

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

# Informatik I Vorlesung

Wintersemester 2016/2017

Mitschrieb von  
Julian Wolff

Aktueller Stand 6. Dezember 2016

# Inhaltsverzeichnis

<b>1</b>	<b>Scheme: Ausdrücke, Auswertung und Abstraktion</b>	<b>3</b>
1.1	REPL . . . . .	3
1.2	Literale . . . . .	3
1.3	Zusammengesetzte Ausdrücke . . . . .	4
1.4	Identifizier . . . . .	4
1.5	Lambda-Abstraktion . . . . .	4
1.6	Kommentare . . . . .	5
1.7	Signaturen . . . . .	5
1.8	Prozedur-Signaturen . . . . .	6
1.9	Testfälle . . . . .	6
1.10	Erinnerung . . . . .	6
1.11	Top-Down-Entwurf (Programmieren mit "Wunschdenken") . . . . .	7
1.12	Reduktionsregeln für Scheme ( $\rightsquigarrow$ ) . . . . .	8
1.12.1	Einschub: Lexikalische Bindung . . . . .	9
1.13	Übliche Notation in der Mathematik: <u>Fallunterscheidung!</u> . . . . .	9
1.14	Spezialform Fallunterscheidung (conditional) . . . . .	9
1.15	Binäre Fallentscheidung: . . . . .	10
1.16	Zusammengesetzte Daten . . . . .	10
1.17	<u>Records</u> in Scheme . . . . .	11
1.18	Spezialform check-property . . . . .	11
1.18.1	Interaktion von Konstruktor und Selektor . . . . .	11
1.19	Längen/Breitengrade . . . . .	13
1.20	Signaturnamen . . . . .	13
<b>2</b>	<b>Gemischte Daten</b>	<b>13</b>
2.1	Polymorphe Signaturen . . . . .	15
2.2	Polymorphe Paare und Listen . . . . .	16
2.3	Liste . . . . .	16
2.4	Visualisierung von Listen . . . . .	17
2.5	Spines (Rückrad) . . . . .	17
2.6	Prozeduren über Listen . . . . .	18
<b>3</b>	<b>Neue Sprachebene "Macht der Abstraktion"</b>	<b>18</b>
3.1	cat . . . . .	18
3.2	Bewertungen . . . . .	19
3.3	Pattern Matching für $\langle \text{pat}_i \rangle$ . . . . .	19
3.4	Rekursion über natürliche Zahlen . . . . .	19
3.4.1	iterative Listenumkehr (backwards) . . . . .	22

3.5	letrec . . . . .	22
-----	------------------	----

# Scheme: Ausdrücke, Auswertung und Abstraktion

## REPL

Definition	DrRacket
Interaktion	REPL

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

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

Allgemein:  $\langle \text{Funktion} \rangle \langle \text{argument} \rangle$

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

(+ 40 2)  $\rightsquigarrow$  42  
 (add? 42)  $\rightsquigarrow$  #f  
Eval Eval

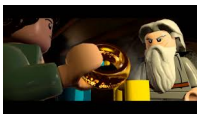
Interaktionsfenster:

Read  $\rightsquigarrow$  Eval  $\rightsquigarrow$  Print  
Loop

REPL

## Literale

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

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

## Zusammengesetzte Ausdrücke

Auswertung zusammengesetzte Ausdrücke (composite expression) in mehreren Schritten (Steps), "von innen nach außen", bis keine weitere Reduktion möglich ist:

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

Beispiel:

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

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

## Identifizier

Ein Wert kann an einen Namen (identifier) gebunden werden, durch (define <id> <expression>). Es erlaubt konsistente Wiederverwendung und dient der Selbstdokumentation von Programmen.

⚠ Achtung: Dies ist eine Spezialform und kein Ausdruck. Insbesondere besitzt diese Spezialform keinen Wert, sondern einen Effekt: der Name <id> wird durch den Wert von <expression> gebunden. Namen können in Scheme fast beliebig gewählt werden, solange

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

Beispiel: Euro  $\rightarrow$  US-\$

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

## Lambda-Abstraktion

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

$$(\text{lambda } (\langle p_1 \rangle \langle p_2 \rangle \dots) \langle \text{expr} \rangle)$$

expr ist der Rumpf und enthält Vorkommen der Parameter  $\langle p_i \rangle$ .

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

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

## Kommentare

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

Prozeduren/Funktionen sollen im Programm eine ein- bis zweizeilige Kurzbeschreibung  
 vorangestellt werden.


## Signaturen

Eine Signatur prüft, ob ein Name ⟨id⟩ an einen Wert einer angegebenen Sorte ge-  
 bunden wird. Signaturverletzungen werden protokolliert.

(: ⟨ id ⟩ ⟨signatur⟩)

Bereits eingebundene Signaturen sind:

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

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

## Prozedur-Signaturen

Prozedur-Signaturen spezifizieren Signaturen sowohl für die Parameter  $\langle p_1 \rangle, \langle p_2 \rangle, \dots$  als auch für den Ergebniswert der Prozedur:

$\parallel (:\langle \text{id} \rangle (\langle \text{signatur} - p_1 \rangle \langle \text{signatur} - p_2 \rangle \dots \rightarrow \langle \text{signatur} - \text{ergebnis} \rangle))$

Prozedur-Signaturen werden bei jeder Anwendung der Funktion  $\langle \text{id} \rangle$  auf Verletzung geprüft.

## Testfälle

Testfälle dokumentieren das erwartende Ergebnis einer Prozedur für ausgewählte Argumente:

$\parallel (\text{check-expect } \langle e_1 \rangle \backslash \text{la} \$ \backslash \text{text} \{e\} \_2 \$ \backslash \text{ra})$

Werte den Ausdruck  $\langle e_1 \rangle$  aus und teste, ob der erhaltene Wert der Erwartung (=Wert des Ausdruck  $\langle e_2 \rangle$ ) entspricht.

Einer Prozedurdefinition sollten Testfälle direkt vorangestellt werden.

$\triangle!$ , „check-expect“ ist eine Spezialform und hat daher keinen Wert. Eine Testverletzung wird als Effekt protokolliert.

## Erinnerung

Konstruktionsanleitung für Prozeduren:

- kurzbeschreibung (ein- bis zweizeiliger Kommentar mit Bezug auf Parameternamen und Ergebnis)
- Signatur  $(:\langle \text{name} \rangle (\dots \rightarrow))$
- Testfälle check-expect/ ceack-within
- Prozedurgerüst (define  $\langle \text{name} \rangle$  (lambda ( $\langle p_1 \rangle \langle p_2 \rangle$  ))
- Rumpf programmieren  $\langle \text{rumpf} \rangle$ )

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

Beispiel: Sunset auf Tatooine (SW Episode IV)

Zeichne Szene zu Zeitpunkt  $t$  ( $t=0 \dots 100$ )

- (1) Himmel verfärbt sich von blau ( $t=0$ ) zu rot ( $t=100$ )
- (2) Sonne(n) versinkt (bei  $t=100$  hinter Horizont)
- (3) Luke startt auf Horizont (bei jeden  $t$ )

Zeichne Szene von hinten nach vorne:



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

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



## Reduktionsregeln für Scheme ( $\rightsquigarrow$ )

Fallunterscheidung je nach Ausdruck:

- Literal  $l$  ( $1$ ,  $\#t$ , "Karotte", ...)  $[eval_1]$   
 $l \rightsquigarrow l$  (keine Reduktion möglich)
- Identifier  $\langle id \rangle$   $[eval_{id}]$   
 $\langle id \rangle \rightsquigarrow$  Wert, an den  $\langle id \rangle$  gebunden
- Lambda-Abstraktion  $[eval_\lambda]$   
 $(\text{lambda } (...)...) \rightsquigarrow (\text{lanmbda } (...)...)$
- Applikation ( $f\ e_1 e_2 \dots$ )

–  $f, e_1, e_2, \dots$  mittels  $\rightsquigarrow$ , erhalte  $f', e'_1, e'_2 \dots$

–	<div style="display: flex; justify-content: space-between; align-items: center; padding: 5px;"> <div style="text-align: left;"> <p>Operation auf <math>e'_1, e'_2 \dots</math> anwenden</p> </div> <div style="text-align: left;"> <p>Falls <math>f</math> primitive (eingebaute) Operation</p> </div> <div style="text-align: right;"> <p><math>[apply/prim]</math></p> </div> </div>
–	<div style="display: flex; justify-content: space-between; align-items: center; padding: 5px;"> <div style="text-align: left;"> <p>Argumentwerte <math>e'_1, e'_2, \dots</math> in den Rumpf einsetzen, den Rumpf mittels <math>\rightsquigarrow</math> reduzieren</p> </div> <div style="text-align: left;"> <p>falls <math>f'</math> Lambda-Abstraktion</p> </div> <div style="text-align: right;"> <p><math>[apply_\lambda]</math></p> </div> </div>

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

Beispiele:

$(+ 40 2)$

$\rightsquigarrow_{eval_{id}} (\# \langle \text{procedure:} + \rangle 40 2)$

$eval_{lit} \cdot 2$

$\rightsquigarrow_{apply_{prim}} 42$

$(\text{sqr } 9) \rightsquigarrow_{eval_{id}} (\text{lambda}(x)(*xx))$

$eval_{lit}$

$\rightsquigarrow_{apply_\lambda} (* 9 9)$

$\rightsquigarrow_{eval_{id}} (\# \langle \text{procedure : } * \rangle 9 9)$

$eval_{lit*2}$

$\rightsquigarrow_{apply_{prim}} 81$

## Einschub: Lexikalische Bindung

Bezeichnen  $(\text{lambda } (x) (* x x))$  und  $(\text{lambda } (r) (* r r))$  die gleiche Funktion?

$(\dots 9) \rightsquigarrow^* 81$

$\Rightarrow$  JA!

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

Das bindende Vorkommen eines Identifiers  $\langle x \rangle$  im Programmtext kann systematisch bestimmt werden: Suche strikt "von innen nach außen" bis zum ersten

(1)  $(\text{lambda } (x) \dots)$

(2)  $(\text{define } x \dots)$

Das ist das Prinzip der lexikalischen Bindung ( $\triangle!$  Syntaxprüfung in DrRacket)

## Übliche Notation in der Mathematik: Fallunterscheidung!

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

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

Typische Primitive in Tests:

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

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

## Spezialform Fallunterscheidung (conditional)

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

$(\langle t_2 \rangle \langle e_2 \rangle)$

...

$(\langle t_n \rangle \langle e_n \rangle)$

$(\text{else } \langle e_{n+1} \rangle))$  <- optional

Führt die Tests in der Reihenfolge  $\langle t_1 \rangle, \langle t_2 \rangle, \dots$  durch. Sobald  $\langle t_i \rangle$  zu  $\#t$  ausgewertet, werte Zweig  $\langle e_i \rangle$  aus.  $\langle e_i \rangle$  ist das Ergebnis der Fallunterscheidung. Wenn  $\langle t_n \rangle \#f$  liefert, dann liefere

$\left( \begin{array}{l} \text{Fehlermeldung "cond: alle Tests ergeben \#f falls kein else- Zweig, sonst} \\ \langle e_{n+1} \rangle \end{array} \right)$

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

$(\text{one-of } \langle e_1 \rangle \langle e_2 \rangle \dots \langle e_n \rangle)$

Reduktion von  $\text{cond}$   $[eval_{cond}]$

- $(\text{cond } ( \langle t_1 \rangle \langle e_1 \rangle ) ( \langle t_2 \rangle \langle e_2 \rangle ) \dots)$ 
  - (1) Reduziere  $\langle t_1 \rangle$ , erhalte  $\langle t'_1 \rangle$
  - (2)  $\langle e_1 \rangle$  falls  $\langle t_1 \rangle = \#t$   
 $(\text{cond } ( \langle t_2 \rangle \langle e_2 \rangle ) \dots)$
- $(\text{cond } (\text{else} \langle e_{n+1} \rangle )) \rightsquigarrow \langle e_{n+1} \rangle$  (  $\langle t_1 \rangle, \langle e_2 \rangle, \dots$  sind nicht ausgewertet  
sonst  
 $\langle e_1 \rangle$  nicht ausgewertet)
- $(\text{cond } ) \rightsquigarrow$  Fehler "cond alle Tests ergeben #f"

## Binäre Fallentscheidung:

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

## Zusammengesetzte Daten

Daten können interessante interne Struktur (Komponenten) aufweisen.

Beispiel: Ein Star Wars Charakter:

name	"Luke Skywalker"
jedi?	#f
force	25

Beispiel:

```

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

(make-character n j f) ~> ( <)Tabelle))) Konstruktion
(character-name <Tabelle>) ~> n Selektor (komponenten auslesen)
(character-jedi? <Tabelle>)) ~> j
(character-force <Tabelle>)) ~> f

```

## Records in Scheme

Record-Definition legt fest:

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

(define-record-procedures  $\langle t \rangle$  Signaturname

make- $\langle t \rangle$  ;Konstruktor

$\langle t \rangle?$  ;Prädikat

( $\langle t \rangle$ - $\langle comp_1 \rangle$  ;Liste der Selektoren

...  $\langle t \rangle$ - $\langle comp_n \rangle$ ))

Liste der Selektoren legt Komponenten (Anzahl, Reihenfolge, Namen) fest. Signatur des Konstruktors/der Selektoren für Record-Signatur  $\langle t \rangle$  mit n Komponenten  $\langle comp_1 \rangle$  ...  $\langle comp_n \rangle$ :

(: make  $\langle t \rangle$  (  $\langle t \rangle$  ...  $\langle t_n \rangle \rightarrow \langle t \rangle$  ))  
n Komponentensignaturen

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

$\forall$  string n, boolean j, natural f:

(character-name (make-character n j f))  $\rightsquigarrow$  n

(character-jedi? (make-character n j f))  $\rightsquigarrow$  j

(character-force (make-character n j f))  $\rightsquigarrow$  f

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

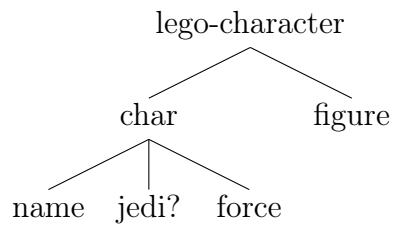
## Spezialform check-property

(check-property (for-all(( $\langle id_1 \rangle \langle signatur_1 \rangle$ ) ... ( $\langle id_n \rangle \langle signatur_n \rangle$ ))  $\langle expr \rangle$ ))  $\langle expr \rangle$  ist das Prädikat, das sich auf  $\langle id_q \rangle$  ...  $\langle id_n \rangle$  bezieht.

Test erfolgreich, falls  $\langle expr \rangle$  für beliebige Bindungen für  $\langle id_1 \rangle$  ...  $\langle id_n \rangle$  immer #t ergibt.

## Interaktion von Konstruktor und Selektor

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



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  $\langle f \rangle$ , die zusammengesetzte Daten der Signatur  $\langle t \rangle$  konsumiert.

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

Prozedur  $\langle f \rangle$ , die zusammengezte Daten der Signatur  $\langle t \rangle$  konstruiert/produziert.

- Konstruktoraufruf für  $\langle t \rangle$  muss enthalten sein!

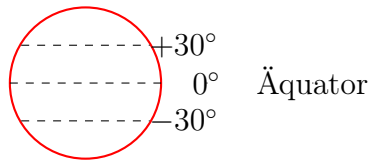
```

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

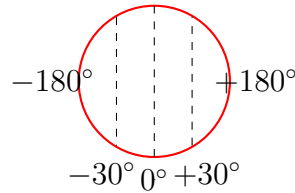
char	<table> <tr> <td>name</td><td></td></tr> <tr> <td>jedi?</td><td></td></tr> <tr> <td>force</td><td></td></tr> </table>	name		jedi?		force	
name							
jedi?							
force							
figure	(BILD)						

## Längen/Breitengrade

Breitengrade (latitude)



Längengrade (longitude)



Sei  $\langle p \rangle$  ein Prädikat mit Signatur  $(\langle t \rangle \rightarrow \text{boolean})$ . Eine Signatur

$\parallel (\text{predicate } \langle p \rangle)$

gilt für jeden Wert  $x$  mit Signatur  $\langle t \rangle$  für den zusätzlich  $(p(\langle p \rangle x) \rightsquigarrow \#t)$  gilt. Signatur  $(\text{predicate } \langle p \rangle)$  ist damit spezifischer (restriktiver) als Signatur  $\langle t \rangle$ .

## Signaturnamen

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

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

Beispiele:

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

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

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

Ein geocode besteht aus:

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

## Gemischte Daten

Die Signatur mixed

$\parallel (\text{mixed } \langle t_1 \rangle \dots \langle t_n \rangle)$

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

Beispiel: Datendefinition:

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

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

Beispiel

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

Das Prädikat  $\langle t \rangle?$  einer Record-Signatur  $\langle t \rangle$  unterscheidet Werte der Signatur  $\langle t \rangle$  von allen anderen Werten:

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

Auch: Prädikate für eingebaute Signaturen.

number?, complex?, real?, rational?, integer?, Prozeduren, die gemischte  
natural?, string?, boolean?

Daten der Singatuen  $\langle t_1 \rangle \dots \langle t_n \rangle$  konsumieren:

```
|| (: <f>((mixed <t1> ... <t2>)-> ...))
|| (define <f>
||   (lambda (x)
||     (cond((<t1>? x) ...)
||           ...
||           ((<t2>? x) ...))))
```

Mittels let lassen sich Werte an lokale Namen! binden:

```
|| (let ((<id1> <e2>) ... (<idn> <en>)) e)
```

Die Ausdrücke  $\langle e_1 \rangle \dots \langle e_n \rangle$  werden parallel ausgewertet.

$\Rightarrow \langle id_1 \rangle \dots \langle id_n \rangle$  können in  $\langle e \rangle$  (und nur dort!) verwendet werden.

Der Wert des let-Ausdruck ist der Wert von e. "nur dort": Verwendung nur in  $\langle e \rangle$ , nicht in den in  $\langle e_i \rangle$ !

Lokal: Verwendung nicht außerhalb des (let...)

 Sprachlevel "Die Macht der Abstraktion"

```
|| (let () ≡ (lambda () ))
```

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

```
|| (check-error <e> <msg>)
```

erwartet Abbruch mit Fehlermeldung  $\langle msg \rangle$ . Erzwingen des Programmabbruches mittels (violation  $\langle msg \rangle$ )

## Polymorphe Signaturen

Beobachtung: Manche Prozeduren arbeiten völlig unabhängig von den Signaturen ihrer Argumente:

parametrisch polymorphe Prozeduren (griechisch: vielgestaltig). Nutze Signaturvariablen:  
Beispiele:

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

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

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

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

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

Beispiel:

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



## Polymorphe Paare und Listen

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

(pair-of  $\langle t_1 \rangle \langle t_2 \rangle$ )

ist eine Signatur für Paare, deren erste und zweite Komponente von der Signatur  $\langle t_1 \rangle$  bzw.  $\langle t_2 \rangle$  sind.

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

## Liste

Eine Liste von Werten der Signatur  $\langle t \rangle$ , (list-of  $\langle t \rangle$ ), ist entweder

- leer (Signatur empty-list) oder
- ein Paar (Signatur pair-of) aus
  - einem Listenkopf (Signatur  $\langle t \rangle$ ) und
  - einer Restliste (Signatur (list-of  $\langle t \rangle$ )))

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

(list-of  $\langle t \rangle$ ). Listen, deren Elemente die Signatur  $\langle t \rangle$  besitzen.

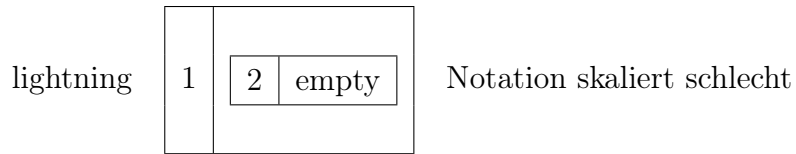
Die Signatur empty-list ist bereits in DrRacket vordefiniert.

Ebenfalls vordefiniert ist:

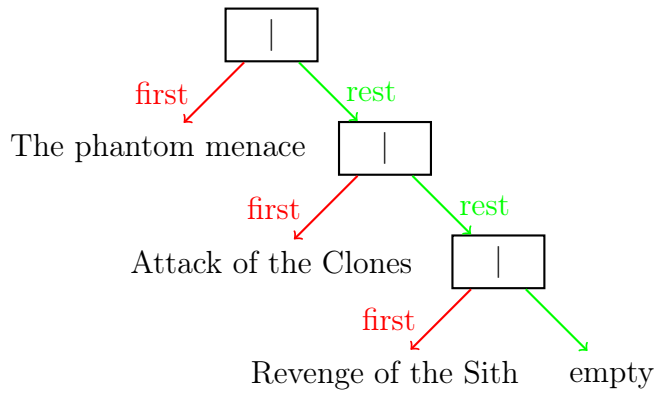
- (: empty empty-list)
- (: empty? (any -> boolean))

## Visualisierung von Listen

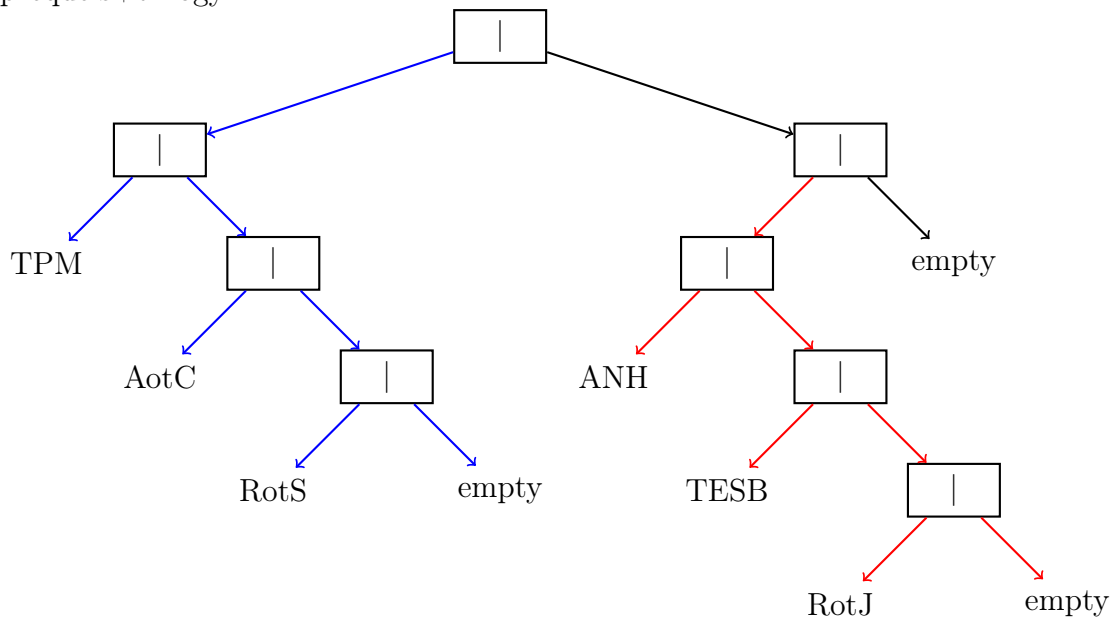
```
|| (make-pair 1 (make-pair 2 empty))
```



### Spines (Rückrad)



prequels+trilogy:



## Prozeduren über Listen

Schablonen für gemischte und zusammengesetzte Daten

Beispiel:

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

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

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

Schablone für Funktion  $\langle f \rangle$ , die Liste xs konsumiert:

```
(: <f> ((list-of <t1>) -> <t2>))
(define <f>
  (lambda (xs)
    (cond ((empty? xs) ... )
          ((pair? xs) ... (first xs) ... (<f> (rest xs)) ...))))
                                          Signatur t_1

                                          signatur t_2
```

## Neue Sprachebene "Macht der Abstraktion"

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

### cat

;Füge Listen xs, xy zusammen (concatenate)

Zwei Fälle (xs leer oder nicht-leer)

- 1)  $\overset{xs}{empty} \quad \overset{ys}{y_1 y_2 y_n} \quad \overset{(catxsxy)}{y_1 y_2 y_n}$
- 2)  $x_1 x_2 x_n \quad y_1 y_2 y_n \quad \overset{x_1 x_2 x_n y_1 y_2 y_n}{\underset{restxs \quad ys}{(cat(restxs)ys)}}$

## Bewertungen

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

Spezialform match vergleicht einen Wert  $\langle e \rangle$  mit gegebenen Patterns  $\langle pat_1 \rangle \langle pat_2 \rangle, \dots \langle pat_n \rangle$ . Falls  $\langle pat_i \rangle$ ,  $1 \leq i \leq n$ , das erste Pattern ist, das auf  $\langle e \rangle$  matched, ist Zweig  $\langle e_i \rangle$  das Ergebnis (ansonsten wird die Aiswertung mit "keiner der Zewige passte") abgegeben.

```

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

```

## Pattern Matching für $\langle pat_i \rangle$

- Literal  $\langle l \rangle$ :  
 $\langle e \rangle$  matched, falls  $\langle e \rangle \rightsquigarrow \langle l \rangle$
- "Don't care"  $\_$  :  
 $\langle e \rangle$  matched immer
- Variable  $\langle v \rangle$   
 $\langle e \rangle$  matched immer, danach ist  $\langle v \rangle$  an den Wert von  $\langle e \rangle$   $n \langle e_i \rangle$  gebunden
- Record-Konstruktor  $(\langle c \rangle \langle pat_{i1} \rangle \langle pat_{ik} \rangle, k \geq \emptyset)$   
 $\langle e \rangle$  matched, wenn es durch  $(\langle c \rangle \langle x_1 \rangle \langle x_k \rangle)$  konstruiert wurde und  $\langle x_j \rangle$  auf  $\langle pat_{ij} \rangle$  matched,  $1 \leq j \leq k$

⚠ Fall 4 ermöglicht Pattern Matching auf komplex konstruierten Werten.

## Rekursion über natürliche Zahlen

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

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

$\mathbb{N} = \{0, (\text{succ } 0), (\text{succ } (\text{succ } 0)), \dots\}$

Konstruktoren:

```

| (: zero natural)
| (define zero 0)

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

```

Bedingte algebraische Eigenschaften (siehe check-property)  $(= \Rightarrow \langle p \rangle \langle e \rangle)$  Nur, wenn  $\langle p \rangle \rightsquigarrow \#t$ , wird der Ausdruck  $\langle e \rangle$  ausgewertet und getestet ob  $\langle e \rangle \rightsquigarrow \#t$ .

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

$0! = 1$

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

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

```

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

```

Schablone für Funktionen  $\langle f \rangle$ , die natürliche Zahlen konsumieren.

```

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

```

Satz:

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

Reduktion kann durchaus zur Konstruktion von Ausdrücken führen, die zunehmende Größe aufweisen (Für factorial bestimmt das Argument die Größe.) Wenn möglich, erzeuge Reduktionsprozesse, die konstanten Platzverbrauch - unabhängig von Funktionsargumenten - benötigen. Beobachtung (Assoziativität von \*)

$$\begin{aligned}
 & (* 10 (* 9 (* 8 (* 7 (* 6 (factorial 5))))) \\
 = & (* (* (* (* (* 10 9) 8) 7) 6) (factorial 5)) \\
 & (* 30240 (factorial 5))
 \end{aligned}$$

⇒ Multiplikationen können vorgezogen werden.

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

Berechne 5!:

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

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

```
|| (: fac-worker (natural natural -> natural))
||
|| (define fac-worker
||   (lambda (n acc)
||     (cond ((= n 0) acc)
||           ((> n 0) (fac-worker (- n 1) (* acc n))))))
||
|| ; Berechne n! [wrapper]
|| (: fac (natural -> natural))
|| (define fac
||   (lambda (n)
||     (fac-worker n 1)))
```

Ein Reduktionsprozess ist iterativ, falls seine Größe konstant bleibt.

Damit: factorial nicht iterativ

fac-worker iterativ

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

Kontext des rekursiven Aufrufes in

- factorial: (\* n  $\square$ )  
Hole
- fac-worker: -keiner-

Ein Prozeduraufruf ist endrekursiv (tail call), wenn er keinen Kontext besitzt.

Prozeduren, die nur endrekursive Prozeduraufrufe enthalten, heißen selbst endrekursiv.

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

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

iterative Listenumkehr (backwards)

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

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

letrec

$$\begin{array}{l} (\text{letrec } ((\langle id_1 \rangle \langle e_1 \rangle) \\ \quad \dots \\ \quad (\langle id_n \rangle \langle e_n \rangle)) \\ \langle e \rangle) \end{array}$$

22