

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

# Informatik I Vorlesung

Wintersemester 2016/2017

Mitschrieb von  
Julian Wolff

Aktueller Stand 8. November 2016

# Inhaltsverzeichnis

<b>1</b>	<b>Scheme: Ausdrücke, Auswertung und Abstraktion</b>	<b>2</b>
1.1	REPL . . . . .	2
1.2	Literale . . . . .	2
1.3	Zusammengesetzte Ausdrücke . . . . .	3
1.4	Identifizier . . . . .	3
1.5	Lambda-Abstraktion . . . . .	3
1.6	Kommentare . . . . .	4
1.7	Signaturen . . . . .	4
1.8	Prozedur-Signaturen . . . . .	5
1.9	Testfälle . . . . .	5
1.10	Erinnerung . . . . .	5
1.11	Top-Down-Entwurf (Programmieren mit "Wunschdenken") . . . . .	6
1.12	Reduktionsregeln für Scheme ( $\rightsquigarrow$ ) . . . . .	7
1.12.1	Einschub: Lexikalische Bindung . . . . .	8
1.13	Übliche Notation in der Mathematik: <u>Fallunterscheidung!</u> . . . . .	8
1.14	Spezialform Fallunterscheidung (conditional) . . . . .	8
1.15	Binäre Fallentscheidung: . . . . .	9

# 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

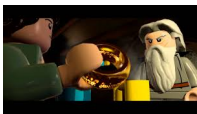
Interaktionsfenster:

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 `⟨pi⟩`.

(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) $\xrightarrow{!}$ (*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:

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

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:

$(\text{check-expect } \langle e_1 \rangle \langle e_2 \rangle)$

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  $(\text{define } \langle \text{name} \rangle (\text{lambda } (\langle p_1 \rangle \langle p_2 \rangle ) )$
- 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 starrt auf Horizont (bei jeden  $t$ )

Zeichne Szene von hinten nach vorne:



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

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

## Reduktionsregeln für Scheme ( $\rightsquigarrow$ )

Fallunterscheidung je nach Ausdruck:

- Literal  $l$  ( $1$ ,  $\#t$ , "Karotte", ...)  $[eval_1]$   
 $l \rightsquigarrow l$  (keine Reduktion möglich)
- Identifier  $\langle id \rangle$   $[eval_{id}]$   
 $\langle id \rangle \rightsquigarrow$  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;"> <div style="text-align: center;"> <p>Operation auf <math>e'_1, e'_2 \dots</math> anwenden</p> </div> <div style="text-align: center;"> <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;"> <div style="text-align: center;"> <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: center;"> <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 *2angle x* 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 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:

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