INFORMATIKI

Skript

Kim Thuong Ngo

January 24, 2019

CONTENTS

1 Sprachebene: Die Macht der Abstraktion - Anfänger

1.1 Scheme: Ausdrücke, Auswertung und Abstraktion

Programm: DrRacket

Die Anwendung von Funktionen wird in Scheme ausschließlich in Präfixnotation durchgeführt:

Mathematik	Scheme
$44-2$	(- 44 2)
$f(x,y)$	(f x y)
$\sqrt{81}$	(sqrt 81)
$\lfloor x \rfloor$	(floor x)
9^2	(expt 9 2)
3!	(! 3)

Allgemein:

```
(<function> <argument> ... <argument> )
```

(+ 40 2) und (odd? 42) sind Beispiele für Ausdrücke, die bei der Auswertung einen Wert liefern.
Notation: $\rightsquigarrow$
(+ 40 2) $\rightsquigarrow$ 42 Auswertung/ Evaluation
(odd? 42) $\rightsquigarrow$ Reduktion
Interaktionsfenster : Read $\to$ Evaluation $\to$ Print $\to$ Read $\to$ ... (REPL)

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

Literal		Signatur
# t # f	Wahrheitswerte: true,false	boolean
" " "abc" "x"	Zeichenkette	string
007 0 1904 -42	ganze Zahlen	integer
0 1 2 3	natürliche Zahlen mit 0	natural
-273.15 0.42	Fließkommazahlen	real
$\frac{1}{2}$ $\frac{3}{4}$	rationale Zahlen	rational
Bild	Bilder	image

Auswertung zusammengesetzter Ausdrücke (composite expression) in mehreren Schritten (steps) "von innen nach außen", bis keine weitere Reduktion möglich ist.

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

Beispiel: $0.7 + \frac{1}{2}/0.25 - \frac{0.6}{0.3}$

Achtung! Scheme rundet bei Arithmetik mit Fließkommazahlen (interne Darstellung nicht präzise). Arithmetik mit rationalen Zahlen ist exakt.

Ein Wert kann an einen Namen (identifier) gebunden werden durch

```
(define <id> <expression>)
```

Erlaubt konsistente Wiederverwendung und dient der Selbstdokumentation von Programmen.
Achtung! Dies ist eine Spezialform und kein Ausdruck. Insbesondere besitzt diese Spezialform keinen Wert, sondern einen Effekt: der Name <id> wird an den Wert von <expression> gebunden.
Namen können in Scheme fast beliebig gewählt werden solange die Zeichen

1. die Zeichen () [] " ´ ‘ , ; | # nicht vorkommen
2. der Name nicht einem numerischen Literal gleicht
3. kein whitespace

enthalten ist.
Beispiel: euro->US$
Achtung! Groß- und Kleinschreibung ist im Identifier nicht relevant.
Eine Lambda-Abstraktion (auch Funktion oder Prozedur) erlaubt die Formulierung von Ausdrücken, in denen mittels Parametern von konkreten Werten abstrahiert wird:

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

(lambda ...) ist eine Spezialform. Wert der Lambda-Abstraktion # <procedure> Anwendung (auch Applikation) der Lambda-Abstraktion führt zur Ersetzung aller Vorkommen der Parameter im Rumpf durch die angegebenen konkreten Argumente:
Beispiel: ((lambda (days) (* days (* 155 minutes-in-a-day))) 365) ⇝ (* 365 (* 155 minutes-in-a-day)) ⇝ ... ⇝ 81 468 000

In Scheme leitet ein Semikolon ; einen Kommentar ein, der bis zum Zeichenende reicht und von Racket bei der Auswertung ignoriert wird. Prozeduren und Funktionen sollten im Programm eine ein- bis zweizeilige Kurzbeschreibung vorangestellt werden.

Eine Signatur prüft, ob ein Name <id> an einen Wert einer angegebenen Sorte gebunden wird. Signaturverletzungen werden protokolliert.

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

Bereits eingebaute Signaturen:

- natural $\mathbb{N}_0$
- integer $\mathbb{Z}$
- rational $\mathbb{Q}$
- real $\mathbb{R}$

- number $\mathbb{C}$
- boolean
- string
- image

Spezialform, kein Wert

Prozedur-Signaturen spezifizieren Signaturen sowohl für die Parameter <p1>,<p2>,... als auch für den Ergebniswert der Prozedur.

```
(: <id> (<signatur-p1> <signatur-p2> ... -> <signatur-ergebnis>))
```

Prozedur-Signaturen werden bei jeder Anwendung der Funktion <id> auf Verletzung geprüft.

Testfälle dokumentieren das erwartete Ergebnis einer Prozedur für ausgewählte Argumente.

```
(check-expect <expression1> <expression2>)
```

Wertet Ausdruck <expression1> aus und teste, ob der erhaltene Wert der Erwartung (= Wert des Ausdrucks <expression2>) entspricht. Eine Prozedurdefinition sollten Testfälle direkt vorangestellt werden.
Spezialform, keinen Wert, Testverletzungen werden protokolliert.

Konstruktionsformel für Prozeduren:

1. Kurzbeschreibung (ein- bis zweizeiliger Kommentar, eingeleitet durch Semikolon ;)
2. Signatur (: <id> (... -> ...))
3. Testfälle check-epect/ check-within
4. Rumpf programmieren

1.1.1 Top-Down-Entwurf

Programmieren mit Wunschdenken

Beispiel: Sunset auf Tatooine (Star Wars IV)
Zeichne Szene zu Zeitpunkt t (t=0 ... 100)

1. Himmel verfärbt sich
2. Sonne versinkt bei t=100
3. Luke starrt auf Horizont (bei jedem t)

Zeichne Tatooine Sunset zu Zeitpunkt t

```
(: tatooine (natural -> image))
(define tatooine
  (lambda (t)
    (overlay/pinhole (luke t)
                     (sun t)
                     (sky t))))
```

1.2 Reduktionsregeln für Scheme

Fallunterscheidungen: je nach Ausdruck

- Literale (1,# t, "abc", ...)
  1 $\rightsquigarrow$ 1 (keine Reduktion möglich)

- Identifier <id>
  <id> $\rightsquigarrow$ Wert
  Wert an den <id> gebunden ist [eval_{id}]

- Lambda-Abstraktion
  (lambda (...) ...) $\rightsquigarrow$ (lambda (...) ...) [eval_{λ}]

- Applikation
  1. f, e1, e2, ... mittels $\rightsquigarrow$ erhalte f', e1', e2' , ...
  2. – Operation auf e1',e2',... falls f' primitive eingebaute Operation [apply_{prim}]
     – Argumentwerte e1',e2',... in Rumpf einsetzen, dann Rumpf mittels $\rightsquigarrow$ reduzieren falls f' Lambda-Abstraktion [apply_{λ}]

Wiederhole Anwendungen von $\rightsquigarrow$ bis keine Reduktion mehr möglich ist.

Beispiele:

1. (+ 40 2)
   $\rightsquigarrow^{eval_{id},2xeval_{lit}}$ (# <procedure: +> 40 2)
   $\rightsquigarrow^{apply_{prim}}$ 42

2. (sqr 9)
   $\rightsquigarrow^{eval_{id},eval_{lit}}$ ((lambda (x) (* x x)) 9)
   $\rightsquigarrow^{apply_{\lambda}}$ (* 9 9)
   $\rightsquigarrow^{eval_{id}}$ (# <procedure: *> 9 9)
   $\rightsquigarrow^{apply_{prim}}$ 81

1.2.1 Lexikalische Bindung

Bezeichnen (lambda (x) (* x x)) und (lambda (r) (* r r)) die gleiche Funktion?
Antwort: JA

Achtung!: Das hat EInfluss auf das korrekte Einsetzen von Argumenten für Parameter (siehe apply$_\lambda$)
Das bindende Vorkommen eines Identifier <x> im Programmtext kann systematisch bestimmt werden:

1. (lambda (x) ...)
2. (define x ...)

Prinzip der lexikalischen Bindung
übliche Notation in der Mathematik: Fallunterscheidung!

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

1.3 Fallunterscheidungen

Tests (auch: Prädikate) sind Funktionen, die einen Wert der Signatur boolean liefern
Typische Primitive in Tests

```
(: = (number number -> boolean))
(: < (real real -> boolean))
(: string=? (string string -> boolean))
(: boolean=? (boolean boolean -> boolean))
(: zero? (number -> boolean))
```

weitere: odd?, even?, positive?, negative?, ...
Spezialform Fallunterscheidung (conditional)

```
(cond (<test1> <expression>)
        (<t2> <e2)
        ...
        (<tn> <en>)
        (else <en+1>)) ; optional
```

Führt die Tests in der Reihenfolge <t1>,<t2>,... durch. Sobald <ti> zu t ausgewertet, werte Zweig <ei> aus. <ei> ist das Ergebnis der Fallunterscheidung. Wenn <tn> f liefert, dann liefere

$$\begin{cases} < en+1 > \\ \text{Fehlermeldung: "cond. alle Tests ergaben f"} \end{cases}$$

Die Signatur one-of lässt genau einen der n aufgezählten Werte zu.

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

Reduktion von cond [eval$_{cond}$]

- (cond (<t1> <e1>) (<t2> <e2>) ...)
    1. Reduziere <t1>, erhalte <t1'>

2. $$\begin{cases} \text{<e1> falls <t1'> t} & <t2>,<e2>,...werden\textcolor{orange}{nicht}ausgewertet \\ (cond(<t2><e2>)...)sonst & <e1> nichtausgewertet \end{cases}$$

- (cond (else <en+1>)) $\leadsto$ <en+1>
- (cond) $\leadsto$ Fehler "cond, alle Tests ergaben #f "

1.3.1 Binäre Fallunterscheidung

```
(if <t1> <e1> <e2>) = (cond (<t1> <e1>) (else <e2>))
```

1.4 Zusammengesetzte Daten

Daten können interessante interne Struktur (Komponenten) aufweisen
Beispiel: ein Star Wars Charakter

name	"Luke Skywalker"
jedi?	#f
force	25
...	

```
; Ein Charakter (character) besteht aus
; - Name (name)
; - Jedi Status (jedi?)
; - St rke der Macht (force)
(define-record-procedures character
     make-character
     character?
     (character-name
     character-jedi?
     character-force))
```

(make-character n j f) $\leadsto$ <Konstruktion> (siehe oben)
(character-name <Konstruktion>) $\leadsto$ n Selektor (Komponenten auslesen)
(character-jedi? <Konstruktion>) $\leadsto$ j
(character-force <Konstruktion>) $\leadsto$ f

Records in Scheme
Record Definition legt fest

- Record-SIgnatur (Name)
- Konstruktor (baut aus Komponenten ein Record)
- Prädikat
- Liste von Selektoren (lesen je eine Komponente des Records)

```
(define-record-procedures <t>
      make-<t>
      <t>?
      (<t>-<component1>
       ...
      <t>-<cn>))
```

Liste der Selektoren legt Komponenten (Anzahl, Reihenfolge, Name) fest.
Signatur des Konstruktors/ der Selektoren für Record-Signatur <t> mit n Komponenten <c1> ... <cn>

```
(: make-<t> (<t1> ... <tn> -> <t>))
                      ; n Komponenten Signaturen
(: <t>-<ci> (<t> -> <ti>))
```

$\forall$ string n, boolean, natural f:
(character-name (make-character n j f)) $\rightsquigarrow$ n
(character-jedi? (make-character n j f)) $\rightsquigarrow$ j
(character-force (make-character n j f)) $\rightsquigarrow$ f

Aussagen über die Interaktion von zwei (oder mehr) Funktionen: algebraische Eigenschaft.

Spezialform check-property

```
(check-property
      (for-all ((<id1> <signatur1>)
                  ...
                  (<idn> <signaturn>))
       <expression>))
       ; Pr dikat, dass sich auf <id1> ... <idn> bezieht
```

Test erfolgreich, falls <expression> für beliebige Bindungen für <id1> ... <idn> immer #t ergibt.
Interaktion von Konstruktor und Selektor

```
(for-all ((n string)
             (j boolean)
             (f natural))
(string=? (character-name
                        (make-character n j f)) n)))
```

Beispiel: Die Summe zweier natürlicher Zahlen ist mindestens so groß wie jede dieser Zahlen.
$\forall x_1, x_2 \in \mathbb{N} : x_1 + x_2 \geq max(x_1, x_2)$

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

Konstruktion von Funktion f, die zusammengesetzte Daten der Signatur <t> konsumiert.
Welche Record-Komponenten <ci> sind relevant für <f>?

```
; Schablone
(: <f> (... <t ... -> ...))
(define <f>
   (lambda ( ... r ...)
       ( ... <t>-<ci> r) ...)))
```

Prozedur <f> die zusammengesetzte Daten der Signatur <t> konstruiert/ produziert Konstruktoraufruf für <t> muss enthalten sein!

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

Sei <p> ein Prädikat mit Signatur (<t> -> boolean).
Eine Signatur (predicate <p>) gilt für jeden Wert x mit Signatur <t> für den zusätzlich (<p> x) $\leadsto$ #t gilt.
Signatur (predicate <p>) ist damit spezifischer (restriktiver) als Signatur <t>.
Einführung eines neuen Signaturnamens <new-t> für die Signatur <t>

```
(define <new-t (signature <t>))
```

Beispiele:

```
; Farbe einer Karte
(define farbe
     (signature (one-of "karo" "herz" "pik" "kreuz")))

; Breitengrad - latitude? ist ein Pr dikat
(define latitude
     (signature (predicate latitude?)))
```

Übersetze eine Ortsangabe mittels Google Geocoding
API in eine Position auf der Erdkugel

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

Ein geocode besteht aus ...

	Signatur
Adresse (address)	string
Ortsangabe (loc)	location
Norostecke (northeast)	location
Südwestecke (southwest)	location
Typ (type)	string
Genauigkeit (accuracy)	string

1.5 Gemischte Daten

Die Signaturen mixed (mixed <t1> ... <tn>) ist gültig für jeden Wert, der mindestens eine der Signaturen <t1> ... <tn> erfüllt.
Beispiel 1: Datendefinition
Die Antwort des Geocoders ist entweder

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

```
(mixed geocode geocode-error)
```

Beispiel 2: (eingebaute Funktion string -> number)

```
(: string->number (string -> (mixed number (one-of #f))))
```

(string-> number "42") ⇝ 42
(string->number "fortytwo") ⇝ #f
Das Prädikat <t>? einer Record-Signatur <t> unterscheidet Werte der Signatur <t> von allen anderen Werten:

```
(: <t>? (any -> boolean))
```

Auch: Prädikate für eingebaute Signaturen

- number?
- complex?
- real?
- rational?
- integer?
- natural?
- string?
- boolean?

Prozeduren, die gemischte Daten der Signaturen <t1> ... <tn> konsumieren

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

Mittels let lassen sich Werte an lokale Namen binden: (let ((<id1> <e1>) ... (<idn> <en>)) <e>)
Die Ausdrücke <e1> ... <en> werden parallel ausgewertet.
⇒ <id1> ... <idn> können in <e> (und nur dort! verwendet werden.
Der Wert des let-Ausdrucks ist der Wert von <e>
nur dort:

- Verwendung nur in <e>, nicht in den <ei>!
- lokal: Verwendung nicht außerhalb des (let ...)

Achtung! Sprachlevel: Die Macht der Abstraktion

```
; syntaktischer Zucker

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

((lambda          (<id1> ... <idn>) <e>)
    <e1> ... <en>)
```

(check-error <e> <message>) erwartet Programmabbruch mit Fehlermeldung <msg>.
Erzwingen des Programmabbruchs mittels

```
(violation <msg>)
```

1.6 Polymorphe Signaturen

Beobachtung: Manche Prozeduren arbeiten unabhängig von den Signaturen ihrer Argumente
parametrisch polymorphe Prozeduren (griech. vielgestaltig).
Nutze Signaturvariablen
Beispiel:

```
; Identit t
(: id (%a -> %a))
(define id (lambda (x) x))

; konstante Funktion (ignoriert 2. Argument)
(: const (%a %b -> %a))
(define const (lambda (x y) x))

; Projektion (ein Argument ausw hlen)
(: proj ((one-of 1 2) %a %b -> (mixed %a %b)))
(define proj
     (lambda (i x y)
          (cond
               ((= i 1) x)
               ((= i 2) y))))
```

Beachte: Parametrisch polymorphe Prozeduren "wissen nichts" über ihre Argumente mit Signatur %a,%b,... und können diese nur reproduzieren oder zu andere polymorphe Prozeduren weiterreichen

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

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

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

1.7 Polymorphe Paare und Listen

```
; ein polymorphes Paar (pair) besteht aus
; - erste Komponente (first)
; - zweite Komponente (rest)
; wobei die Komponenten beliebige Werte sind
(define-record-procedures-parametric pair pair-of
      make-pair
      pair?
      (first
      rest))
```

(pair-of <t1> <t2>) ist eine Signatur für Paare deren erste und zweite Komponente von der Signatur <t1> bzw. <t2> sind.

```
(: make-pair (%a %b -> (pair-of %a %b)))

(: first ((pair-of %a %b) -> %a))
(: rest ((pair-of %a %b) -> %b))
```

Eine Liste von Werten der Signatur <t>, (list-of <t>), ist entweder

- leer (Signatur empty-list) oder
- ein Paar (Signatur pair-of) aus
  - einem Listenkopf (Signatur <t>) und
  - einer Restliste (Signatur (list-of <t>)

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

(list-of <t>) Listen, deren Elemente die Signatur <t> besitzen
Signatur empty-list bereits in Racket vordefiniert
ebenfalls vordefiniert

```
(: empty empty-list)
(: empty? (any -> boolean))
```

1.7.1 Visualisierung von Listen

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

Spines (Rückgrat)

1.7.2 Prozeduren über Listen

Schablonen für gemischte und zusammengesetzte Daten
Beispiel:

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

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

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

Schablone für Funktion <f>, die Liste xs konsumiert

```
(: <f> ((list-of <t1>) -> <t2>))

(define <f>
     (lambda (xs)
          (cond
               ((empty? xs) ... )
               ((pair? xs) ... (first xs)
                               ... (<f> (rest xs)) ... ))))
```

- Signatur (list-of %a) eingebaut
- neuer syntaktischer Zucker eingebaut

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

Ausgabeformat für nicht leere Listen
<list <e1> <e2> ... <en»

Füge Listen xs, ys zusammen (concatenate)
2 Fälle (xs leer oder nicht leer)

$$1)\quad \overbrace{empty}^{xs} \quad \overbrace{y_1 y_2 \ldots y_m}^{ys} \quad \overbrace{y_1 y_2 \ldots y_m}^{(catxsys)} \quad 2)$$

$$x_1 x_2 \ldots x_n \quad y_1 y_2 \ldots y_m \quad \underbrace{\underbrace{x_1}_{(firstxs)} \; \underbrace{x_2 \ldots x_n}_{(restxs} \underbrace{y_1 y_2 \ldots y_m}_{ys}}_{(cat(restxs)ys)}$$

Bemerkungen:

- die Länge von xs (hier n) bestimmt die Anzahl der rekursiven Aufrufe
- auf ys werden keine Selektoren angewandt

2.1 MATCH

Spezialform match
vergleicht einen Wert <e> mit gegebenen Patterns <pat1> <pat2> ... <patn>
Falls <pati>, $1 \leq i \leq n$, das erste Pattern ist, das auf <e> matched, ist Zweig <ei> das Ergebnis (ansonsten wir die Auswertung mit "keiner der Zweige passte" abgebrochen

```
(match <e>
   (<pat1> <e1>)
   (<pat2> <e2>)
   ...
   (<patn> <en>))
```

Pattern Matching

1. Literal <l>
   <e> matched, falls <e> $\rightsquigarrow$ <l>

2. Don't Care $<e>matchedimmerVariable<v><e>matchedimmer,danachist<v>andenWertvon<e>in<ei>gebunden$

3. Record Konstruktor (<c> <pat1> ... <patk>) $k \geq 0$
   <e> matched, wenn es durch (<c> <x1> ... <xk>) konstruiert und <xj> auf <patj> matched, $1 \leq j \leq k$
   Achtung! Fall 4 ermöglicht Pattern Matching auf komplex konstruierten Werten

2.2 Rekursion über natürliche Zahlen

Die natürlichen Zahlen (vgl. gemischte Daten)
Eine natürliche Zahl (natural) ist entweder

- die Null 0 (zero)
- die Nachfolger (succ) einer natürlichen Zahl $\mathbb{N} = 0, (succ 0), (succ(succ 0)), \ldots$

Konstruktoren

```
(: zero natural)
(define zero 0)

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

Bedingte algebraische Eigenschaften (siehe check-property)
(... $\underbrace{<p>}_{Prädikat}$ $\underbrace{<e>}_{Ausdruck}$)
Nur wenn <p> $\rightsquigarrow$ #t, wird der Ausdruck <e> ausgewertet und getestet, ob <e> $\rightsquigarrow$ #t
Beispiel: Fakultätsfunktion $n!(n \in \mathbb{N})$:
$0! = 1$
$n! = n * (n-1)! \equiv (succ n)! = (succ n) * n!$

$3! = 3 * 2! = 3 * (2 * 1!) = 3 * (2 * (1 * 0!)) = 3 * (2 * (1 * 1)) = 6$

```
; Berechne n!
(: factorial (natural -> natural))
(define factorial
    (lambda (n)
         (cond
             ((zero? n) 1)
             (else (* (factorial (- n 1)) n)))

(define factorial2
     (lambda (n)
```

```
        (cond
            ((= n 0) 1)
            (else (* (factorial (- n 1)) n))))
```

Schablone für Funktionen <f>, die natürliche Zahl konsumieren:

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

SATZ
Eine Funktion, die nach der Schablone für Listen oder natürliche Zahlen geschrieben ist, terminiert immer (= liefert immer ein Ergebnis)

Reduktion kann durchaus zur Konstruktion von Ausdrücken führen, die zunehmende Größe aufweisen (für factorial bestimmt das Argument die Größe)

Wenn möglich erzeuge Reduktionsprozesse, die konstanten Platzverbrauch - unabhängig von Funktionsargumenten - benötigen

Beobachtung:
(Assoziativität von Multiplikation)
(* 10 (* 9 (* 8 (* 7 (* 6 (factorial 5))))))
= (* (* (* (* (* 10 9) 8) 7) 6) (factorial 5)9
= (* 30240 (factorial 5))
⇒ Multiplikation kann verzogen werden

Idee:
Führe Multiplikationen jeweils sofort aus
Schleife das Zwischenergebnis (akkumulierendes Argument) durch die Berechnung
am Ende enthält der Akkumulator das Endergebnis

Berechne 5!

n	acc
5	1
4	5
3	20
2	60
1	120
0	120

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

```
    (lambda (n acc)
        (cond
            ((= n 0) acc)
            ((> n 0) (fac-worker (- n 1) (* acc n))))))

; wrapper
(: fac (natural -> natural))
(define fac
    (lambda (n)
        (fac-worker n 1)))
```

Ein Reduktionsprozess ist iterativ, falls seine Größe konstant bleibt.
Damit ist factorial nicht iterativ, aber fac-worker ist iterativ

Wieso ist fac-worker iterativ?
Der rekursive Aufruf ersetzt den aktuell reduzierten Ausdruck vollständig. Es gibt keinen Kontext (ungebundenen Ausdruck), der auf das Ergebnis des rekursiven Aufrufs "wartet"
Kontext des rekursiven Aufrufs

- factorial (* n /hbox)
- fac-worker -keiner-

Ein Prozeduraufruf ist endrekursiv, wenn er keinen Kontext besitzt
Prozeduren, die nur endrekursive Prozeduraufrufe enthalten, heißen selbst endrekursiv.
Endrekursive Prozeduren führen zu iterativen Reduktion

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

$$\overbrace{(cat \quad \underbrace{(list 1000 999 ... 2)}_{\underbrace{(cat (list 1000 999 ... 3) (list 2 1))}_{999 Aufrufe von make-pair}} \quad (list 1))}^{1000 Aufrufe von make-pair}$$

$\Rightarrow$ Anzahl der Aufrufe von make-pair: $1000 + 999 + ... + 1$
rev auf einer Liste der Länge n
$\sum_{i=n}^{n} = \frac{1}{2} * n * (n+1)$ quadratisch in n

Konstruiere iterative Listenumkehr (backwards)
Berechnung von (backwards (list 1 2 3))

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

xs	acc
(list 1 2 3)	empty
(list 2 3)	(list 1)
(list 3)	(list 2 1)
empty	(list 3 2 1)

2.3 LETREC

Mittels letrec lassen sich Werte an lokale Namen binden

```
(letrec ((<id1> <e1>)
             ...
             <idn> <en>))
   <e>)
```

Die Ausdrücke <e1> ... <en> dürfen selbst auf die Namen <id1> ... <idn> beziehen. Der Wert des gesamten letrec-Asudrucks ist der Wert von <e>.

2.4 Induktive Definitionen

Konstruktive Definition der natürlichen Zahlen N

Definition (Peano Axiome)

P1	$0 \in \mathbb{N}$ (Null)
P2	$\forall n \in \mathbb{N} : succ(n) \in \mathbb{N}$ (Nachfolger)
P3	$\forall n \in \mathbb{N} : succ(n) \neq 0$
P4	$\forall m, n \in \mathbb{N} : succ(m) = succ(n) \Rightarrow m = n$
P5 Induktsionsaxiom	$\mathbb{N}$ enthält nicht mehr als 0 und die durch die succ() generierte Elemente nichts sonst ist in $\mathbb{N}$ für jede Menge $M \subseteq \mathbb{N}$: Falls $0 \in M$ und $\forall n : (n \in M \Rightarrow succ(n) \in M)$, dann ist $M = \mathbb{N}$

2.4.1 Beweisschema der vollständigen Induktion

Sie P(n) ist eine Eigenschaft einer Zahl $n \in \mathbb{N}$ (Prädikat)

```
(: P (natural -> boolean))
```

Ziel: Zeige $\forall n \in \mathbb{N} : P(n)$
Definiere: $M := n \in \mathbb{N} | P(n) gilt \subseteq \mathbb{N}$
Induktionsaxiom P5 für M
Falls $0 \in M$ und $\forall n : (n \in M \Rightarrow succ(n) \in M)$, dann ist $M = \mathbb{N}$
Falls P(0) und $\forall n : (P(n) \Rightarrow P(succ(n))$, dann $\forall n \in \mathbb{N} : \mathbb{P}()$
Beispiel 1:
$1 = 1$
$1 + 3 = 4$

$1+3+5=9$
$1+3+5+7=16$
[
$\sum_{i=0}^{n}(2i+1)$Summer der ersten n+1 ungeraden natürlichen Zahlen=(n+1)^2] $\equiv P(n)$

Zeige: $\forall n \in \mathbb{N}: P(n)$
Induktionsbasis P(0)
$\sum_{i=0}^{0}(2i+1) = 2*0+1 = 1 = (0+1)^2$
Induktionsschritt $\forall n: P(n) = P(n+1)$
$\sum_{i=0}^{n+1}(2i+1) = \sum_{i=0}^{n} +2*(n+1)+1$
$=^{I.V.} (n+1)^2 + 2n + 3 = n^2 + 4n + 4 = (n+2)^2$

Beispiel 2:

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

$P(n) \equiv [(factorailn) = n!]$
Zeige: $\forall n \in \mathbb{N}: P(n)$
Induktionsbasis P(0)
(factorial 0)
$\rightsquigarrow$ ((lambda (k) ...) 0)
$\rightsquigarrow$ (if (= 0 0) 1 ...)
$\rightsquigarrow$ (if #t 1 ...)
$\rightsquigarrow$ 1=0!
Induktionsschritt $\forall n: (P(n) \Rightarrow P(n+1))$
(factorial (n+1))
$\rightsquigarrow$ ((lambda (k) ...) n+1)
$\rightsquigarrow$ (if (= n+1 0) ... (* ...))
$\rightsquigarrow$ (if f ... (* ...))
$\rightsquigarrow$ (* n+1 (factorial (- n+1 1)))
$\rightsquigarrow$ (* n+1 (factorial n))
$=^{I.V.}$ (* n+1 n!) = (n+1)!

Annahme: - realisiert Differenz korrekt, * realisiert Multiplikation korrekt

Beispiel
Jedes f, das sich an die Schablone für Funktionen über natürliche Zahlen hält liefert immer ein Ergebnis (terminiert immer)

Sei f also definiert durch:

```
(: f (natural -> %a))
(define f
    (lambda (n)
        (if (= n 0)
            basis
            (step (f (- n 1)) n))))

; Bemerkung
(: basis %a)                        ; Ausdruck
(: step (%a natural -> %a))   ; totale Funktion
```

Dann gilt: $P(n) \equiv$ f(n) terminiert mit Ergebnis der Signaur $\%a$

Beweis:
Induktionsbasis P(0)
f(0)
$\leadsto$ (if (= 0 0) basis ...)
$\leadsto$ (if t basis ...)
$\leadsto$ basis
Induktionsschritt $\forall n : (P(n) \Rightarrow P(n+1))$
(f n+1)
$\leadsto$ (if (= n+1 0) ... (step ...))
$\leadsto$ (if f ... (step ...))
$\leadsto$ (step (f (- n+1 1)) n+1)
$\leadsto$ (step $\underbrace{(f\,n)}$terminiert mit EIgenschaft R (n+1))
$\leadsto$ (step R n+1) terminiert

2.4.2 Definition Listen

die Menge M* (= listen mit Elementen aus M (list-of M)) ist induktiv definiert
L1 empty ∈ M*
L2 $\forall x \in m, xs \in M*$:(make-pair x xs) ∈ M* Schema für Listeninduktion
L3 sonst ist in M*
Sei P(xs) eine Eigenschaft von Listen über M:

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

Falls P(empty) Induktionsanfang bzw. Induktionsbasis ud $\forall x \in M, xs \in M* : (P(xs) \Rightarrow$ P((make-pair x xs))) Induktionsschritt, dann $\forall xs \in M* : P(xs)$

Beispiel: Eigenschaften von cat und append

```
(define cat
    (lambda (xs ys)
```

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

1	(cat empty ys)	ys
2	(cat xs empty)	xs
3	(cat (cat xs ys) zs)	(cat xs (cat ys zs))

(M*, cat, empty) ist ein Monoid

Beweis:

Beispiel: Interaktion von length und cat

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

$ys \in M*$ sei beliebig
$P(xs) \equiv$ (length (cat xs ys)) = (+ (length xs) (length ys))

Beweis:
Induktionsbasis P(empty):

Induktionsschritt

2.5 Prozeduren höherer Ordnung (Higher Order Procedures HOP)

Abstraktion von Funktionsparameter

```
; extrahiere die Elemente xs, die das Pr dikat p? erf llen
(: filter ((%a -> boolean) (list-of %a) -> (list-of %a)))

(define filter
  (lambda (p? xs)
    (cond
      ((empty? xs) empty)
      ((pair? xs) (if (p? (first xs))
                      (make-pair (first xs)
                                 (filter p? (rest xs)))
                      (filter p? (rest xs)))))))
```

Prozeduren höherer Ordnung (Higher Order Procedures kurz. HOP)

1. akzeptieren Prozeduren als Parameter und/oder
2. liefern eine Prozedur als Ergebnis

⇒ filter ist vom Typ1

HOP vermeiden Duplizierung von Code und führen zu:

- kompaktere Programme
- verbesserte Lesbarkeit
- verbesserte Wartbarkeit

2.5.1 MAP

Beispiel: (map f xs)

```
; wende f auf alle Elemente von xs an
(: map ((%a -> %b) (list-of %a) -> (list-of %b))

(define map
    (lambda (f xs)
        (cond
            ((empty? xs) empty)
            ((pair? xs) (make-pair (f (first xs))
                                         (map f (rest xs)))))))
```

Hinweis
Verwende einfache Lambda-Abstraktion direkt als anonyme Funktion. Wenn eine globale Benennung (via define) nicht gerechtfertigt erscheint. (z.B. lokaler/ einmaliger Benutzung)

2.5.2 LISTENFALTUNG (FOLD)

Allgemeine Transformation von Listen Listenfaltung (list folding)
Idee: die Listenkonstruktoren und empty werden systematisch ersetzt:

(foldr z c xs) wirkt als Spine Transformer

- empty → z
- make-pair → c

Eingabe: Liste (list-of %a)
Ausgabe: im allgemeinen keine Liste (etwa %b)

```
(: 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)))))))
```

Beispiel: Reduktion von xs
Spine Transformation

- empty → 0
- (make-pair y ys) → (lambda (y ys) (+ 1 ys))

Teachpack "universe" nutzt HOP, um Animation (= Sequenzen von Szenen / Bildern) zu definieren

```
(big-bang <init>
               (on-tick <tock>)
               (to-draw <render> <w> <h>))
```

- (: <init> %a)
  Startzustand
- (: <tock> (%a -> %a))
  Funktion, die neuen aus alten Zustand berechnet, wird 28 Mal/Sekunde aufgerufen
- (: <render> (%a -> image))
  Funktion, die aus aktuellen Zustand eine Szene berechnet (wird in Fenster mit <w>x<h> Pixeln angezeigt)
- beim Schließen der Animation wird der letzte aktuelle Zustand zurückgegeben

2.5.3 Komposition von Funktionen (allgemein)

((compose f g) x) ≡ (f (g x))
Mathematik: (compose f g) ≡ $f \circ g$ "f nach g"
⇒ compose konstruiert aus f und g eine neue Funktion ("Funktionsfabrik")

```
(: compose ((%b -> %c) (%a -> %b) -> (%a -> %c)))

(define compose
     (lambda (f g)
        (lambda (x)
            (f (g x)))))
```

repeat: n-fache Komposition einer Funktion f mit sich selbst
(n-fache Anwendung von f, Exponentiation):

- $f^0 = id$ (Identität id ≡ (lambda (x) x)
- $f^n = f^{n-1} \circ f$

```
(: repeat (natural (%a -> %a) -> (%a -> %a)))

(define repeat
    (lambda (n f)
        (cond
            ((= n 0) (lambda (x) x))
            ((> n 0) (compose (repeat (- n 1) f) f)))))

; greife auf das n-te Element von xs zu (n > 0)
(: nth (natural (list-of %a) -> %a))

(define nth
    (lambda (n xs)
         ((compose first (repeat (- n 1) rest)) xs)))
```

2.5.4 Currying

Reduktion von ((add 1) 41)
$\leadsto_{eval(id)}$ (((lambda (x) (lambda (y) (+ x y))) 1) 41)
$\leadsto_{apply\lambda(\lambda x)}$ ((lambda (y) (+ 1 y)) 41)
$\leadsto_{apply\lambda(\lambda y)}$ (+ 1 41)
$\leadsto_{apply,prim(\lambda)}$ 42

- Currying (Haskell B. Curry, Moses Schönfinkel)
  Anwendung einer Funktion auf ihre erstes Argument liefert ein Funktion der restlichen Argumente
- jede n-stellige Funktion lässt sich in eine alternative curried Variante transformieren, die in n-Schritten jeweils nur ein Argument konsumiert: curry

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

(: uncurry ((%a -> (%b -> %c)) -> (%a %b -> %c)))
(define uncurry
   (lambda (f)
```

```
        (lambda (x)
            (lambda (y)
                (f x y))))
```

Erinnerung: Bestimmung der ersten Ableitung der reellen Funktion f durch Bildung des Differenzenquotienten:

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

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

Operator ' (Ableitung) konsumiert FUnktion f und produziert f'
⇒ ' ist higher order

2.6 Streams

```
(stream-of %a)
```

unendliche Ströme von Elementen x, der Signatur $\%a$
Ein Stream ist ein Paar

Verzögerte Auswertung eines Ausdrucks (delayed evaluation):

```
(: delay (%a -> (-> %a)))
```

(delay <e>): verzögere die Auswertung des Ausdrucks <e> und liefere "Versprechen" (promise) , <e> bei Bedarf später auswerten zu können
Implementation
(delay <e>) ≡ (lambda () <e>)
(force <p>): Erzwinge Auswertung des Promise <p>, liefere den Wert des verzögerten Ausdrucks

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

2.6.1 Sieb des Eratosthenes

(Generierung aller Primzahlen; Stream-Programm über 200 Jahre alt)

1. starte mit dem Stream str der Zahlen 2,3,4,...
2. Die erste Zahl n im str ist eine Primzahl
3. Streiche alle Vielfachen von n im Stream str

4. weiter bei 2.

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~ ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~ ...

2.7 Binärbäume

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

T1	empty-tree ∈ T(M)
T2	$\forall x \in M, l, r \in T(M)$: (make-node l x r) ∈ T(M)
T3	Nichts sonst ist in T(M)

Hinweise:

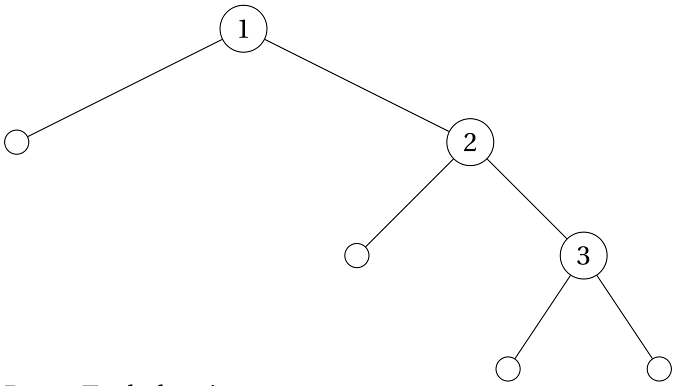
- Jeder Knoten (made-node) in einem Binärbaum hat 2 Teilbäume l und r sowie eine Markierung (label) $x \in M$
- vgl. M* und T(M)
  empty-list und empty-tree
  make-pair und make-node

Visualisierung und Terminologie

- empty-tree:
- (make-node l x r):
- der Knoten mit Markierung x ist Wurzel (root) des Baumes
- ein Knoten, der nur leere Teilbäume besitzt, heißt Blätter
  alle anderen Knoten sind innere Knoten (inner nodes)

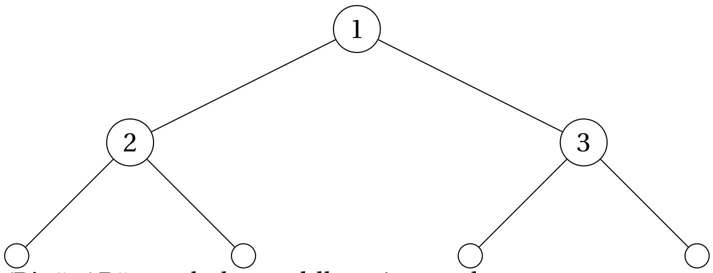
Beispiele für Binärbäume der Menge T(ℕ)
Baum T1: listenartig (rechtstief)

- Knoten mit Label 1 ist Wurzel
- Knoten 3 ist Blatt
- Knoten 1,2 sind innere Knoten



Baum T2: balanciert

- Knoten 1 ist Wurzel und innerer Knoten
- Knoten 2,3 sind Blätter



(Binär-)Bäume haben zahllose Anwendungen

- Suchbäume (schneller Zugriff, z.B. Datenbanksysteme)
- Datenkompressionen
- Darstellung von Programmen/ Ausdrücken im Rechner
- ...

Bäume sind DIE induktive Datenstruktur der Informatik!

2.7.1 Tiefe eines Baumes

Die Tiefe (depth) eines Binärbaumes t ist die maximale Länge eines Weges von der Wurzel bis zu einem leeren Teilbaum

(btree-depth empty-tree)	0
(btree-depth T2)	2
(btree-depth T1)	3
(btree-depth classifier)	4

Schablone (gemischte + zusammengesetzte 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)))
                                                    )))))
```

2.7.2 Pretty Printing

Einschub: Pretty Printing von Binärbäume
Prozedur (pp <t>) erzeugt formatierte String Binärbäume <t>

Idee: Repräsentiere formatierenden String als Liste von Zeilen (String)

1. Nutze (string-append ...) um Zeilen-Strings zu definieren (horizontal)
2. Nutze (append ...) um die einzelnen Zeilen zu einer Liste von Zeilen zusammen zu setzen (vertikal)

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

2.7.3 Induktion über Binärbäume

Sei P(t) eine Eigenschaft von Binärbäumen $t \in T(M)$, also

```
(: P ((b-tree-of M) -> boolean))
```

Falls P(empty-tree) [Induktionsbasis] und $\forall x \in M, l, r \in T(m) : P(l) \land P(r) \Rightarrow$ P((make-node l x r)) [Insuktionsschritt] dann $\forall t \in T(M) : P(t)$

Beispiel: Zusammenhang zwischen Größe (btree-size) und Tiefe (btree-depth) eines Binärbaumes t:

$$P(t) \equiv (btree - deptht) \leq (btree - sizet) \leq 2^{(btree-deptht)} - 1$$

Induktionsbasis: P(empty-tree)
Induktionsschritt: $P(l) \land P(r) \Rightarrow$ P((make-node l x r)) /box

2.7.4 Tree-Transformer

Erinnerung: fold ist Spine-Transformer üfr Liste xs
Wie müsste btree-fold, eine fold-Operation für Binärbäume, verhalten?

Tree-Transformer für Bäume t:

```
; falte Baum bzgl. z und c
(: btree-fold (%b (%b %a %b -> %b) (btree-of %a) -> %b)))

(define btree-fold
    (lambda (z c t)
        (cond
            ((empty-tree? t) z)
            ((node? t) (c (btree-fold z c (node-left-branch t))
                                  (node-label t)
                                  (btree-fold z c (node-right-branch t)))))))
```

Beispiel: Bestimme die Markierung lm links-außen im Baum t (oder empty, falls t leer ist)

```
(: leftmost ((btree-of %a) -> (list-of %a)))
(define leftmost
    (lambda (t)
        (btree-fold empty
                         (lambda (l1 x l2)
                              (if (empty? l1) (list x) l1))
                         t)))
```

2.7.5 Baumdurchläufe

Ein Tiefendurchlauf (depth-first-traversal) eines Baumes t sammelt die Markierungen jedes Knoten n in t auf. Die markierungen der Teilbäume l,r des Knotens n=(make-node l x r) werden vor x eingesammelt (Durchlauf zuerst in der Tiefe)
Je nachdem, ob x (a) zwischen, (b) vor, (c) nach den Markierungen von l,r eingeordnet wird, erhält man ein

(a)	inorder traversal	1 2 3
(b)	preorder traversal	2 1 3
(c)	postorder traversal	1 3 2

```
(: inorder ((btree-of %a) -> (list-of %a)))
(define inorder
    (lambda (t)
        (btree-fold empty
                         (lambda (xs1 x xs2)
                             (append xs1 (list x) xs2))
                          t)))
```

Beispiel: Baumdarstellung des arithmetischen Ausdrucks (Term)

$$2 + \underbrace{3 * 4}_{*bindetstärker+}$$

Daher:

Ein Breitendurchlauf (breath-first-traversal) eines Baumes t sammelt die Markierungen der Knoten ebenenweise vo der Wurzel ausgehend auf:

(levelorder t) ⇝ (list "s" "c" "h" "e" "m" "e")

Idee: gegeben sei eine Liste ts von Bäumen

1. Sammle die Liste der Markierungen der Wurzeln der (nicht-leeren) Liste in ts (roots ts)
2. Bestimme die Liste ts' der Teilbäume der (nicht-leeren) Bäume in ts (subtrees ts)
3. Führe 1. rekursiv auf ts' aus
4. Konstruiere die Listen aus 1. und 3.

```
(: traverse ((list-of (btree-of %a)) -> (list-of %a)))
(define traverse
    (lambda (ts)
        (cond
            ((empty? ts) empty)
            ((pair? ts) (append (roots ts)
                                        (traverse (subtrees ts)))))))
```

3 Neue Sprachebene: Die Macht der Abstraktion - Fortgeschrittene

- neues Ausgabeformat in der REPL
  (list $x_1 x_2 ... x_n$) ⇝ ($x_1 x_2 ... x_n$)
  empty ⇝ ()

- neuer Gleichheitstest für Werte aller (auch benutzerdefinierte) Signaturen
(: equal? (%a %b -> boolean))

- Quote
Sei <e> ein beliebiger Scheme Ausdruck
Dann liefert (quote <e>) die Repräsentation von <e> (<e> wird nicht ausgewertet)
Beispiele:
  - (quote 42) $\rightsquigarrow$ 42
  - (quote "Leia") $\rightsquigarrow$ "Leia"
  - (quote t) $\rightsquigarrow$ t
  - (quote (+ 40 2)) $\rightsquigarrow$ (+ 40 2)

  Abkürzung: (quote <e>) ≡ ’<e>

- Symbole Was ist (first (* 1 2))?
Was sind lambda, x, + in (lambda (x) (+ x 1)) ?
Neue Signatursymbol zur Repräsentation von Namen in Programmen. Effiziente interne Darstellung, effizient vergleichbar (mit equal?)
kein Zugriff auf die einzelnen Zeichen des Symbols.

Mögliche Operanden von Symbolen:

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

3.1 Natürliche Repräsentation/ Auswertung arithmetischer Ausdrücke (Terme)

3.2 Auswertung eines arithmetischen Ausdrucks e in Environment D

3.3 Freie/ Gebundene Variablen

Zur Auswertung $E_1 \equiv ((\lambda x.(\text{f x y}))\ \text{z})$

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

In E_1 ist

- Variable x (durch das λx als Parameter) gebunden (bound), während
- Variable f,y,z frei (free) sind

Welche Variablen eines Ausdrucks frei/ gebunden?
free(v) = v
free$((e_1 e_2))$ = free(e_1) ∪ free(e_2)
free$((\lambda x.e_1))$ = free(e_1) \ x

bound(v) = ∅
bound$((e_1 e_2))$ = bound(e_1) ∪ bound(e_2)
bound$((\lambda x.e_1))$ = bound(e_1) \ x

free(E_1) = $E_1 \equiv ((\lambda x.$(f x y)) z)
= $\{f, y, z\}$
Achtung: Bindung/ Freiheit muss für jedes Vorkommen einer Variable seperat entschieden werden

$E_2 \equiv (x(\lambda x.x))$
free(E_2) = x
bound(E_2) = x

3.4 Beta-Reduktion

Werte einer Applikation $((\lambda v.e_1)e_2)$ aus

1. Kopie von e_1 herstellen
2. In Kopie: freie Vorkommen von v durch e_2 ersetzen

Beispiel:
$((((\underbrace{\lambda x.(\lambda y.x)}_{k})(\lambda z.y))\ ↯\ )$ a)
⇒ sollte y liefern
→ $(((\lambda y.(\lambda z.y)))\ ↯\ )$ a)
→ $((\lambda z.↯)a)$
→ ↯

$((((\lambda x.(\lambda p.x))(\lambda z.y))\ ↯\ )$ a)
→ $(((\lambda p.(\lambda z.y))\ ↯\ )$ a)
→ $((\lambda z.y)$a)
→ y

3.5 Definition Beta-Reduktion

$((\lambda x.e)\text{a})\ \overrightarrow{\beta}\ \text{e}[\text{x} \mapsto \text{a}]$
"ersetze freie Vorkommen von x in e durch a"

$(\mapsto 1)$	$x[x \mapsto a] = a$	
$(\mapsto 2)$	$v[x \mapsto a] = v$	
$(\mapsto 3)$	$(e_1 e_2)[x \mapsto a] = (e_1\ [x \mapsto a]\ e_2\ [x \mapsto a])$	
$(\mapsto 4)$	$(\lambda x.e)[x \mapsto a] = (\lambda x.e)$	v’ neuer Variablenname
$(\mapsto 5)$	$(\lambda v.e[x \mapsto a] = (\lambda v.e[x \mapsto a])$ falls v nicht frei von a	
$(\mapsto 6)$	$(\lambda v.e[x \mapsto a] = (\lambda v'.e[v \mapsto v'])[x \mapsto a]$ sonst	

Beispiel: $((((\lambda x.(\lambda y.x))(\lambda z.y))\lightning)a)$

$\overrightarrow{\beta}\ ((((\lambda y.x)\ [x \mapsto (\lambda z.y)\ ])\ \lightning)\ a)$

$\overrightarrow{(\mapsto 6)}\ ((((\lambda y'.x[y \mapsto y'])\ [x \mapsto (\lambda z.y)])\ \lightning)\ a)$

$\overrightarrow{(\mapsto 2)}\ ((((\lambda y'.x)[x \mapsto (\lambda z.y)])\ \lightning)\ a)$

$\overrightarrow{(\mapsto 5)}\ ((((\lambda y'.x[x \mapsto (\lambda z.y)]))\ \lightning)\ a)$

$\overrightarrow{(\mapsto 1)}\ (((\lambda y'.(\lambda z.y))\lightning)\ a)$

$\overrightarrow{\beta}\ ((\lambda z.y)\ [y' \mapsto \lightning]\ a)$

$\overrightarrow{(\mapsto 5)}\ ((\lambda z.y\ [y' \mapsto \lightning])\ a)$

$\overrightarrow{(\mapsto 2)}\ ((\lambda z.y)\ a)$

$\overrightarrow{\beta}\ y[x \mapsto a]$

$\overrightarrow{(\mapsto 2)}\ y$