

Informatik II Skript Sommersemester 2015

Finn Ickler

10. Juni 2015

Inhaltsverzeichnis

14.4.2015	3
16.4.2015	4
21.4.2015	6
23.4.2015	8
28.4.2015	10
30.4.2015	13
5.5.2015	17
7.5.2015	18
12.5.2015	23
19.5.2015	27
21.5.2015	32
9.6.2015	37

Codebeispiele

1	Arithmetik mit Fließkommazahlen	4
2	Schlüsselwort define	5
3	Lambda Abstraktion	5
4	Bilderzusammenstellung am Beispiel einer Uhr	7

5	Die one-of Signatur	10
6	Konstruktion eines eigenen Ifs?	10
7	Absolutbetrag durch cond	12
8	Boolsche Ausdrücke mit and und or	13
9	Record Definitionen	13
10	Check-property	15
11	Übersetzung mathematischer Aussagen in check-property	15
12	Konstruktoren und Selektoren	16
13	predicate Signaturen am Beispiel von Längen- und Breitengrade	18
14	Ersetzung one-of durch predicate Signaturen	18
15	Geocoding	20
16	cond mit gemischten Daten	21
17	Wrapper und Worker	23
18	make-pair, ein polymorpher Datentyp	25
19	Listen mit Signatur list-of	26
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

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) ~> #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
<code>#f, #t</code>	(true, false, Wahrheitswert)	boolean
<code>"x"</code>	(Zeichenketten)	String
<code>0 1904 42 -2</code>	(ganze Zahl)	Integer
<code>0.42 3.14159</code>	(Fließkommazahl)	real
<code>1/2, 3/4, -1/10</code>	(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)) ~> (+ 40 (+ 1 1)) ~> (+ 40 2) ~> 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)))
5
(- (+ 0.7
    (/ 1/2 0.25))
  (/ 0.6 0.3))
10
; Arithmetik mit rationalen Zahlen (rational) ist exakt

```

```
(- (+ 7/10
      (/ 1/2 1/4))
  (/ 6/10 3/10))
```

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

```
(define <id> <e>)      <id>Identifier <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

```
(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>)
```

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

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

. *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 Weltbevoelkerung 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 Sinaturen

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.

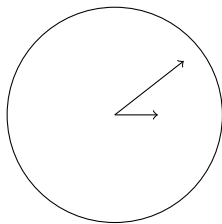
Spezialform: kein Wert, sondern Effekt: Testverletzung protokollieren

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 $(\text{lambda } (\dots) \dots) \rightsquigarrow (\text{lambda } (\dots) \dots)$ [eval_λ]
- Applikationen (f e₁ e₂...)

$$f, e_1, e_2 \text{ reduzieren erhalte: } f', e_1', e_2' \quad (1)$$

- (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)` $\rightsquigarrow_{\text{eval id}}$ `(#<procedure+> 40 2)` \rightsquigarrow 42

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

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

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

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

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

```

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 $t_i \# t$ ergibt, werte Zweig e_i aus. e_i ist Ergebnis der Fallunterscheidung.

Wenn $t_n \# t$ liefert, dann liefert

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

Codebeispiel 7: Absolutwert von x

```

5 (: 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
  (define my-abs
    (lambda (x)
      (cond ((< x 0) (- x))
            ((> x 0) x)
            (else 0))))

```

Reduktion von `cond` [$\text{eval}_{\text{cond}}$]

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

① Reduziere t_1 erhalte $t'_1 \rightsquigarrow \begin{cases} \langle e_{_1} \rangle & \text{falls } t'_1 = \#t \\ \text{② } (\text{cond } \langle t_{_2} \rangle \langle e_{_2} \rangle) & \text{sonst} \end{cases}$

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

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

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

```

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

  (if (<t1>
      <e1>
      (if <t2>
          (if <e2>
              ...
              (if <tn>
                  <en>
                  (else <en+1>))
              <en+1>))
      <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 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)
10      (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?
15     character-force))

```

```

; 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:

```

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))
      (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 11: Mathematische \forall -Aussage in Racket

```

; Für alle natürlichen Zahlen x1,x2 gilt: x1 + x2 ≥
  max(x1,x2)
5 (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)
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

```

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

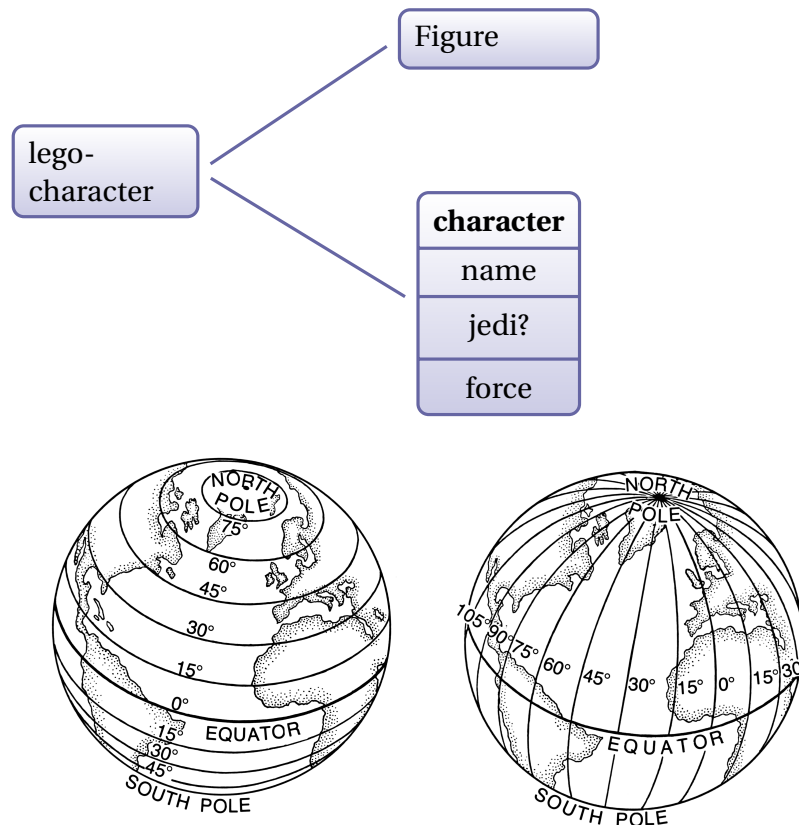
10 ; Bilde den Charakter c zum Jedi aus (sofern c überhaupt
    Macht besitzt)
(: train-jedi (character -> character))

(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)
20                      (> (character-force c) 0)
                      (* 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 $\langle \text{newt} \rangle$ (signature $\langle t \rangle$)

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)
(define longitude?
15 (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))

```

10

```
(define g
  (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 Ein geocode-error besteht aus:
- Südwestecke (southwest) location
- Typ (type) string
- Genauigkeit (accuracy) string

```
(: geocode-adress (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)
(define A81
25 (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>))

```

$$\begin{array}{ccc}
 & (\langle \text{id}_1 \rangle \langle e_1 \rangle) & \\
 & (\dots) & \\
 & (\langle \text{id}_n \rangle \langle e_n \rangle) & \\
 \text{5 } \langle e \rangle & \equiv & \langle e \rangle \\
) & & \langle e_1 \rangle \\
 & & \langle e_2 \rangle \dots \\
 & & \langle e_n \rangle
 \end{array}$$

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 ( $\pi = 180^\circ$ )
(: 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 lat2) (cos lng2))
            (* (cos lat1) (sin lng1)
              (cos lat2) (sin lng2))
            (* (sin lat1) (sin
              lat2))))))

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)
    (cond ((= i 1) x)
          ((= i 2) y))))

15

```

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"))
```

```

15 ; "Liste" der Zahlen 1,2,3,4
    (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 $\langle t \rangle$ 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 -> boolean))</code>	liegt leere Liste vor?
	<code>(: pair? (any -> boolean))</code>	Nicht leere Liste?
Selektoren:	<code>(: first (list-of %a)-> %a)</code>	Kopf-Element
	<code>(: rest (list-of %a)-> (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

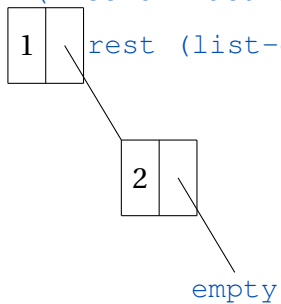
```
(make-pair 1 (make-pair 2 empty))
```

Visualisierung Listen

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

*Spine (Rückgrat)*

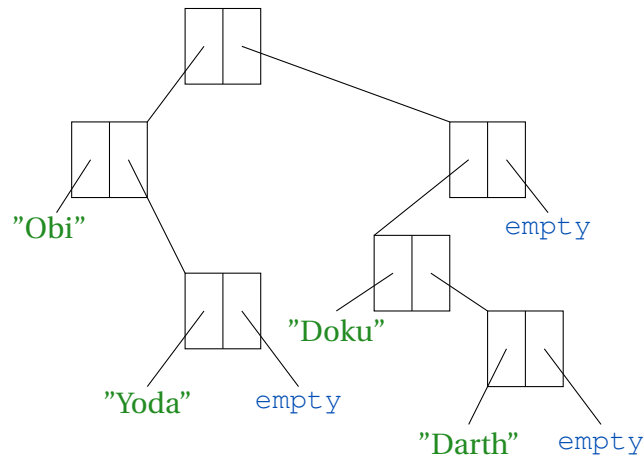
```
(pair-of natural (list-of natural))  
(natural first) 1 rest (list-of natural)
```



```
(: 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: Jedis und Siths in einer geschachtelten Liste

```

5 ; 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)))
10 ; 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
5                          (make-pair 2
                                      empty)))) 42)

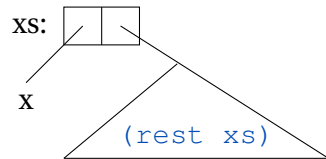
(check-expect (list-sum one-to-four) 10)

```

```

(define list-sum
10  (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) ... (first xs) ...
                       (underbrace{<t1>})
                       (underbrace{<f> (rest xs)}) ...))
    (underbrace{<t1>})

```

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 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)
10    (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)
10    (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 23: Ausflug: Bluescreen Berechnung wie in Starwars mit Listen:



```
(define yoda
```



```
(define dagobah
```

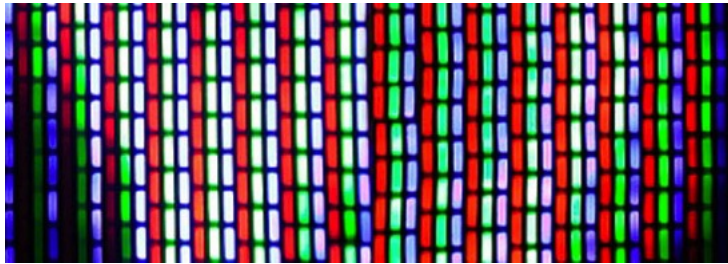
```

; -----
; 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)

```

```

                    (first fg))
                    (bluescreen (rest fg) (rest bg))))))

55
; 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 Einschränkungsoperator

```
; 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:

```

(<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
- (`<f> (- 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
(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))` \rightsquigarrow `(* 30240 (factorial 5))`
 Assoziativität von \cdot

→ Multiplikationen können vorgezogen werden :-)

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

Beispiel: Berechne 5!

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

n	acc	
\downarrow 5	1 $\downarrow \cdot 5$	neutrales Element
\downarrow 4	5 $\downarrow \cdot 4$	
\downarrow 3	20 $\downarrow \cdot 3$	
\downarrow 2	60 $\downarrow \cdot 2$	
\downarrow 1	120 $\downarrow \cdot 1$	
\downarrow 0	120	

```

; Berechne n!
; Wrapper
(: fac (natural -> natural))
5 (check-expect (fac 0) 1)
  (check-expect (fac 3) 6)
(define fac
  (lambda (n)
    (fac-worker n 1)))
10
; Berechne n! (mit Zwischenergebnis/Akkumulator acc),
  endrekursiv
; Worker
(define fac-worker
  (lambda (n acc)
15   (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 11000))`

$\overbrace{1000 \cdot \text{make-pair}}$

`(cat (list 1000 ... 2) (list 1))`
`(cat (list 1000 ... 3) (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 28: Effizientere Variante eine Liste umzudrehen

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

(check-expect (backwards empty) empty)
5 (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))))))
      (backwards-worker xs empty))))
```