

# Informatik II Skript

Steffen Lindner

July 23, 2015

# Contents

<b>1 Einführung - 14.04.15</b>	<b>4</b>
<b>2 Ausdrücke, Defines, usw. - 16.04.2015</b>	<b>5</b>
<b>3 Signaturen, Testfälle - 21.04.15</b>	<b>6</b>
<b>4 Substitutionsmodell, Fallunterscheidung - 23.04.15</b>	<b>8</b>
<b>5 One-of Signatur - 28.04.15</b>	<b>10</b>
<b>6 Zusammengesetzte Daten - 30.04.15</b>	<b>11</b>
<b>7 Fortsetzung zusammengesetzte Daten - 05.05.15</b>	<b>14</b>
<b>8 Gemischte Daten - 07.05.15</b>	<b>15</b>
<b>9 Parametrisch polymorphe Funktionen - 12.05.15</b>	<b>17</b>
<b>10 Listen - 12.05.15</b>	<b>19</b>
<b>11 Listenprozeduren - 19.05.2015</b>	<b>20</b>
<b>12 Rekursion auf Listen - 21.05.15</b>	<b>22</b>
<b>13 Endrekursive Prozeduren - 09.06.2015</b>	<b>24</b>
<b>14 Induktive Definitionen - 11.06.2015</b>	<b>26</b>
<b>15 Prozeduren höherer Ordnung (high-order procedures) - 16.06.2015 und 18.06.2015</b>	<b>27</b>
<b>16 Teachpack "universe" - 23.6.</b>	<b>29</b>
16.0.1 Implementierung Star Wars VII . . . . .	29
16.0.2 Komposition von Prozeduren allgemein: . . . . .	29
16.1 Parametrische Kurven (Higher Order Datenstrukturen) . . . . .	30
<b>17 Curryng (Haskell B. Curry) 25.6.</b>	<b>31</b>
17.1 Beispiel Ableitung als HOP . . . . .	31
17.2 Beispiel Charakteristische Funktion einer Menge $s \subseteq M$ . . . . .	32
<b>18 set-inset, Streams, delay, force 30.6.</b>	<b>33</b>
18.0.1 Charakterisiere Funktion zur Repräsentation von Mengen: . . . . .	34
18.0.2 Streams (stream-of %a) . . . . .	34

<b>19 Stream von fib - 2.7.</b>	<b>36</b>
19.0.3 Hinweis: . . . . .	36
19.0.4 Visualisierung . . . . .	36
19.0.5 Schablone (gemischte Daten): . . . . .	37
<b>20 Fortsetzung Bäume - 7.7.</b>	<b>38</b>
20.1 Einschub: Pretty-Printing von Bäumen . . . . .	38
20.2 Induktion über Binärbäume . . . . .	38
20.2.1 Beispiel: . . . . .	39
<b>21 Fortsetzung Bäume (btree-fold) - 9.7.</b>	<b>41</b>
21.1 Tiefendurchläufe (depth-first-traversals) . . . . .	41
<b>22 Zeichencodierungen - 14.07</b>	<b>43</b>
22.1 Prozeduren zur Huffman-Codierung . . . . .	44
<b>23 Fortsetzung Codierung mit Bäumen - 16.07</b>	<b>46</b>
23.1 Erstellung eines Huffman-Tree für einen gegebenen Text txt . . . . .	46
<b>24 DMdA-fortgeschritten - 16.7.</b>	<b>47</b>
24.1 Quote . . . . .	47
24.1.1 Beispiele . . . . .	47
24.1.2 Listennotation in Programmen: . . . . .	47
24.2 Symbole . . . . .	47
24.3 Operationen: . . . . .	47
24.4 Repräsentation und Auswertung arithmetischer Ausdrücke . . . . .	48
<b>25 Der <math>\lambda</math>-Kalkül - 21.7.</b>	<b>50</b>
25.0.1 Syntax des $\lambda$ -Kalküls . . . . .	50
25.0.2 Abkürzungen . . . . .	50
25.1 Freie/Gebundene Variablen . . . . .	50
<b>26 <math>\lambda</math>-Kalkül Fortsetzung - 23.07</b>	<b>52</b>
26.1 Definition $\beta$ -Reduktion . . . . .	53
26.1.1 Beispiele: $\beta$ -Reduktion . . . . .	53

# 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
$44-2$	<code>(- 44 2)</code>
$f(x,y)$	<code>(f x y)</code>
$\sqrt{81}$	<code>(sqrt 81)</code>
$9^2$	<code>(expt 9 2)</code>
$3!$	<code>(! 3)</code>

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

$$(\text{define } \langle id \rangle \langle e \rangle) \text{ } (\langle id \rangle \text{ Identifier, } \langle e \rangle \text{ Expression})$$

Erlaubt konsistente Wiederverwendung und dient der Selbstdokumentation von Programmen.

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

Namen können in Scheme 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.:  $\text{euro} \rightarrow \text{us\$}$

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:

$$(\text{lambda } (\langle p1 \rangle \langle p2 \rangle \dots) \langle e \rangle), \langle e \rangle \text{ Rumpf}$$

$\langle e \rangle$  enthält Vorkommen der Parameter  $\langle p1 \rangle, \langle p2 \rangle \dots$

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

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

$$(\text{lambda } (\text{days}) (* \text{ days } (* 155 \text{ min-in-a-day}))) \rightsquigarrow (* 365 (* 155 \text{ min-in-a-day})) \rightsquigarrow 81468000$$

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  $p_1, p_2, \dots, p_n$  als auch den Ergebniswert der Prozedur:

$$(< signaturp_1 > \dots < signaturp_n > - > < signatur - ergebnis >)$$

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 Ausdruck  $< e_1 >$  aus und teste, ob der erhaltene Wert der Erwartung (= 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.: Zeichnen Ziffernblatt (Stunden- und Minutenzeiger) zur Uhrzeit H:m auf einer analogen 24h-Uhr

- Minutenzeiger legt  $360^\circ/60$  pro Minute zurück ( $360/60 * m$ )
- Stundenzeiger legt  $360^\circ/12$  pro Stunde zurück ( $360/12 * h + 360/12 * m/60$ )

# Chapter 4

## Substitutionsmodell, Fallunterscheidung - 23.04

Reduktionsregeln für Scheme (Fallunterscheidung je nach Ausdrucksart)

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

- Lambda-Abstraktion

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

- 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

$(+ \ 40 \ 2)$

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

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

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

$\text{eval}_{lit} (2)$ •

(position-minute-hand 30)

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

$\rightsquigarrow (* \ \text{degrees-per-minute } 30)$

$\rightsquigarrow (* \ \text{degrees-per-minute } 30)$

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

Bezeichnen  $(\text{lambda } (x) \ (* \ x \ x))$  und  $(\text{lambda } (r) \ (* \ r \ r))$  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

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

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

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

Binäre Fallunterscheidung: if

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

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

# Chapter 5

## One-of Signatur - 28.04.15

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

$$(\text{one-of } \langle e_1 \rangle \dots \langle e_n \rangle)$$

Reduktion von if:

$$(\text{if } t_1 \ e_1 \ e_2) \rightsquigarrow \begin{cases} \langle e_1 \rangle, & \text{falls } t_1' = \#t ; e_2 \text{ wird niemals ausgewertet} \\ \langle e_2 \rangle, & \text{sonst; } e_1 \text{ wird niemals ausgewertet} \end{cases}$$

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

Spezialform Fallunterscheidung (conditional expression):

$$(\text{cond } (\langle t_1 \rangle \langle e_1 \rangle) \dots (\langle t_n \rangle \langle e_n \rangle) (\text{else } \langle e_{n+1} \rangle)) (\text{else optional})$$

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

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

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

$$(\text{cond } (\langle t_1 \rangle \langle e_1 \rangle) (\langle t_2 \rangle \langle e_2 \rangle) \dots) \rightsquigarrow \begin{cases} \langle e_1 \rangle, & \text{falls } t_1' = \#f \\ (\text{cond } (\langle t_2 \rangle \langle e_2 \rangle) \dots), & \text{sonst} \end{cases}$$

Reduziere  $t_1$ , erhalte  $t_1'$ .

$(\text{cond } ) \rightsquigarrow$  Fehlermeldung "Alle Tests..."

$(\text{cond } (\text{else } \langle e_{n+1} \rangle)) \rightsquigarrow e_{n+1}$

cond ist "systematischer Zucker"

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

$(\text{cond } (\langle t_1 \rangle \langle e_1 \rangle) (\langle t_1 \rangle \langle e_1 \rangle) \dots))$  entspricht  $(\text{if } \langle t_1 \rangle \langle e_1 \rangle (\text{if } \langle t_1 \rangle \langle e_1 \rangle (\text{if } \dots)))$

Spezialformen 'and' und 'or':

$(\text{or } \langle t_1 \rangle \langle t_2 \rangle \dots \langle t_n \rangle)$  entspricht  $(\text{if } \langle t_1 \rangle \#t (\text{or } \langle t_2 \rangle \dots))$

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

$(\text{and } \langle t_1 \rangle \dots \langle t_n \rangle) \rightsquigarrow (\text{if } \langle t_1 \rangle (\text{and } \langle t_2 \rangle \dots \langle t_n \rangle) \#f)$

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

# 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 jeweils eine Komponenten des Records)

```
(define-records-procedures < t >
  make-< t > ; Konstruktor
  < t >?
  (< t > - < comp1 > ... < t > - < compn > 4)) ; Liste der n-Selektoren
```

Verträge des Konstruktors / der Selektoren für Record-Signatur  $\langle t \rangle$  mit n Komponenten namens  $\langle comp_1 \rangle \dots \langle comp_n \rangle$ :

- $(: \text{make-}\langle t \rangle (\langle t_1 \rangle \dots \langle t_n \rangle \rightarrow \langle t \rangle))$
- $(: \langle t \rangle - \langle comp_1 \rangle (\langle t \rangle \rightarrow \langle t_1 \rangle))$
- ...
- $(: \langle t \rangle - \langle comp_n \rangle (\langle t \rangle \rightarrow \langle t_n \rangle))$

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

$(\text{character-name} (\text{make-character } n \ j \ f)) = n$

(analog für den Rest)

Interaktion von Funktionen (algebraische Eigenschaften).

Spezialform check-property:

```
(check-property
  (for-all ((<id1> <sig1> ... <idn> <sign>))
    <e>))
```

$\langle e \rangle$  bezieht sich auf  $\langle id_1 \rangle \dots \langle id_n \rangle$ .

Test erfolgreich, falls  $\langle e \rangle$  für bel. gewählte Bindungen für  $\langle id_1 \rangle \dots \langle id_n \rangle$  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 ((x1 natural)
            (x2 natural))
    (\geq (+ x1 x2) (max x1 x2))))
```

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) ... ))
```

Konstruktion 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
    - \* Name
    - \* jedi?
    - \* force

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

Einführung eines neuen Signaturnamens  $\langle new - t \rangle$  für die Signatur  $\langle t \rangle$ :

$(\text{define } \langle new - t \rangle (\text{signatur } \langle t \rangle))$

Bsp.:

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

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

# Chapter 8

## Gemischte Daten - 07.05.15

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

$$(: \text{ geocoder } (\text{string} \rightarrow (\text{mixed } \text{geocode } \text{geocode-error})))$$

Ein Geocode besteht aus:

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

$$(: \text{ geocode-address } (\text{geocode} \rightarrow \text{string})) \dots$$

Ein geocode-error besteht aus:

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

Teachpack: geocoder.rkt

Gemischte Daten Die Signatur

$$(\text{mixed } < t_1 > \dots < t_n >)$$

ist gültig für jeden Wert, der mindestens eine der Signatur  $< t_1 > \dots < t_n >$  erfüllt.

Beispiel: Datendefinition:

Eine Antwort des Geocoders ist entweder

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

Beispiel (eingebaute Funktion  $\text{string} \rightarrow \text{number}$ ):

$$(: \text{ string} \rightarrow \text{number } (\text{string} \rightarrow (\text{mixed } \text{number } (\text{one of } \#f))))$$
$$(\text{string} \rightarrow \text{number } "42") \rightsquigarrow 42$$

$(\text{string} \rightarrow \text{number } \text{"foo"}) \rightsquigarrow \#f$

Erinnerung:

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

$(: < t >? (\text{any} \rightarrow \text{boolean}))$

Auch Prädikate für eingebaute Signaturen:

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

Prozeduren die gemischte Daten der Signaturen  $< t_1 > \dots < t_n >$  konsumieren:

Konstruktionsanleitung:

$(: < f > ((\text{mixed } < t_1 > \dots < t_n > \rightarrow \dots))$

$(\text{define } < f > (\text{lambda } (x) (\text{cond } ((< t_1 >? x) \dots) \dots (< t_n >? x) \dots)))$

Mittels let lassen sich Werte an lokale Namen binden:

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

Die Ausdrücke  $< e_1 > \dots < e_n >$  werden parallel ausgewertet  $\rightarrow < id_1 > \dots < id_n >$  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.

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

$\leftrightarrow$

$((\text{lambda } (< id_1 > \dots < id_n >) < e >) < e_1 > \dots < e_n >)$



# Chapter 9

## Parametrisch polymorphe Funktionen - 12.05.1

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

$$\text{dist}(l_1, l_2) = \text{Erdradius in km} \cdot \text{acos}(\cos(l_1.\text{lat}) \cdot \cos(l_1.\text{lng}) \cdot \cos(l_2.\text{lat}) \cdot \cos(l_2.\text{lng}) + \cos(l_1.\text{lat}) \cdot \sin(l_1.\text{lng}) \cdot \cos(l_2.\text{lat}) \cdot \sin(l_2.\text{lng}) + \sin(l_1.\text{lat}) \cdot \sin(l_2.\text{lat}))$$

### Parametrisch polymorphe Funktionen

Beobachtung: Manche Prozeduren arbeiten unabhängig von den Signaturen ihrer Argumente: parametrisch polymorphe Prozeduren (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 x2) (cond ((= i 1) x1) ((= i 2) x2))))

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

#### Beispiel:

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

(number string boolean → string)

(number boolean natural → boolean)

(number number number → number)

; 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  $\rightarrow$  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  $\lambda l$ , 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 Länge von `xs` bestimmt die Anzahl der rekursiven Aufrufe von `cat`
- Auf `ys` werden niemals Selektoren angewandt

# Chapter 12

## Rekursion auf Listen - 21.05.15

Generierung aller natürlichen Zahlen (vgl. gemischte Daten). Eine natürliche Zahl (`natural`) 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`):

$(\Rightarrow \langle p \rangle \langle t \rangle)$

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

### Beispiel:

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

$0! = 1$

$n! = n \cdot (n-1)!$

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  $n > 0 \Rightarrow \text{pred}$  anwendbar
- $(\langle f \rangle (-n\ 1))$  hat die Signatur  $\langle t \rangle$

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 möglich, 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 -> natural))

(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".

Kontext 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:  $1000 + 999 + 998 + \dots + 1$

⇒ 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  $e_1, \dots, e_n$  und  $e$  dürfen sich auf die Namen von  $<id_1> \dots <id_n>$  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)  $\forall n \in \mathbb{N} : \text{succ}(n) \in \mathbb{N}$  (Nachfolger)
- (P3)  $\forall n \in \mathbb{N} : \text{succ}(n) \neq 0$
- (P4)  $\forall m, n \in \mathbb{N} : \text{succ}(m) = \text{succ}(n) \Rightarrow m = n$

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

- (P5) Induktionsaxiom

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

- $\mathbb{N}$  enthält nicht mehr als die durch 0 und die durch  $\text{succ}()$  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  $\forall n(n \in M \Rightarrow \text{succ}(n) \in M)$

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? erfüllen

(: 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  $c = (\text{lambda } (x \ r) \ (+ \ 1 \ r))$

# 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:  $(\text{compose } f \ g) \equiv f \circ g$ . '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):

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

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

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

(define repeat
  (lambda (n f)
    (cond ((= 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)

$f(t) = \langle x(t), y(t) \rangle$  siehe rkt-Datei

Reduktion:

$((\text{add } 1) \ 21)$

$\text{eval}_i d \rightsquigarrow (((\text{lambda } (x) (\text{lambda } (y) (+ x y))) \ 1) \ 21)$

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

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

# Chapter 17

## Currying (Haskell B. Curry) 25.6.

Erstmals implementiert 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.

$(\%a \%b \rightarrow \%c) \rightarrow$  Applikation auf 2 Argumente (Signaturen  $\%a \%b \rightarrow \%c$ )

$\text{curry} \uparrow \downarrow \text{uncurry} = (\%a (\%b \rightarrow \%c)) \rightarrow$  Applikation auf 1 Argument (Signaturen  $\%a \rightarrow (\%b \%c) \rightarrow$  Applikation auf 1 Argument (Signatur  $\%b \rightarrow \%c$ )

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:

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

- Operator `'` (Ableitung) konsumiert Funktion f und produziert Funktion f'  $\Rightarrow$  `'` ist higher-order!

```
(: diffquot (h f -> f')  
(: diffquot (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: ( $\chi_S: M \rightarrow \text{boolean}$ )

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

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

$$\chi_s(m) = \#f$$

$$\chi_s(s) = \#t$$

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

Element  $x$  in Menge S einfügen:

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

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



# Chapter 18

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

Konvertiere Liste xs in eine Menge gleicher Elemente:

```
;      xs
;  +---+---+
;  |   |   |
;  +---+---+
;  /       \
; x1      +---+---+                xs
;          |   |   |                \
;          +---+---+      (foldr empty-set set-inset))
;          /       \
;      x2          .
;                  \
;                  +---+---+
;                  |   |   |
;                  +---+---+
;                  /       \
;                  xn      empty
```

$\rightsquigarrow$  (fold empty-set set-in-set xs)) - empty  $\rightarrow$  empty-set - make-pair  $\rightarrow$  set -inset

```
;      set-inset
;      /       \
;  xn      set-inset
;      /       \
;                  .
;                  .
;                  \
;                  set-inset
;                  /       \
;                  xn      empty-set
```

### 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-Element (daher auch: lazy list)

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

->

;   +-----+
;   | x2 |   tail |
;   +-----+
```

- Vergleich:

<pre>;   (list-of %a) ;   xs ;   +---+ ;         ;   +---+ ;   /   \ ; x1     +---+ ;               ;         +---+ ;         /   \ ;      x2     . ;             \ ;             +---+ ;                   ;             +---+ ;             /   \ ;          xn     empty</pre>	<pre>(stream-of %a) +-----+   xs       +-----+ /       \ xn       +---+                2           +---+          / x2</pre>
---	--

**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  $(\text{delay } e) \equiv (\text{lambda } () \text{ e \{nicht ausgewertet\}})$
- `(force p)`: Erzwingt Auswertung des Promise `p`, liefert Wert zurück  
 $(\text{: force } ((\rightarrow \%a) \rightarrow \%a))$

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

# Chapter 19

## Stream von fib - 2.7.

Generiere den unendlichen Strom fibs der Fibonacci-Zahlen

$\text{fib}(0) = 1; \text{fib}(1) = 1; \text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$

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

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:  $\square$   
(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):

```

;    +---+
;    | 1| Wurzel
;    +---+
;    /      \
; | _|      +---+
;           |  |      innerer Knoten
;           +---+
;           /      \
;         | _|      .
;                   \
;                   +---+
;                   |  | Blatt
;                   +---+
;                   /      \
;                   | _|      | _|

```

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:

$(\text{btree-depth empty tree}) = 0$

$(\text{btree-depth } t_1) = 3$

$(\text{btree-depth } t_2) = 2$

### 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)))))))

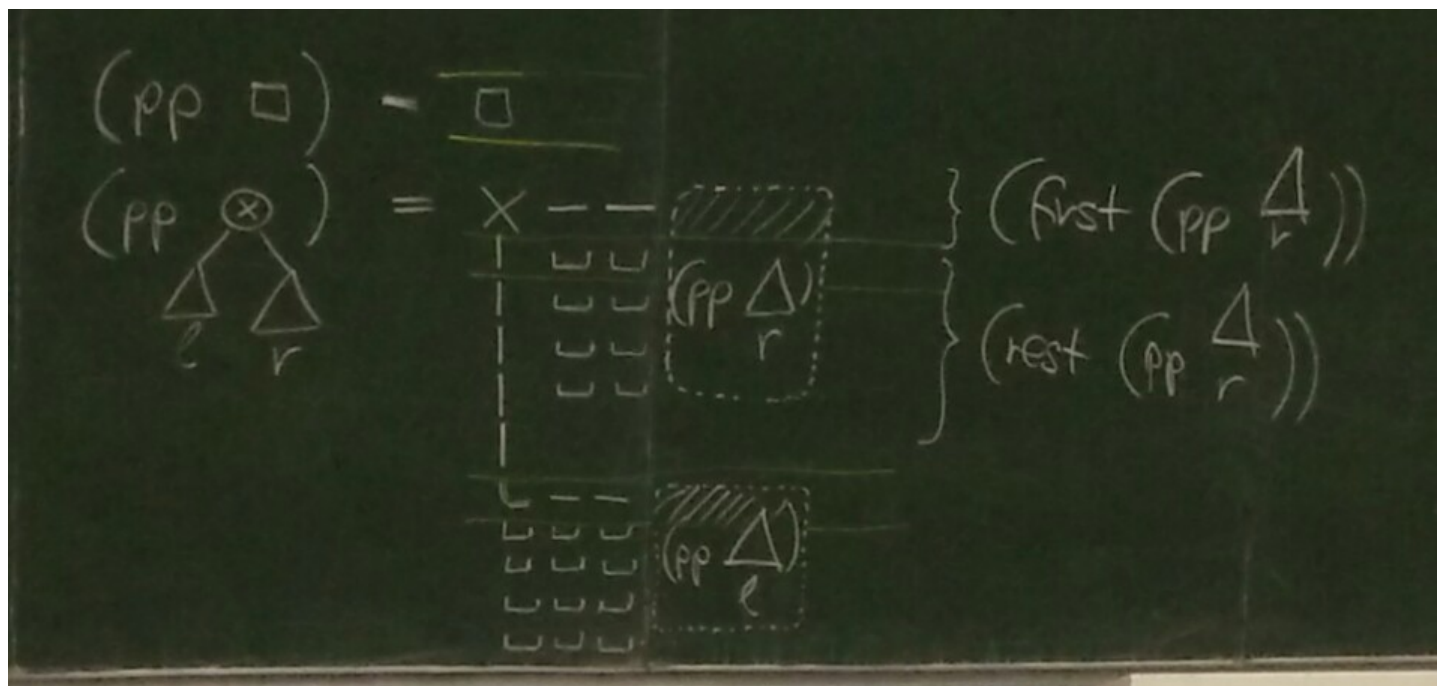
```

# 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  $\rightarrow$  string)

### 20.2 Induktion über Binärbäume

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

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

Falls  $P(\text{empty-tree})$  [Induktionsbasis]

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

$P(l) \wedge P(r) \Rightarrow P((\text{make-node } l \times r))$

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"):

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

Erstes  $\leq$  trivial. Zweites:

Induktionsbasis  $P(\text{empty-tree})$ :

`(size empty-tree)`

$\rightsquigarrow [\text{depth}] 0$

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

Induktionsschritt:  $(P(l) \wedge P(r) \Rightarrow P((\text{make-node } l \times r)))$

`(size (make-node l x r))`

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

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

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

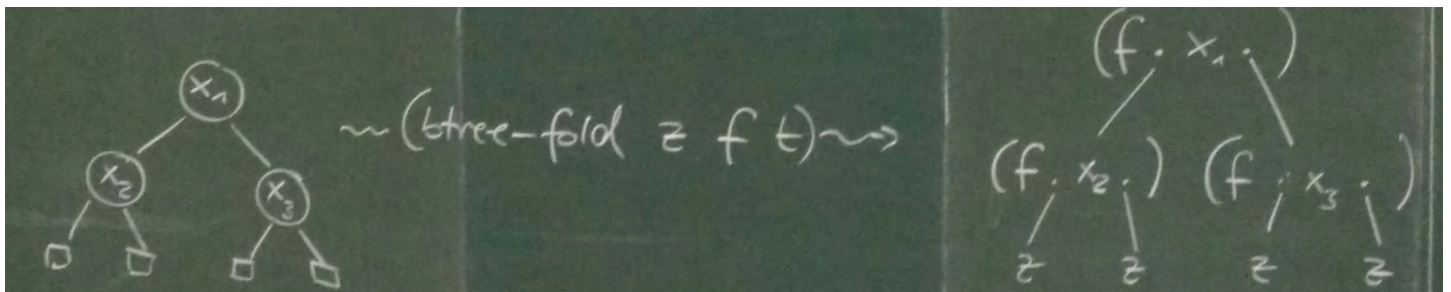
$$\leq 2 \times \max(2^{(\text{depth } l)} + 2^{(\text{depth } r)} - 1)$$

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

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

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

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



opku9zfswdefrtgzhuji kolpölokijuzhtgrffgtzhuji kolpkijuzhtgrftgzhuji klpö [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 (node-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).

$(\text{leftmost } (x) \square) = (\text{list } x)$

$(\text{leftmost } (x) \Delta) = (\text{leftmost } \Delta)$

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 (leftmost t1) (list 1))
(check-expect (leftmost 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:  $f \circ f^{-1} = \text{id}$

```
(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  $n = (\text{make-node } L \ x \ r)$  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)

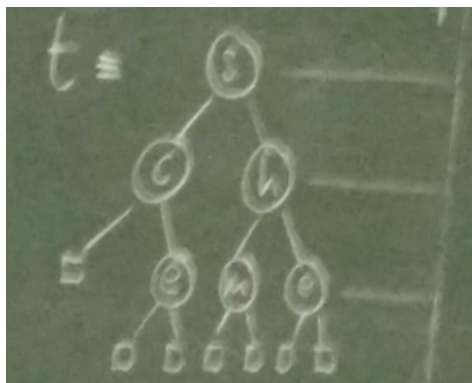
(c) postorder traversal (xs1 xs2 x).

```
(: 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-first-traversal) eines Baumes  $t$  sammelt die Markierungen der Knoten ebenenweise von der Wurzel ausgehend auf.



$(\text{levelorder } t) \rightsquigarrow (\text{list } "s" "c" "h" "e" "m" "e")$

Idee: Gegeben sei eine Liste  $ts$  von Bäumen.

1. Sammle die Liste der Markierungen der Wurzeln der (nicht-leeren) Bäume in  $ts$  auf  $(= (\text{roots } ts))$
2. Bestimme  $ts'$  der nicht-leeren Teilbäume der Bäume in  $ts$   $(= (\text{subtrees } ts))$
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. Rekursion: ( (e) (m) (e) )

roots: "e" "m" "e" subtrees: ( )

# Chapter 22

## Zeichencodierungen - 14.07

bilden Zeichen auf Sequenzen von Bits ab. Derzeit sind Codes fixer Länge weit verbreitet.

- ASCII (Codes 0-127, 7 Bit, American Standard Code for Information Interchange)
- ISO-8859-1 (Codes 0-255, 8 Bit, 191 lateinische + Steuerzeichen)
- Unicode (20 Bit, codiert Zeichen 129 aktueller/historischer Sprachen, inklusive Klingon)  
Beispiel: Zeichen '€': 0000 0010 0000 1010 1100 ("020AC")

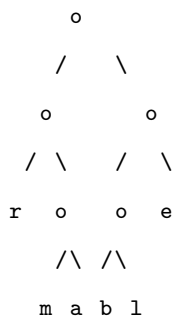
Huffman-Codes nutzen Bitsequenzen variabler Länge.

Idee: Zeichen mit hoher Frequenz werden mit weniger Bits codiert, als seltene Zeichen.

⇒ Datenkompression

Huffman-Codes sind Binärbäume mit markierten Blättern (Knoten haben keine Codierung).

Beispiel: Humman-Code für "erdbeermarmelade":



Code für Zeichen x Pfad von Wurzel bis Blatt mit Markierung x.

- Abstieg in linken Teilbaum Bit 0
- Abstieg in rechten Teilbaum Bit 1

Zeichen	Frequenz	Code
e	5	11
r	3	00
m	2	010
a	2	011
d	2	101
b	1	1000
l	1	1001

Huffman-Codes sind präfixfrei.: alle Bits eines bestimmten Zeichens sind niemals Präfix eines anderen Zeichens  $\Rightarrow$  eindeutige Decodierung.

11|00|101|1000|11|11|...  $\Rightarrow$  e r d b e e ...

Länge 42 Bit, Unicode 320 Bit

Einsatz in JPEG, MP3, ZIP, ...

## 22.1 Prozeduren zur Huffman-Codierung

1. Decodieren einer Bitsequenz ( $(: \text{ huff-decode } ((\text{huff-tree-of string}) (\text{list-of bit}) \rightarrow \text{string}))$ )
2. Codieren eines Strings: ( $(: \text{ huff-encode } ((\text{huff-tree-of string}) \text{ string} \rightarrow (\text{list-of bit})))$ )  
 $\forall \text{ string } s: (\text{huff-decode ht } (\text{huff-encode ht } s)) = s$
3. Huffman-Tree für gegebenen Text erstellen: ( $(: \text{ huffman-code } (\text{string} \rightarrow (\text{huff-tree-of string}))$ )

Decodieren eines huffman-codierten Strings (= Code von 0/1-Bits)

Plan: Baue

`(: decode ( (huff-tree-of %a) (list-of bit) -> (list-of %a)))`

1.  $(\text{decode ht } \square[\dots \text{bits} \dots]) = (\text{make-pair } x (\text{decode ht ht } [\dots \text{ bits } \dots]))$
2.  $(\text{decode ht } \Delta[]) = \text{empty}$
- 3.a)  $(\text{decode ht baum } [0 \text{ bits } \dots]) = (\text{decode ht linker-teilbaum } [\dots \text{ bits } \dots])$
- 3.b)  $(\text{decode ht baum } [1 \text{ bits } \dots]) = (\text{decode ht rechter-teilbaum } [\dots \text{ bits } \dots])$

??? Neueinstieg an Wurzel des Huffman-Tree  $\Rightarrow$  Wurzel des Huffman-Trees als Parameter durchführen.

Huffman-Codierung eines Strings s als Liste von 0/1-Bits

Plan:

- (a) Codierung eines Zeichens c: Suche c mittels einer Tiefensuche von der Wurzel des Huffman-Tree aus. Protokolliere den Pfad beim Abstieg als Liste von 0/1-Bits.

Frage: Wie reagieren, wenn die Tiefensuche zu einem Blatt  $\neq c$  führt?

Idee: Verfolge an innerem Knoten  $(\text{make-huff-node } l \ r)$  jeweils linken UND rechten Teilbaum. Suche nach c schlägt fehl in l oder in r. Bei Fehlschlag liefere die leere Bitliste. Beachte:

$(\text{append empty } xs) = xs$

$(\text{append } xs \text{ empty}) = xs$

`(: encode ( (huff-tree-of %a) (list-of bit) %a -> (list-of bit)))`

bisheriges Protokoll	Zeichen c

- 1)  $(\text{encode } \square x \text{ } [\dots \text{ bits } \dots] \text{ } c) = \text{empty} ; x \neq c$
  - 2)  $(\text{encode } \square c \text{ } [\dots \text{ bits } \dots] \text{ } c) = (\text{reverse } [\dots \text{ bits } \dots])$
  - 3)  $(\text{encode baum } [\dots \text{ bits } \dots] \text{ } c) = (\text{append } (\text{encode l-baum } [0 \text{ bits } \dots] \text{ } c) (\text{encode r-baum } [1 \text{ bits } \dots] \text{ } c))$
- (b) Codiere Zeichen oder Strings  $s$  mit  $\text{encode } (\text{map})$ , verbinde die einzelnen Bit-Listen mit  $\text{flatten } (\text{concat})$

# Chapter 23

## Fortsetzung Codierung mit Bäumen - 16.07

(huff-encode  $\Delta_{ht} s$ )  $\rightsquigarrow$  0111 0100...

(huff-decode  $\Delta_{ht}$  0111 0100...)  $\rightsquigarrow$  s

### 23.1 Erstellung eines Huffman-Tree für einen gegebenen Text txt

Plan:

(H1) Stelle Häufigkeit des Vorkommens jedes Zeichens in txt fest. Sortiere in Reihenfolge steigender Häufigkeit. (occurences)  
Beispiel (occurences "erdbeermarmelade")  $\rightsquigarrow$  (list <"l",1>, <"b",1>, ... <"e",5>) Vorkommen <i,n> (occur): Ding  
i (item) kommt mit Häufigkeit n (freq) vor.

(H2) Baue Huffman-Tree von den Blättern her auf. Initialisiere den Aufbau für Vorkommen <i,n> konstruiere < $\square_i, n$ >.

(H3) Die beiden Huffman-Trees, die die seltensten Zeichen repräsentieren, stehen am Anfang der Liste.

(list < $\square_i, n$ >, < $\square_j, m$ >...)

Konstruktion des Huffman-Trees, die diese Invariante bewahrt:

Iteration: Wiederhole, bis Liste Länge 1 hat

(a) Fasse Vorkommen < $\Delta_l, n$ > und < $\Delta_r, m$ > zu einem Vorkommen: < $\Delta_{l-O-\Delta_r}, n+m$ >

(b) Sortiere dieses Vorkommen bzgl. Häufigkeit  $n+m$  in die Restliste ein.

(H4) Baum ht in (list < $\Delta_{ht}, n$ >) ist der gesuchte Huffman-Tree.

# Chapter 24

## DMdA-fortgeschritten - 16.7.

- Neues Ausgabeformat in REPL:

`(list x1 ... xn) → (x1 ... xn)`

`empty → ()`

- Neuer (struktureller) Gleichheitstest für Werte aller (auch benutzer-definierter) Signaturen:

`(: equal? (%a %b -> boolean))`

### 24.1 Quote

Sei  $e$  ein beliebiger Scheme-Ausdruck. Dann liefert `(quote e)` die Repräsentation von  $e$  -  $e$  wird nicht ausgewertet.

#### 24.1.1 Beispiele

`(quote 42) ↪ 42`

`(quote "U Tü") ↪ "U Tü"`

Konstante (Literele) repräsentieren sich selbst

`(quote (+ 40 2)) ↪ (+ 40 2)` Funktionsapplikation repräsentiert als Liste

Abkürzung `(quote e) ≡ 'e`

#### 24.1.2 Listennotation in Programmen:

`(list x1 ... xn) ≡ '(x1 .. xn)`

`empty ≡ '()`

### 24.2 Symbole

Was ist `(first '(* 1 2))`? Was sind `lambda`, `x`, `+`, in `'(lambda (x) (+ x 1))`?

→ Neue Signatur symbol zur Repräsentation von Namen in Programmen. Effiziente interne Darstellung (KEINE STRINGS), effizient vergleichbar. Kein Zugriff auf einzelne Zeichen des Symbols.

### 24.3 Operationen:

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

## 24.4 Repräsentation und Auswertung arithmetischer Ausdrücke

list

Beispiel:  $e \equiv \overbrace{(* (! (+ 1 2)) x)}$

```
(define arith
  (signature (mixed number          ;Konstanten
              symbol                ;Variablen
              (list-of arith))) ; zusammengesetzte Ausdruecke
```

Auswertung möglich, wenn Bindung für Symbole (Variablen und Operatoren) an Werte gegeben. Dictionary (Environment):

d1: { x → 3,

\* →<procedure: \* > ,

+ →<procedure: + > ,

! → fac }

⇒ e  $\rightsquigarrow$  18

d2: { x → 1,

\* →<procedure: \* > ,

+ →<procedure: + > ,

! → (lambda (x) (- x)) }

⇒ e  $\rightsquigarrow$  -3

Auswertung eines arithmetischen Ausdrucks e (unter Dictionary d):

((eval d) e)

konfigurierter Auswerter

(E1) ((eval d) c) = c ; c Konstante

(E2) ((eval { $x_1 \rightarrow v_1, \dots, x_n \rightarrow v_n$ } ) $x_i$ ) =  $v_i$ , ; Xi Variable

(E3) ((eval d) ( $e_1 e_2 \dots e_n$ )) = (... (((eval d)  $e_1$ ) ((eval d)  $e_2$ )) ... ((eval d)  $e_n$ ))

Graphisch: ( $e_1 e_2$ )  $\equiv e_1 @ e_2$

```
(: eval ((dict-of symbol %a) -> (arith -> %b))
(define eval
  (lambda (d)
    (lambda (e)
```



```
(cond ((constant? e) e)
      ((variable? e) (lookup-dict e d))
      ((compound? e)
       (let (es (map (eval d) e)))
         (foldl (first es) (lambda (f x) (f x))
                  (rest es))))))
```

# Chapter 25

## Der $\lambda$ -Kalkül - 21.7.

Der  $\lambda$ -Kalkül ist eine Notation für beliebige (berechenbare) Funktionen. Entwickelt in den 1930er Jahren von Alonzo Church (\* 1903, †1995) als neue Grundlage für die Mathematik. Seither verwendet als theoretischer Unterbau von Programmiersprachen.

### 25.0.1 Syntax des $\lambda$ -Kalküls

Die Menge der Ausdrücke (expression)  $E$  des  $\lambda$ -Kalküls ist induktiv definiert. Sei  $V$  eine unendliche Menge von Variablennamen.

- $\forall v \in V : v \in E$  [Variablen]
- $\forall e_1, e_2 \in E : (e_1 \ e_2) \in E$  [Applikation] ( $e_1$  Funktion,  $e_2$  Argument)
- $\forall v \in V, e_1 \in E : (\lambda v. \ e_1) \in E$  [Abstraktion] ( $v$  Parameter,  $e_1$  Body)

Beispiel:

$y \in E$ ,  $(\lambda y. \ y) \in E$  Identitätsfunktion,  $(\lambda y. \ z) \in E$  Funktion ignoriert  $y$  liefert  $z$ ,  $((f \ x) \ y) \in E$  Currying,  $(\lambda f. (f \ x)) \in E$  Anwendung von  $f$  auf  $x$  (H.O.P.)

### 25.0.2 Abkürzungen

$(\dots ((e_1 \ e_2) e_3) \dots e_n) \equiv (e_1 e_2 \dots e_n)$

$(f \ x \ y) \equiv ((f \ x) \ y)$

## 25.1 Freie/Gebundene Variablen

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

- wird der hier nicht bekannte Wert von 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  $\lambda x$  als Parameter) gebunden, während

- Variablen  $f, y, z$  frei sind.

Welche Variablen eines Ausdrucks sind frei/gebunden?

$$\text{free}(v) = \{v\}$$

$$\text{free}((e_1 e_2)) = \text{free}(e_1) \cup \text{free}(e_2)$$

$$\text{free}((\lambda v . e_1)) = \text{free}(e_1) \setminus \{v\}$$

$$\text{bound}(v) = \{\}$$

$$\text{bound}((e_1 e_2)) = \text{bound}(e_1) \cup \text{bound}(e_2)$$

$$\text{bound}((\lambda v . e_1)) = \text{bound}(e_1) \cup \{v\}$$

### Beispiel:

$$\text{free}(((\lambda x . ((f x) y)) z)) = \{f, y, z\} \text{ NACHRECHNEN}$$

$$\text{bound}(((\lambda x . ((f x) y)) z)) = \{x\}$$

Achtung:

Bindung/Freiheit muss für jedes Vorkommen separat entschieden werden.

$$E_2 \equiv (x_1 (\lambda x . x_2))$$

$$\text{free}(E_2) = \{x_1\}$$

$$\text{bound}(E_2) = \{x_2\}$$

# Chapter 26

## $\lambda$ -Kalkül Fortsetzung - 23.07

Die Anwendung einer Funktion (=  $\lambda$ -Abstraktion) auf ein Argument wird im  $\lambda$ -Kalkül durch  $\beta$ -Reduktion ( $\overset{\rightarrow}{\beta}$ ) beschrieben.

Die Applikation  $((\lambda v . e_1) e_2)$  wird durchgeführt, indem

1. eine Kopie von Rumpf  $e_1$  hergestellt wird und
2. alle freien Vorkommen von  $v$  in  $e_1$  durch  $e_2$  ersetzt werden.

Beispiel:

$$\begin{aligned} & ((\lambda x. (* ((\lambda x. (+ x a)) b) x)) c) \text{ [x ist gebunden]} \\ & \overset{\rightarrow}{\beta} (* ((\lambda x. (+ x a)) b) \mathbf{c}) \\ & \overset{\rightarrow}{\beta} (* (+ b a) c) \end{aligned}$$

Beispiel II:

$$\begin{aligned} & ((\lambda x . \text{foo}) \text{bar}) \\ & \overset{\rightarrow}{\beta} \text{foo} \end{aligned}$$

Beispiel III:

$$\begin{aligned} & ((\lambda x . ((\lambda y. (* x y)) a)) b) \\ & \overset{\rightarrow}{\beta} ((\lambda y. (* b y)) a) \\ & \overset{\rightarrow}{\beta} (* b a) \end{aligned}$$

Beispiel IV:

$$\begin{aligned} & ((\lambda f . (f c)) (\lambda x . (+ x a))) \\ & \overset{\rightarrow}{\beta} ((\lambda x . (+ x a)) c) \\ & \overset{\rightarrow}{\beta} (+ c a) \end{aligned}$$

Beispiel V:

$$\begin{aligned} & (((\lambda x . (\lambda y . x)) (\lambda z . y) \text{foo}) a) \\ & \lambda x \text{ ignoriert zweites Argument } \lambda z \text{ ignoriere Arg, liefert } y \\ & \overset{\rightarrow}{\beta} (((\lambda y . ((\lambda z . y)) \text{foo}) a) \text{ [y ist jetzt gebunden (Variable capturing)]}) \\ & \overset{\rightarrow}{\beta} ((\lambda z . \text{foo}) a) \\ & \overset{\rightarrow}{\beta} \text{foo} \end{aligned}$$

Problem: freies  $y$  "wandert" unter das  $\lambda y$  und wird damit gebunden.  $\therefore$ -(

## 26.1 Definition $\beta$ -Reduktion

$$((\lambda x . e) a) \xrightarrow{\beta} e [x \rightarrow a]$$

”ersetze freie Vorkommen von x in e durch a”

$$(\rightarrow 1) \quad x [x \rightarrow a] = a$$

$$(\rightarrow 2) \quad v [x \rightarrow a] = v$$

$$(\rightarrow 3) \quad (e_1 e_2) [x \rightarrow a] = (e_1 [x \rightarrow a] e_2 [x \rightarrow a])$$

$$(\rightarrow 4) \quad (\lambda x . e) [x \rightarrow a] = (\lambda x . e)$$

$$(\rightarrow 5) \quad (\lambda v . e) [x \rightarrow a] = (\lambda v . e [x \rightarrow a]) \text{ wenn } v \text{ nicht frei in } a \text{ ist}$$

$$(\rightarrow 6) \quad (\lambda z . e [v \rightarrow z])[x \rightarrow a] \text{ sonst } (z : \text{neuer Name})$$

### 26.1.1 Beispiele: $\beta$ -Reduktion

$$(\underline{((\lambda x . (\lambda y . x)) (\lambda z . y) \text{foo}) a}) [\text{Redex, Reductional Expression}]$$

$$\xrightarrow{\beta} (((\lambda y . x) [x \rightarrow (\lambda z . y)] \text{foo}) a)$$

$$(\rightarrow 6) \quad ((\lambda y' . x[y \rightarrow y']) [x \rightarrow (\lambda z . y)] \text{foo}) a)$$

$$(\rightarrow 2) \quad ((\lambda y' . x) [x \rightarrow (\lambda z . y)] \text{foo}) a)$$

$$(\rightarrow 5) \quad ((\lambda y' . x [x \rightarrow (\lambda z . y)]) \text{foo}) a)$$

$$(\rightarrow 1) \quad (\underline{((\lambda y' . (\lambda z . y)) \text{foo}) a})$$

$$\xrightarrow{\beta} ((\lambda z.y) [y' \rightarrow \text{foo}] a)$$

$$(\rightarrow 5) \quad ((\lambda z.y [y' \rightarrow \text{foo}]) a)$$

$$(\rightarrow 2) \quad \underline{((\lambda z.y) a)}$$

$$\xrightarrow{\beta} y[z \rightarrow a]$$

$$(\rightarrow 2) \quad y$$

THE END