Eberhard Karls Universität Tübingen

Informatik I Vorlesung

Wintersemester 2016/2017

Mitschrieb von
Julian Wolff

Aktueller Stand 2. Februar 2017

Inhaltsverzeichnis

1 Scheme: Ausdrücke, Auswertung und Abstraktion 4

1.1 REPL 4
1.2 Literale 4
1.3 Zusammengesetzte Ausdrücke 5
1.4 Identifier 5
1.5 Lambda-Abstraktion 5
1.6 Kommentare 6
1.7 Signaturen 6
1.8 Prozedur-Signaturen 7
1.9 Testfälle 7
1.10 Erinnerung 7
1.11 Top-Down-Entwurf (Programmieren mit "Wunschdenken") 8
1.12 Reduktionsregeln für Scheme ($\rightsquigarrow$) 9
1.12.1 Einschub: Lexikalische Bindung 10
1.13 Übliche Notation in der Mathematik: Fallunterscheidung! 10
1.14 Spezialform Fallunterscheidung (conditional) 10
1.15 Binäre Fallentscheidung: 11
1.16 Zusammengesetzte Daten 11
1.17 Records in Scheme 12
1.18 Spezialform check-property 12
1.18.1 Interaktion von Konstruktor und Selektor 12
1.19 Längen/Breitengrade 14
1.20 Signaturnamen 14

2 Gemischte Daten 14

2.1 Polymorphe Signaturen 16
2.2 Polymorphe Paare und Listen 17
2.3 Liste 17
2.4 Visualisierung von Listen 18
2.5 Spines (Rückrad) 18
2.6 Prozeduren über Listen 19

3 Neue Sprachebene "Macht der Abstraktion" 19

3.1 cat 19
3.2 Bewertungen 20
3.3 Pattern Matching für $\langle \text{pat}_i \rangle$ 20
3.4 Rekursion über natürliche Zahlen 20
3.4.1 iterative Listenumkehr (backwards) 23

3.5 letrec . . . 23
3.6 Induktive Definitionen . . . 24
3.7 Beweisschema der vollständigen Induktion . . . 24
3.8 Induktionsaxiom (P5) für M . . . 24
3.8.1 Beispiel . . . 24
3.8.2 Beispiel . . . 25
3.8.3 Beispiel . . . 25
3.8.4 Bemerkung . . . 26
3.9 Def (Listen) . . . 26
3.10 Schema der Listeninduktion . . . 26
3.11 Prozeduren höherer Ordnung (HIGHER-ORDER FUNCTIONS) . . . 28
3.11.1 Beispiel (map f xs) . . . 29
3.11.2 Hinweis . . . 29
3.11.3 Listenfaltung . . . 29
3.11.4 Beispiele (Reduktionen von xs) . . . 30
3.12 Universe . . . 31
3.13 Komposition von Funktionen (allgemein) . . . 31
3.14 Currying . . . 32
3.15 Erinnerung . . . 33
3.16 Streams (stream-of % a) . . . 33
3.17 Vergleich . . . 34
3.17.1 Verzögerte Auswertung eines Ausdrucks (delayed evaluation) . 34
3.18 Sieb des Erastostgenes (Generierung aller Primzahlen) . . . 34
3.19 Binärbäume . . . 35
3.19.1 Visualisierung/Terminologie . . . 35
3.19.2 Einschub: Pretty-Printing von Binärbäumen . . . 37
3.20 Induktion über Binärbäumen . . . 37
3.20.1 Beispiel: . . . 37
3.20.2 Erinnerung . . . 38
3.20.3 Beispiel . . . 38
3.21 Baumdurchläufe . . . 39
3.21.1 Beispiel . . . 39

4 Neue Sprachebene DMdA-fortgeschritten 41
4.1 Quote . . . 41
4.1.1 Beispiele: . . . 41
4.2 Symbole . . . 41
4.3 Operatoren . . . 41
4.4 Natürliche Repräsentation und Auswertung . . . 42
4.5 Auswertung eines arithmetischen Ausdrucks e . . . 42

5 Lambda-Kalkül **43**
5.1 Syntax des λ-Kalküls 43
5.1.1 Freie/Gebundene Variablen 43
5.2 β-Reduktion 44
5.2.1 Definition β-Reduktion 45

Scheme: Ausdrücke, Auswertung und Abstraktion

REPL

Definition	DrRacket
Interaktion	REPL

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: ⟨Funktion⟩⟨ argument⟩)

(+ 402) und (odd? 42) sind Beispiele für die Ausdrücke, die bei Auswertung einen Wert liefern. (Notation $\rightsquigarrow$) heißt Auswertung/Evaluation/Reduktion.

(+ 40 2) $\underset{Eval}{\rightsquigarrow}$ 42

(add? 42) $\underset{Eval}{\rightsquigarrow}$ #f

Interaktionsfenster:

$\underbrace{\text{Read} \rightsquigarrow \text{Eval} \rightsquigarrow \text{Print}}_{\text{Loop}}$ REPL

Literale

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

Literal		Signatur
#t #f	(true, false, Wahrheitswerte)	boolean
„ac“ „x“ „ “	(Zeichenketten)	string
0 1904 -42 007	(ganze Zahlen)	integer
0.42 3.1415 -273.15	(Fließkommazehlen)	real
1/2 3/4 -1/10	(rationale Zahlen)	rational
	(Bilder)	image

Zusammengesetzte Ausdrücke

Auswertung zusammengesetzte 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:
$0.7 + (\frac{1}{2}/0.25) - (0.6/0.3) = 0.7$

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

Identifier

Ein Wert kann an einen Namen (identifier) gebunden werden, durch(define$\langle$id$\rangle\langle$expression$\rangle$)
Es 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 $\langle$id$\rangle$ wird durch den Wert von $\langle$expression$\rangle$ gebunden. Namen können in Scheme fast beliebig gewählt werden, solange

- die Zeichen ()[]{}",'; #\ | nicht vorkommen
- der name nicht einem numerischen Literal gleicht
- keinen Whitespaße (Leerzeichen, Tabulatoren, Neuwlines) enthalten sind

Beispiel: Euro $\rightarrow$ US-$

⚠ Achtung: Groß-/Kleinschreibung ist in Identifiern nicht relevant.

Lambda-Abstraktion

Eine Lambda-Abstraktion (auch: Funktion, Prozedur) erlaubt die Formulierung von Ausdrücken, in denen mittels Parametern von konkreten Werten abstrahiert wird:
(lambda ($\langle p_1\rangle\langle p_2\rangle$...)$\langle$expr$\rangle$)
expr ist der Rumpf und enthält Vorkommen der Paramenter $\langle p_i\rangle$.

(lambda...) ist eine Spezialform. Der Wert der Lambda-Abstraktion #⟨procedure⟩
Die Anwendung (auch: Applikation) der Lambda-Abstraktion führt zur Ersetzung aller Vorkommen der Parameter im Rumpf durch die angegebenen konkreten Argumente:

> (lambda (days)(*days(*155 minutes-in-a-day)) 365)
> $\overset{!}{\rightsquigarrow}$ (*365 (155 minutes-in-a-day))
> $\rightsquigarrow$ ...
> $\rightsquigarrow$ 81468000

Kommentare

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

Signaturen

Eine Signatur prüft, ob ein Name ⟨id⟩ an einen Wert einer angegebenen Sorte gebunden wird. Signaturverletzungen werden protokolliert.
(: ⟨ id ⟩ ⟨signatur⟩)
Bereits eingebundene Signaturen sind:

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

⚠ Der Doppelpunkt „ : “ ist eine Spezialform und hat daher keinen Wert, aber einen Effekt: Eine Signaturprüfung wird durchgeführt.

Prozedur-Signaturen

Prozedur-Signaturen spezifizieren Signaturen sowohl für die Parameter $\langle p_1 \rangle$, $\langle p_2 \rangle$, ... 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

Testfälle dokumentieren das erwartende Ergebnis einer Prozedur für ausgewählte Argumente:

```
(check-expect ⟨e1⟩ \la$\text{e}_2$\ra)
```

Werte den Ausdruck $\langle e_1 \rangle$ aus und teste, ob der erhaltene Wert der Erwartung (=Wert des Ausdruck $\langle e_2 \rangle$) entspricht.
Einer Prozedurdefinition sollten Testfälle direkt vorangestellt werden.
⚠„check-expect“ ist eine Spezialform und hat daher keinen Wert. Eine Testverletztung wird als Effekt protokolliert.

Erinnerung

Konstruktionsanleitung für Prozeduren:

- kurzbeschreibung (ein- bis zweizeiliger Kommentar mit Bezug auf PArameternamen und Ergebnis)
- Signatur (:⟨ name ⟩ (... →))
- Testfälle check-expect/ ceack-within
- Prozedurgerüst (define ⟨name⟩ (lambda (⟨p_1⟩⟨p_2⟩))
- Rumpf programmieren ⟨rumpf⟩)

Top-Down-Entwurf (Programmieren mit "Wunschdenken")

Beispiel: Sunset auf Tatooine (SW Episode IV)

Zeichne Szene zu Zeitpunkt t (t=0 ... 100)

(1) Himmel verfärbt sich von blau (t=0) zu rot (t=100)

(2) Sonne(n) versinkt (bei t=100 hinter Horizont)

(3) Luke starrt auf Horizont (bei jeden t)

Zeichne Szene von hinten nach vorne:

Abbildung 1: Frodo auf dem Weg nach Mord... äh ich meine natürlich Luke auf Tatooine

```
;Zeichne Tatooine Sunset zu Zeitpunkt t
(:tatooine (natural -> image))
(define tatooine
    (lambda (t)
        (overlay/pinhole (luke t)
                         (sun t )
                         (sky t ))))
```

Reduktionsregeln für Scheme ($\rightsquigarrow$)

Fallunterscheidung je nach Ausdruck:

- Literal l (1, #t, "Karotte", ...) [$eval_1$]
  l $\rightsquigarrow$ l (keine Reduktion möglich)
- Identifier ⟨id⟩ [$eval_{id}$]
  ⟨id⟩ $\rightsquigarrow$ Wert, an den ⟨id⟩ gebunden
- Lambda-Abstraktion [$eval_\lambda$]
  (lambda (...)...) $\rightsquigarrow$ (lanmbda (...)...)
- Applikation (f $e_1 e_2$...)
  - f, e_1, e_2, ... mittels $\rightsquigarrow$, erhalte f',e'_1, e'_2 ...
Operation auf e'_1, e'_2... anwenden	Falls f primitive (eingebaute) Operation	[apply/prim]
Argumentwerte $e'_1, e'_2, ...$ in den Rumpf einsetzen, den Rumpf mittels $\rightsquigarrow$ reduzieren	falls f' Lambda-Abstraktion	[$apply_\lambda$]

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

Beispiele:

(+ 40 2)
$\underset{eval_{id}}{\rightsquigarrow}$ (#⟨procedure:+⟩ 40 2)
$eval_{lit} \cdot 2$
$\underset{applyprim}{\rightsquigarrow}$ 42

(sqr 9) $\underset{eval_{id}}{\rightsquigarrow}$ $(lambda(x)(*xx))$
$eval_{lit}$
$\underset{apply_\lambda}{\rightsquigarrow}$ (* 9 9)
$\underset{eval_{id}}{\rightsquigarrow}$ $(\#\langle procedure : *\rangle$ 9 9)
$eval_{lit*2}$
$\underset{apply_{prim}}{\rightsquigarrow}$ 81

Einschub: Lexikalische Bindung

Bezeichnen (lambda (x) (* x x)) und (lambda (r) (* r r)) die gleiche Funktion?
(... 9) $\overset{*}{\rightsquigarrow}$ 81
$\Rightarrow$ JA!
⚠ Das hat Einfluss auf das korrekte Einsetzten von Argumenten für Parameter ($s.apply_\lambda$).
Das <u>bindende Vorkommen</u> eines Identifiers $\langle x \rangle$ im Programmtext kann systematisch bestimmt werden: Suche strikt "von innen nach außen"bis zum ersten
(1) (lambda (x) ...)
(2) (define x ...)
Das ist das Prinzip der <u>lexikalischen Bindung</u> (⚠ Syntaxprüfung in DrRacket)

Übliche Notation in der Mathematik: <u>Fallunterscheidung!</u>

maximum $(x_1, x_2) = \begin{pmatrix} x_1 \text{ falls } x_1 \geq x_2 \\ x_2 \leftarrow \text{sonst} \end{pmatrix}$
<u>Tests</u> (auch <u>Prädikate</u>) 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))
```

Weiter: add?, even?, positive?, negative?, ...

Spezialform Fallunterscheidung (conditional)

(cond ($\langle t_1 \rangle \langle e_1 \rangle$)
($\langle t_2 \rangle \langle e_2 \rangle$)
...
($\langle t_n \rangle \langle e_n \rangle$)
(else $\langle e_{n+1} \rangle$))) <- optional
Führt die Tests in der Reihenfolge $\langle t_1 \rangle, \langle t_2 \rangle$, ... durch. Sobald $\langle t_i \rangle$ zu #t auswertet, werte Zweig $\langle e_i \rangle$ aus. $\langle e_i \rangle$ ist das Ergebnis der Fallunterscheidung. Wenn $\langle t_n \rangle$ #f liefert, dann liefere
$\begin{pmatrix} \text{Fehlermeldung "cond: alle Tests ergeben } \#f \text{falls kein else- Zweig, sonst} \\ \langle_{n+1} \rangle \end{pmatrix}$
Die Signatur <u>one-of</u> lässt genau einen der n aufgezählten Werte zu:
(one-of $\langle e_1 \rangle \langle e_2 \rangle$ ... $\langle e_n \rangle$)
Reduktion von cond [$eval_{cond}$]

- (cond ($\langle t_1 \rangle$ $\langle e_1 \rangle$) ($\langle t_2 \rangle$ $\langle e_2 \rangle$) ...)

  (1) Reduziere $\langle t_1 \rangle$, erhalte $\langle t_1' \rangle$

  (2) $\langle e_1 \rangle$ falls $\langle t_1 \rangle = \#t$
(cond ($\langle t_2 \rangle \langle e_2 \rangle$)...)

- (cond (else$\langle e_{n+1} \rangle$)) $\rightsquigarrow$ $\langle e_{n+1} \rangle$ ($\langle t_1 \rangle$, $\langle e_2 \rangle$, ... sind <u>nicht</u> ausgewertet sonst
  $\langle e_1 \rangle$ nicht ausgewertet)

- (cond) $\rightsquigarrow$ Fehler "cond alle Tests ergeben #f

Binäre Fallentscheidung:

$$\begin{array}{rcl} (\text{if } \langle t_1 \rangle\ \langle e_2 \rangle & & (\text{cond } (\ \langle t_1 \rangle \\ \langle e_3 \rangle) & \equiv & (\text{else } \langle e_2 \rangle)) \\ \langle e_1 \rangle)) & & \end{array}$$

Zusammengesetzte Daten

Daten können interessante intere Struktur (<u>Komponenten</u>) aufweisen.
<u>Beispiel:</u> Ein Star Wars Charakter:

name	"Luke Skywalker"
jedi?	#f
force	25

<u>Beispiel:</u>

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

(make-character n j f) ⇝ ) ⟨)Tabelle⟩)) Konstruktion
(charcter-name ⟨Tabelle⟩) ⇝) n Selektor (komponenten auslesen)
(character-jedi? ⟨)Tabelle⟩)) ⇝) j
(character-force ⟨)Tabelle⟩)) ⇝) f
```

Records in Scheme

Record-Definition legst fest:

- Record-Signatur (Name)
- Konstruktor (bau aus komponenten einen Record)
- Prädikat (später)
- Liste von Selektoren (lesen je eine Komponente des Record)

(define-record-procedures $\langle t \rangle$ Signaturname
make-$\langle t \rangle$;Konstruktor
$\langle t \rangle$? ;Prädikat
($\langle t \rangle$-$\langle comp_1 \rangle$;Liste der Selektoren
... $\langle t \rangle$-$\langle comp_n \rangle$))
Liste der Selektoren legt Komponenten (Anzahl, Reihenfolge, Namen) fest. Signatur des Konstruktors/der Selektoren für Record-Signatur $\langle t \rangle$ mit n Komponenten $\langle comp_1 \rangle$ ... $\langle comp_n \rangle$:
(: make $\langle t \rangle$ ($\underbrace{\langle t \rangle}_{\text{n Komponentensignaturen}}$ ... $\langle t_n \rangle \rightarrow \langle t \rangle$))

(: $\langle t \rangle$-$\langle comp_1 \rangle$ ($\langle t \rangle \rightarrow \langle t_1 \rangle$))

$\forall$ string n, boolean j, 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

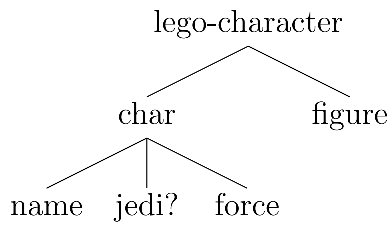
Aussagen üver die Interaktion von zwei (oder mehr) Funktionen: algebraische Eigenschaft.

Spezialform check-property

(check-property (for-all(($\langle id_1 \rangle\langle signatur_1 \rangle$) ... ($\langle id_n \rangle\langle signatur_n \rangle$)) $\langle expr \rangle$)) expr ist das Prädikat, das sich auf $\langle id_q \rangle$ ... $\langle id_n \rangle$ bezieht.
Test erfolgreich, falls $\langle expr \rangle$ für beliebige Bindungen für $\langle id_1 \rangle$ ... $\langle id_n \rangle$ immer #t ergibt.

Interaktion von Konstruktor und Selektor

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



<u>Beispiel:</u> 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 ((x_1 natural)
(x_2 natural))
($\geq$(+ x_1 x_2) (max x_1 x_2))))

Konstruktion von Funktionen $\langle f \rangle$, die zusammengesetzte Daten der Signatur $\langle t \rangle$ <u>konsumiert</u>.

- Welchen Record-Komponenten $\langle comp_i \rangle$ sind relevant für $\langle f \rangle$?
- $\Rightarrow$ Schablone:
  (:$\langle f \rangle$ (... $\langle t \rangle$ ... $\rightarrow$ ...))
  (define ⟨ f ⟩
  (lambda (... r ...)
  ... ($\langle t \rangle$-$\langle comp_i \rangle$ r)...))
  (: not (boolean -> boolean))

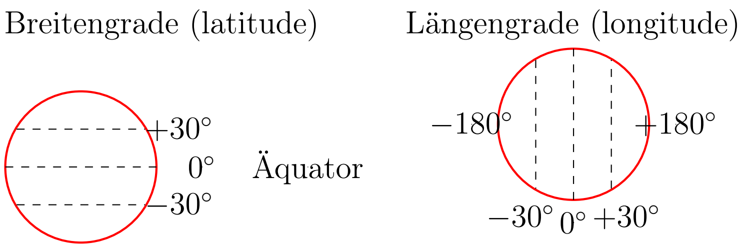
Prozedur ⟨f⟩, die zusammengetzte Daten der Signatur ⟨t⟩ konstruiert/produziert.

- Konstruktoraufruf für ⟨t⟩ <u>muss</u> enthalten sein!

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

<table>
<tr><td>char</td><td><table><tr><td>name
jedi?
force</td><td></td></tr></table></td></tr>
<tr><td>figure</td><td>(BILD)</td></tr>
</table>

Längen/Breitengrade



Sei ⟨p⟩ ein Prädikat mit Signatur (⟨t⟩ → boolean). Eine Signatur

```
(predicate ⟨p⟩)
```

gilt für jeden Wert x mit Signatur ⟨t⟩ für den zusätzlich (p⟨p⟩ x)⇝ #t gilt. Signatur (predicate ⟨p⟩) ist damit spezifischer (restriktiver) als Signatur ⟨t⟩.

Signaturnamen

Einführung eines neuen Signaturnamens ⟨new-t⟩ für die Signatur ⟨t⟩:

```
(define ⟨new-t⟩ (signature ⟨t⟩))
```

Beispiele:

```
(define farbe
  (signature (one-of "Karo" "Herz" "Pik" "Kreuz")))
(define latitude
  (signature (predicate latitude?)))
```

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

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

Ein geocode besteht aus:

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

Gemischte Daten

Die Signatur mixed

```
(mixed ⟨t1⟩...⟨tn⟩)
```

ist gültig für jeden Wert, der mindestens eine Signatur $\langle t_1\rangle$ ... $\langle t_n\rangle$ erfüllt.
Beispiel: Datendefinition:

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

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

Beispiel

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

Das Prädikat ⟨t⟩? einer Record-Signatur ⟨t⟩unterscheidet Werte der Signatur ⟨t⟩von <u>allen anderen</u> Werten:

```
(: ⟨t⟩? (any -> boolean))
```

Auch: Prädikate für eingebaute Signaturen.
number?, complex?, real?, rational?, integer?, Prozeduren, die gemischte
natural?, string?, boolean?
Daten der Singatuen $\langle t_1\rangle$... $\langle t_n\ \rangle$konsumieren:

```
(: ⟨f⟩(()mixed ⟨t1⟩ ... ⟨t2⟩)-> ...))
(define ⟨f⟩
 (lambda (x)
  (cond((⟨t1⟩? x)...)
   ...
   ((⟨tn⟩? x)...))))
```

Mittels <u>let</u> lassen sich Werte an <u>lokale Namen!</u> binden:

```
(let ((⟨id1⟩ ⟨e2⟩) ... (⟨idn⟨ ⟨en⟩)) e)
```

Die Ausdrücke $\langle e_1\rangle$... $\langle e_n\rangle$werden parallel ausgewertet.
⇒ $\langle id_1\rangle$... $\langle id_n\rangle$können in ⟨e⟩(und <u>nur dort!</u>) verwendet werden.
Der Wert des let-Ausdruck ist der Wert von e. "nur dort": Verwendung nur in in ⟨e⟩, nicht in den in $\langle e_i\rangle$!
Lokal: Verwendung nicht außerhalb des (let...)
⚠Sprachlevel "Die Macht der Abstraktion"

```
(let    )≡(lambda ()   )
```

„Syntaktischer Zucker“= Dinge die nett sind aber ersetzt werden können.

```
(check-error ⟨e⟩ ⟨msg⟩)
```

erwartet Abbruch mit Fehlermeldung ⟨msg⟩. Erzwingen des Programmabbruches mittels (violation ⟨msg⟩)

Polymorphe Signaturen

Beobachtung: Manche Prozeduren arbeiten völlig unabhängig von den Signaturen ihrer Argumente:
parametrisch polymorphe Prozeduren (griechisch: vielgestaltig). Nutze Signaturvariablen:
Beispiele:

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

;konstante Funktion (ignoriert zweites Argument)
(: const( %a %b -> %a)) "Anstatt %b kann auch any benutzt werden
  "
(define const
 (lambda (x y) x))

;Projektion (ein Argument auswaehlen)
(: 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 an andere polymorphe Prozeduren weiterreichen.
Eine polymorphe Signatur steht für die Signaturen, in denen die Signaturvariablen konistent durch konkrete Signaturen ersetzt werden.

Beispiel:

Wenn eine Prozedur (%a number %b -> %a) erfüllt, dann auch
(string number boolean -> string)
(boolean number natural -> boolean)
(string number string -> string)
(number number number -> number)

Polymorphe Paare und Listen

```
; Ein polymorphes Paar (pair) besteht aus
;- erster Komponente (first)
;- zweite Komponente (rest)
; wobei die komponenten jeweils beliebige Werte sind:

(define-record-procedures-parametric pair pair-of
 make-pair
 pair?
 (first
  rest))
```

(pair−of ⟨t_1⟩⟨t_2⟩)

ist eine Signatur für Paare, deren erste und zweite Komponente von der Signatur ⟨t_1⟩bzw. ⟨t_2⟩sind.

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

Liste

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⟩)))

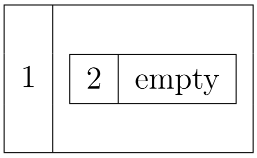
```
(define list-of
 (lambda (t)
  (signature (mixed empty-list
                    (pair-of t (list-of t))
                                         ))))
```

(list-of ⟨t⟩). Listen, deren Elemente die Signatur ⟨t⟩besitzen.
Die Signatur empty-list ist bereits in DrRacket vordefiniert.
Ebenfalls vordefiniert ist:

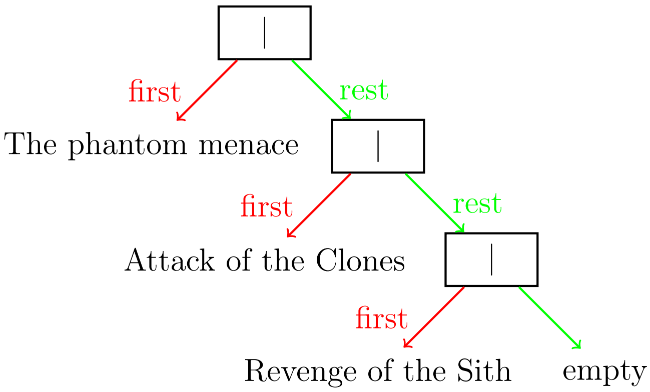
- (: empty empty-list)
- (: empty? (any -> boolean))

Visualisierung von Listen

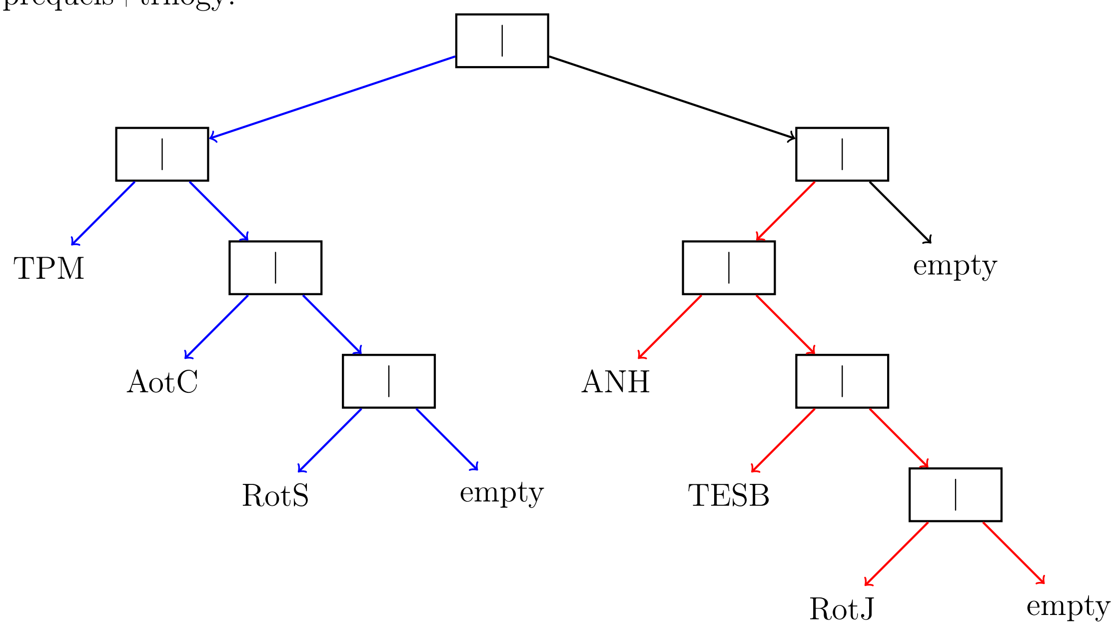
```
(make-pair 1 (make-pair 2 empty))
```

lightning  Notation skaliert schlecht

Spines (Rückrad)



prequels+trilogy:



Prozeduren über Listen

Schablonen für gemischte und zusammengesetzte Daten
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)

(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 t_1

                                                signatur t_2
```

Neue Sprachebene "Macht der Abstraktion"

- Signatur (list-of %a) eingebaut
- Neuer syntaktischer Zucker eingebaut:
  (list $\langle e_1\rangle\langle e_2\rangle$... $\langle e_n\rangle$)
  $\equiv$
  (m-p $\langle e_1\rangle$(m-p $\langle e_2\rangle$
  ...
  (m-p $\langle e_n$ empty) ..))
- Ausgabeformat für nicht-leere Listen
  #⟨list $\langle e_1\rangle\langle e_2\rangle$... $\langle e_n\rangle$⟩

cat

;Füge Listen xs, xy zusammen (concatenate)
Zwei Fälle (xs leer oder nicht-leer)

$$\begin{array}{llll}
1) & \overset{xs}{empty} & \overset{ys}{y_1 y_2 y_n} & \overset{(catxsxy)}{y_1 y_2 y_n} \\
2) & x_1 x_2 x_n & y_1 y_2 y_n & \underset{(cat(restxs)ys)}{\underset{restxs \quad ys}{x_1 x_2 x_n y_1 y_2 y_n}}
\end{array}$$

Bewertungen

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

Spezialform match vergleicht einen Wert ⟨e⟩mit gegebenen Patterns ⟨pat$_1$⟩⟨pat$_2$⟩, ... ⟨pat$_n$. Falls ⟨pat$_1$, $1 \leq i \leq n$, das erste Pattern ist, das auf ⟨e⟩matchted, ist Zweig ⟨e$_i$⟩das Ergebnis (ansonsten wird die Aiswertung mit "keiner der Zewige passte") abgegeben.

```
(match ⟨e⟩
 (⟨pat₁⟩ ⟨e₁⟩)
 (⟨pat₂⟩ ⟨e₂⟩)
 (⟨patₙ⟩ ⟨eₙ⟩))
```

Pattern Matching für ⟨pat$_i$⟩

- Literal ⟨l⟩:
  ⟨e⟩matched, falls ⟨e⟩$\rightsquigarrow$⟨l⟩
- "Don't care"_ :
  ⟨e⟩matched immer
- Variable ⟨v⟩
  ⟨e⟩matched immer, danach ist ⟨v⟩an den Wert von ⟨e⟩n ⟨e$_i$⟩gebunden
- Record-Konstruktor (⟨c⟩⟨pat$_{i1}$⟩⟨pat$_{ik}$), $k \geq \emptyset$
  ⟨e⟩matched, wenn es durch (⟨c⟩⟨x$_1$⟩⟨x$_k$⟩) konstruiert wurde und ⟨x$_j$⟩auf ⟨pat$_{ij}$ matched, $1 \leq j \leq k$

⚠Fall 4 ermöglicht Pattern Matching auf komplex konstuierten Werten.

Rekursion über natürliche Zahlen

Die natürlichen Zaglen (vgl. gemischte Daten). Eine natüliche Zahl (natural) ist entweder

- die 0 (zero)
- die Nachfolger (succ) einer natülichen Zahl

$\mathbb{N} = \{0, (\text{succ } 0), (\text{succ } (\text{succ } 0)), ...\}$
Konstuktoren:

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

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

Bedingte algebraische Eigenschaften (siehe check-property) $(= \Rightarrow \langle \text{p} \rangle \langle \text{e} \rangle)$ Nur, wenn $\langle \text{p} \rangle \rightsquigarrow \#\text{t}$, wird der Ausdruck $\langle \text{e} \rangle$ausgewertet und getestet ob $\langle \text{e} \rangle \rightsquigarrow \#$ t.
Beispiel: Fakultätsfunktion n! $(n \in \mathbb{N})$:
$0! = 1$
$n! = n \cdot (n-1)! \equiv (\text{succ } n)! = (\text{succ } n)! \cdot n!$

$$\begin{aligned} 3! &= 3 \cdot 2! \\ &= 3 \cdot (2 \cdot 1!) \\ &= 3 \cdot (2 \cdot (1 \cdot 0!)) \\ &= 3 \cdot (2 \cdot (1 \cdot 1)) \\ &= 6 \\ 10! &= 3628800 \end{aligned}$$

```
; Berechne n!
(: factorial (natural -> natural))
(define factorial
  (lambda (n)
    (cond ((= n 0) ...
          ((< n 0) (* (factorial (- n 1)) n))))))
```

Schablone für Funktionen ⟨f⟩, die natürliche Zahlen 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 *)

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

⇒ Multiplikationen können vorgezogen werden.

Idee: Führe Multiplikation jeweils sofort aus. Schleife des Zwischenergebnis (akkumulierendes Argument) durch die Berechnung. Am Ende enthält der Akkumulator das Endergebnis.
Berechne 5!:

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

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

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

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

```
; Berechne n! [wrapper]
(: fac (natural -> natural))
(define fac
  (lambda (n)
    (fac-worker n 1)))
```

Ein Reduktionsprozess 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 Ausdruck vollständig. Es gibt keinen Kontext (umgebenden Ausdruck), der auf das Ergebnis des rekursiven Aufrufs "wartet".
Kontext des rekursiven Aufrufes in

- factorial: (* n $\underbrace{\square}_{Hole}$)
- fac-worker: -keiner-

Ein Prozeduraufruf ist endrekursiv (tail call), wenn er keinen Kontext besitzt. Prozeduren, die nur endrekursive Prozeduraufrufe enthalten, heißen selbst endrekursiv.

Endrekursive Prozeduren führen zu iterativen Reduktionen.

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

```
(rev (from-to 1 1000))

|.....1000 Aufrufe von make-pair....|
(cat (list 1000 999 ... 2) (list 1))
     |                    \
     |                     |______________
     |                                    |
     (cat (list 1000 999 ... 3) (list 2))
     |....999 Aufrufe von make-pair......|
```

⇒ Anzahl Aufrufe von make-pair 1000+999+998+...+1 auf einer Liste der Länge n:

$\sum_{i=1}^{n} = \frac{1}{2} \cdot n \cdot (n+1)$ Quadratisch in n

iterative Listenumkehr (backwards)

Konstruiere iterative Listenumkehr (backwards)

Berechnung von (backwards (list 1 2 3)).

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

	xs	acc	
	(list 1 2 3)	empty	
rest			(make-pair 1 acc)
	(list 2 3)	(list 1)	
rest			(make-pair 2 acc)
	(list 3)	(list 2 1)	
rest			(make-pair 3 acc)
	empty	(list 3 2 1)	

linear viele Aufrufe von make-pair!

letrec

Mit letrec lassen sich Werte an lokale Namen binden:

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

Die Ausdrücke $\langle e_1\rangle ... \langle e_n\rangle$dürfen selbst auf die Namen $\langle id_1\rangle ...$ $\langle id_n\rangle$beziegen. Den Wert des gesamten letrec-Ausdruck ist der Wert von $\langle e\rangle$.

Induktive Definitionen

Konstruktive Definition der natürlichen Zahlen $\mathbb{N}$:

Def. (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)$\neq 0$	succ ist
(P4)	$\forall n, m \in \mathbb{N} : succ(m) = succ(n) \Leftarrow m = n$	injektiv (erzeugt neue Elemente)

(BILD TAFEL)

(P_5) Induktionsaxiom:

Für jede Menge $M \subseteq \mathbb{N}$:

Falls $0 \in M$ und $\forall n : (n \in M \Rightarrow succ(n) \in M)$,

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

"Nichts sonst ist in $\mathbb{N}$"

Beweisschema der vollständigen Induktion

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

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

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

Definiere: $M := \{n \in \mathbb{N} | P(n) \text{ gilt}\} \subseteq \mathbb{N}$ "M enthält alle n, für die P(n) gilt."

Induktionsaxiom (P5) für M

Falls	Falls
$0 \in M$	P(0) (INDUKTIONSBASIS)
und	und
$\forall n : (n \in M \Rightarrow succ(n) \in M)$	$\forall n : (P(n) \Rightarrow P(succ(n)))$ (INDUKTIONSSCHRITT)
dann	dann
$M = \mathbb{N}$	$\forall n \in \mathbb{N} : P(n)$

Beispiel

1	=1
1+3	=4
1+3+5	=9
1+3+5+7	= 16

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

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

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

(2) Induktionsschritt: $\forall n : (P(n) \Rightarrow P(n+1))$
$\sum_{i=0}^{n+1}(2i+1) \underset{\sum}{=} \sum_{i=0}^{n}(2i+1) + 2(n+1) + 1$
$\underset{P(n)}{=} (n+1)^2 + 2n + 3$
$= n^2 + 4n + 4$
$= (n+2)^2\checkmark$

Beispiel

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

$P(n) \equiv$ (factorial n) $=$ $\underline{\text{n!}}$ $\qquad$ $\underline{\text{X}}$: Racket-Repräsentation der Zahl X

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

(1) Induktionsbasis P(0)
(factorial 0) $\rightsquigarrow$((lambda (k) ...) $\underline{0}$)
$\rightsquigarrow$ (if (= 0 0) 1 ...)
$\rightsquigarrow$ (if #t 1 ...)
$\rightsquigarrow$ 1 = $\underline{0!}$

(2) Induktionsschritt
$\forall n : (P(n) \Rightarrow P(n+1))$:
(factorial $\underline{\text{n+1}}$)
$\rightsquigarrow$ ((lambda (k) ...) $\underline{\text{n+1}}$)
$\rightsquigarrow$ (if (= $\underline{\text{n+1}}$ 0)... (* ...))
$\rightsquigarrow$ (if #f ... (* ...))
$\rightsquigarrow$ (* $\underline{\text{n+1}}$ (factorial (- $\underline{\text{n+1}}$ 1))) Annahme: - realisiert Differenz korrekt
$\rightsquigarrow$ (* $\underline{\text{n+1}}$ (factorial n))
$\underline{\text{P(n)}}$ (* $\underline{\text{n+1}}$ $\underline{\text{n!}}$) = ($\underline{\text{n+1}}$)! $\checkmark$ Annahme: + realisiert Multiplikation korrekt.

Beispiel

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

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

also definiert durch:

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

Bemerkung

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

Dann gilt:

P(n)≡ (f n) terminiert mit Ergebnis
der Signatur %a

Beweis

(1) Induktionsbasis P(0)

(f $\underline{0}$)

$\overset{*}{\rightsquigarrow}$ (if (= $\underline{0}$ 0) basis ...)

$\rightsquigarrow$ (if #t basis)

$\rightsquigarrow$ basis ✓

(2) Induktionsschritt $\forall n$:(P(n) ⇒ P(n+1))

(f $\underline{\text{n+1}}$)

$\rightsquigarrow$ (if (= $\underline{\text{n+1}}$ 0) ... (step...))

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

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

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

$\overset{P(n)}{\rightsquigarrow}$ (step R $\underline{\text{n+1}}$) terminiert ✓

Def (Listen)

Die Menge M* (= Listen mit Elementen aus M, (list-of M)) ist induktiv definiert:

(l1) empty $\in M*$

(l2) $\forall c \in M, xs \in M*$): (make-pair x xs) $\in M$

(l3) Nichts sonst ist in M*

Schema der Listeninduktion

Sei P(xs) eine Eigenschaft von Listen über M:

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

Falls
P(empty)

und
$\forall x \in M, xs \in M*$:(P(xs)$\Rightarrow$ P((make-pair x xs)))

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

<u>Beispiel:</u> Eigenschaften von cat (append).

```
(define cat
  (lambda (xs xs
    (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"	
	($\mathbb{N}$, +, 0)	
	($\mathbb{N}$, +, 1)	

<u>Beweise</u>

(1) (cat empty ys) $\overset{*}{\leadsto}$ ys ✓

(2) P(xs) $\equiv$ (cat xs empty) = xs

Induktionsbasis P(empty): (cat empty empty) $\overset{(1)}{=}$ empty ✓

Induktionsschritt $\forall x \in M, xs' \in M*$: (P(xs') $\Rightarrow$ P((make-pair x xs')))

(cat (make-pair x xs') empty)

$\overset{*}{\leadsto}$ (make-pair) (HIER FEHLT NOCH WAS)

$\overset{*}{\leadsto}$ (make-pair x (cat xs' empty))

$\overset{I.V.}{=}$ (make-pair x xs')

(3) P(xs) $\equiv$ (cat (cat xs ys) zs) = (cat xs (cat ys zs))

ys, zs $\in M*$ beliebig

Induktionsbasis P(empty) (cat (cat empty ys) zs)

$\overset{(1->)}{=}$ (cat ys zs)

$\overset{(1<-)}{=}$ (cat empty (cat ys zs)) ✓

Induktionsschritt $\forall x \in M, xs' \in M*$: (P(xs') $\Rightarrow$ P ((make-pair x xs')))

(cat (cat (make-pair x xs') ys) zs)

$\stackrel{*}{=}$ (cat (make-pair x (cat xs' ys)) zs)

$\stackrel{*}{\rightsquigarrow}$ (make-pair x (cat (cat xs' ys) zs))

$\stackrel{I.V.}{=}$ (make-pair x (cat xs' (cat ys zs)))

$\stackrel{*}{\rightsquigarrow}$ (cat (make-pair x xs') (cat ys) zs)) ✓

Beispiel: Interaktion von length/cat

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

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

Indultionsbasis P(empty)

(length (cat emtpy) ys))

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

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

Induktionsschritt $\forall x \in M, xs' \in M*$: (P(xs') $\Rightarrow$ P ((make-pair x xs')))

(length (cat (make-pair x xs') ys))

$\stackrel{*}{\rightsquigarrow}$ (length (make-pair x (cat xs' ys)))

$\stackrel{*}{\rightsquigarrow}$ (+ 1 (length (rest (make-pair x (cat xs' ys)))))

$\rightsquigarrow$ (+1 (length (cat xs' ys)))

$\stackrel{I.V.}{=}$ (+1 (+ (length xs') (length ys)))

$\stackrel{(+)Assoziativ}{=}$ (+ (+ 1 (length xs')) (length ys))

$\stackrel{*}{\rightsquigarrow}$ (+ (length (make-pair $\underset{\text{beliebig}}{\text{x}}$ xs')) (length ys)) ✓

Prozeduren höherer Ordnung (HIGHER-ORDER FUNCTIONS)

Abstraktion von Funktionsparametern

```
;Extrahiere die Elemente xs, die das Praedikat p? erfuellen

(: 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 H.O.P)
H.O.P. ...

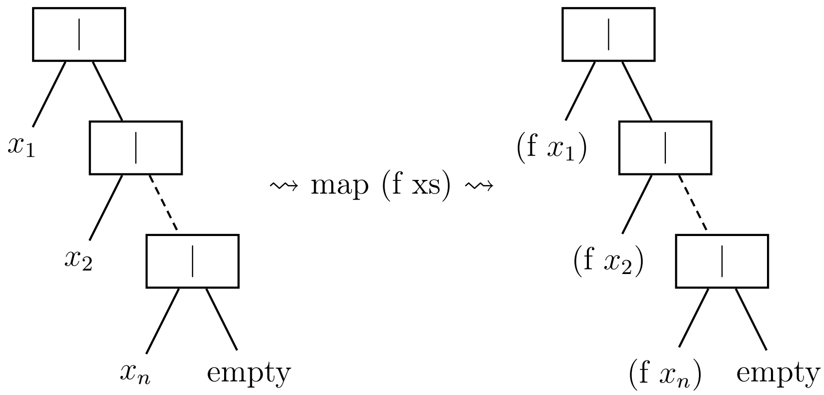
(1) akzeptieren Prozeduren als Parameter und/oder

(2) liefern eine Prozedur als Ergebnis.

filter ist vom Typ (1)
H.O.P. vermeiden Duplizierung von Code und führen zu

- kompakteren Programmieren
- verbesserte Lesbarkeit
- verbesserte Wartbarkeit

Beispiel (map f xs)



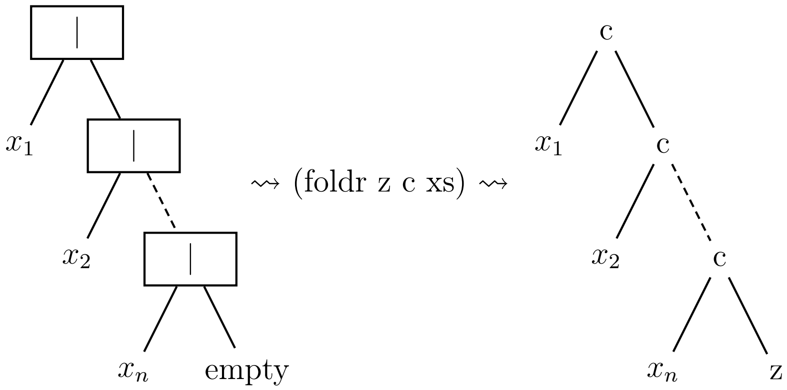

```
; Wende f auf alle Elemente von xs an
(: map (( %a -> %b)  (list-of %a) -> (list-of %b)))
(define map
  (lambda (f xs)
    (cond ((empty? xs) empty)
          ((pair? xs) (make-pair (f (first xs)) (map f (rest xs)
             ))))))
```

Hinweis

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

Listenfaltung

Allgemeinere Transformation von Listen: Listenfaltung (list folding).
Idee: die Listenkonstruktion make-pair und empty werden systematisch ersetzt:

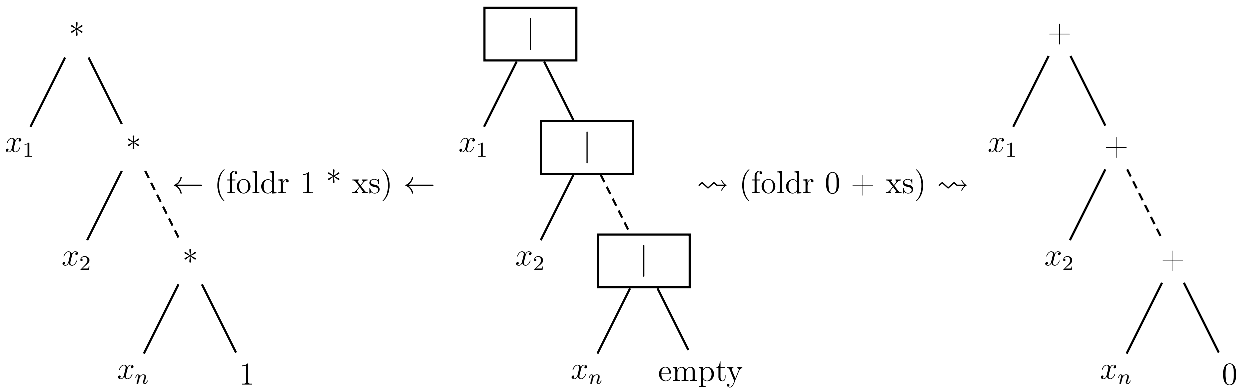


(foldr z c xs) wirkt als Spine Transformer:

- empty → z
- make-pair → c
- Eingabe: Liste (list-of %a)
- Ausgabe im allg. <u>keine</u> Liste (etwa %b)

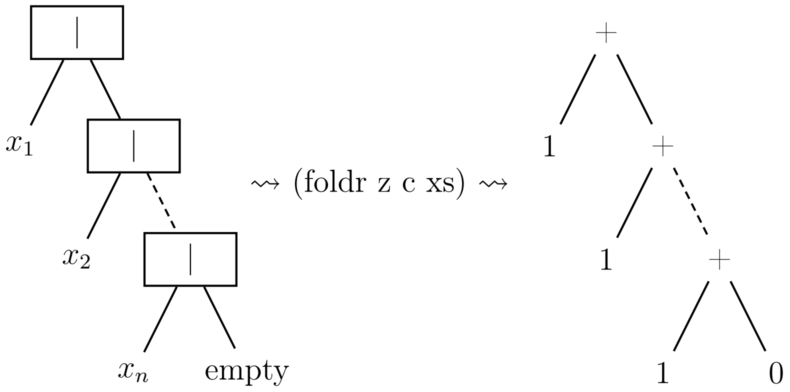
```
;Falte Liste xs bzgl. c und z
(: foldr ( %b (%a %b -> %b) (list-of %a)-> %b))
(define foldr
  (lambda (z c xs)
    (cond ((empty? xs) z)
          ((pair? xs) (c (first xs) (foldr z c (rest xs)))))))
```

Beispiele (Reduktionen von xs)




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

Länge eine Liste durch Listenfaltung:



Spine-Transformation

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

Universe

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

```
(big-bang ⟨init⟩
        (an-tick ⟨ tok⟩)
        (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 aktuellem Zustand
   eine Szene berechnet (wird in Fenster mit ⟨w⟩ x ⟨h⟩ Pixeln
   angezeigt)

;Bei Schliessen der Animation wird der letzte aktuelle Zustand
   zurueckgegeben.
```

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
  (lamda f g)
    (lambda (x)     ;Ergebnis ist eine
```

```
        (f (g x))))) ;Funktion (nicht angewandt)
```

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

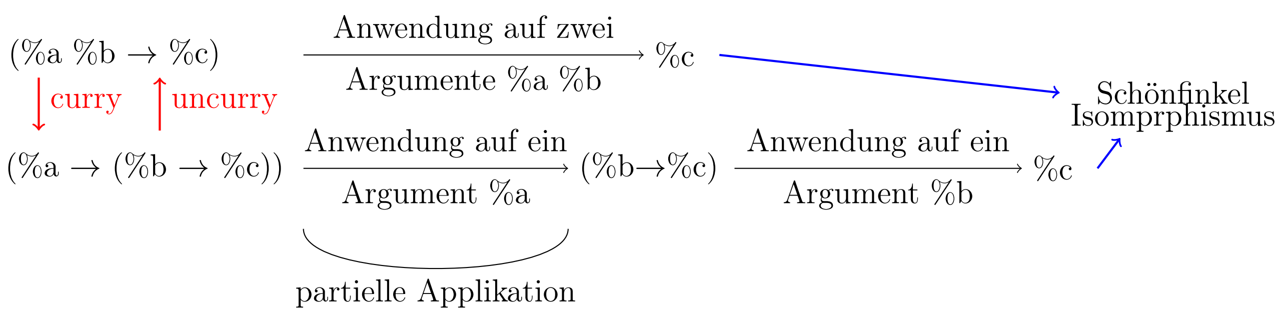
$$f^0 = id \text{ (Identität id} \equiv \text{(lambda (x) x))}$$
$$f^n = f^{n-1} \circ f$$

```
(: 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)))
```

Currying

Reduktion von ((add 1) 41)

$\overset{\text{eval id}}{\rightsquigarrow}$ ((lambda (x)(lambda (y) (+ x y))) 1 41)
$\overset{\text{apply } \lambda}{\rightsquigarrow}$ ((lambda (y) (+ 1 y)) 41)
$\overset{\text{apply } \lambda}{\rightsquigarrow}$ (+ 1 41) $\rightsquigarrow$ 42



- Currying (Haskell B. Curry, Moses Schönfinkel)

  Anwendung einer Funktion auf ihr erstes Argument liefert eine 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 (Transformator)
  Gegenrichtung: uncurry

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

(define curry
  (lambda (f)
    (lambda (x)     ;neue
      (lambda (y)   ;Funktion
        (f x y)     ;
      )
    )
  )
)
```

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

(define uncurry
  (lambda (f)
    (lambda (x y)
      ((f x) y)
      )
    )
  )
```

Erinnerung

Bestimmung der ersten Ableitung der reellen Funktion f durch Bildung des Differenzenquotienten: (Hier Bild mit x und y achse und schaubild) $\frac{f(x+h)-f(x)}{h}$ Differenzenquotient $\lim\limits_{0} \frac{f(x+h)-f(x)}{h} = f'(x)$ Differentialquotient

- Operator ' (Ableitung) konsumiert funktion f ind produziert f' ⇒ ' ist higher-order

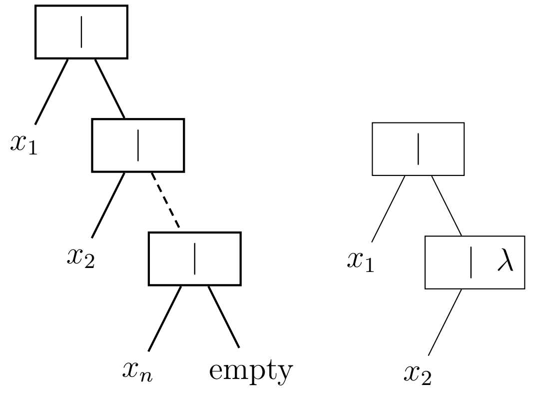
Streams (stream-of % a)

unendliche Ströme von Elementen x_i der Signatur %a.
Ein Stream ist ein Paar stream head (X1) stream tail

$\boxed{x_1 \mid \text{tail}}$ %a (-> (stream-of %a))

Vergleich

(list-of %a) (stream.of %a)



Verzögerte Auswertung eines Ausdrucks (delayed evaluation)

- ```
  (: delay ( %a -> (-> %a)))
  (delay ⟨e⟩) ;Verzoegere die Auswertung des Ausdrucks ⟨e⟩ und
     liefere "Versprechen" (promise) ⟨e⟩ bei Bedarf spaeter
     auswerten zu koennen.
  ```

  ⚠ funktioniert nicht:

  ```
  (define delay
    (lambda (e)
      (lambda () e)))
  ```

  Implementation:

  ```
  (delay ⟨e⟩) ≡ (lambda ()⟨e⟩)
  ```

- ```
  (: force ((-> %a) -> %a))
  (force ⟨p⟩) ;Erzwinge Auswertung des Promise ⟨p⟩, liefere den
     Wert des verzoegerten Ausdrucks
  (define force
    (lambda (p)
      (p)))
  ```

Sieb des Erastostgenes (Generierung aller Primzahlen)

Stream-Programm (über 2200 Jahre alt):

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

$\underline{2}$ 3 5 7 9 11 13 15 17 19 21 23 25

$\underline{2}$ $\underline{3}$ 5 7 11 13 17 19 23 25

$\underline{2}$ $\underline{3}$ $\underline{5}$ 7 11 13 17 19 23

Binärbäume

Die Menge der Binärbäume T(M) über M ist Induktiv definiert.

(T1) empty-tree $\in$ T(M)

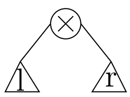
(T2) $\forall x \in M$, l,r $\in T(M)$: (make-node l x r) $\in$ T(M)

(T3) Nichts sonst ist in T(M)

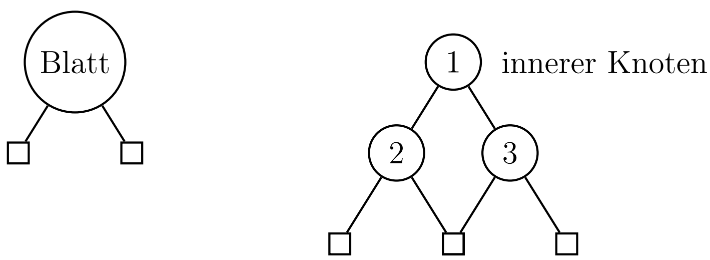
Hinweise:

- Jeder Knoten (make-node) in einem Binärbaum hat zwei 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/Terminologie

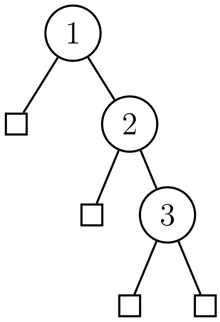
- empty-tree: □
- (make-node l x r): 

- Der Knoten mit Markierung x ist Wurzel (root) des Baumes
- Ein Knoten, der nur leere Teilbäume besitzt, heißt Blatt (leaf) Alle anderen Knoten sind innere Knoten (inner nodes)



Beispiel für Binärbäume der Menge T(ℕ).

- Baum t_1: listenartig (rechtstief)

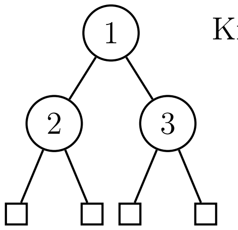


Knoten mit Label 1 ist Wurzel

Knoten 1,2 sind innere Knoten

Knoten 3 ist Blatt

- Baum t_2: balanciert



Knoten 1 ist Wurzel und innerer Knoten

Knoten 2,3 sind Blätter

(Binär-)Bäume haben zahllose Anwendungen:

- Suchbäume (schneller Zugriff, z.B. in Datenbanksystemen)
- Datenkompression
- Darstellung von Programmen/Ausdrücken im Rechner
- ...

Bäume sind DIE induktive Datenstruktur in der Informatik.

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

```
(btree-depth empty-tree)   = 0
(btree-depth t2)           = 2
(btree-depth t3)           = 3
(btree-depth classifier)   = 4
```

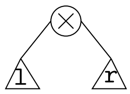
Schablone (gemischte + zusammengesetzte Daten):

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

Einschub: Pretty-Printing von Binärbäumen

Prozedur (pp ⟨t⟩) erzeugt formatierten String für Binärbaum ⟨t⟩.

```
(pp   □) = (list "  □")
```



```
(pp  l     r) = x
```

Idee: Repräsentiere formatierten String als Liste von Zeichen (String):

(1) Nutze (string-append ...) um Zeilen-Strings zu definieren [horizontal]

(2) Nutze (append...) um die einzelnen Zeilen zu einer Liste von Zeilen zusammenzusetzen [vertikal]

Erst direkt vor der Ausgabe werden die Zeilen-Strings zu einem auszugebendem String zusammengesetzt.

```
(strings-list->string)
```

Induktion über Binärbäumen

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

Falls

P(empty-tree) [Induktionsbeweis]

und

$\forall x \in M, l \in T(M), r \in T(M)$:

$P(l) \wedge P(r) \Rightarrow$P((make-node l x r))

dann $\forall t \in T(M) P(t)$.

Beispiel:

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

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

Induktionsbasis P(empty-tree):

(size empty-tree) $\rightsquigarrow 0 = 2^0 - 1 \rightsquigarrow 2^{(depthempty-tree)} - 1$ ✓

Induktionsschritt $P(l) \wedge P(r) \rightarrow P((make - nodelxr))$

(size (make-node l x r)) $\rightsquigarrow$ (size l) + 1+ 1 (size r) $\leq 2^{(depthl)} - 1 + 1 + 2^{(depthr)} - 1$

$= 2^{(}depthl) + 2^{(swpthr)} - 1$

$\leq 2 \cdot max(2^{(depthl)}, 2^{(depthr)}) - 1$

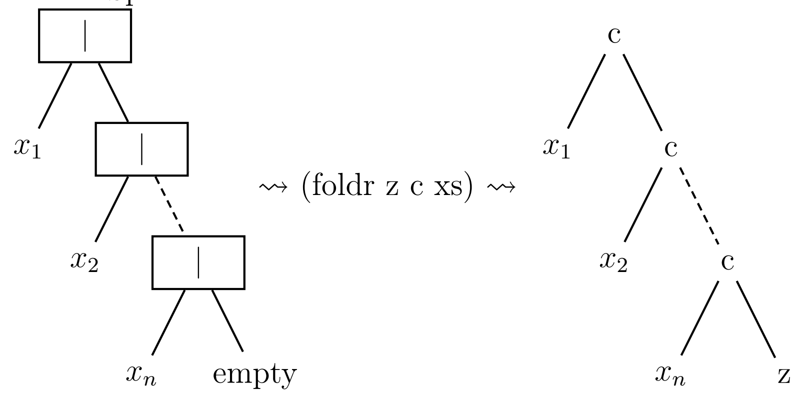
$= 2 \cdot 2^{max((depthl),(depthr))} - 1$

$= 2^{1+max((depthl),(depthr))} - 1$

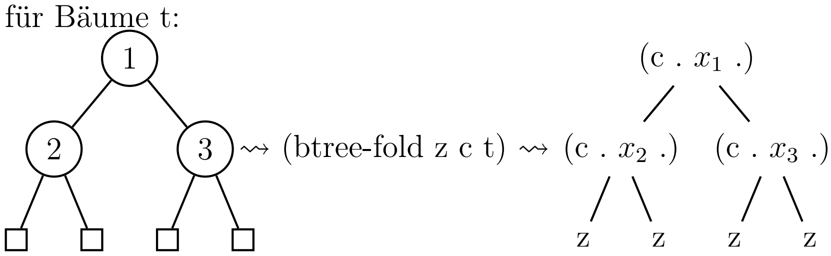
$\rightsquigarrow 2^{(depth(make-nodelxr))} + 1$

Erinnerung

fold ist Spine-Transformer für Listen xs:



Wie müsste btree-fold, eine fold-Operation für Binärbäume, verhalten? Tree-Transformer für Bäume t:

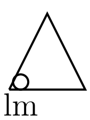


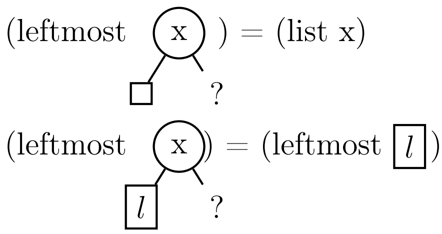

```
;Falte Baum bzgl. z und c

(: 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

Bestimmte die Markierung lm links-aussen im Baum t (oder empty, falls t leer ist).

$t \equiv\Downarrow$






```
(: 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)))
```

Baumdurchläufe

Ein Tiefnedurchlauf (depth-first traversal) eines Baumes t sammelt die Markierungen jedes Knoten n un 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)

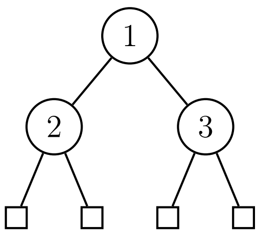
Je anchdem, ob x (a) zwischen, (b) vor, (c) nach den Markierungen von l,r eingeordnet wird, erhält man

(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          ;(1)
                                            (list x)     ;(2)
                                            xs2          ;(3)
                                   ))
                t)))
```

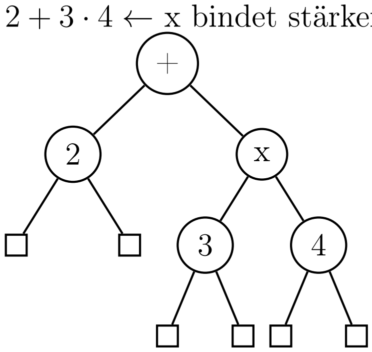


Daraus wird die Liste 2 1 3

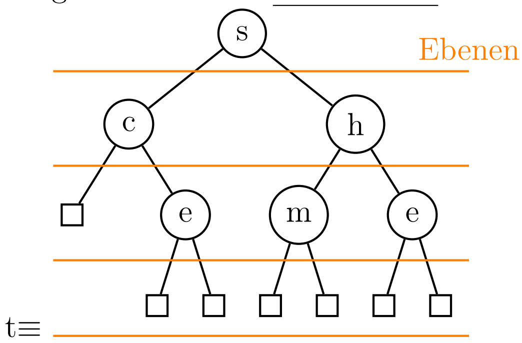
Beispiel

Baumdarstellung des arithmetischen Ausdrucks (Term)

$2 + 3 \cdot 4 \leftarrow$ x bindet stärker als +



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




```
(leveloder t) = (list "s" "c" "h" "e" "m" "e")
```

Idee: Gegeben sei eine Liste ts von Bäumen

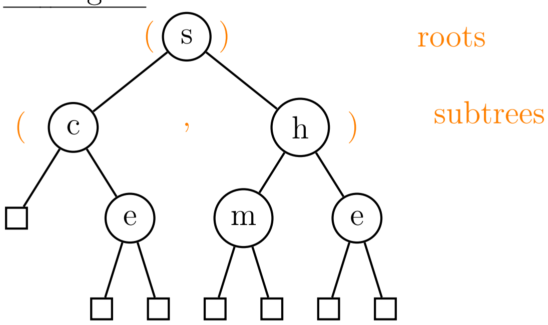
(1) Sammle die Liste der Markierungen der Wurzeln der (nicht-leeren) Bäume n ts (= (roots ts))

(2) Bestimme die Liste ts' der Teilbäume der (nicht-leeren) Bäume n ts (= (subtrees ts))

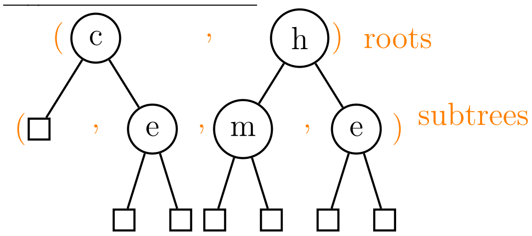
(3) Führe (1) rekursiv auf ts' aus.

(4) komkatiniere die Listen aus (1) und (3).

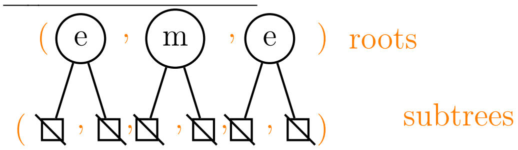
Zu Beginn



1. Rekursionaufruf



2. Rekursionaufruf



3. Rekursionsaufruf

( )

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

Neue Sprachebene DMdA-fortgeschritten

- neues Ausgabevormat in der REPL (list x_1 x_2 x_n) $\rightsquigarrow$ (x_1 x_2 x_n)
  empty $\rightsquigarrow$ ()

- Neuer Gleichheitstest für Werte aller (auch benutzerdef.) Signaturen (:equal? (%a %b -> boolean))

Quote

Sei $\langle$e$\rangle$ein beliebiger Scheme-Ausdruck. Dann liefert (quote $\langle$e$\rangle$) die Representation bon $\langle$e$\rangle$- $\langle$e$\rangle$wird nicht ausgewertet.

Beispiele:

(quote 42) $\rightsquigarrow$ 42
(quote "Leia") $\rightsquigarrow$ "Leia"
(quote #t) $\rightsquigarrow$ #t
(quote + 40 2) $\rightsquigarrow$ (+ 40 2) Funktionsapplikation repräsentiert als Liste
Abkürzung (quote $\langle$e$\rangle$) $\equiv$ '$\langle$e$\rangle$

Notation von Listenliteralen in Programmen (c_i Komstanten) (list c_1 c_2 c_n) $\equiv$ '(c_1 c_2 c_n) empty $\equiv$ '()

Symbole

Was ist (first (* 1 2))? Was sind lambda, x + in '(lambda (x) (+ x 1))?
Neue Signatur symbol zur Repräsentation von Namen in Programmen. Effizientere interne Darstellung, effizient vegleichbar (not equal?). KEin Zugriff auf die einzelnen Zeichen des Symbols. lambda $\neq$ "lambda"

Operatoren

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

Natürliche Repräsentation und Auswertung

Natürliche Repräsentation und Auswertung arithmetischer Ausdrücke (Terme):
Beispiel:
e $\equiv$ '(* (! (+ 1 2)) x)

```
(define term
  (signature (mixed number
                    symbol
                    (list-of term))))
```

- kein Parser benötigt
- kein Prettyprinter

Auswertung möglich, wenn Bindungen für Symbole (Variablen und Operatoren) an Werte gegeben sind. Dictionary (Enviroment):

- d1: { x$\rightarrow$ 3,
  *$\rightarrow$ $\langle$procedure: *$\rangle$,
  + $\rightarrow$ $\langle$procedure: +$\rangle$,
  ^$\rightarrow$ $\langle$procedure: expt$\rangle$,
  ! $\rightarrow$ fac}
  e $\rightsquigarrow\rightsquigarrow$ 18
- d2: { x $\rightarrow$ 1,
  *$\rightarrow$ *,
  +$\rightarrow$ +,
  ^$\rightarrow$ expt,
  !$\rightarrow$ (lambda (x) (-x)) }
  e $\rightsquigarrow\rightsquigarrow$ -3

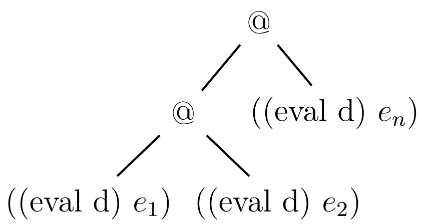
Auswertung eines arithmetischen Ausdrucks e

Auswertung eines arithmetischen Ausdrucks e (im Enviroment d):
((eval d) e)

(E_1) ((eval d) c) = c

(E_2) ((eval $\{x_1 \rightarrow v_1, \quad , x_n \rightarrow v_n\})x_i) = v_i$

(E_3) ((eval d) (e_1 e_2 ... e_n)) = (... (((eval d) e_1) ((eval d) e_2))...((eval d) e_n))



```
(: eval ((dict-of symbol value) -> (term -> value)))

(define eval
  (lambda (d)
    (lambda (e)
      (cond ((constant? e) e) ;(E1)
            ((variable? e) (lookup-dict e d)) ;(E2)
            ((compound? e) (let ((es (map (eval d) e)))
                             (foldl (first es) @ (rest es))))))))

(: @ ((%a -> %b) %a -> %b))
(define @
  (lambda (f x)
    (f x)))
```

Lambda-Kalkül

Das λ-Kalkül ist eine Notation für beliebige Funktionen. Entwicket in den 1930er Jahren von Alonzo Church (1903-1995) als neue Grundlage der Mathematik [aber die Mathematiker bevorzugen die Mengenlehre]. Seither verwendet als thearetischer Unterbau für Programmiersprachen.

Syntax des λ-Kalküls

Die Menge der Ausdrücke (expressions) E des λ-Kalküls ist induktiv definiert (sei V unendliche Memge von Variablennamen).

- $\forall v \in V : v \in E$ [Variable]
- $\forall e_1, e_2 \in E : (e_1 e_2) \in E$ [Applikation]
- $\forall v \in V, e_1 \in E : (\lambda v.e_1) \in E$ [Abstraktion]

Beispiele $y \in E$
$(\lambda y.y) \in E$ Identitätsfunktion
$(\lambda y.z) \in E$ Funktion ignoriert Arg. y, liefert z
$((fx)y) \in E$ Anwendung von f auf x und y (Currying)
$(\lambda f.(f\ x)) \in E$ Anwendung von Argument f auf x (H.O.P., Typ 1)

$((e_1\ e_2)\ e_3) \equiv (e_1\ e_2\ e_3)$ Currying

Freie/Gebundene Variablen

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

- wird der hier nicht bekannte Wert der Variablen f,y,z benötigt, 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
- Variablen f,y,z frei (free) sind.

Welche Variablen eines Ausdrucks sind frei/gebunden?
free (v) = {v}
free ((e_1 e_2)) = free (e_1) $\cup$ free (e_2)
free ((λx.e_1)) = free(e_1) {x}
bound (v) = {}
bound ((e_1 e_2)) = bound (e_1) $\cup$ bound (e_2)
bound ((λv.e_1)) = bound (e_1) $\cup$ { v}
Beispiel:
free (t_1)
= free (((λx((f x) y)) z))
= free ((λx ((f x) y))) $\cup$ free (z)
= (free (((f x) y)) \{x}) $\cup$ {z}
= ((free ((f x)) $\cup$ free (y)) \{x}) $\cup$ {z}
= (((free (f) $\cup$ free (x)) $\cup$ {y}) \{x}) $\cup$ {z}
= ((({f}$\cup${x}) $\cup$ {y})\{x}) $\cup$ {z}
= {f,x,y} \ {x} $\cup$ {z} = {f y z}
Achtung: Bindung/Freiheit muss für jedes Vorkommen einer Variable seperat entschieden werden.
$E_2 \equiv$ (x (λx.x))
free (t_2) = {x}
bound (t_2) = {x}

β-Reduktion

Die Anwendung einer Funktion (= λ-Abstraktion) auf ein Argument wird im λ-Kalkül durch β-Reduktion($\underset{\beta}{\rightarrow}$) beschrieben. Die Applikation(λv.e_1)e_2) wird durchgeführt, indem

(1) eine Kopie des Rumpfes e_1 hergestellt wird und

(2) alle freien Vorkommen von v in e_1 durch e_2 ersetzt werden.

Beispiel:

- $((\lambda x.(*\ ((\lambda x.(+\ x\ a))\ b)\ x))\ c)$
  $\underset{\beta}{\rightarrow}$ $(*\ ((\lambda x.(+\ x\ a))\ b)\ c)$
  $\underset{\beta}{\rightarrow}$ $(*\ (+\ b\ a)\ c)$

- $((\lambda x.foo)\ bar) \underset{\beta}{\rightarrow} foo$

- $((\lambda x.((\lambda y.(*\ y\ x))\ a))\ b)$
  $\underset{\beta}{\rightarrow}$ $((\lambda y.(*\ y\ b))\ a)$
  $\underset{\beta}{\rightarrow}$ $(*\ a\ b)$ Curry

- $((\lambda f.(f\ c))(\lambda x.(+\ x\ a)))$
  $\underset{\beta}{\rightarrow}$ $((\lambda x.(+\ x\ a))\ c)$
  $\underset{\beta}{\rightarrow}$ $(+\ c\ a)$ Higher-Order

- $(((\lambda x.(\lambda y.x))\ (\lambda z.y)\ \maltese)\ a)$ *muesste y liefern*
  $\underset{\beta}{\rightarrow}(((\lambda y.(\lambda z.y))\ \maltese)\ a)$
  $\underset{\beta}{\rightarrow}((\lambda z.\maltese)\ a)$
  $\underset{\beta}{\rightarrow}\maltese$

Definition β-Reduktion

$((\lambda x.e)\ a) \underset{\beta}{\rightarrow} e[x \rightarrowtail a]$ "*ersetze freie Vorkommen von x in a*"

$(\rightarrowtail_1)$ $x[x \rightarrowtail a] = a$

$(\rightarrowtail_2)$ $v[x \rightarrowtail a] = v$

$(\rightarrowtail_3)$ $(e_1\ e_2)[x \rightarrowtail a] = (e_1[x \rightarrowtail a]\ e_2\ [x \rightarrowtail a])$

$(\rightarrowtail_4)$ $(\lambda x.e)[x \rightarrowtail a] = (\lambda x.e)$

$(\rightarrowtail_5)$ $(\lambda x.e)[x \rightarrowtail a] = (\lambda v.e[x \rightarrowtail a])$ falls v nicht frei in a!

$(\rightarrowtail_6)$ $(\lambda x.e)[x \rightarrowtail a] = (\lambda v'.e[v \rightarrowtail v'])\ [x \rightarrowtail a]$ sonst v': neuer Variablenname

Beispiel: $(((\lambda x.(\lambda y.x))\ (\lambda z.y)\ \maltese)\ a)$ *muegsste y liefern*
$\underset{\beta}{\rightarrow}$ $((((\lambda y.x)[x \rightarrowtail (\lambda z.y)])\ \maltese)\ a)$
$\underset{(\rightarrowtail_6)}{\rightarrow}$ $((((\lambda y'.x[y \rightarrowtail y'])[x \rightarrowtail (\lambda z.y)])\ \maltese)\ a)$
$\underset{(\rightarrowtail_2)}{\rightarrow}$ $((((\lambda y'.x)[x \rightarrowtail (\lambda z.y)])\maltese)\ a)$
$\underset{(\rightarrowtail_5)}{\rightarrow}$ $((((\lambda y'.x[x \rightarrowtail (\lambda z.y)]))\maltese)\ a)$
$\underset{(\rightarrowtail_1)}{\rightarrow}$ $(((\lambda y'.(\lambda z.y))\ \maltese)\ a)$
$\underset{\beta}{\rightarrow}$ $((\lambda z.y)[y' \rightarrowtail \maltese]\ a)$
$\underset{(\rightarrowtail_5)}{\rightarrow}$ $((\lambda z.y[y' \rightarrowtail \maltese])\ a)$

$$\underset{(\rightarrowtail_2)}{\rightarrow} ((\lambda z.y)\ a)$$
$$\underset{\beta}{\rightarrow} y[z \rightarrowtail a]$$
$$\underset{(\rightarrowtail_2)}{\rightarrow} y$$