

# Informatik II Skript Sommersemester 2015

Finn Ickler

8. Juli 2015

## Inhaltsverzeichnis

<b>14.4.2015</b>	<b>4</b>
<b>16.4.2015</b>	<b>5</b>
<b>21.4.2015</b>	<b>7</b>
<b>23.4.2015</b>	<b>9</b>
<b>28.4.2015</b>	<b>11</b>
<b>30.4.2015</b>	<b>14</b>
<b>5.5.2015</b>	<b>18</b>
<b>7.5.2015</b>	<b>19</b>
<b>12.5.2015</b>	<b>23</b>
<b>19.5.2015</b>	<b>27</b>
<b>21.5.2015</b>	<b>32</b>
<b>9.6.2015</b>	<b>37</b>
<b>11.6.2015</b>	<b>39</b>
<b>16.6.2015</b>	<b>44</b>
<b>18.6.2015</b>	<b>46</b>
<b>23.6.2015</b>	<b>49</b>

<b>25.6.2015</b>	<b>55</b>
<b>30.6.2015</b>	<b>59</b>
<b>2.7.2015</b>	<b>64</b>
<b>7.7.2015</b>	<b>69</b>

## Codebeispiele

1	Arithmetik mit Fließkommazahlen . . . . .	5
2	Schlüsselwort define . . . . .	6
3	Lambda Abstraktion . . . . .	6
4	Bilderzusammenstellung am Beispiel einer Uhr . . . . .	8
5	Die one-of Signatur . . . . .	11
6	Konstruktion eines eigenen Ifs? . . . . .	11
7	Absolutbetrag durch cond . . . . .	13
8	Boolsche Ausdrücke mit and und or . . . . .	14
9	Record Definitionen . . . . .	14
10	Check-property . . . . .	16
11	Übersetzung mathematischer Aussagen in check-property . . . . .	16
12	Konstruktoren und Selektoren . . . . .	17
13	predicate Signaturen am Beispiel von Längen- und Breitengrade . . . . .	19
14	Ersetzung one-of durch predicate Signaturen . . . . .	19
15	Geocoding . . . . .	21
16	cond mit gemischten Daten . . . . .	22
17	Wrapper und Worker . . . . .	23
18	make-pair, ein polymorpher Datentyp . . . . .	25
19	Listen mit Signatur list-of . . . . .	27
20	Geschachtelte Listen . . . . .	29
21	Rekursion auf Listen: Länge einer Liste . . . . .	30
22	Rekursion: Zusammenfügen zweier Listen . . . . .	31
23	Bildmanipulation mit Listen aus Pixeln . . . . .	32
24	Check-property mit Einschränkungen . . . . .	35
25	Rekursion auf natürlichen Zahlen: Fakultät . . . . .	35
26	Fehlerhafte Rekursionen . . . . .	36
	Endrekursion.rkt . . . . .	37
27	Umdrehen einer Liste durch lambda Rekursion . . . . .	38
28	Letrec und endrekursives Umdrehen einer Liste . . . . .	39
	HigherOrderProcedures.rkt . . . . .	46
29	Anwendungsbeispiele foldr . . . . .	48

Animationen-und-HOP-Typ2.rkt . . . . .	49
30 Animation 1: Ein Zähler . . . . .	50
31 Animation 2: Ein Raumschiff . . . . .	50
32 Anwendungen von Combined . . . . .	52
33 + als Higher Order Funktion . . . . .	53
CurryUndMengen.rkt . . . . .	55
34 Einfache Curry Beispiele . . . . .	55
35 Ableitungen berechnen mit Curry . . . . .	56
36 Mengenoperationen Teil 1 . . . . .	57
37 Mengenoperationen Teil 2 . . . . .	58
StreamsUndMengen.rkt . . . . .	59
38 Listen zu Mengen Konvertierung . . . . .	59
39 Mengenoperationen . . . . .	59
40 Implementation von Streams . . . . .	61
41 Rekursiv defnierter Sream . . . . .	64
Baeume.rkt . . . . .	65
42 Implementation von Bäumen . . . . .	66
43 Berechnung der Größe eines Baumes . . . . .	69
44 Entwicklung einer Pretty Print Methode für Bäume . . . . .	70

**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*

```
(odd? 42)  $\rightsquigarrow$  #f
```

Interaktionsfenster:

*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	(Fließkommazahl)	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)))

(- (+ 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 <id> wird an den Wert von <e> 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

```
(define absoluter-nullpunkt -273.15)
(define pi 3.141592653)
(define Gruendungsjahr-SC-Freiburg 1904)
(define top-level-domain-germany "de")
5 (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>)**

<e>Rumpf: enthält Vorkommen der Parameter <p<sub>n</sub>>

(lambda(...)) ist eine Spezialform. Wert der lambda-Abstraktion ist #<procedure>

. *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 P1> ... <Signatur Pn> -> <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.

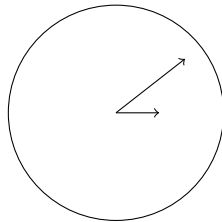
Spezialform: kein Wert, sondern Effekt: Testverletzung protokollieren

*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 zuruecklegt
(define degrees-per-minute 360/60)

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

; Zeichne Ziffernblatt zur Stunde h und Minute m
(: draw-clock (natural natural -> image))
(check-expect (draw-clock 4 15) (draw-clock 16 15))
10 (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)
    (* m degrees-per-minute)))

; 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)
     (+ (* (modulo h 12) degrees-per-hour)
30 ; h mod 12 in {0,1,...,11}
       (* (/ m 60) degrees-per-hour))))

; Zeichne Ziffernblatt mit Minutenzeiger um dm und
; Stundenzeiger um dh Grad gedreht
35 (: clock-face (rational rational -> image))
   (define clock-face
   (lambda (dh dm)
     (clear-pinhole
     (overlay/pinhole
40 (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"))))))))

```

## 23.4.2015

### Substitutionsmodell

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

- literal (1, "abc", #t, ...)  $l \rightsquigarrow$   $[eval_{lit}]$
- Identifier id(pi, clock-face,...)  $id \rightsquigarrow$  gebundene Wert  $[eval_{id}]$
- lambda Abstraktion  $(\text{lambda } (...) ...) \rightsquigarrow (\text{lambda } (...) ...)$   $[eval_{\lambda}]$
- Applikationen (f  $e_1$   $e_2$ ...)

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

`(position-minute-hand 30)`  $\xrightarrow[\text{eval id}]{\sim}$  `((lambda (m) (* degrees-per-minute m)) 30)`  
 $\xrightarrow[\text{eval lambda}]{\sim}$  `(* degrees-per-minute 30)`  
 $\xrightarrow[\text{eval id}]{\sim}$  `(#<procedure *> 360/60 30)`  
 $\xrightarrow[\text{apply prim}]{\sim}$  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 *bindene 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:  
 $\langle e_2 \rangle \begin{cases} e_1 & \text{falls } t_1 \\ e_2 & \text{sonst} \end{cases}$   
 $\langle e_2 \rangle$

## 28.4.2015

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

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

### Codebeispiel 5: 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)
(check-expect (heim-punkte 1 4) 0)
5 (check-expect (heim-punkte 3 3) 1)
(define heim-punkte
  (lambda (h a)
    (cond ((> h a) 3)
          ((< h a) 0)
10         (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 6: Koennen wir unser eigenes 'if' aus 'cond' konstruieren? (Nein!)

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

; Sichere Division x/y, auch fuer y = 0
10 (: safe-/ (real real -> real))
```

```

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

(safe-/ 42 0)      ; Fuehrt zu Fehlermeldung "division by
zero"              ; (Reduktion mit Stepper durchfuehren)

```

Spezifikation Fallunterscheidung (conditional expression):

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

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

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

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

## Codebeispiel 7: Absolutwert von x

```
(: my-abs (real -> real))
(check-within (my-abs -4.2) 4.2 0.001) ; Wichtig:
(check-within (my-abs 4.2) 4.2 0.001) ; Tesfaelle decken
    alle Zweige
(check-within (my-abs 0) 0 0.001) ; der conditional
    expression an
5 (define my-abs
  (lambda (x)
    (cond ((< x 0) (- x))
          ((> x 0) x)
          (else 0))))
```

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

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

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

(cond)  $\rightsquigarrow$  „Fehlermeldung: alle Test ergaben false“

(cond (else <e<sub>n+1</sub>>))  $\rightsquigarrow e_{n+1}$

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

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

Spezialform 'and' und 'or'

(or <t<sub>1</sub>> <t<sub>2</sub>> ... <t<sub>n</sub>>)  $\rightsquigarrow$  (if <t<sub>1</sub>> (or <t<sub>2</sub>> ... <t<sub>n</sub>>) #t)

(or)  $\rightsquigarrow$  #f

(and <t<sub>1</sub>> <t<sub>2</sub>> ... <t<sub>n</sub>>)  $\rightsquigarrow$  (if <t<sub>1</sub>> (and <t<sub>2</sub>> ... <t<sub>n</sub>>) #f)

(and)  $\rightsquigarrow$  #t

## Codebeispiel 8: Konstruktion komplexer Prädikate mittels 'and' und 'or'

```

5 (and #t #f) ; ~> #f (Mathematik: Konjunktion)
  (or #t #f) ; ~> #t (Mathematik: Disjunktion)
; Kennzeichen am/pm fuer 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)
        (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 9: 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))
  (: character-jedi? (character -> boolean))
  (: character-force (character -> real))
10 (define-record-procedures character
    make-character
    character?
    (character-name
     character-jedi?
     character-force))
15

```

```

; Definiere verschiedene Charaktere des Star Wars Universums
(define luke
20   (make-character "Luke_Skywalker" #f 25))
(define r2d2
   (make-character "R2D2" #f 0))
(define dooku
   (make-character "Count_Dooku" #f 80))
25 (define yoda
   (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 10: Interaktion von Selektoren und Konstruktor:

```
(check-property
  (for-all ((n string)
            (j boolean)
            (f real))
    (expect (character-name (make-character n j f)) n)))

5

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

10

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

15
```

*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 11: 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))))

5
```

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)
    5    ... (character-force c) ...))
```



Konstruktion von Funktionen, die zusammengesetzte Daten *konstruieren*

- Der konstruktor *muss* aufgerufen werden

→ Schablone:

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

- Konkrete Beispiele:

## Codebeispiel 12: Abfragen der Eigenschaften von character Records

```
5 ; Könnte Charakter c ein Sith sein?
  (: sith? (character -> boolean))
  (check-expect (sith? yoda) #f)
  (check-expect (sith? r2d2) #f)
  (define sith?
    (lambda (c)
      (and (not (character-jedi? c))
            (> (character-force c) 0))))

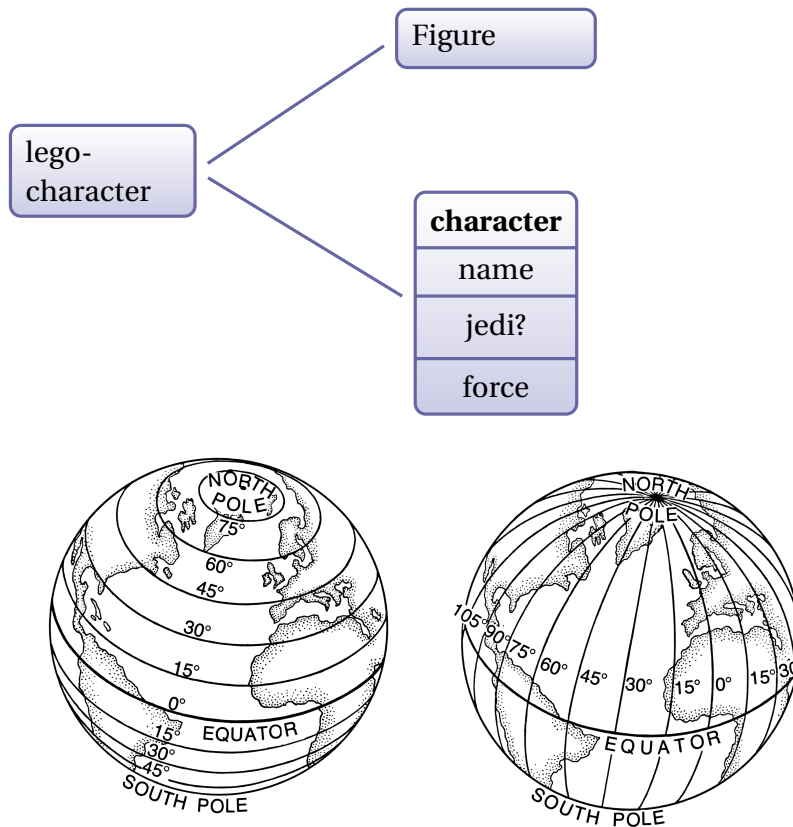
10 ; Bilde den Charakter c zum Jedi aus (sofern c überhaupt
    Macht besitzt)
  (: train-jedi (character -> character))

  (check-expect (train-jedi luke) (make-character "Luke_
    Skywalker" #t 50))
15 (check-expect (train-jedi r2d2) r2d2)

  (define train-jedi
    (lambda (c)
      (make-character (character-name c)
                      (> (character-force c) 0)
                      (* 2 (character-force c)))))

20
```

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  $(\text{predicate } \langle p \rangle)$  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 13: 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 14: 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 15: 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:

```
(: @\argt{}@? (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 16: 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 (
  ((lambda (<id1> ... <idn>))

```

$$\begin{array}{ccc}
 \langle \langle id_1 \rangle \langle e_1 \rangle \rangle & & \langle e \rangle \\
 \langle \dots \rangle & \equiv & \langle e_1 \rangle \\
 \langle \langle id_n \rangle \langle e_n \rangle \rangle & & \langle e_2 \rangle \dots \\
 5 \quad \langle e \rangle & & \langle e_n \rangle \\
 ) & &
 \end{array}$$

## 12.5.2015

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

### Codebeispiel 17: 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°)
(: 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
                                lat2) (cos lng2))
                            (* (cos lat1) (sin lng1) (cos
                                lat2) (sin lng2))
                            (* (sin lat1) (sin lat2))))))))
30
35  (gc1 (geocoder o1))
    (gc2 (geocoder o2))
    (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 18: 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  $\langle t_i \rangle$  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 19: 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))
```

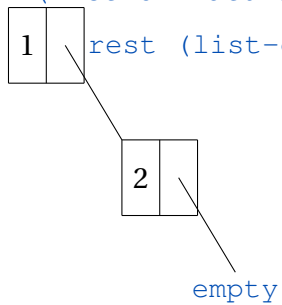
*Visualisierung Listen*

1	2	empty
---	---	-------



*Spine (Rückgrat)*

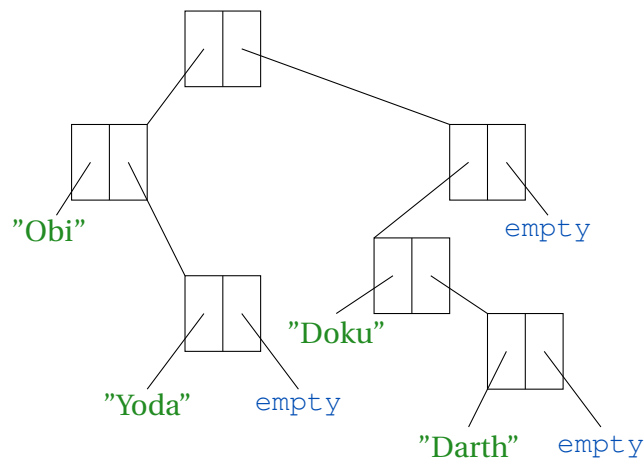
```
(pair-of natural (list-of natural))
(natural first) 1 rest (list-of natural)
```



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

5

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



Codebeispiel 20: Jedi 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))

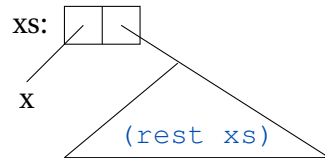
(check-expect (list-sum empty) 0)
(check-expect (list-sum (make-pair 40
  (make-pair 2
    empty)))) 42)

(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>)
  ... (make-pair <en>) empty) ...)
```

- Ausgabeformat für nicht leere Listen:

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

### Codebeispiel 21: 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!
```

```

(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)

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

Beobachtung:

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

#### Codebeispiel 22: 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)

(define cat
  (lambda (xs ys)
    (cond ((empty? xs)
          ys)
          ((pair? xs)
           (make-pair (first xs) ; <- cat dennoch param.
                      polymorph
                      (cat (rest xs) ys))))))
10
15 ; Hinweis: Verfügbar als eingebaute Funktion `append'

```

**21.5.2015**

Codebeispiel 23: Ausflug: Bluescreen Berechnung wie in Starwars mit Listen:



```
(define yoda
```

```
)
```

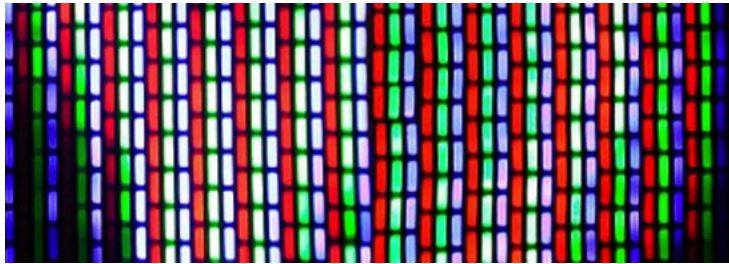


```
(define dagobah
```

```
)
```

```
; _____  
; 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)

```

```

55         (first fg))
        (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))
70                fg-w
                fg-h)
              (violation "Dimensionen_von_Vorder-/Hintergrund_
                          verschieden")))))

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 24: ==> 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 25: 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)
```

```

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

```

Konstruktionsanleitung für Prozeduren über natürlichen Zahlen:

```

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

```

Beobachtung:

- Im letzten Zweig ist  $n > 0 \rightarrow \text{pred}$  angewandt
- $(\text{<f> } (- \text{ n } 1))$  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 26: Fehlerhafte Rekursionen

```

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

; Fehlerhaft: kein definierter Abbruch der Rekursion
; => Abbruch der Reduktion bei n = 0 ("cond: alle Tests
10 ergaben #f")
(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))`  $\rightsquigarrow$  `(* 30240 (factorial 5))`

Assoziativität von  $\cdot$

→ 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	
$\cdot 1 \checkmark 5$	1 $\curvearrowright \cdot 5$	neutrales Element
$\cdot 1 \checkmark 4$	5 $\curvearrowright \cdot 4$	
$\cdot 1 \checkmark 3$	20 $\curvearrowright \cdot 3$	
$\cdot 1 \checkmark 2$	60 $\curvearrowright \cdot 2$	
$\cdot 1 \checkmark 1$	120 $\curvearrowright \cdot 1$	
$\cdot 1 \checkmark 0$	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
15 (define fac-worker
    (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 27: Liste xs umdrehen

```
; Aufwand: 1/2 × n × (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))

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

Beobachtung: von `(rev (from-to 1 1000))`

$$\overbrace{(cat \quad (list \ 1000 \ \dots \ 2) \quad (list \ 1))}^{1000 \cdot \text{make-pair}}$$

$$(cat \ (list \ 1000 \ \dots \ 3) \ (list \ 2))$$

→ Aufrufe von `make-pair`: 1000+999+998+...+1

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

Konstruiere iterative Listenumkehrfunktion `backwards`:

	n
	rest ✓ (list 123)
(: backwards-worker ((list-of %a) (list-of %a) -> (list-of %a)))	rest ✓ (list 23)
	rest ✓ (list 3)
	empty

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 28: 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
    ; Liste xs umdrehen (mit Akkumulator acc, endrekursiv)
    ; Worker
    ; Aufwand: n Aufrufe von make-pair, wenn xs die Länge n hat
    (letrec ((backwards-worker
15              (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

Induktionsschritt

Falls  
 $P(0)$   
 und  
 $\forall (P(n) \Rightarrow P(\text{succ}(n)))$   
 dann  
 $\forall n \in \mathbb{N} P(n)$

*Beispiel:*



$$\begin{aligned}
 1 &= 1 \\
 1 + 3 &= 4 \\
 1 + 3 + 5 &= 9 \\
 1 + 3 + 5 + 7 &= 16 \\
 &\dots
 \end{aligned}$$

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

Induktionsschluss  $P(0)$

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

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

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

Beispiel:

```
(define factorial
  (lambda (k)
    (if
```

```
(= k 0) 1
(* k (factorial (- k 1))))))
```

5

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

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

Induktionsbasis  $P(0)$

```
(factorial(0))
```

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

```
~> ((lambda (k) ...) 0)
```

```
~> (if (= 0 0) 1 ...)
```

```
~> (if #t 1 ...)
```

```
~> 1 = 0!  $\checkmark$ 
```

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

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

$\rightsquigarrow^*$  `((lambda (n) ...) n+1)`

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

$\rightsquigarrow$  `(if #f 1 ... (...))`

$\rightsquigarrow$  `(* n+1 (factorial (- n+1 1)))`

$\rightsquigarrow$  `(* n+1 (factorial (- n)))`

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

$= (n+1)! \checkmark$

*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))))
```

5

`(: 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`  $\checkmark$

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

`(f n+1)`

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

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

$$\rightsquigarrow (\text{step } (f \ (- \ \boxed{n+1} \ 1)) \ \boxed{n+1})$$

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

$$\Rightarrow (\text{step } (f \ \boxed{n}) \ \boxed{n+1}) \text{ terminiert}$$

*Definition:* (Listen. endliche Folge)

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

- |      |                               |                   |
|------|-------------------------------|-------------------|
| (L1) | <code>empty</code> $\in M^*$  | Nicht leere Liste |
| (L2) | $\forall x \in M, xs \in M^*$ | $\in M^*$         |
| (L3) | Nichts sonst in $M^*$         |                   |

Beweisschema *Listeninduktion*

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

(:  $P \ ((\text{list-of } M) \rightarrow \text{boolean})$ )

(`make-pair x xs`)

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

Induktionsanfang

Induktionsschritt

## 16.6.2015

Beispiel:

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

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

$$\begin{cases} (1) & \text{cat empty ys} = \text{ys} \\ (2) & (\text{cat xs empty}) = \text{xs} \\ (3) & (\text{cat (cat xs ys) ys}) = (\text{cat xs (cat ys zs)}) \end{cases} \quad \text{Beweise:}$$

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

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

Induktionsanfang  $P(\text{empty})$

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

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

(define make-pair mp)

(cat (mp x xs) empty)

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

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

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

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

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

Induktionsanfang  $P(\text{empty})$

(cat (cat empty ys) zs)

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

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

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

(cat (cat (mp x xs) ys) zs)

$$\begin{aligned}
 & \rightsquigarrow^* (\text{cat } (\text{mp } x \text{ (cat } xs \text{ } ys)) \text{ } zs) \\
 & \rightsquigarrow^* (\text{mp } (\text{cat } (\text{cat } xs \text{ } ys)) \text{ } zs) \\
 & \stackrel{iv.}{=} (\text{mp } (\text{cat } (\text{cat } xs \text{ } ys) \text{ } zs)) \\
 & \rightsquigarrow (\text{cat } (\text{mp } x \text{ } xs) \text{ } (\text{cat } ys \text{ } zs)) \checkmark
 \end{aligned}$$

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

```

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

```

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

*Induktionsbasis:*

$$\begin{aligned}
 & (\text{length } (\text{cat } \text{empty } ys)) \\
 & \stackrel{(1)}{=} (\text{length } ys) \\
 & \stackrel{+}{=} (+ 0 (\text{length } ys)) \\
 & \rightsquigarrow (+ (\text{length } \text{empty}) (\text{length } ys)) \checkmark
 \end{aligned}$$

*Induktionsschritt*

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

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

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

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

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

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

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

*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

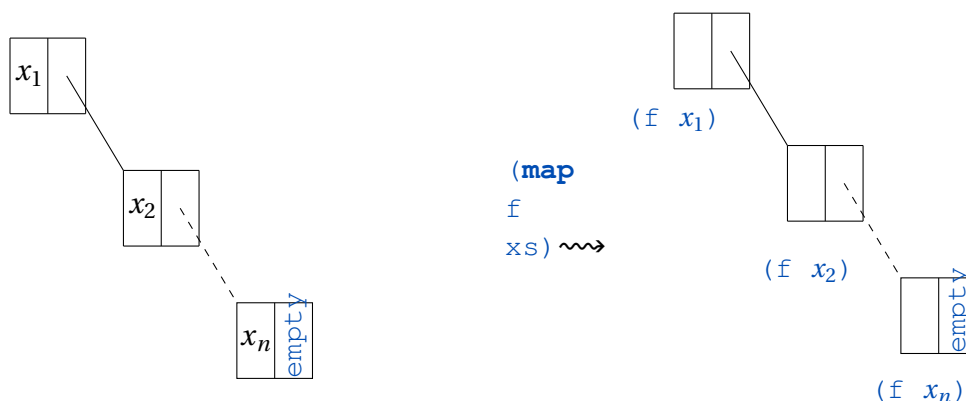
Zwei Arten von *Higher Order Prozeduren* (H.O.P)

- (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)`



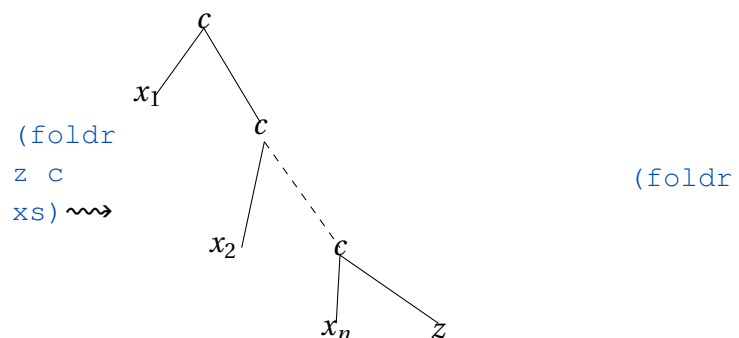
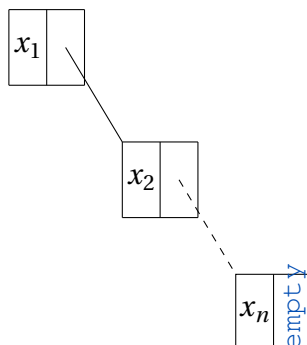
```

;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)))))))

```

Allgemeine Transformation von Listen *Listenfaltung* (list folding)

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



`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`

TODO: Großes Bild von `foldr` Funktionen

```

(: 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)))
5

```

**Codebeispiel 29: 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)))
5

; Listenreduktion via foldr: Produkt der Liste xs
(: my-product ((list-of number) -> number))
(define my-product
  (lambda (xs)
    (foldr 1 * xs)))
10

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

; Identität (auf Listen), implementiert via foldr
20 (: 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)))
35 (define my-map
  (lambda (f xs)
    (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)
60                        (make-pair y ys)
                        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 30: Ein animierter Zähler

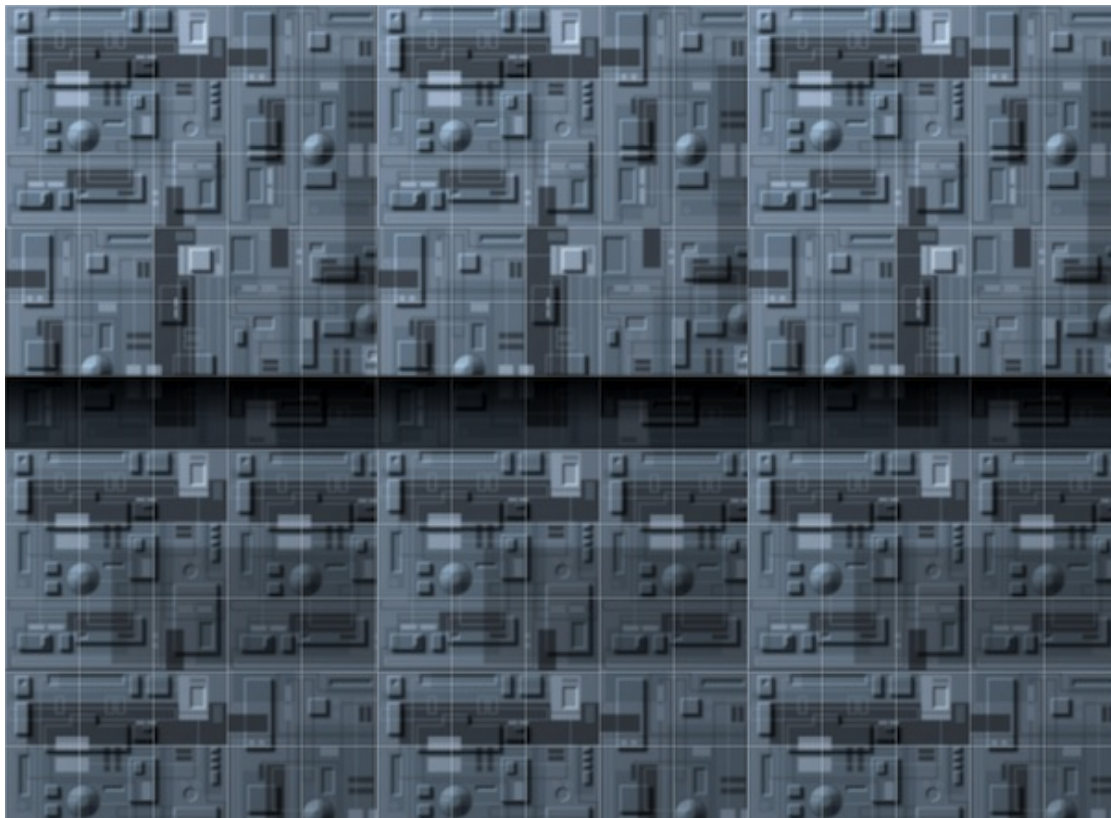
```
; Erstellung von Animationen mit Teachpack "universe"
; (1) Zähler

(: scene (natural -> image))
5 (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 31: 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
; 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)

```

10

```

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:

$(\underbrace{\text{compose } f \text{ } g}_{\text{neue Prozedur realisiert Komposition von } f \text{ und } g}) \ x) \equiv (f \ (g \ x))$

neue Prozedur realisiert  
Komposition von  $f$  und  $g$

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

```

(: compose (%b -> %c) ($a -> %b) -> (%a -> %c))
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
5

```

### Codebeispiel 32: 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)

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

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

```
(: 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 33: 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))))

(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)
```

```
20 | (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)`

$\rightsquigarrow$   
 $\text{eval}_{id}$  `((lambda (x) (lambda (y) (+ x y)) 1) 41)`

$\rightsquigarrow$   
 $\text{apply}_\lambda$  `((lambda (y) (+ 1 y) 41)`  
Funktion die 1 auf  
ihr Argument anwenden

$\rightsquigarrow$   
 $\text{apply}_\lambda$  `(+ 1 41)`  
 $\text{[lambda(y)]}$

## 25.6.2015

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

```

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

Es gilt für jeder Prozedur p:

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

„Schönfinkel Isomorphismus“

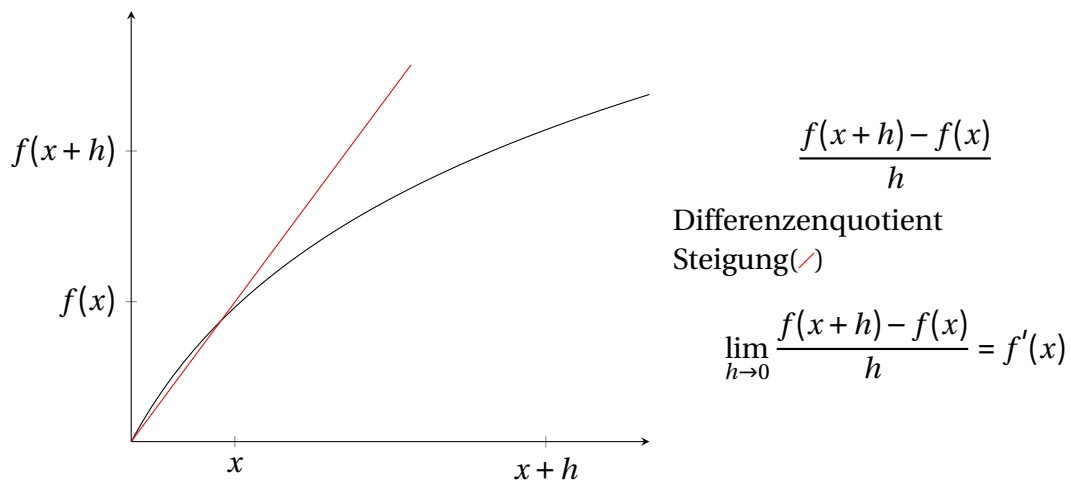
### Codebeispiel 34: Einfache Anwendung von Curry

```

(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)
5 (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:



Operator ' (Ableitung konsumiert Funktionen und produziert Funktion)  $\rightarrow$  ' ist höherer Order

### Codebeispiel 35: 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) = x3 + 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))

```

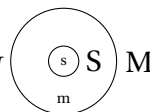


```

    (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 36: 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 ( $x \in S$ )?

```

```
(: set-member? (%a (set-of %a) -> boolean))
(define set-member?
  (lambda (x S)
    (S x)))
```

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

-> 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 37: Erweiterte Mengenoperationen

```
; Element x in Menge S hinzufügen: S ∪ {x}
(: set-insert (number (set-of number) -> (set-of number)))
(define set-insert
5   (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
25   (set-insert
      5
      (set-insert
        4
        (set-insert
          3
```

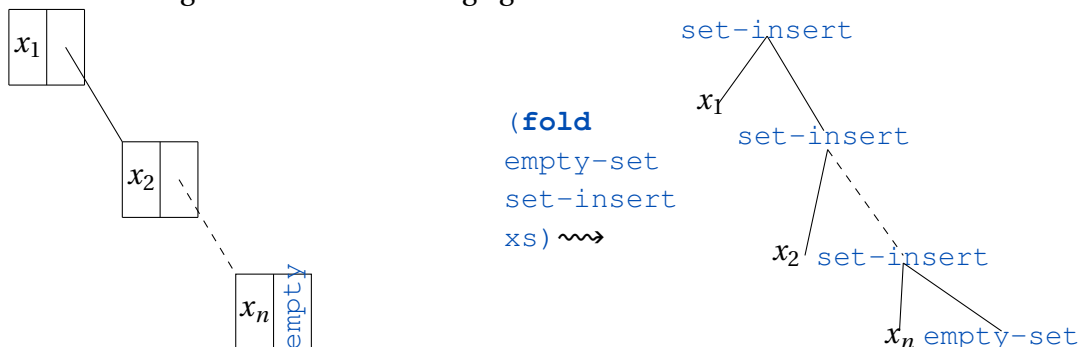
```

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

```

## 30.6.2015

Konvertierung Liste xs in eine Menge gleicher Elemente.



Codebeispiel 38: Konvertiert eine Liste zu einer Menge

```

; Konvertiere Liste xs in Menge
(: list->set ((list-of number) -> (set-of number)))
(define list->set
5   (lambda (xs)
      (fold empty-set set-insert xs)))

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

```

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

Weitere Mengenoperationen analog:

Codebeispiel 39: Mengenoperationen  $\setminus$ ,  $\cup$ ,  $\cap$ ,  $\Delta$

```

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

```

```

    (lambda (y)
      (if (= y x)
          #f
          (S y))))))
10
; S U T
; x ∈ S U T ⇔ x ∈ S ∨ x ∈ T
(: set-union ((set-of %a) (set-of %a) -> (set-of %a)))
(define set-union
15   (lambda (S T)
     (lambda (x)
       (or (S x) (T x))))))

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

; S \ T
; x ∈ S \ T ⇔ x ∈ S ∧ x ∉ T
30 (: set-difference ((set-of %a) (set-of %a) -> (set-of %a)))
(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

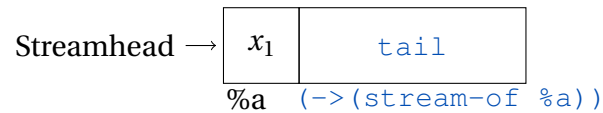
(3) Nachteile

- Elemente sind nicht auf zählbar

*Streams* (`stream-of %a`): unendliche Ströme von Elementen x, mit Signatur %a

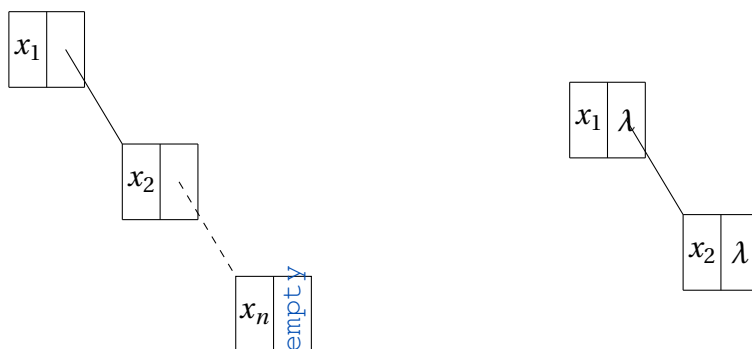
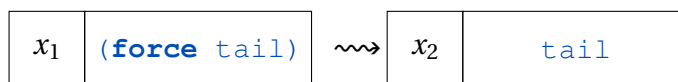
Ein Stream ist ein Paar:

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



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.

$(\text{delay } e) \equiv (\text{lambda } () \quad \begin{matrix} \uparrow \\ \text{nicht} \\ \text{ausgewertet} \end{matrix} \quad e \quad )$

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

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

### Codebeispiel 40: Streams

```
; Promise, ein Wert des Vertrags t zu liefern (0-stellig
  Prozedur)
(define promise
  (lambda (t)
```

```

    (signature (-> t))))

; Verzögerte Auswertung (delay)
;
10 ; Variante 1:
; (delay e)  (lambda () e)
;
; Variante 2 (nutzt selbstdefinierte Scheme-Syntax-Regel,
; verfügbar ab
; Sprachebene "DMdA - fortgeschritten"):
15 ;
; (define-syntax delay
;   (syntax-rules ()
;     ((_ e)
;      (lambda () e))))
20 ;
; Erzwungene Auswertung
(: force ((promise %a) -> %a))
(define force
  (lambda (p)
25    (p)))

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

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

; Polymorphe Paare (isomorph zu `pair')
40 (: make-cons (%a %b -> (cons-of %a %b)))
(: head ((cons-of %a %b) -> %a))
(: tail ((cons-of %a %b) -> %b))
(define-record-procedures-parametric cons cons-of
  make-cons
45  cons?
  (head

```

```

    tail))

; Ein Stream besteht aus
; - einem ersten Element (head)
50 ; - einem Promise, den Rest des Streams generieren zu können
    (tail)
(define stream-of
  (lambda (t)
    (signature (cons-of t (promise (stream-of t))))))
55

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

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

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

70 (define stream-take
    (lambda (n str)
      (if (= n 0)
          empty
          (make-pair (head str)
                     (stream-take (- n 1) (force (tail str))))))
75

; Beispiel (Stream Stream):
; Filtere Stream str bzgl. Prädikat p?
(: stream-filter ((%a -> boolean) (stream-of %a) -> (stream-of
  %a)))
80

(check-expect (stream-take 10
                        (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

```

```

90 (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

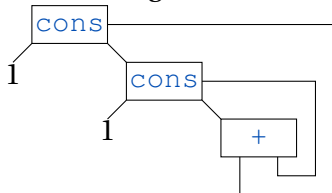
*Beobachtung:*

```

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

```

Stream-Diagramm zu fibs:



Codebeispiel 41: Stream aller Fibonacci Zahlen

```

; Beispiel (Streams Stream):
; Erzeuge neuen Stream durch die Anwendung von f
; auf die Heads der Streams str1, str2
5 (: stream-zipWith ((%a %b -> %c) (stream-of %a) (stream-of %b)
  -> (stream-of %c)))
(define stream-zipWith
  (lambda (f str1 str2)
    (make-cons (f (head str1) (head str2))
                (lambda () (stream-zipWith f
10                                (force (tail str1))
                                (force (tail
                                        str2)))))))

```



```

; Die unendliche Folge der Fibonacci-Zahlen 1, 1, 2, 3, 5, ...
15 (: fibs (stream-of natural))
   (check-expect (stream-take 10 fibs) (list 1 1 2 3 5 8 13 21 34
      55))
   (define fibs
     (make-cons
      1
      (lambda ()
        (make-cons
         1
         (lambda ()
           (stream-zipWith +
            fibs
            (force (tail fibs))))))))))
20
25

```

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  $T(M)$

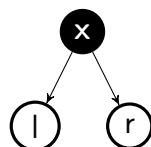
*Hinweis:*

- Jeder Knoten (`make-node`) in einem Binärbaum hat zwei Teilbäume sowie eine Markierung (`label`).
- Vergleiche:
  - $M^*$  und  $T(M)$
  - `empty` und `empty-tree`
  - `make-pair` und `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:

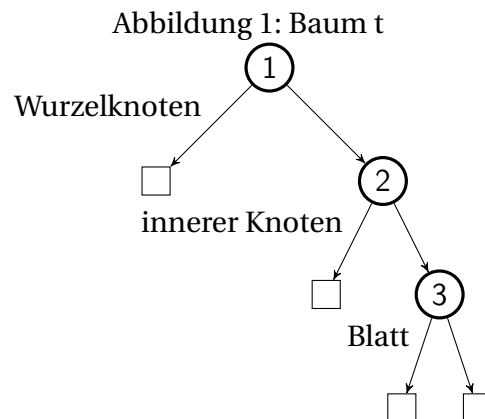
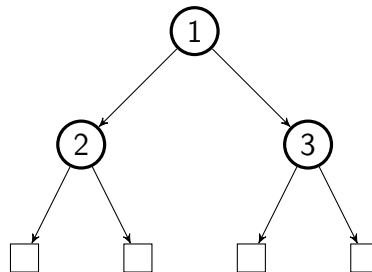


Abbildung 2: Baum  $t_2$  *balanciert*, alle Teilbäume auf einer Tiefe haben die selbe Anzahl an Knoten



- Suchbäume (z.B Datenbanken)
- Datenkompression
- Darstellung von Termen (Ausdrücken)

Bäume sind *die* Induktiv definierte Datenstruktur

#### Codebeispiel 42: 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")
                            "prequel?"
                            (make-leaf "Mace_Windu"))
                "dark_side?"
                (make-node (make-leaf "Emperor")
                            "pilot?"
                            (make-leaf "Darth_
80   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)))))))
```

## 7.7.2015

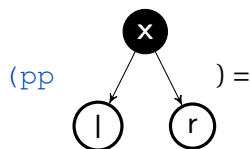
### Codebeispiel 43: Die Größe eines Baumes

```
; 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
10 (lambda (t)
    (cond ((empty-tree? t)
          0)
          ((node? t)
           (+ (btree-size (node-left-branch t))
              (btree-size (node-right-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 44: 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)))
                        (map ((curry string-append)
25

```

```

                                (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)
        (expect (print t) (write-string "\n")))))
40

```

### Induktion über Binärbäume

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

Falls  $(\text{empty-tree})$  und

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

dann

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

Induktionsbasis

Induktionsschritt

Beispiel:

Zusammenhang zwischen Größe ( $\text{btree-size}$ ) und Tiefe ( $\text{btree-depth}$ ) 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  $(P(l) \wedge P(r) \Rightarrow P(\text{make-node } l \ x \ r))$

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

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

[size]

$$\begin{aligned}
&= 2^{(\text{depth } l) - 1 + 1 + 2^{(\text{depth } r) - 1}} \\
&= 2^{(\text{depth } l)} + 2^{(\text{depth } r) - 1} \\
&\leq 2 \cdot \max(2^{(\text{depth } l)}, 2^{(\text{depth } r)}) - 1 \\
&= 2 \cdot 2^{\max(\text{depth } l, \text{depth } r)} - 1 \\
&\stackrel{\sim}{=} 2^{(\text{depth } (\text{make-node } l \times r))} - 1 \checkmark \\
&[\text{depth}]
\end{aligned}$$

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)))))))

```