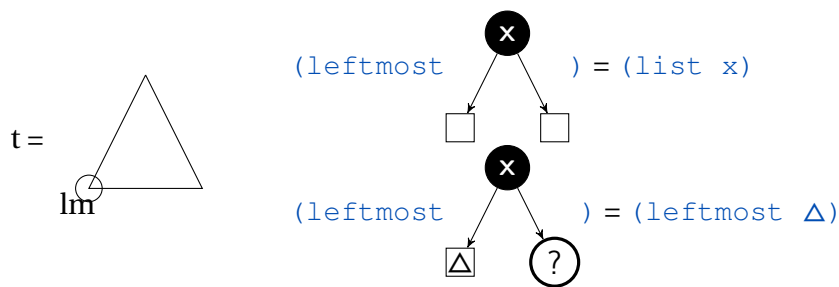
; Ist x eine Markierung im Baum t?
(: btree-member? (number (btree-of number) -> boolean))
20  (define btree-member?
      (lambda (x t)
        (btree-fold #f
                    (lambda (m1 y m2) (or m1 (= x y) m2))
                    t)))

25  ; Spiegelbild des Baums t
(: btree-mirror ((btree-of %a) -> (btree-of %a)))
(define btree-mirror
30   (lambda (t)
      (btree-fold empty-tree
                  (lambda (t1 x t2) (make-node t2 x t1))
                  t)))

```

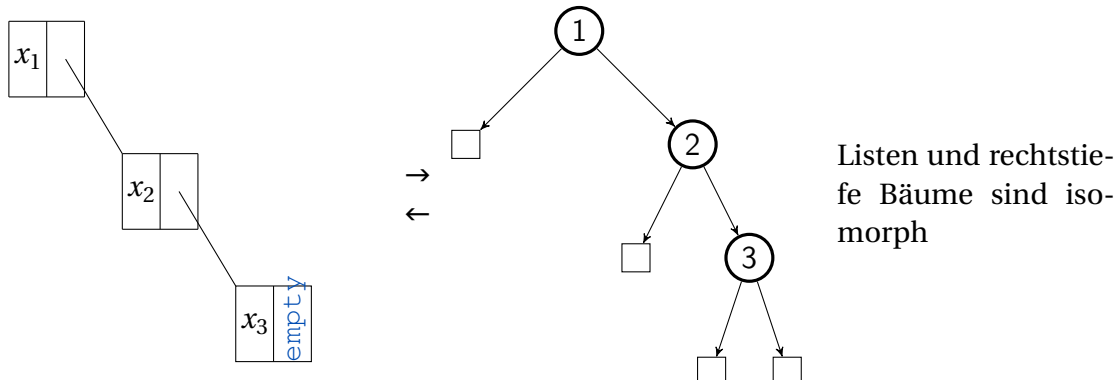
Bestimme die Markierung im links-Außen im Baum `t` (oder `empty` falls `t` leer ist).<sup>1</sup>

<sup>1</sup>Nach dem Prinzip von „How to Replace Failure by a List of Successes“, Wadler 1985



```
(: leftmost (btree-of %a) -> (list-of %a))
(check-expect (leftmost empty-tree) empty)
(check-expect (leftmost t1) (list 1))
(check-expect (leftmost t2) (list 2))
```

```
5 (define leftmost
  (lambda (t)
    (btree-fold
     empty
     (lambda (l1 x l2)
       (if (empty? l1)
           (list x)
           l1))
     t)))
```



### Codebeispiel 47: Listen sind rechtstiefe Bäume

```
; Listen und rechts-tiefe Bäume sind isomorph

; Konvertiere Liste xs in rechts-tiefen Baum
5 (: list->btree ((list-of %a) -> (btree-of %a)))
(check-expect (list->btree empty) empty-tree)
(check-expect (list->btree (list 1 2 3)) t1)
(define list->btree
```

```

10  (lambda (xs)
      (fold empty-tree
            (lambda (x t) (make-node empty-tree x t))
            xs)))

15  ; Konvertiere rechts-tiefen Baum t in Liste
    ; (ÜBUNG: (: right-deep? ((btree-of %a) -> boolean))) )
    (: btree->list ((btree-of %a) -> (list-of %a)))
    (check-expect (btree->list empty-tree) empty)
    (check-expect (btree->list t1) (list 1 2 3))
20  (define btree->list
      (lambda (t)
        (btree-fold empty
                    (lambda (xs1 x xs2) (make-pair x xs2))
                    t))) ;
25                                ; empty-list, da t rechts-tief

    ; Listen und rechts-tiefe Bäume sind isomorph:
    (check-property
30  (for-all ((xs (list natural)))
    (expect (btree->list (list->btree xs)) xs)))

```

Ein *Tiefendurchlauf* (*depth-first-traversal*) eines Baumes *t* sammelt die Markierungen der Teilbäume *l*, *r* des Knotens.

*n* = *make-node* *l* *x* *r* werden *vor* *x* eingesammelt (Durchlauf zuerst in der Tiefe).  
Je nachdem ob *x*:

(a) zwischen, (b) vor, (c) nach den Markierungen von *l*, *r* eingezeichnet wird, erhält man einen

(a) *inorder* traversal      **123**

(b) *preorder* traversal    **213**

(c) *postorder* traversal    **132**

```

    (: inorder ((btree-of %a) -> (list-of %a)))
    (define inorder
      (lambda (t)
        (btree-fold
5      empty

```

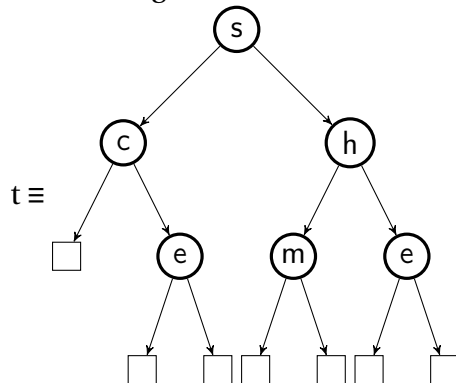


```

10 (lambda (xs1 x xs2)
    (append
      xs1 ❶
      (list x) ❷
      xs2) ❸
    t)))

```

Ein *Breitendurchlauf* (*breadth-first-traversal*) eines Baumes *t* sammelt die Markierungen der Knoten ebenenweise von der Wurzel ausgehend auf.



```
(levelorder t) ⇨ (list "s" "c" "h" "e" "m" "e")
```

Idee: Gegeben sei eine Liste von Bäumen

(1) Sammle die Liste der Markierungen der Wurzeln der nicht leeren Bäume in *ts* auf

```
(roots ts)
```

(2) Bestimme Liste *ts'* der nicht leeren Teilbäume der Bäume in *ts* (*subtrees ts*)

(3) Führe (1) rekursiv auf *ts'* aus

(4) Konkateniere die Listen aus (1) und (3)

### Codebeispiel 48: Breitendurchlauf

```

5 ; Repräsentiert 2 + (3 ⊗ 4)
  (: term (btree-of string))
  (define term
    (make-node (make-leaf "2")
               "+"
               (make-node (make-leaf "3")
                           "⊗"
                           (make-leaf "4")))))

```

```

10 (check-expect (inorder term) (list "2" "+" "3" "⊗" "4")) ; C
   (check-expect (preorder term) (list "+" "2" "⊗" "3" "4")) ;
   Scheme
   (check-expect (postorder term) (list "2" "3" "4" "⊗" "+")) ;
   Forth

15 ; Breitendurchlauf eines Baumes

; Breitendurchlauf für die Liste der Bäume ts
(: traverse ((list-of (btree-of %a)) -> (list-of %a)))
20 (define traverse
    (lambda (ts)
      (cond ((empty? ts) empty)
            ((pair? ts) (append (roots ts)
                                (traverse (subtrees ts)))))))

25 ; Liste der Wurzelmarkierungen der nicht-leeren Bäume in ts
(: roots ((list-of (btree-of %a)) -> (list-of %a)))
(define roots
  (lambda (ts)
    (map node-label
30       (filter node? ts))))

; Liste der Teilbäume der nicht-leeren Bäume in ts
(: subtrees ((list-of (btree-of %a)) -> (list-of (btree-of
35 %a))))
(define subtrees
  (lambda (ts)
    (flatten
      (map (lambda (t) (list (node-left-branch t)
40                      (node-right-branch t)))
           (filter node? ts)))))

; Breitendurchlauf für Baum t
; (Wrapper für traverse)
(: levelorder ((btree-of %a) -> (list-of %a)))
45 (define levelorder
    (lambda (t)
      (traverse (list t))))

50 ; Beispielbaum

```

```

(: scheme (btree-of string))
(define scheme
  (make-node (make-node empty-tree
                        "c"
                        (make-leaf "e"))
    "s"
    (make-node (make-leaf "m")
      "h"
      (make-leaf "e"))))
(check-expect (levelorder scheme) (string->strings-list
  "scheme"))

```

## 14.7.2015

*Zeichenkodierungen* bilden Zeichen auf Sequenzen von Bits ab. Derzeit sind Codes fester Länge sehr beliebt.

ASCII (Code 0-127, 7Bit, American Standard Code for Information Interchange)

ISO8859-1 (Code 0-255, 8Bit, besteht aus lateinischen und Steuerzeichen)

Unicode (20Bit, codiert Zeichen aus 129 aktuellen und historischen Sprachen, inkl. Klingon)

Beispiel Zeichen '€' : 0000 0010 0000 1010 1100

*Huffman-Codes* nutzen Bitsequenzen variabler Länge.

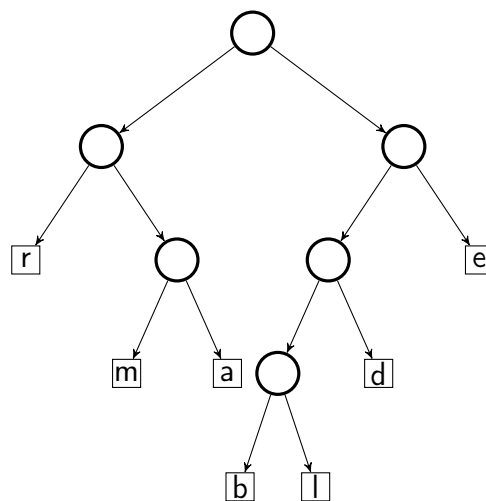
Idee: Zeichen mit hoher Frequenz werden mit weniger Bits codiert, als seltene Zeichen. ⇒ Datenkompression

Huffman-Codes sind Binärbäume mit markierten Blättern:

Beispiel Huffman-Codes für: "erdbeermarmelade"

Code für Zeichen x: Pfad von Wurzel bis Blatt mit Markierung x

- Abstieg in linken Teilbaum : Bit 0
- Abstieg in Rechten Teilbaum: Bit 1



Zeichen	Frequenz	Code
e	5	11
r	3	00
m	2	010
a	2	101
b	1	1000
l	1	1001

Huffman-Codes sind präfix frei, die Bits eines Zeichens sind niemals Präfix eines anderen Zeichens → eindeutige Codierung.

|11|00|101|1000|11|11... ⇒ erdbee...

(Länge 42 Bit, Unicode = 320 Bit)

Einsetzung in JPEG, MP3, ZIP

### Codebeispiel 49: Implementierung von Huffman-Codes

```

; Huffman-Trees zur Datenkompression

; Beispiel: Unicode Zeichen ""
5 (: euro-symbol string)
(define euro-symbol "\U020AC")

; Ein Blatt eines Huffman-Tree (huff-leaf)
10 ; - trägt eine Markierung (label):

```

```

(: make-huff-leaf (%a -> (huff-leaf-of %a)))
(: huff-leaf-label ((huff-leaf-of %a) -> %a))
(define-record-procedures-parametric huff-leaf huff-leaf-of
  make-huff-leaf
15  huff-leaf?
    (huff-leaf-label))

; Ein innerer Knoten eines Huffman-Tree (huff-node) besitzt
; - einen linken Teilbaum (left) und
20 ; - einen rechten Teilbaum (right):
(: make-huff-node (%a %b -> (huff-node-of %a %b)))
(: huff-node-left ((huff-node-of %a %b) -> %a))
(: huff-node-right ((huff-node-of %a %b) -> %b))
(define-record-procedures-parametric huff-node huff-node-of
25  make-huff-node
    huff-node?
    (huff-node-left
     huff-node-right))

30 ; Signatur (huff-tree-of t): Huffman-Tree mit Blättern
; mit Markierungen der Signatur t
(define huff-tree-of
  (lambda (t)
    (signature (mixed (huff-leaf-of t)
35                      (huff-node-of (huff-tree-of t)
                                      (huff-tree-of t))))))

; Ein Bit eines Zeichencodes
(define bit
  (signature (one-of 0 1)))
40

; Beispiel: Huffman-Tree für Text "erdbeermarmelade" (s. oben)
(: code-for-erdbeermarmelade (huff-tree-of string))
45 (define code-for-erdbeermarmelade
  (make-huff-node
    (make-huff-node
      (make-huff-leaf "r")
      (make-huff-node
50      (make-huff-leaf "m")
      (make-huff-leaf "a"))))
    (make-huff-node
      (make-huff-node
        (make-huff-leaf "e")
        (make-huff-leaf "l"))
      (make-huff-node
        (make-huff-leaf "a")
        (make-huff-leaf "d"))))
  )

```

```

55 (make-huff-node
    (make-huff-leaf "b")
    (make-huff-leaf "l"))
    (make-huff-leaf "d")
    (make-huff-leaf "e"))))
60 (print code-for-erdbeermarmelade)

```

Prozeduren zur Huffman-Codierung:

(1) Decodierung einer Bitsequenz

```
(:huff-decode ((huff-tree-of string) (list-of bit) -> string))
```

(2) Codierung eines Strungs

```
(huff-tree-of string) string -> (list-of bit)
```

$\forall \text{ string} : (\text{huff-decode } (\text{huff-encode } s)) = s$

(3) Huffman-Tree für gegebenen Text erstellen

```
(:huffman-code (string -> (huff-tree-of string)))
```

Decodieren eines huffman-codierten string (= eine Liste aus Bits)

Plan: Baue

```
(: decode ((huff-tree-of %a) (huff-tree-of %a) (list-of bit) -> (list-of %a)))
```

(1)  $(\text{decode } \Delta_{ht} \boxed{x} [...\text{Bits}...]) = (\text{make-pair } x \ (\text{decode } \Delta_{ht} \Delta_{ht} [...\text{Bits}...]))$

(2)  $(\text{decode } \Delta_{ht} \Delta [ ]) = \text{empty}$

(3a)  $(\text{decode } \Delta_{ht} \begin{array}{c} \bullet \\ / \quad \backslash \\ l \quad r \end{array} [0... \text{Bits}...]) = (\text{decode } \Delta_{ht} \Delta_l [...\text{Bits}...])$

(3b)  $(\text{decode } \Delta_{ht} \begin{array}{c} \bullet \\ / \quad \backslash \\ l \quad r \end{array} [1... \text{Bits}...]) = (\text{decode } \Delta_{ht} \Delta_r [...\text{Bits}...])$

Zu (1): Neueinstieg an Wurzel des Huffman-Tree  $\Rightarrow$  Wurzel des Huffman-Tree als Parameter durchführen.

## Codebeispiel 50: Decodierung einer Liste von Bits in einen String

```

; Worker
; Decodiere Bitsequenz bits bzgl. Huffman-Tree t, Wurzel
5 ; des Huffman-Codes ist ht
(: decode ((huff-tree-of %a) (huff-tree-of %a) (list-of bit)
  -> (list-of %a)))
(define decode
  (lambda (ht t bits)
    (cond
      10 ((huff-leaf? t) ;
          (1)
          (make-pair (huff-leaf-label t)
                     (decode ht ht bits)))
      ((huff-node? t)
       (cond
         15 ((empty? bits) ;
              (2)
              empty)
         ((pair? bits)
          (cond
            20 ((= 0 (first bits)) ;
                  (3a)
                  (decode ht (huff-node-left t) (rest bits)))
              ((= 1 (first bits)) ;
                  (3b)
                  (decode ht (huff-node-right t) (rest
                    bits))))))))))

; Wrapper
25 ; Decodiere Bitsequenz bits bzgl. Huffman-Tree ht
(: huff-decode ((huff-tree-of string) (list-of bit) -> string))
(define huff-decode
  (lambda (ht bits)
    (strings-list->string (decode ht ht bits))))
30

; Beispiel
(: erdbeermarmelade-bits (list-of bit))
(define erdbeermarmelade-bits
35 (list 1 1 0 0 1 0 1 1 0 0 0 1 1 1 1 0 0 0 1 0 0 1 1
        0 0 0 1 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1))

```

```

40 (: bread-bits (list-of bit))
    (define bread-bits
      (list 1 0 0 0 0 0 1 1 0 1 1 1 0 1))

; Funktioniert Decodierung wie erwartet?
45 (check-expect
    (huff-decode code-for-erdbeermarmelade erdbeermarmelade-bits)
    "erdbeermarmelade")

(check-expect
  (huff-decode code-for-erdbeermarmelade bread-bits)
  "bread")

```

### Huffman-Codierung eines Strings als Liste von Bits

Plan:

- a) Codierung eines Zeichens  $c$ . Suche mittels einer Tiefensuche von der Wurzel des Huffman-Trees aus. Protokolliere den Pfad beim Absteig als Liste von Bits.

Frage : Wie soll ich reagieren, wenn die Tiefensuche zu einem Blatt mit  $x \neq c$  führt?

Idee : Verfolge an inneren Knoten `(make-huff-node l r)` jeweils linken und rechten Teilbaum. Suche nach  $c$  schlägt fehl in entweder  $l$  oder  $r$ . Bei Fehlschlag liefern wir die leere Bitliste. Beachte `(append empty xs) = xs` und `(append xs empty) = xs`


```

(: encode ((huff-tree-of %a) (list-of bit)) %a -> (list-of bit)))
                                     akku      Zeichen

```

(1) `(encode [x] [...Bits...] c) = empty`

(2) `(encode [c] [...Bits...] c) = (reverse [...Bits...])`

(3) `(encode  [...Bits...] c) = (append (encodel [0...Bits...] c) (encoder [1...Bits...] c))`

- b) Codiere Zeichen des Strings  $s$  mit `encode` (mittels `map`), verbinde einzelne Bitlisten mit `flatten` (bzw. `concat`)

### Codebeispiel 51: Encodierung eines String in Bit

```

; Codierung

```



```

; Worker
5 ; Codiere einzelnes Zeichen c mittels Huffman-Tree t
  (: encode ((huff-tree-of string) (list-of bit) string ->
    (list-of bit)))
  (define encode
    (lambda (t bits c)
      (cond
10      ((huff-leaf? t)
        (if (string=? (huff-leaf-label t) c)
            (reverse bits)
            empty))
        ; (2)
        ;
        (1)
        ((huff-node? t)
          ; (3)
          (append (encode (huff-node-left t) (make-pair 0 bits)
            c)
            (encode (huff-node-right t) (make-pair 1 bits)
              c))))))

; Wrapper
; Codiere Text s mittels Huffman-Tree ht
20 (: huff-encode ((huff-tree-of string) string -> (list-of bit)))
  (define huff-encode
    (lambda (ht s)
      (flatten
        (map (lambda (c) (encode ht empty c))
          (string->strings-list s))))
25

; Beispiel
(huff-decode code-for-erdbeermarmelade
30 (huff-encode code-for-erdbeermarmelade
  "erdbeermarmelade"))

; Eigenschaft:
; Wenn wir String s mit demselben Huffman-Tree codieren und
  dann
; decodieren, erhalten wir wieder s:
35 (check-property
  (for-all
    ((xs (list-of (one-of "e" "r" "d" "b" "m" "a" "l"))))
    (let ((s (strings-list->string xs)))

```

40

```
(expect (huff-decode code-for-erdbeermarmelade
                  (huff-encode
                    code-for-erdbeermarmelade s))
        s)))
```

## 16.7.2015

Erstellung eines Huffman-Tree eines gegebenen Textes txt. Plan:

(H1) Stelle Häufigkeit des Vorkommens jedes Zeichen in txt fest. Sortiere in Reihenfolge steigender Häufigkeit (*occurences*)

Beispiel: (*occurences* "erdbeermarmelade")  $\rightsquigarrow$

```
(list <"l" 1> <"b" 1> ... <"e" 5>)
```

Vorkommen  $\langle i \ n \rangle$  (*occur*) : Ding  $i$  (*item*) kommt mit Häufigkeit  $n$  (*freq*) vor.

(H2) Baue Huffman-Tree von Blättern her auf. Initialisiere den Aufbau: Für Vorkommen  $\langle i \ n \rangle$  konstruiere  $\langle \boxed{i} \ n \rangle$

(H3) Die beiden Huffman-Trees die die seltensten Zeichen repräsentieren stehen am Anfang der Liste. ( $\text{list } \langle \boxed{i} \ n \rangle \langle \boxed{j} \ m \rangle$ ). Konstruktion des Huffman-Tree, die diese *Invariante* bewahren.

Iteration: Wiederhole bis Liste Länge 1 hat:

- Fasse Vorkommen  $\langle \Delta_i \ n \rangle$  und  $\langle \Delta_r \ m \rangle$  zu einem Vorkommen  $\langle \bullet \ n+m \rangle$
- Sortiere dieses Vorkommen bzgl. Häufigkeit  $n+m$  in die Restliste ein.



(H4) Baum  $ht$  in ( $\text{list } \langle \Delta_{ht} \ n \rangle$ ) ist der gesuchte Huffman-Tree

### Codebeispiel 52: Implementation der Erzeugung von Huffman-Trees

5

```
; Konstruktion von Huffman-Trees für gegebenen Text

; Zu (H1)

; Worker
```

```

; Grupperie Liste von String xs bzgl. eq?
(: group-by ((%a %a -> boolean) (list-of %a) (list-of %a)
            -> (list-of (list-of %a))))
10
(define group-by
  (lambda (eq? gs xs)
    (cond
      ((empty? xs)
       (list gs))
      ((pair? xs)
       (if (eq? (first gs) (first xs))
           (group-by eq?
                     (make-pair (first xs) gs)
                     (rest xs))
           (make-pair gs (group-by eq?
                                   (list (first xs))
                                   (rest xs)))))))
15
20
25 ; Wrapper
; Gruppieren die Liste von Strings xs
(: group ((list-of string) -> (list-of (list-of string))))
(define group
  (lambda (xs)
    (cond
      ((empty? xs)
       empty)
      ((pair? xs)
       (let ((ys (sort (curry string=? ) xs)))
         (group-by string=? (list (first ys)) (rest ys))))))
30
35

; "Star Wars scroll": Intro zu Episode IV (A New Hope)
40 ; Video im Web: http://vimeo.com/1589113
(define starwars-scroll "\
starwarsepisodeivnewhope\
alongtimeagoinagalaxyfarfaraway\
aperiodofcivilwar\
45 rebelspacehipsstrikingfromahiddenbasehavewontheirfirstvictory\
againsttheevilgalacticempireduringthebattlerebelspiesmanagedto\
stealsecretplanstotheempiresultimateweaponthedeathstar\
anarmoredspacestationwithenoughpowertodestroyanentireplanet\
pursuedbytheempiressinisteragentsprincessleiaraceshome\
50 aboardherstarshipcustodianofthestolenplansthatcansaveherpeople\
andrestorefreedomtothegalaxy")

```

```

55 ; Ein Vorkommen (occur) ist charakterisiert durch
; - ein Ding (item) und
; - seine Häufigkeit (freq)
(: make-occur (%a %b -> (occur-of %a %b)))
(: occur-item ((occur-of %a %b) -> %a))
60 (: occur-freq ((occur-of %a %b) -> %b))
(define-record-procedures-parametric occur occur-of
  make-occur
  occur?
  (occur-item
65   occur-freq))

; Vergleiche zwei Vorkommen bzgl. Häufigkeit
(: occur<=? ((occur-of %a natural) (occur-of %a natural) ->
  boolean))
(define occur<=?
70   (lambda (o1 o2)
     (<= (occur-freq o1)
         (occur-freq o2))))

75 ; Liste der Zeichen (und ihrer Häufigkeiten) in s,
; nach aufsteigender Häufigkeit sortiert
(: occurrences (string -> (list-of (occur-of string natural))))
(define occurrences
80   (lambda (s)
     (sort (curry occur<=?)
           (map (lambda (g)
                  (make-occur      ; i,n
                    (first g)      ; i
50                  (length g)))   ; n
               (group (string->strings-list s))))))

85 ; Zu (H2)

90 ; Konstruiere Liste der Huffman-Tree-Blätter aus Liste os
; der Häufigkeiten der einzelnen Zeichen:
(: huffman-leaves ((list-of (occur-of %a
                                     %b)) ->

```

```

95         (list-of (occur-of (huff-tree-of %a) %b))))
(define huffman-leaves
  (lambda (os)
    (map (lambda (o)
          (make-occur (make-huff-leaf (occur-item o))
100              (occur-freq o)))
         os)))

105 ;      Länge k  2                Länge k-1
;
; (list ,n ,m, ...) ~~> (list ... , n+m ...)
;      l      r                /  \
;
110 ;
;      l      r

(: merge ((list-of (occur-of (huff-tree-of %a) natural)) ->
          (list-of (occur-of (huff-tree-of %a) natural))))
(define merge
115   (lambda (os)
     (let* ((o1 (first os))
            (o2 (first (rest os)))
            (o1+2 (make-occur (make-huff-node (occur-item o1)
120              (occur-item o2))
                             (+ (occur-freq o1) (occur-freq
              o2)))))
       (sort (curry occur<=?) (make-pair o1+2 (rest (rest
         os)))))))

; Zu (H3)

125 ; Iteriere Anwendung von f auf x bis done? erfüllt ist
(: until ((%a -> boolean) (%a -> %a) %a -> %a))
(define until
  (lambda (done? f x)
    (if (done? x)
130        x
        (until done? f (f x)))))

; Enthält Liste xs genau ein Element?
(: singleton? ((list-of %a) -> boolean))
135 (define singleton?
  (lambda (xs)

```

```

    (and (pair? xs)
          (empty? (rest xs))))
140 ; Zu (H4)

; Generiere optimalen Huffman-Tree für String s
(: huffman-code (string -> (huff-tree-of string)))
(define huffman-code
145 (lambda (s)
      (occur-item                               ; (H4)
        (first
          (until singleton? merge                ; (H3)
            (huffman-leaves                       ; (H2)
              (occurrences s))))))              ; (H1)
150

```

### Neue Sprachebene: DMdA-fortgeschritten

- Neues Ausgabeformat im REPL

```

(list x1... xn) → (x1.. xn)
empty          → ()

```

- Neuer (struktureller) Gleichheitstest für Werte aller (auch benutzerdefinierte)

Signatur: `(: equal? (%a %b -> boolean))`

### Quote:

Sei *e* ein beliebiger Scheme-Ausdruck. Dann liefert `(quote e)` die Repräsentation von *e* – *e* wird *nicht* ausgewertet.

### Beispiele:

<code>(quote 42)</code>	$\rightsquigarrow$ 42	} Konstante Literale repräsentieren sich selbst
<code>(quote "UTÜ")</code>	$\rightsquigarrow$ "UTÜ"	
<code>(quote (+ 402))</code>	$\rightsquigarrow$ (+ 402)	} Funktionsapplikation als Liste

### Listennotation in Programmen

```

(list x1 ... xn) ≡ '(x1... xn)
empty           ≡ '()

```

### Symbole:

Was ist `(first '(x 12))`?

Was sind `lambda`, `x`, `+` in `'(lambda (x) (+ x 1))`

Neue Signatur *Symbol* zur Repräsentation von Namen in Programmen. Effiziente interne Darstellung/effizient vergleichbar. Kein Zugriff auf die einzelnen Zeichen des

### Symbols.

*Operationen:*

- (symbol? (%a -> boolean))
- (: symbol->string (symbol -> string))

### Repräsentation und Auswertung arithmetischer Ausdrücke:

Beispiel:  $e \equiv \text{'('} \underbrace{\text{* (! (+ 1 2))}}_{\text{list}} \text{'}$

```
(define arith
  (signature (mixed number
                    symbol
                    (list-of airth))))
```

Auswertung möglich, wenn Bindungen für Symbole (Variablen *und* Operatoren) an Wert gegeben. *Dictionary (Envirement)*.

 $d_1:$ 

```
{x → 3
 * → <procedure : *>
 + → <procedure : +>
 ! → fac}
```

$$e \rightsquigarrow 18$$
 $d_2:$ 

```
{ x → 1
  * → <procedure : *>
  + → <procedure : +>
  ! → (lambda (x) (- x)) }
```

$$e \rightsquigarrow -3$$

### Codebeispiel 53: Eigene Programmiersprache die arithmetische Ausdrücke auswerten kann

```
; Arithmetische Ausdrücke (Konstanten, Variablen,
  Operatorapplikationen)
(define arith
  (signature (mixed number          ; Konstanten
              symbol                 ; Variablen/Operatoren
```

```

5      (list-of arith)))    ; zusammengesetzter
      Ausdruck (op e1 e2)

; Ist e eine Konstante?
(: constant? (arith -> boolean))
(define constant?
10   (lambda (e) (number? e)))

; Ist e ein Operator?
(: operator? (arith -> boolean))
(define operator?
15   (lambda (e) (elem? e '(* + ^ !))))

; Ist e eine Variable?
; NB: + ist auch eine (funktionswertige) Variable!
(: variable? (arith -> boolean))
20 (define variable?
    (lambda (e) (symbol? e)))

; Ist e ein zusammengesetzter Ausdruck?
(: compound? (arith -> boolean))
25 (define compound?
    (lambda (e) (pair? e)))

; Ein Eintrag (entry) key val im Dictionary besteht aus
30 ; - einem Schlüssel (key) und
; - einem Wert (val)
(: make-entry (%a %b -> (entry-of %a %b)))
(: entry-key ((entry-of %a %b) -> %a))
(: entry-val ((entry-of %a %b) -> %b))
35 (define-record-procedures-parametric entry entry-of
    make-entry
    entry?
    (entry-key
     entry-val))

40 ; Vertrag für Dictionary mit Schlüsseln k und Werten v
(define dict-of
  (lambda (k v)
    (signature (list-of (entry-of k v)))))

45 ; Das leere Dictionary ohne Einträge
(: empty-dict (dict-of %a %b))

```



```

(define empty-dict '())

50 ; Signatur t, erweitert um Wert 'nothing
(define maybe
  (lambda (t)
    (signature (mixed t
                      (one-of 'nothing))))))

55 ; Wert des Eintrags für Schlüssel k im Dictionary d
; (oder k falls k nicht in d existiert)
(: lookup-dict (%a (dict-of %a %b) -> (maybe %b)))
(define lookup-dict
60   (lambda (k d)
     (cond
      ((empty? d) 'nothing)
      ((pair? d)  (if (equal? k (entry-key (first d)))
                      (entry-val (first d))
                      (lookup-dict k (rest d)))))))

65 ; Beispiel

(: fac (natural -> natural))
70 (define fac
    (lambda (n)
      (if (= 0 n)
          1
          (* n (fac (- n 1))))))

75 ; Dictionary eingebauter Funktionen (curried)
(: builtin-dict (dict-of symbol %a))
(define builtin-dict
80   (list (make-entry '* (curry *))
          (make-entry '+ (curry +))
          (make-entry '^ (curry expt))
          (make-entry '! fac)))

```

Auswertung eines arithmetischen Ausdrucks e (unter Dictionary d)

$$\underbrace{((\text{eval } d) e)}_{\text{Konfigurierter Ausdruck}}$$

(E1)  $((\text{eval } d) c) = c$

Konstante

(E2)  $((\text{eval } \{x_1 \rightarrow v_1 \dots x_n \rightarrow v_n\})x_i) = v_i$   $x_i$  Variable

(E3)  $((\text{eval } d) e_1 e_2 \dots e_n) = (\dots ((\text{eval } d) e_1) ((\text{eval } d) e_2) \dots ((\text{eval } d) e_n))$

```
(: eval ((dict-of symbol) %a -> (arith %b)))
(define eval
  (lambda (d)
    (lambda (e)
      (cond ((constant? e)e)
            ((variable? e)(lookup-dict d e))
            ((compound? e)
             (let ((es (map (eval d) e)))
               (foldl (first es) (lambda (f x) (f x)) (rest es)))))))
```

## 21.7.2015

Das  $\lambda$ -Kalkül ist eine Notation für beliebige (berechenbare) Funktionen : Entwickelt in den 1930er Jahren von Alonzo Church (\*1903 †1995) als neue Grundlage der Mathematik. Seither verwendet als Theoretischer Unterbau von Programmiersprachen.

*Syntax des  $\lambda$ -Kalküls*

Die Menge der Ausdrücke (expressions)  $E$  des  $\lambda$ -Kalküls ist induktiv definiert. Sei  $V$  eine unendliche Menge an Variablennamen

- $\forall v \in V : v \in E$  [Variablen]
- $\forall e_1, e_2 \in E : (\underbrace{e_1}_{\text{Funktion}} \underbrace{e_2}_{\text{Argument}}) \in E$  [Applikation]
- $\forall v \in V, e_1 \in E : (\lambda_{\text{Parameter}} v \underbrace{e_1}_{\text{body}}) \in E$  [Abstraktion]

*Beispiele:*

$y \in E$

$(\lambda y y) \in E$  Identitätsfunktion

$(\lambda y z) \in E$  Funktion ignoriert y, liefert z

$((f x) y) \in E$  Currying

$(\lambda f (f x)) \in E$  Anwendung von Funktion  $f$  auf  $x$  (H.O.P)

*Abkürzungen:*

$(\dots ((e_1 e_2) e_3) \dots e_n) \equiv (e_1 e_2 \dots e_n)$

$(f x y) \equiv ((f x) y)$

#### Codebeispiel 54: Das Lambda Kalkül durch Schemes Quote Technik implementiert

```

; -Kalkül

5 ; Form von Ausdrücken (E)
(define λ-syntax
  (signature (mixed symbol
                  (one-of 'λ)
                  (list-of λ-syntax)))
            ; v
            ; λ
            ; Applikation (e1
            ; Abstraktion ( v
            ; e1)

; Ist Ausdruck e eine Variable?
(: var? (λ-syntax -> boolean))
(define var?
15   (lambda (e)
      (and (symbol? e)
            (not (equal? e '))))))

; Ist Ausdruck e eine Applikation (e1 e2)?
20 ;
;                               kein λ
(: app? (λ-syntax -> boolean))
(define app?
25   (lambda (e)
      (and (pair? e)
            (not (equal? (first e) '))))))

; Zugriff auf Funktion e1 / Argument e2 der Applikation (e1 e2)
(: fun (λ-syntax -> λ-syntax))
30 (: arg (λ-syntax -> λ-syntax))
(define fun
  (lambda (e)
    (if (app? e) (first e) (violation "fun: not an_
                                     application"))))
(define arg

```

```

35  (lambda (e)
    (if (app? e) (first (rest e)) (violation "arg:_not_an_
        application"))))

; Ist Ausdruck e eine Abstraktion (λ v e1)?
(: lam? (λ-syntax -> boolean))
40  (define lam?
    (lambda (e)
      (and (pair? e)
            (pair? (rest e))
            (equal? (first e) 'λ)
            (symbol? (first (rest e))))) ; Liste dreistellig?
                                           ; λ
                                           ; Symbol v

; Zugriff auf Parameter v / Body e der Abstraktion (λ v e)
(: parm (λ-syntax -> λ-syntax))
(: body (λ-syntax -> λ-syntax))
50  (define parm
    (lambda (e)
      (if (lam? e) (first (rest e)) (violation "parm:_not_an_
        abstraction"))))
    (define body
      (lambda (e)
65      (if (lam? e) (first (rest (rest e))) (violation "body:_not_
        an_abstraction"))))

; Beispiele:

60  ; Kombinator I
    ; I (λx.x)
    (: I λ-syntax)
    (define I
      '(λ x x))

65  ; Kombinator K
    ; K (λx.(λy.x))
    (: K λ-syntax)
    (define K
70      '(λ x ( λ y x)))

; Kombinator S
; S (λf.(λg.(λx.((f x) (g x)))))
(: S λ-syntax)
75  (define S

```

```

      ' (λ f (λ g (λ x ((f x) (g x))))) )

; Beispiele: Operationen auf Syntax des λ-Kalküls
;
80 (var? 'x)
   (var? 'λ) ; ~> #f

   (app? '(f x))
85 (fun '(f x))
   (arg '(f x))

   (lam? '(λ x (f x)))
   (parm '(λ x (f x)))
90 (body '(λ x (f x)))

;

```

### Freie/Gebundene Variablen

Zur Auswertung von  $E_1 \equiv ((\lambda x (f x y)) z)$

- wird der hier nicht bekannte Wert von Variablen  $f, y, z$  benötigt, während
- der Wert von  $x$  im Rumpf  $(f x y)$  durch das Argument  $z$  festgelegt ist.

in  $E_2$  ist

- Variable  $x$  (durch das  $\lambda x$ ) als Parameter *gebunden*, während
- Variablen  $f, y, z$  *frei* sind

Welche Variablen eines Ausdrucks sind frei/gebunden?

$\text{free}(v) = \{v\}$

$\text{free}((e_1 e_2)) = \text{free}(e_1) \cup \text{free}(e_2)$

$\text{free}((\lambda v e_1)) = \text{free}(e_1) \setminus \{v\}$

$\text{bound}(v) = \emptyset$

$\text{bound}((e_1 e_2)) = \text{bound}(e_1) \cup \text{bound}(e_2)$

$\text{bound}((\lambda v e_1)) = \text{bound}(e_1) \cup \{v\}$

*Beispiel*

$$\text{Klausur} \begin{cases} \text{free}(E_1) &= \{fyz\} \\ \text{bound}(E_1) &= \{x\} \end{cases}$$

*Achtung:* Bindung/Freiheit muss für jedes Vorkommen separat entschieden werden.

$E_2 \equiv (x_1 (\lambda x_2 x_2))$

$\text{free}(E_2) = \{x\}$

$\text{bound}(E_2) = \{x\}$

Codebeispiel 55: Prozeduren die herausfinden ob Variablen frei oder gebunden sind

```

; Menge der freien Variablen in Ausdruck e
(: free (λ-syntax -> (set-of symbol)))
(define free
5   (lambda (e)
      (cond
        ((var? e)
         (set-insert e empty-set)) ; {v}
        ((app? e)
10        (set-union (free (fun e)) (free (arg e))))
        ((lam? e)
         (set-delete (parm e) (free (body e))))))

; Menge der gebundenen Variablen in Ausdruck e
15 (: bound (λ-syntax -> (set-of symbol)))
(define bound
    (lambda (e)
      (cond
        ((var? e)
20         empty-set)
        ((app? e)
         (set-union (bound (fun e)) (bound (arg e))))
        ((lam? e)
         (set-insert (parm e) (bound (body e))))))

25 ; Beispiele

; E1 ((λx. ((f x) y)) z)
(: E1 λ-syntax)
30 (define E1
    '((λ x ((f x) y)) z))

; E2 (x (λx.x))
(: E2 λ-syntax)

```

```

35 (define E2
    '(x (λ x x)))
;
; frei    gebunden

```

## 23.7.2015

Die Anwendung einer Funktion ( $\lambda$ - Abstraktion) auf ein Argument wird im  $\lambda$ -Kalkül durch  $\beta$ -Reduktion ( $\rightarrow_\beta$ ) beschrieben. Die Applikation  $((\lambda v e_1) e_2)$  wird durchgeführt indem

- (1) Eine Kopie vom Rumpf  $e_1$  hergestellt wird und
- (2) alle *freien* Vorkommen von  $v$  in  $e_1$  durch  $e_2$  ersetzt werden

Beispiele:

$$\text{a) } ((\lambda x \underbrace{(* ((\lambda x (+ x a)) b) x)}_{\text{Rumpf}}) c)$$

gebunden

$$\xrightarrow[\beta]{} (* ((\lambda x (+ x a)) b) c)$$

$$\xrightarrow[\beta]{} (* (+ b a) c)$$

$$\text{b) } ((\lambda x \text{foo}) \text{bar})$$

$$\xrightarrow[\beta]{} \text{foo}$$

$$\text{c) } ((\lambda x ((\lambda y (* x y)) a)) b)$$

$$\xrightarrow[\beta]{} (\lambda y (* b y) a)$$

$$\xrightarrow[\beta]{} (* b a)$$

$$\text{d) } ((\lambda f (f c)) (\lambda x (+ x a)))$$

$$\xrightarrow[\beta]{} ((\lambda x (+ x a)) c)$$

$$\xrightarrow{\beta} (+ \ c \ a)$$

e)  $((\underbrace{(\lambda x \ (\lambda y \ x))}_{\text{K, ignoriert zweites Argument}} \ \underbrace{(\lambda z \ y)}_{\text{ignoriere Argument, liefere y}}) \ \textcolor{blue}{\lambda} a)$

$$\xrightarrow{\beta} ((\lambda y \ (\lambda x \ y)) \ \textcolor{blue}{\lambda} a)?$$

$$\xrightarrow{\beta} ((\lambda \ x \ \textcolor{blue}{\lambda} a)$$

Problem : freies y "wandert" unter das  $\lambda \ y$  und wird damit gebunden :-)

$$\xrightarrow{\beta} \textcolor{blue}{\lambda}$$

*Definition  $\beta$ -Reduktion*  $((\lambda \ x \ e) a) \xrightarrow{\beta} e[x \mapsto a]$  „Ersetze freie Vorkommen von x in e durch a“

$$(\mapsto 1) \ x[x \mapsto a] = a$$

$$(\mapsto 2) \ v[x \mapsto a] = v$$

$$(\mapsto 3) \ (e_1 \ e_2)[x \mapsto a] = (e_1[x \mapsto a] \ e_2[x \mapsto a])$$

$$(\mapsto 4) \ (\lambda \ x \ e)[x \mapsto a] = (\lambda \ x \ e)$$

$$(\mapsto 5) \ (\lambda v \ e)[x \mapsto a] = \begin{cases} (\lambda v \ e[x \mapsto a]) & \text{wenn v nicht frei in a ist} \\ (\lambda z \ e[v \mapsto z])[x \mapsto a] & \text{sonst, z neuer Name} \end{cases}$$

$$(\mapsto 6)$$

*Beispiel  $\beta$ -Reduktion*  $((\underbrace{(\lambda x \ (\lambda y \ x))}_{\text{Redex}} \ (\lambda z \ y)) \ \textcolor{blue}{\lambda} a)$

$$\xrightarrow{\beta} ((\lambda \ y \ x)[x \mapsto (\lambda z \ y)] \ \textcolor{blue}{\lambda} a)$$

$$\xrightarrow{\mapsto 6} ((\lambda y' \ x \ [y \mapsto y'])[x \mapsto (\lambda z \ y)] \ \textcolor{blue}{\lambda} a)$$

$$\xrightarrow{\mapsto 2} ((\lambda y' \ x) [x \mapsto (\lambda z \ y)] \ \textcolor{blue}{\lambda} a)$$

$$\xrightarrow{\mapsto 5} ((\lambda y' \ x \ [x \mapsto (\lambda z \ y)]) \ \textcolor{blue}{\lambda} a)$$

$$\xrightarrow{\mapsto 1} (\underbrace{((\lambda y' \ (\lambda z \ y)) \ \textcolor{blue}{\lambda} a)}_{\text{Redex}})$$



$$\xrightarrow[\beta]{} ((\lambda z y)[y' \mapsto t]a)$$

$$\xrightarrow{\mapsto 5} ((\lambda z y)[y' \mapsto t]a)$$

$$\xrightarrow{\mapsto 2} \underline{((\lambda z y) a)}$$

$$\xrightarrow[\beta]{} y[z \mapsto a]$$

$$\xrightarrow{\mapsto 2} y$$

## Codebeispiel 56: Beta Reduktion

```

;  $\beta$ -Reduktion für Funktionsanwendung
; (( $\lambda x.e$ ) a)  e[x a]
(: beta ( $\lambda$ -syntax  $\rightarrow$   $\lambda$ -syntax))
5 (define beta
  (lambda (e)
    (if (and (app? e)
              (lam? (fun e)))
        (subst (body (fun e))
                (arg e)
                (parm (fun e)))
        (violation
         (string-append "beta:_cannot_reduce")))))

15 ; e[x a]: Ersetze freie Vorkommen von x in e durch a
(: subst ( $\lambda$ -syntax  $\lambda$ -syntax symbol  $\rightarrow$   $\lambda$ -syntax))
(define subst
20 (lambda (e a x)
  (cond
    ((var? e) (if (equal? e x)
                   a
                   ; ( $\mapsto 1$ )
                   e))
    ; ( $\mapsto 2$ )
    ((app? e) (list (subst (fun e) a x)
                     ; ( $\mapsto 3$ )
                     (subst (arg e) a x)))
    ; ( $\mapsto 4$ )
    ((lam? e)
     (let ((v (parm e)))
       (cond ((equal? v x)
                ; ( $\mapsto 5$ )
                (subst (body e) a x)
                ; ( $\mapsto 6$ )
                (let ((v (parm e)))
                  (subst (body e) a x))))
              (let ((v (parm e)))
                (subst (body e) a x)))))))

```

```

    e)
      ; (→4)
30  ((not (free? v a))
      (list 'λ v (subst (body e) a x)))
      ; (→5)
    (else
      (let ((z (fresh (set-union (free e) (free
        a)))))
        (subst (list 'λ z (subst (body e) z v))
          ; (→6)
          a x)))))))))

; Unendlicher Vorrat (Stream) an Variablennamen x0, x1, x2, ...
(: name-pool (natural -> (stream-of symbol)))
40 (define name-pool
    (lambda (n)
      (make-cons
        (string->symbol (string-append "x" (number->string n)))
        (lambda () (name-pool (+ n 1))))))

45 ; Erzeuge neuen Variablennamen, der nicht in Liste vs vorkommt
(: fresh ((set-of symbol) -> symbol))
(define fresh
  (lambda (vs)
50    (head
      (until
        (lambda (np) (not (set-member? (head np) vs)))
        (compose force tail)
        (name-pool 0)))))

55 ; Beispiele

(name-pool 0)
60 (stream-take 10 (name-pool 0))
(fresh empty-set)
(fresh (set-insert 'x0 empty-set))
(fresh (set-insert 'x1 (set-insert 'x0 empty-set)))

65 ; Beispiele
;
(beta '((λ x (λ y (x y))) a)) ; no capture

```

70

```
(beta '(( $\lambda$  x ( $\lambda$  y (x y))) y))  
;  
; free y would be captured by  $\lambda y$ ; renaming (case  $\rightarrow$  6)
```