# Informatik II Skript

Steffen Lindner

July 9, 2015

# Contents

# Chapter 1

# Einführung - 14.04.15

Scheme: Ausdrücke, Auswertung und Abstraktion

Dr.Racket: Definitionsfenster (oberer Bereich), Interaktionsfenster (unterer Bereich)

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

**Beispiele**

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

Allgemein: $(< function >< arg1 >< arg2 > ...)$

(+ 40 2) und (odd? 42) sind Beispiele für Ausdrücke, die bei Auswertung einen Wert liefern. (Notation: $\rightsquigarrow$)

(+ 40 2) $\rightsquigarrow$ 42 ($\rightsquigarrow$ = Auswertng / Reduktion / Evalutation)

(odd? 42) $\rightsquigarrow$ #f

Interaktionsfenster: Read $\rightarrow$ Eval $\rightarrow$ Print $\rightarrow$ Read ... (Read-Eval-Print-Loop aka. REPL)

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

Literal:

#t, #f (true, false, Wahrheitswerte) (boolean)

"abc", "x", " " (Zeichenkette) (String)

0 1904 42 -2 (ganze Zahlen) (Integer)

0.42 3.1415 (Fließkommazahl) (Reel)

1/2, 3/4 (rationale Zahl) (Rational)

_(″)_/" (Bilder) (Image)

# Chapter 2

# Ausdrücke, Defines, usw. - 16.04.2015

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

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

Achtung: Scheme rundet bei Arithmetik mit Fließkommazahlen (interne Darstellung ist binär).

Bsp.: Auswertung des zusammengesetzten Ausdrucks 0.7 + (1/2)/0.25 - 0.6/0.3

Arithmetik mit rationalen Zahlen ist exakt.

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

$$(\text{define} < id > < e >)\ (< id > \text{Identifier}, < e > \text{Expression})$$

Erlaubt konsistente Wiederverwendung und dient der Selbstdokumentation von Programmen.

Achtung: Dies ist eine sogenannte Spezifikation und kein Ausdruck. Insbesodnere besitzt diese Spezialform keinen Wert, sondern einen Effekt: Name $< id >$ wird an den Wert von $< e >$ gebunden.

Namen können in Scheme fast beliebig gewählt werden, solange:

1. die Zeichen (kommt noch) nicht vorkommen
2. der Name nicht einem numerischen Literal gleicht
3. kein whitespace (Leerzeichen, Tabulatoren, Return) enthalten ist.

Bsp.: euro → us$

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

Eine Lambda-Abstraktion (auch: Funktion, Prozedur) erlaubt die Formulierung von Ausdrücken, die mittels Parametern konkreten Werten abstrahieren:

$$(\text{lambda}\ (< p1 >< p2 > ...) < e >), < e > \text{Rumpf}$$

$< e >$ enthälft Vorkommen der Parameter $< p1 >, < p2 >$...

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

Anwendung (auch: Applikation/Aufruf) der Lambda-Abstraktion führt zur Ersetzung der vorkommenden Parameter im Rumpf durch die angegebenen Argumente:

(lambda (days) (* days (* 155 min-in-a-day))) ⇝ (* 365 (* 155 min-in-a-day)) ⇝ 81468000

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

Prozeduren sollten im Programm eine ein-bis zweizeiliger Kurzberschreibung direkt voran gestellt werden.

# Chapter 3

# Signaturen, Testfälle - 21.04.15

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

$$(: < id > < signatur >)$$

Bereits eingebaute Signaturen:

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

(: ...) ist eine Spezialform ohne Wert, aber Effekt: Signaturprüfung

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

$$(< signaturp_1 > ... < signaturp_n > - >< signatur - ergebnis >)$$

Prozedur-Signaturen werden bei jeder Anwendung eine Prozedur auf Verletzung geprüft.

Testfälle dokumentieren das erwartete Ergebnis einer Prozedur für ausgewählte Argumente:

$$(check - expect < e_1 >< e_2 >)$$

Werte Ausruck $< e_1 >$ aus und teste, ob der erhaltene Wert der Erwarung (= der Wert von $< e_2 >$) entspricht.

Einer Prozedurdefinition sollten Testfälle direkt vorangestellt werden.
Spezialform: Kein Wert, aber Effekt: Testverletzung protokollieren.

Konstruktionsanleitung für Prozeduren

- ; ... (1) Kurzbeschreibung (1-2 zeiliger Kommentar mit Bezug auf Parameter)
- (: ...) (2) Signatur
- (check-expect ...) (3) Testfälle
- (define (lambda (...) ...) (4) Prozedur + Rumpf

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

Bsp.: Zeichen Ziffernblatt (Stunden- und Minutenzeiger) zur Uhrzeit H:m auf einer analogen 24h-Uhr

- Minutenzeiger legt 360°/60 pro Minute zurück (360/60 * m)
- Stundenzeiger legt 360°/12 pro Stunde zurück (360/12 * h + 360/12 * m/60)

# Chapter 4

# Substitutionsmodell, Fallunterscheidung - 23.0

Reduktionsregeln für Scheme (Fallunterscheidug je nach Ausrucksart)

Wiederhole, bis keine Reduktion mehr möglich:

- Literal (1, "abc", #t, ...) [$\text{eval}_{lit}$]

  l $\rightsquigarrow$ l

- Identifier id (pi, clock-face, ...) [$\text{eval}_{id}$]

  id $\rightsquigarrow$ gebundener Wert

- Lambd-Abstraktion

  (lambda () ) $\rightsquigarrow$ (lambda () ) [$\text{eval}_{\lambda}$]

- Applikation (f, e1, e2)

  - (1) f, e1, e2 reduziere, erhalte f', e1', e2'
  - (2)
    - Operation f' auf e1', e2', ... falls f' primitive Operation (+, *, ...) [$\text{apply}_{prim}$]
    - Argumentenwert e1', e2', ... Rumpf von f' einsetzen, dann Rumpf reduzieren , falls f' Lambdaabstraktion [$\text{apply}_{\lambda}$]

Beispiel: Applikation

(+ 40 2)

$\rightsquigarrow$ (#< procedure + > 40 2) $\rightsquigarrow$ 42

$\text{eval}_{lit}$ (+)

$\text{eval}_{lit}$ (40)

$\text{eval}_{lit}$ (2)•

(position-minute-hand 30)

$\rightsquigarrow$ ((lambda (m) (* degrees-per-minute m)) 30)

$\rightsquigarrow$ (* degrees-per-minute 30)

$\rightsquigarrow$ (* degress-per-minute 30)

$\rightsquigarrow$ (#<procedure * > 360/60 30)

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

Achtung: Das hat Einfluss auf das korrekte Einsetzen von Argumenten für Parameter! (s. $\text{apply}_{\lambda}$)

Das bindenen Vorkommen eines Identifiers x kann im Programmtext systematisch bestimmt werden: suche strik von "innen nach außen" bis zum ersten

- (lambda (x) )
- (define x )

(Prinzip der lexikalischen Bindung)

Übliche Notation in der Mathematik: Fallunterscheidung

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

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

- (: = (number number → boolean))
- (: < (real real → boolean)), auch $>, \leq, \geq$
- (: string=? (string string → boolean)), auch string>?, string≤?
- (: boolean? (boolean boolean → boolean))
- (: zero? (number → boolean))
- odd?, even?, positive?, negative?, ...

Binäre Fallunterscheidung: if

(if $< t_1 >< e_1 >< e_2 >$)

Mathematisch: $\begin{cases} e_1, & \text{falls } t_1 \\ e_2, & \text{sonst} \end{cases}$

# Chapter 5

# One-of Signatur - 28.04.15

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

$$(\text{one-of} < e_1 > ... < e_n >)$$

Reduktion von if:

$$(\text{if } t_1 \ e_1 \ e_2) \rightsquigarrow \begin{cases} < e_1 >, & \text{falls t1' = \#t ; e2 wird niemals ausgewertet} \\ < e_2 >, & \text{sonst; e1 wird niemals ausgewertet} \end{cases}$$

(1) Reduziere $t_1$, erhalte $t_1'$

Spezialform Fallunterscheidung (conditional expression):

$$(\text{cond } (< t_1 >< e_1 >) \ ... \ (< t_n >< e_n >) \ (\text{else} < e_{n+1} >)) \ \text{(else optional)}$$

Werte die Tests in der Reihenfolge $t_1, t_2, ..., t_n$ aus. Sobald $t_i$ # t ergibt werte Zweig $e_i$ aus. $e_i$ ist das Ergebnis der Fallunterscheidung. Wenn $t_n$ # f liefert, dann liefere

$$\begin{cases} Fehlermeldung, & \text{"cond: alle Tests ergaben \#f", falls kein else-Zweig} \\ < e_{n+1} >, & \text{sonst} \end{cases}$$

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

$$(\text{cond } (< t_1 >< e_1 >)(< t_2 >< e_2 >)...) \rightsquigarrow \begin{cases} < e_1 >, & \text{falls t1' = \#f} \\ (cond(< t_2 >< e_2 >)(...)), & \text{sonst} \end{cases}$$

Reduziere $t_1$, erhalte $t_1$'.

(cond ) $\rightsquigarrow$ Fehlermeldung "Alle Tests..."

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

cond ist "systematischer Zucker"

(auch: abgleitete Form) für eine verschachtelte Anwendung von 'if':

(cond $(< t_1 >< e_1 >)$ $(< t_1 >< e_1 >)$ ... ))) entspricht (if $< t_1 >< e_1 >$ $(if < t_1 >< e_1 > (if...))$

Spezialformen 'and' und 'or':

$$(or < t_! >< t_2 > ... < t_n >) \text{ entspricht } (if < t_1 > \#t(or < t_2 > ...))$$

$$(\text{or}) \rightsquigarrow \#\text{f}$$

$$(and < t_1 > ... < t_n > \rightsquigarrow (if < t_1 > (and < t_2 > ... < t_n >)\#f)$$

$$(\text{and}) \rightsquigarrow \#t$$

# Chapter 6

# Zusammengesetzte Daten - 30.04.15

Ein Charakter besteht aus drei Komponenten.

- Name des Charakters (name)
- Handelt es sich um einen Jedi? (jedi?)
- Stärke der Macht (force)

→ Datendefinition für zusammengesetzte Daten.

Konkreter Charakter:

| Name | "Luke Skywalker" |
|---|---|
| jedi? | #f |
| force | 25 |

; Ein Charakter (character) besteht aus

; - Name (name)

; - Jedi-Status (jedi?)

; - Stärke der Macht (force)

(define-records-procedures charakter

make-character

character?

(character-name

character-jedi

character-force))

(make-character n j f) $\rightsquigarrow$ $<records>$ (konstruktion)

(character-name $<record>$ $\rightsquigarrow$ n (Komponentenzugriff)

(character-jedi? $<record>$) $\rightsquigarrow$ j (Komponentenzugriff)

(character-force $<record>$) $\rightsquigarrow$ f (Komponentenzugriff)

Zusammengesetzte Daten = Records in Scheme.

Record-Definition legt fest:

- Record-Signatur
- Konstruktor (Baut aus Komponenten einen Record)

- Prädikat (liegt Record vor?)
- Liste von Selektoren (lesen jewils eine Komponenten des Records)

$$\text{(define-records-procedures} < t >$$
$$\text{make-}< t > \text{ ; Konstruktor}$$
$$< t >?$$
$$(< t > - < comp_1 > ... < t > - < comp_n > 4)) \text{ ; Liste der n-Seleketoren}$$

Verträge des Konstruktors / der Selektoren für Record-Signatur $< t >$ mit n Komponenten namens $< comp_1 > ... < comp_n >$:

- (: make-$< t >$ ($< t_1 > ... < t_n > \rightarrow < t >$))
- (: $< t > - < comp_1 >$ ($< t > \rightarrow < t_1 >$))
- ...
- (: $< t > - < comp_n >$ ($< t > \rightarrow < t_n >$))

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

(character-name (make-character n j f)) = n

(analog für den Rest)

Interaktion von Funktionen (algebraische Eigenschaften).

Spezialform check-property:

```
(check-property
  (for-all ((<id_1>  <sig_1> ... <id_n>  <sig_n>))
  <e>))
```

$< e >$ bezieht sich auf $< id_1 > ... < id_n >$.

Test erfolgreich, falls $< e >$ für bel. gewählte Bindungen für $< id_1 > ... < id_n >$ immer #t ergibt.

Interaktion von Selektor und Konstruktor:

```
(check-property
  (for-all ((n string)
            (j booleans)
            (f integer))
  (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 ((x_1 natural)
            (x_2 natural))
  (\geq (+ x_1 x_2) (max x_1 x_2))))
```

Konstruktion von Funktionen, die zusammengesetzte Daten konsumieren:

- Welche Record-Komponenten sind relevant für Funktionen?

  → Schablone:

  ; könnte Charakter e ein Sith-Lord sein?

  (: sith? (character → boolean))

```
(define sith?
  (lambda (e)
    ... (character-jedi? c) ... (character-force c) ... ))
```

Konstrukton von Funktionen, die zusammengesetzte Daten konstruieren:

- Der Konstruktor muss aufgerufen werden.

  → Schablone:

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

# Chapter 7

# Fortsetzung zusammengesetzte Daten - 05.05.1

- lego-character
  - figure
  - character
    - Name
    - jedi?
    - force

Sei $< p >$ ein Prädikat mit Signatur $(< t > \rightarrow boolean)$. Eine Signatur der Form (predicate $< p >$) gilt für jeden Wert x der Signatur $< t >$ sofern $(< p >$ x$) \rightsquigarrow \#t$. Signatur (predicate $< p >$) ist somit spezifischer (restriktiver) als die Signatur $< t >$ selbst.

Einfürhung eines neuen Signaturnamens $< new - t >$ für die Signatur $< t >$:

$$(\text{define} < new - t > (\text{signatur} < t >))$$

Bsp.:

```
(define farbe
  (signatur
    (one-of "Karo" "Herz" "Pik" "Kreuz")))

(define latitude
  (signature (predicate latitude?)))
```

# Chapter 8

# Gemischte Daten - 07.05.15

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

(: geocoder (string → (mixed geocode geocode-error)))

Ein Geocode besteht aus:

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

(: geocode-address (geocode → string)) ...
Ein geocode-error besteht aus:

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

Teachpack: geocoder.rkt
Gemischte Daten Die Signatur

$$(\text{mixed} < t1 > ... < t_n >)$$

ist gültig für jeden Wert, der mindestens eine der Signatur $< t_1 > ... < t_n >$ erfüllt.
Beispiel: Datendefinition:
Eine Antwort des Geocoders ist entweder

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

Beispiel (eingebaute Funktion string → number):
(: string→number (string → (mixed number (onfe of #f))))
(string→number "42") ⇝ 42

(string→number "foo") ⇝ #f

Erinnerung:

Das Prädikat $< t >?$ einer Signatur $< t >$ unterscheidet Werte der Signatur $< t >$ von allen anderen Werten:

(: $< t >?$ (any → boolean))

Auch Prädikafür eingebaute Signaturen:

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

Prozeduren die gemische Daten der Signaturen $< t_1 > ... < t_n >$ konsumieren:

Konstruktionsanleitung:

(: $< f >$ ((mixed $< t_1 > ... < t_n >$ → ...))

(define $< f >$ (lambda (x) (cond (($< t_1 >?$ x) ...) ... ($< t_n >?$ x)...)))))

Mittels let lassen sich Werte an lokale Namen binden:

$$(\text{let } ((< id_1 >< e_1 >) \ ... \ (< id_n >< e_n >)) < e >)$$

Die Ausdrücke $< e_1 > ... < e_n >$ werden parallel ausgewertet → $< id_1 > ... < id_n >$ können in $< e >$ (und nur hier) verwendet werden. Der Wert let-Ausdruck ist der Wert von $< e >$.

Achtung: 'let' ist verfügbar ab Sprachebene "DMdA".

'let' ist syntaktischer Zucker.

(let $((< id_1 >< e_1 >) \ ... \ (< id_n >< e_n >)) < e >$)

↔

((lambda $(< id_1 > ... < id_n >) < e >) < e_1 > ... < e_n >$)

# Chapter 9

# Parametrisch polymorphe Funktionen - 12.05.1

Abstand zwier geografischer Positionen $l_1$, $l_2$ auf der Erdkugel in km (lat, lng jeweils in Radian):

dist($l_1$, $l_2$) = Erdradius in km · acos(cos($l_1$.lat) · cos($l_1$.lng) · cos($l_2$.lat) · cos($l_2$.lng) + cos($l_1$.lat) · sin($l_1$.lng) · cos($l_2$.lat) · sin($l_2$.lng) + sin($l_1$.lat) · sin($l_2$.lat))

Parametrisch polymorphe Funktionen

Beobachtung: Manche Prozeduren arbeiten unabhängig von den Signaturen ihrer Argumente: parametrisch polymophe Prozduren (gr.: vielgestaltig). Nutze Signaturvariablen: %a, %b, ...

Beispiel:

; Identität

(: id (%a → %a))

(define id (lambda (x) x))

; Konstante Funktion (ignoriert zweites Argument)

(: const (%a %b → %a))

(define cost (lambda (x y) x))

; Projection (ein Argument auswählen)

(: proj ((one-of 1 2) %a %b → (mixed %a %b)))

(define proj (lambda (i $x1$ $x_2$) (cond ((= i 1) $x_1$) ((= i 2) $x_2$))))

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

Beispiel:

Wenn eine Prozedur (number %a %b → %a) erfüllt, dann auch :

(number string boolean → string)

(number boolean natural → boolean)

(number number number → number)

; Ein polymorphes Paar (pair) besteht aus

; - erster Komponente (first)

; - zweiter Komponente (rest)

; wobei die Komponenten bel. Signaturen besitzen

```
(define-record-procedures-pair pair-of
```

```
make-pair
pair?
(first rest))
```

(pair-of $< t_1 > < t_2 >$) ist eine Signatur für Paare, deren erste bzw. zweite Komponente die Signaturen $< t_1 >$ bzw. $< t_2 >$ erfüllen

→ pair-of: Signatur mit (zwei) Signaturparametern

(: make-par (%a %b → (pair-of %a %b)))
(: pair? (any → boolean))
(: first ((pair-of %a %b) → %a))
(: rest ((pair-of %a %b) → %b))

# Chapter 10

# Listen - 12.05.15

Eine Liste von werten der Signatur $< t >$ (list-of $< t >$) ist entweder:

- leer (Signatur empty-list) oder
- ein Paar (Signatur pair-of) aus einem Wert der Signatur $< t >$ und einer Liste von Werten der Signatur $< t >$

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

Signatur empty-list bereits in Racket vordefiniert. Ebenfalls vordefiniert:

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

Operationen auf Listen

- Konstruktoren:

```
(: empty empty-list) ; leere Liste
(: make-pair (%a (list-of %a) -> (list-of %a)))
```

- Prädikate:

```
(: empty? (any -> boolean)) ; leer Liste?
(: pair? (any -> boolean)) ; nicht-leere Liste?
```

- Selektoren:

```
(: first ((list-of %a) -> %a)) ; Kopfelement
(: rest (list-of %a) -> (list-of %a))) ; Restliste
```

# Chapter 11

# Listenprozeduren - 19.05.2015

### Prozeduren, die Listen konsumieren

Konstruktionsanleitung befolgen!

Beispiel:

```
; Summe der Zeichen der Liste xs
(: list-sum (list-of number) -> number)
(check-expect (list-sum empty) 0)
(check-expect (list-sum one-to-four) 10)
```

Schablone (gemische + zusammengesetzte Daten)

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

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

(list-sum (rest xs)) erzielt Fortschritt !.

Konstruktionsanleitung für Prozeduren ¡f¿, die Liste xs konsumiert.

```
(: <f> ((list-of <t_1>) -> <t_2>))
(define <f>
  (lambda (xs)
    (cond ((empty? xs) ...)
          ((pair? xs) ... (first xs) ... (<f> (rest xs)) ...))))
```

Neue Sprachebene "Macht der Abstraktion"

- Signatur (list-of % a) eingebaut
- Neuer syntaktischer Zucker:

  ```
  (list <e_1> <e_2> ... <e_n>)
  ```

- Ausgabeformat für nicht leere Liste:

  $\#< list\ x_1\ x_2 ... x_n >$

Füge Listen xs, ys zusammen (concatenation):

Beobachtung:

- Die Lnge von xs bestimmt die Anzahl der rekusriven Aufrufe von cat
- Auf ys werden niemals Selektoren angewandt

# Chapter 12

# Rekursion auf Listen - 21.05.15

Generierung aller natürlichen Zahlen (vgl. gemische Daten). Eine natürliche Zahl (natuarl) ist entweder:

- die 0 (zero)
- der Nachfolger (succ) einer natürlichen Zahl.

Konstruktoren

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


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

Vorgängerfunktion (pred), definiert für $n > 0$:

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

Bedinge algebraische Eigenschaften (s. check-property):

$(\Rightarrow < p > < t >)$

Nur wenn $< p > \not\Rightarrow \#t$, wird Ausdruck $< t >$ ausgewertet und getestet ob $< t > \not\Rightarrow \#t$.

Beispiel:

Fakultätsfunktion n! $(n \in \mathbb{N})$

$0! = 1$

$n! = n \cdot (n\text{-}1)!$

Konstruktionsanleitung für gem. Daten:

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

Beobachtung:

- Im letzten Zweig ist $n > 0 \Rightarrow$ pred anwendbar
- $(< f >$ (- n 1)) hat die Signatur $< t >$

Satz

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

Beweis: in Kürze

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

$\Rightarrow$ Wenn mäglich, erzeugte Reduktionsprozesse, die konstanten Platzverbrauch - unabhängig vom Eingabeparametern - benötigen.

# Chapter 13

# Endrekursive Prozeduren - 09.06.2015

Idee für die Multiplikation:

Führe Multiplikation sofort aus. Schleife das Zwischenergebnis (akkumulierendes Argument) durch die Berechnung. Am Ende enthält Akkumulator das Ergebnis.

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

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

Ein Berechnungsprozess ist iterativ, falls seine Größe konstant bleibt.

Damit:

- factorial nicht iterativ
- fac-worker iterativ

Wieso ist fac-worker iterativ?

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

Kontekt des rekursiven Aufrufs in

- factorial : (* n [])
- fac-worker : keiner

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

Endrekursive Prozeduren generieren iterative Berechnungsprozesse.

Beobachtung:

```
(cat (list 1000 ... 2) (list 1)
```

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

⇒ Quadratischer Aufwand

Konstruiere iterative listenumkehr (backwards)

Berechnung von (backwards (list 1 2 3))

```
(: backwards-worker ((list-of %a) (lsit-of %a) \rightarrow (list-of %a)))
```

Mittels letrec lassen sich Werte an lokale Namen binden:

```
(letrec ((<id_1> <e_1>)
           ...
          (<id_n> <e_n>))
         <e>)
```

Die Ausdrücke $e_1, ...e_n$ und e dürfen sich auf die Namen von $< id_1 > ... < id_n >$ beziehen.

# Chapter 14

# Induktive Definitionen - 11.06.2015

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) $\forall n \in \mathbb{N} : succ(n) \neq 0$
- (P4) $\forall m, n \in \mathbb{N} : succ(m) = succ(n) \Rightarrow m = n$

  (P3) und (P4) $\Rightarrow \mathbb{N}$ ist induktiv definiert.
- (P5) Induktionsaxiom

  Für jede Menge $M \subseteq \mathbb{N}$ mit $0 \in M$ und $\forall n : n \in M \Rightarrow succ(n) \in M$, gilt $M = \mathbb{N}$

  - $\mathbb{N}$ enhält nicht mehr als die durch 0 und die durch succ() generierten Elemente
  - Nichts sonst ist in $\mathbb{N}$

Beschweisschema der vollständigen Induktion:

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

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

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

Definiere $M = \{n \in \mathbb{N} | P(n)\} \subseteq \mathbb{N}$

Induktionsaxiom:

Falls $0 \in M$

und $\forall n(n \in M \Rightarrow succ(n) \in M)$

dann $M = \mathbb{N}$

# Chapter 15

# Prozeduren höherer Ordnung (high-order procedures) - 16.06.2015 und 18.06.2015

```
; Extrahiere die Elemente von xs, die Predikat p? erfuellen

(: filter ((%a -> boolean) (list-of %a) -> (list-of %a)))
(define filter
  (lambda (p? xs)
    (cond ((empty? xs) xs)
          ((pair? xs)
            (if (p? (first xs))
              (make-pair (first xs) (filter p? (rest xs)))))))
```

Wert des Parameters p? ist Prozedur

Higher-order procedures (H.O.P)

(a) akzeptieren Prozeduren als Parameter und/oder

(b) liefern Prozeduren als Ergebnis

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

Bsp.: (map f xs)

```
; Wende f auf alle ELemente von Liste 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)))))))
```

Allgemeinere Transformation von Listen: Listenfaltung (list-folding), Idee: Ersetze die Listenkonstruktoren make-pair und empty systematisch:

- (foldr z c xs) wirkt als Spine Transformer
  - empty $\rightsquigarrow$ z
  - make-pair $\rightsquigarrow$ c

- Eingabe: Liste (list-of %a)
- Ausgabe: Im allg. keine Liste mehr (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)))))))
```

Beispiel Summe:

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

Beispiel Länge einer Liste durch Listenreduktion:

```
(foldr 0 c xs)
```

mit c = (lambda (x r) (+ 1 r))

# Chapter 16

# Teachpack ”universe” - 23.6.

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

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

optional

- (: <init> %a)

Startzustand -(: <tock> (%a → %a))

Funktion, die neuen Zustand aus altem Zustand berechnet, wird 28 Mal pro Sekunde aufgerufen.

- (: <render> (%a → image))

Funktion, die aus aktuellem Zustand ein Bild einer Szene berechnet (wird in Fenster mit Dimension <w>x<h> Pixel angezeigt).

- Bei Schliessen der Animation wird der letzte Zustand zurück gegeben.

### 16.0.1 Implementierung Star Wars VII

Ausgabe der römischen Episodennummer für Film f. (roman (film-episode f))

Gesuchte Funktion ist *Komposition* von zwei existierenden Prozeduren.

1. Wende ’film-episode’ an, dann
2. wende ’roman’ auf das Ergebnis von (1) an.

### 16.0.2 Komposition von Prozeduren allgemein:

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

neue Prozedur realisiert

Komposition von f und g.

Mathematik: (compose f g) ≡ f ∘ g. ’f nach g’

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

```
                (lambda (x)
                        (f (g x)))))
```

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

$f^0$ = id - (id ≡ (lambda (x) x))

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

```
(: repeat (natural (%a -> %a) -> (%a -> %a)))
(define repeat
        (lambda (n f)
                (cond ((= n 0) (lambda (x) x))
                      ((> n 0) (compose f (repeat (- n 1) f))))))
```

; Greife auf das n-te Element der Liste xs zu

```
(: nth (natural (list-of %a) -> %a))
(define nth
        (lambda (n xs)
                ((compose first (repeat (- n 1) rest)) xs)))
```

## 16.1 Parametrische Kurven (Higher Order Datenstrukturen)

$f(t) =< x(t), y(t) >$ siehe rkt-Datei

Reduktion:

((add 1) 21)

$^{eval_id} \rightsquigarrow$ (((lambda (x) (lambda (y) (+ x y))) 1) 21)

$^{apply-lambda(x)} \rightsquigarrow$ $\underline{((lambda(y)(+1y))}21)$

$^{apply-lambda(y)} \rightsquigarrow$ (+ 1 21)

# Chapter 17

# Currying (Haskell B. Curry) 25.6.

Erstmals implementier von Moses Schönfinkel.

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

- Jede n-stellige Prozedur lässt sich in eine alternative curried Prozedur transformieren, die in n Schritten jeweils ein Argument: curry. Umgekehrte Transformation: uncurry.

(%a %b → %c) → Applikation auf 2 Argumente (Signaturen %a %b) → %c

curry ↑ ↓ uncurry = (%a (%b → %c)) → Applikation auf 1 Argument (Signaturen %a) → (%b %c) → Applikation auf 1 Argument (Signatur %b) → %c

partielle Applikation

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

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

## 17.1 Beispiel Ableitung als HOP

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

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

- Operator _' (Ableitung) konsumiert Funktion f und produziert Funktion f' ⇒ _' ist higher-order!

```
(: diffquot (h f -> f'))
(: diffquot (reatl (real -> real) -> (real -> real)))
(define difquot
        (lambda (h f)
                (lambda (x)
```

```
(/ (- (f (+ x h)) (f x))
   h))))
```

## 17.2 Beispiel Charakteristische Funktion einer Menge $s \subseteq M$

Charakteristische Funktion für S: (: $\chi_S(M \to \text{boolean})$)

(Sei 0 = #f und 1 = #t)

$$\chi_S(x) = \begin{cases} 1 & , \quad x \in S \\ 0 & , \quad \text{sonst} \end{cases}$$

$\chi_s(m) = \#\text{f}$

$\chi_s(s) = \#\text{t}$

- Idee: Repräsentative $S \subseteq M$ durch Prozedur (M $\to$ boolean) und Mengenoperation durch Operation auf Prozeduren (H.O.P.)
- :-) Darstellung unendlicher Mengen ($S_{42} = \{x \in \mathbb{Z} | x > 42\}$)
- :-) Mengenoperationen ($\cup, \cap, \setminus$) in konstanter Zeit.

Element $x$ in Menge S einfügen:

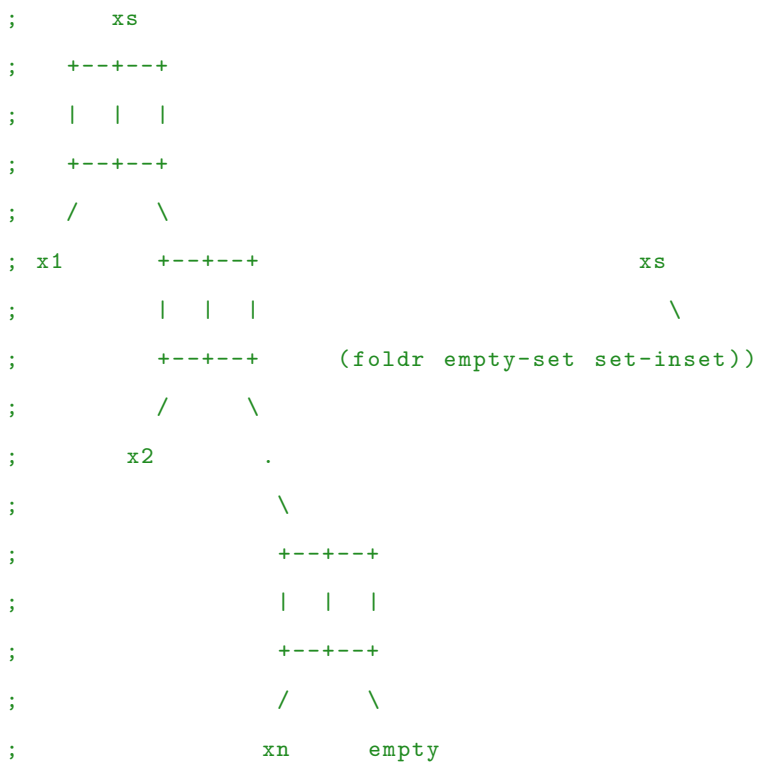
(Sei 0 = #f und 1 = #t )

$$\chi_{S\{x\}}(y) = \begin{cases} 1 & , \quad x \in S \\ \chi_S(y) & , \quad \text{sonst} \end{cases}$$

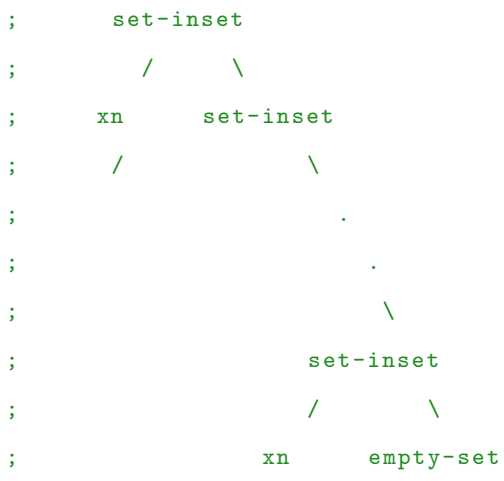
# Chapter 18

# set-inset, Streams, delay, force 30.6.

Konvertiere Liste xs in eine Menge gleicher Elemente:




⇝ (fold empty-set set-in-set xs)) - empty → empty-set - make-pair → set -inset



### 18.0.1 Charakterisiere Funktion zur Repräsentation von Mengen:

1. Performance: set-member 2 hat lineare Laufzeit bei mit set-inset konstruierten Mengen (wie Liste!)
2. Vorteile:
   - unendliche Mengen darstellbar
   - Mengenoperationen in konstanter Zeit durchführbar
3. Nachteil: Elemente sind nicht aufzählbar

### 18.0.2 Streams (stream-of %a)

Unendliche Ströme von Elementen $x_i$ mit Signaturen %a. Ein Stream ist ein Paar.

```
;   Stream-head
;   +----+-------+
;   | x1 |  tail |
;   +----+-------+
```

%a ($\rightarrow$ (stream-of %a)) -Erst eine Ausführung des tails (*fore*) erzeugt nächstes Stream-Elemt (daher auch: lazy list)

```
;   +----+---------------+                          +----+-------+
;   | xn |  (force tail) |            ->            | x2 |  tail |
;   +----+---------------+                          +----+-------+
```

- Vergleich:

```
;   (list-of %a)                          (stream-of %a)
;      xs
;   +--+--+                                +----+--+
;   |  |  |                                | xs |  |
;   +--+--+                                +----+--+
;   /     \                                /       \
; x1      +--+--+                        xn      +--+--+
;         |  |  |                                |  | 2|
;         +--+--+                                +--+--+
;         /     \                                /
;       x2       .                            x2
;                 \
;                 +--+--+
;                 |  |  |
;                 +--+--+
;                 /     \
;               xn      empty
```

**delayed evaluation**

Verzögerte Auswertung eines Ausdrucks (delayed evaluation)

- (delay e) : Verzögere die Auswertung des Ausdrucks e und liefere "Versprechen " (promise), e bei Bedarf später auswerten zu können (delay e) ≡ (lambda () e {nicht ausgewertet})
- (force p): Erzwingt Auswertung des Promise p, liefert Wert zurück

  (: force ((→ %a) → %a))

```
(define force
        (lambda (p)
                (p)))
```

# Chapter 19

# Stream von fib - 2.7.

Generiere den unendlichen Strom fibs der Fibonacci-Zahlen

f(0) = 1; fib(1) = 1; fib(n) = fib(n - 1) + fib(n - 2)

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

Beobachtung:

1 1* 2 3 5 8 Start des Streams fibs

\+ 1* 2 3 5 8 tail von fibs

= 2 3 5 8 tail von tail von fibs

Stream-Diagramm zu fibs:

yet-to-follow

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

$(T_1)$ empty-tree $\in T(M)$

$(T_2)$ $\forall x \in M$ und $l, r \in T(M)$: (make-node l x r) $\in T(M)$

$(T_3)$ Nichts sonst ist in $T(M)$

### 19.0.3 Hinweis:

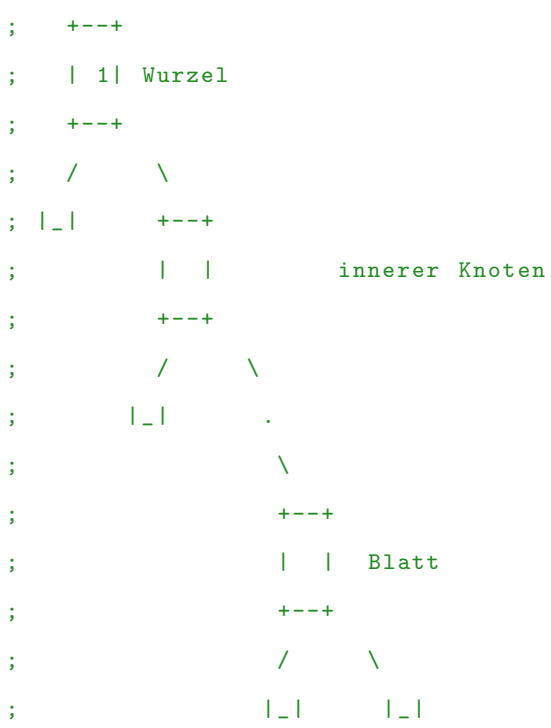
- Jeder Knoten (make-node) in einem Binärbaum hat zwei Teilbäume sowie eine Markierung (label) x
- Vergleiche: $M*$ und $T(M)$, empty und empty-tree, make-pair und make-node

### 19.0.4 Visualisierung

- empty-tree: □
  (make-node l x r): (Zeichnung yet-to-come)
- 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)
  (Zeichnung yet-to-come)

**Beispiele für Binärbäume der Menge $T(M)$**

Baum $t_1$ (listenartig, rechts-tief):




Baum $t_2$ (balanciert): alle Teilbäume auf einer Tiefe haben gleiche Anzahl von Knoten

(Zeichnung)

(Binär-) Bäume haben zahllose Anwendungen:

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

Bäume sind **die** induktive Datenstruktur!

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

(btree-depth empty tree) = 0

(btree-depth t1) = 3

(btree-depth t2) = 2

### 19.0.5 Schablone (gemischte Daten):

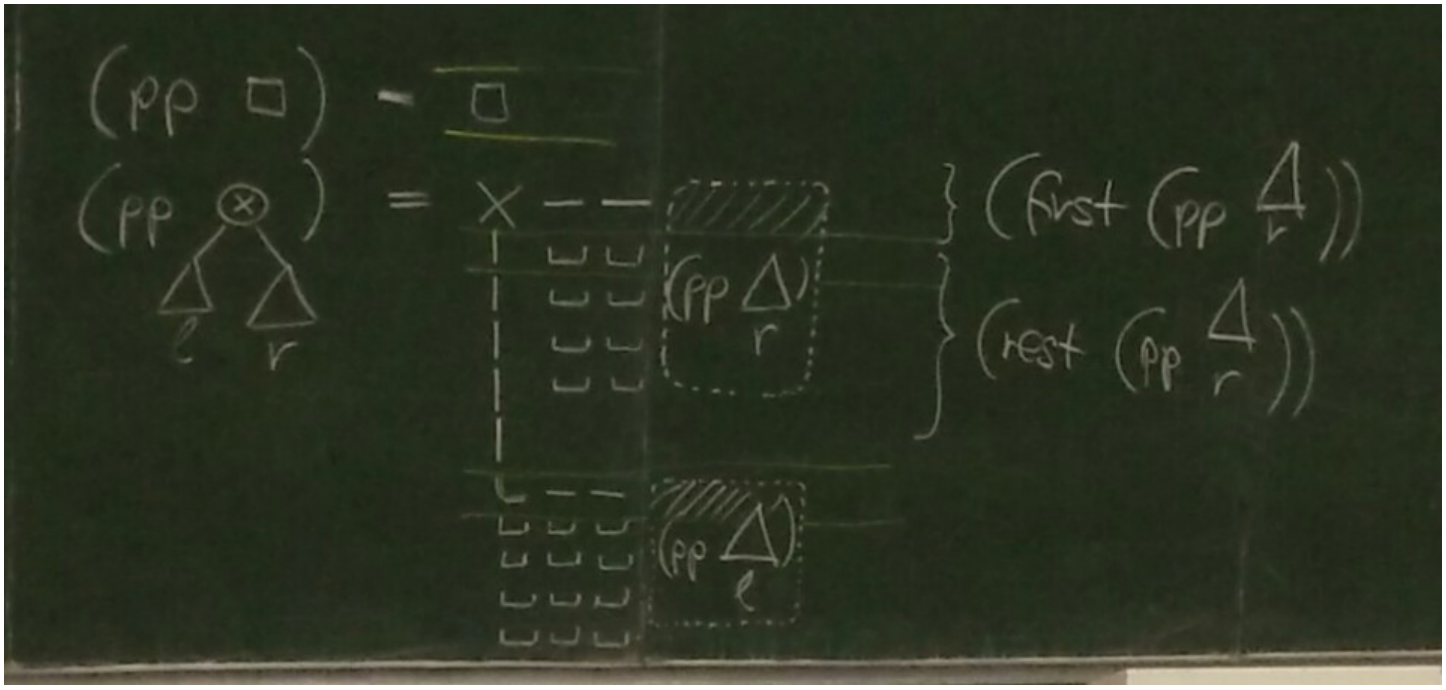
```
(: btree-depth ((btree-of %a) -> natural))
(define btree-depth
        (lambda (f)
                (cond ((empty-tree? t) 0)
                      ((node? t) (+ 1 (max (btree-depth (node-left-branch))
                                           (btree-depth (node-right-branch)))))))
```

# Chapter 20

# Fortsetzung Bäume - 7.7.

## 20.1 Einschub: Pretty-Printing von Bäumen

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




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

1. Nutze (string-append ...) um Zeilen-Strings zu definieren. (horizontale Konkatenation).

2. Nutze (append ...) um die einzelnen Zeilen zu einer Liste von Zeilen zusammenzusetzen (vertikale Konkatenation). Erst direkt vor der Ausgabe werden die Zeilen-Strings zu einem auszugebenden String zusammengesetzt. (strings-list → string)

## 20.2 Induktion über Binärbäume

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

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

Falls P(empty-tree) [Induktionsbasis]

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

### 20.2.1 Beispiel:

Zusammenhang zwischen Grösse (btree-size) und Tiefe (btree-depth) eines Binärbaums t ("ein Baum der Tiefe n enthält mindestens n Knoten und höchstens $2^n - 1$ Knoten"):

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

Erstes $\leq$ trivial. Zweites:

Induktionsbasis P(empty-tree):

```
(size empty-tree)
```

$*\rightsquigarrow$[depth] 0

$= 2^{(0)} - 1\surd$

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

(size (make-node l x r))

$\rightsquigarrow$[size] (size l) + 1 + (size r)

$\leq$ [I.V.] $2^{(depth\ l)} - 1 + 1 + 2^{(depth\ r)} - 1$

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

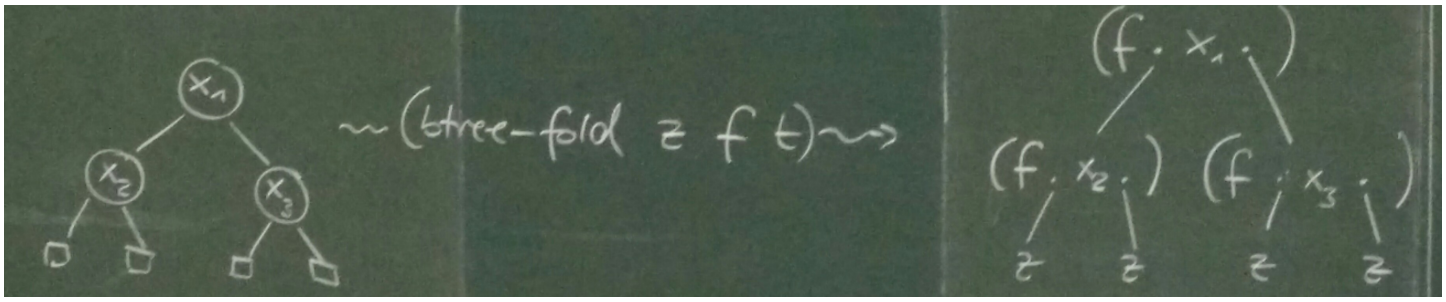
$\leq 2$ x max($2^{(depth\ l)} + 2^{(depth\ r)}$) -1

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

$= 2^{(1+max((depth\ l),\ (depth\ r)))}$

$\leftarrow$[depth] $2^{depth((make\text{-}node\ l\ x\ r))} - 1\surd$

Wie müsste sich btree-fold, eine fold-Operation für Binärbäume verhalten? Tree transformer für Baum t:




opku9zfswdefrtgzhujikolpölokijuzhtgrffgtzhujikolpkijuzhtgrftgzhujiklpö [Anmerkung Marvin]

Falte Baum t bzgl. f und z:

```
(: btree-fold (%a %b %a -> %a) (btree-of %b) -> %a))
```

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

# Chapter 21

# Fortsetzung Bäume (btree-fold) - 9.7.

Bestimme die Markierung Lm (left most) links-aussen im Baum t (oder empty, falls t leer ist).

(leftmost (x) □ ) = (list x)

(leftmost (x) Δ) = (leftmost Δ)

Hinweis:

- Nutzt die leere Liste (empty) als Fehlerindikator (insbesondere kein Abbruch mit violation)

(Prinzip: "Replacing Failure by a list of successes", Philip Wadler)

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

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

**Rechtstiefe Bäume und Listen sind isomorph (gleichgestaltig)**

Beweis: $f \circ f^{-1} = \mathrm{id}$

```
(list->btree (btree->list x))
(right-deep? btree-fold ->  Tuete Gummibaerchen)
```

## 21.1 Tiefendurchläufe (depth-first-traversals)

Ein Tiefendurchlauf (depth-first-traversal) eines Baumes t sammelt die Markierung jedes Knotens 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 (xs1 x xs2)

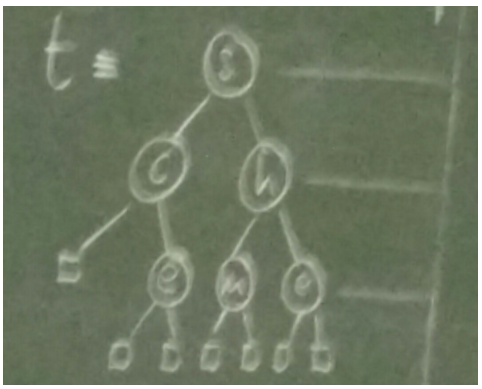
(b) preorder traversal (x xs1 xs2)

(c) postorder traversal (xs1 xs2 x).

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

Baumdarstellung eines arithmetischen Ausdrucks

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




(levelorder t) $\rightsquigarrow$ (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 in ts auf (= (roots ts) )
2. Bestimme ts' der nicht-leeren Teilbäume der Bäume in ts (= (subtrees ts))
3. Führe (1) rekursiv auf ts' aus.
4. Konkateniere die Listen aus (1) und (3).

zu Beginn:

roots: "s"

subtrees: ( (c), (h) ) 1. Rekursion: ( (c), (h) )

roots: "c" "h"

subtrees: ( (e) (m) (e) )

2. Rekusion: ( (e) (m) (e) )

roots: "e" "m" "e" subtress: ( )