

**14.4.2015****Scheme**Ausdrücke , Auswertung und Abstaktion**Dr Racket**

Definitonsfenster

Willkommen bei [DrRacket](#), Version 6.1.1 [3m].Sprache: **Die Macht der Abstraktion**; memory limit: **128 MB**.> **Interaktionsfenster**

Die Anwendung von Funktionen wird in Scheme ausschließlich in Präfixnotation durchgeführt

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 92)</code>
$3!$	<code>(! 3)</code>

Allgemein: (`<funktion><argument1><argument2>...`)


`(+ 40 2)` und `(odd? 42)` sind Beispiele für Ausdrücke, die bei Auswertung einen Wert liefern

(Notation:  $\rightsquigarrow$ )`(+ 40 2)`  $\rightsquigarrow$  42`(odd? 42)`  $\overset{\text{Reduktion}}{\rightsquigarrow}$  #f

Interaktionsfenster:

$$\underbrace{Read \rightarrow Eval \rightarrow Print \rightarrow Loop}_{REPL}$$

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

Literal		Sorte, Typ
#f, #t	(true, false, Wahrheitswert)	boolean
"x"	(Zeichenketten)	String
0 1904 42 -2	(ganze Zahl)	Integer
0.42 3.14159	(Fließkommazahl)	real
1\2, 3\4, -1\10	(rationale Zahlen)	rational
	(Bilder)	image

## 16.4.2015

Auswertung zusammengesetzter Ausdrücke in mehreren Schritten (Steps), von “innen nach außen“, bis keine Reduktion mehr möglich

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

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

Beispiel: Auswertung des zusammengesetzten Ausdrucks

```
; Achtung: Arithmetik mit Fliesskommazahlen (real)\\
unterliegt Rundung!
(+ 0.7
  (- (/ 1/2 0.25)
      (/ 0.6 0.3)))

(- (+ 0.7
      (/ 1/2 0.25))
   (/ 0.6 0.3))

; Arithmetik mit rationalen Zahlen (rational) ist exakt
(- (+ 7/10
      (/ 1/2 1/4))
   (/ 6/10 3/10))
```

Ein Wer kann an einen Namen (auch Identifier) gebunden werden, durch  
 (define <id><e>)           <id>Identifier <e>Ausdruck

Erlaubte konsistente Wiederverwendung, dient der Selbstdokumentation von Programmen

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

Namen können in Scheme beliebig gewählt werden, solange

- (1) Die Zeichen ( ) [ ] { } “ , ‘ ‘ ; # | \ nicht vorkommen
- (2) der nicht einem numerischen Literal gleicht.
- (3) kein Whitespace (Leerzeichen, Tabulator, Return) enthalten ist.

Beispiel: euro → US\$

**Achtung:** Groß\Kleinschreibung ist irrelevant

```
; Bindung von Werten an Namen
(define absoluter-nullpunkt -273.15)
(define pi 3.141592653)
(define Gruendungsjahr-SC-Freiburg 1904)
(define top-level-domain-germany "de")
(define minutes-in-a-day (* 24 60))
(define vorwahl-tuebingen (sqrt 1/2))
```

Eine lambda-Abstraktion (auch Funktion, Prozedur) erlaubt die Formatierung von Ausdrücken, in denen mittels Parametern von konkreten Werten abstrahiert wird.

(lambda (<p1><p2>...) <e>

<e>Rumpf ⇒ enthält Vorkommen der Parameter <p<sub>n</sub>>

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

Anwendung (auch : Application) des lambda Aufrufs führt zur Ersetzung aller Vorkommen der Parameter im Rumpf durch die angegebenen Argumente.

```
; Abstraktion: Ausdruck mit "Loch" \odot
(lambda (\odot) (* \odot (* 155 minutes-in-a-day)))

; Zuwachs der Weltbevölkerung innerhalb von days Tagen
(define population-growth-in-days
  (lambda (days) (* days (* 155 minutes-in-a-day))))

(population-growth-in-days 7)
```

(lambda (days) (\* days (\* 155 minutes-in-a-day))) 365) ~~~~  
 (\* 365 (\* 155 minutes-in-a-day)) ~~~~81468000

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

Prozeduren sollten im Programm ein- bis zwei zeilige Kurzbeschreibungen direkt vorangestellt werden

## 21.4.2015

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

(: <id><signatur>)

Bereits eingebaute Sinaturen

natural  $\mathbb{N}$  boolean

integer  $\mathbb{Z}$  string

rational  $\mathbb{Q}$  image

real  $\mathbb{R}$  ...

numver  $\mathbb{C}$

(: ...) ist eine Spezialform und hat keinen Wert, aber einen Effekt: Signaturprüfung

Prozedur Signatur spezifizieren sowohl Signaturen für die Parameter  $P_1, P_2, \dots, P_n$  als auch den Ergebniswert der Prozedur

(: <Signatur  $P_1$ >... <Signatur  $P_n$ >- > <Signatur Ergebnis>)

Prozedur Signaturen werden bei jeder Anwendung einer Prozedur auf Verletzung geprüft Testfälle dokumentieren das erwartete Ergebnis einer Prozedur für ausgewählte Argumente:

(check-expect <e<sub>1</sub>><e<sub>2</sub>>)

Werte Ausdruck <e<sub>1</sub>>aus und teste, ob der erhaltene Wert der Erwartung <e<sub>2</sub>>entspricht (= der Wert von <e<sub>2</sub>>) Einer Prozedur sollte Testfälle direkt vorangestellt werden.

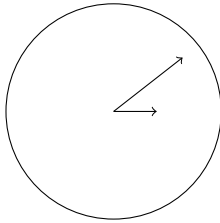
Spezialform: kein Wert, sondern Effekt: Testverletzung protokollieren

### Konstruktionsanleitung für Prozeduren:

- (1) Kurzbeschreibung (ein- bis zweizeiliger Kommentar mit Bezug auf Parametername)
- (2) Signaturen
- (3) Testfälle
- (4) Prozedurrumpf

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

Beispiel:Zeichne Ziffernblatt (Stunden- und Minutenzeiger) zu Uhrzeit auf einer analogen 24 Uhr



Minutenzeiger legt  $\frac{360^\circ}{60}$  Grad pro Minute zurück (also  $\frac{360}{60} \cdot m$ )  
 Stundenzeiger liegt  $\frac{360}{12}$  pro Stunde zurück ( $\frac{360}{12} \cdot h + \frac{360}{12} \cdot \frac{m}{60}$ )

```
; Grad, die Minutenzeiger pro Minute zuruecklegt
(define degrees-per-minute 360/60)

; Grad, die Stundenzeiger pro voller Stunde zuruecklegt
(define degrees-per-hour 360/12)

; Zeichne Ziffernblatt zur Stunde h und Minute m
(: draw-clock (natural natural -> image))
(check-expect (draw-clock 4 15) (draw-clock 16 15))
(define draw-clock
  (lambda (h m)
    (clock-face (position-hour-hand h m)
                 (position-minute-hand m))))

; Winkel (in Grad), den Minutenzeiger zur Minute m einnimmt
(: position-minute-hand (natural -> rational))
(check-expect (position-minute-hand 15) 90)
(check-expect (position-minute-hand 45) 270)
(define position-minute-hand
  (lambda (m)
    (* m degrees-per-minute)))

; Winkel (in Grad), den Stundenzeiger zur Stunde h einnimmt
(: position-hour-hand (natural natural -> rational))
(check-expect (position-hour-hand 3 0) 90)
(check-expect (position-hour-hand 18 30) 195)
(define position-hour-hand
  (lambda (h m)
    (+ (* (modulo h 12) degrees-per-hour)
       ; h mod 12 in {0,1,...,11}
       (* (/ m 60) degrees-per-hour))))
```

```

; Zeichne Ziffernblatt mit Minutenzeiger um dm und
; Stundenzeiger um dh Grad gedreht
(: clock-face (rational rational -> image))
(define clock-face
  (lambda (dh dm)
    (clear-pinhole
     (overlay/pinhole
      (circle 50 "outline" "black")
      (rotate (* -1 dh) (put-pinhole 0 35 (line 0 35 "red")))
      (rotate (* -1 dm) (put-pinhole 0 45 (line 0 45 "blue"))))))))

```

## 23.4.2015

### Substitutionsmodell

Reduktionsregeln für Scheme (Fallunterscheidung je nach Ausdrücken) wiederhole, bis keine Reduktion mehr möglich

- literal (1, "abc", #t, ...)  $l \rightsquigarrow l$  [eval<sub>lit</sub>]
- Identifier id(pi, clock-face,...)  $id \rightsquigarrow$  gebundene Wert [eval<sub>id</sub>]
- lambda Abstraktion  $(\text{lambda } (\dots) \dots) \rightsquigarrow \text{lambda}(\dots) \dots$  [eval<sub>λ</sub>]
- Applikationen (f e<sub>1</sub> e<sub>2</sub> ...)

$$f, e_1, e_2 \text{ reduzieren erhalte: } f', e_1', e_2' \quad (1)$$

- (2)  $\begin{cases} \text{Operation } f' \text{ auf } e_1' \text{ und } e_2' \text{ [apply}_{\text{prim}}] & \text{falls } f' \text{ primitiv ist} \\ \text{Argumentenwerte in den Rumpf von } f' \text{ einsetzen, dann reduzieren} & \text{falls } f' \text{ lambda Abstraktion} \end{cases}$

Beispiel:

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

$(\text{position-minute-hand } 30)$	$\rightsquigarrow_{\text{eval id}}$ $\rightsquigarrow_{\text{eval lambda}}$ $\rightsquigarrow_{\text{eval id}}$ $\rightsquigarrow_{\text{apply prim}}$	$((\text{lambda } (m) (* \text{degrees-per-minute } m)) 30)$ $(* \text{degrees-per-minute } 30)$ $(\# <\text{procedure } *> \frac{360}{60} 30)$ $180$
-------------------------------------	---	--

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 Prozeduren (siehe apply)

Das bindene Vorkommen eines Identifiers id kann im Programmtext systematisch bestimmt werden: Suche strikt von innen nach außen, bis zum ersten

(1)  $(\text{lambda } (r) <\text{Rumpf}>)$

(2)  $(\text{define } <e>)$

Prinzip der lexikalische BindungÜbliche Notation in der Mathematik: Fallunterscheidung

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

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

Typische primitive Tests.

(: = (number number -&gt; boolean))

(: &lt; (real real -&gt; boolean))

auch &gt;, &lt;=, &gt;=

(: String=? (string string -&gt; boolean))

auch string&gt;?, string&lt;=?

(: zero? (number -&gt; boolean))

odd?, even?, positive?, negative?

Binäre Fallunterscheidung if*if*<  $e_1$  > Mathematik:

$$< e_2 > \begin{cases} e_1 & \text{falls } t_1 \\ e_2 & \text{sonst} \end{cases}$$

<  $e_2$  >)**28.4.2015**Die Signatur one of lässt genau einen der ausgewählten Werte zu.(one of < $e_1$ > < $e_2$ > ... < $e_n$ >)

```

; Punkte der Heimmannschaft bei Ergebnis h:a
(: heim-punkte (natural natural -> (one-of 3 0 1)))
(check-expect (heim-punkte 2 0) 3)
(check-expect (heim-punkte 1 4) 0)
(check-expect (heim-punkte 3 3) 1)
(define heim-punkte
  (lambda (h a)
    (cond ((> h a) 3)
          ((< h a) 0)
          (else 1))))

```

Reduktion von if:

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

- ① Reduziere  $t_1$ , erhalte  $t'_1 \rightsquigarrow$  ②  $\begin{cases} \langle e_1 \rangle & \text{falls } t'_1 = \#t, \langle e_2 \rangle \text{ niemals ausgewertet} \\ \langle e_2 \rangle & \text{falls } t'_1 = \#f, \langle e_1 \rangle \text{ niemals ausgewertet} \end{cases}$

```
; Koennen wir unser eigenes `if' aus `cond' konstruieren? (Nein!)

; Bedingte Auswertung von e1 oder e2 (abhaengig von t1)
(check-expect (my-if (= 42 42) "Yes!" "No!") "Yes!")
(check-expect (my-if (odd? 42) "Yes!" "No!") "No!")
(define my-if
  (lambda (t1 e1 e2)
    (cond (t1 e1)
          (else e2))))

; Sichere Division x/y, auch fuer y = 0
(: safe-/ (real real -> real))
(define safe-/
  (lambda (x y)
    (my-if (= y 0)      ; <-- Funktion my-if wertet ihre Argumente
            x           ; vor der Applikation aus: (/ x y) wird
            (/ x y)))) ; in *jedem* Fall reduziert. :-()

(safe-/ 42 0)          ; Fuehrt zu Fehlermeldung "division by zero"
                      ; (Reduktion mit Stepper durchfuehren)
```

Spezifikation Fallunterscheidung (conditional expression):

(cond	Mathematik:
( $\langle t_1 \rangle \langle e_1 \rangle$ )	$\left\{ \begin{array}{l} e_1 \text{ falls } t_1 \\ e_2 \text{ falls } t_2 \\ \dots \\ e_n \text{ falls } t_n \\ e_{n+1} \text{ sonst} \end{array} \right.$
( $\langle t_2 \rangle \langle e_2 \rangle$ )	
...	
( $\langle t_n \rangle \langle e_n \rangle$ )	
(else $\langle e_{n+1} \rangle$ )	

Werte die Tests in den Reihenfolge  $t_1, t_2, t_3, \dots, t_n$  aus.

Sobald  $t_i \# t$  ergibt, werte Zweig  $e_i$  aus.  $e_i$  ist Ergebnis der Fallunterscheidung. Wenn  $t_n \# t$  liefert, dann liefert

{	Fehlermeldung „cond: alle Tests ergaben false“	falls kein else Zweig
	$\langle e_{n+1} \rangle$	sonst

```
; Absolutwert von x
(: my-abs (real -> real))
(check-within (my-abs -4.2) 4.2 0.001) ; Wichtig:
```



```

(check-within (my-abs 4.2) 4.2 0.001)      ; Tesfaelle decken alle
  Zweige
(check-within (my-abs 0) 0 0.001)          ; der conditional
  expression an
(define my-abs
  (lambda (x)
    (cond ((< x 0) (- x))
          ((> x 0) x)
          (else 0))))

```

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

$(\text{cond } (<t_1><e_1>)(<t_2><e_2>)\dots(<t_n><e_n>))$

① Reduziere  $t_1$  erhalte  $t'_1 \rightsquigarrow$  ②  $\begin{cases} <e_1> & \text{falls } t'_1 = \#t \\ (\text{cond } <t_2> <e_2>) & \text{sonst} \end{cases}$

$(\text{cond}) \rightsquigarrow$  „Fehlermeldung : alle Test ergaben false“

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

cond ist syntaktisches Zucker (auch abgeleitete Form) für eine verbundene Anwendung von if

$$\begin{array}{ll}
 (\text{cond } (<t_1><e_1>) & \text{if } <t_1> \\
 (<t_2><e_2>) & <e_1> \\
 \dots & \text{if } <t_2> \\
 \dots & <e_2> \\
 \dots & \dots \\
 (<t_n><e_n>) & \text{if } <t_n> \\
 & <e_n> \\
 (\text{else } <e_{n+1}> & <e_{n+1}>)) \dots))
 \end{array}$$

Spezialform 'and' und 'or'

$(\text{or } <t_1> <t_2> \dots <t_n>) \rightsquigarrow (\text{if } <t_1> (\text{or } <t_2> \dots <t_n>) \#t)$

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

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

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

; Konstruktion komplexer Praedikate mittels 'and' und 'or':

```

(and #t #f) ; eval #f (Mathematik: Konjunktion)
(or #t #f) ; eval #t (Mathematik: Disjunktion)

```

; Kennzeichen am/pm fuer Stunde h

```

(: am/pm (natural -> (one-of "am" "pm" "???")))
(check-expect (am/pm 10) "am")
(check-expect (am/pm 13) "pm")
(check-expect (am/pm 25) "???")
(define am/pm
  (lambda (h)

```

```
(cond ((and (>= h 0) (< h 12)) "am")
      ((and (>= h 12) (< h 24)) "pm")
      (else "???")))
```

## 30.4.2015

### Zusammengesetzte Daten

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

Konkrete Charakter:

name	„Luke Skywalker“
jedi?	#f
force	25

```
; Ein Charakter (character) besteht aus
; - Name (name)
; - Jedi-Status (jedi?)
; - Staerke der Macht (force)
(: make-character (string boolean real -> character))
(: character? (any -> boolean))
(: character-name (character -> string))
(: character-jedi? (character -> boolean))
(: character-force (character -> real))
(define-record-procedures character
  make-character
  character?
  (character-name
   character-jedi?
   character-force))

; Definiere verschiedene Charaktere des Star Wars Universums
(define luke
  (make-character "Luke_Skywalker" #f 25))
(define r2d2
  (make-character "R2D2" #f 0))
(define dooku
  (make-character "Count_Dooku" #f 80))
(define yoda
  (make-character "Yoda" #t 85))
```

Zusammengesetzte Daten = Records in Scheme Record-Definition legt fest:

- Record-Signatur
- Konstruktor (baut aus Komponenten einen Record)
- Prädikat (liegt ein Record vor?)
- Liste von Selektoren (lesen jeweils eine Komponente des Records)

```
(define-record-procedure <t>
  make-<t>
  <t>?
  (<t>-<comp1> ... <t>-<comp2>))
;Liste der n Selektoren
```

Verträge des Konstruktors der Selektoren für Record- Signatur  
 <t> mit Komponenten namens <comp<sub>1</sub>> ... <comp<sub>n</sub>>

```
(: make-<t> (<t1>...<t2>) -> <t>)
(: <t>-<comp1> (<t> -> <t1>))
(: <t>-<compn> (<t> -> <tn>))
```

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

```
(character-name (make-character n j f) n)
(character-jedi? (make-character n j f) j)
(character-force (make-character n j f) f )
```

Spezialform check-property:

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

↓

;Bezieht sich auf <id1> ... <idn>

Test erfolgreich, falls <e> für beliebig gewählte Bedeutungen für <id<sub>1</sub>> ... <id<sub>n</sub>>  
 immer #t ergibt

Interaktion von Selektoren und Konstruktor:

```
(check-property
  (for-all ((n string)
            (j boolean)
            (f real))
    (expect (character-name (make-character n j f)) n)))

(check-property
  (for-all ((n string)
            (j boolean)
```

```

        (f real))
    (expect (character-jedi? (make-character n j f)) j)))

(check-property
  (for-all ((n string)
            (j boolean)
            (f real))
    (expect-within (character-force (make-character n j f)) f
      0.001)))

```

Beispiel: Die Summe von zwei natürlichen Zahlen ist mindestens so groß wie jeder dieser Zahlen:  $\forall x_1 \in \mathbb{N}, x_2 \in \mathbb{N} : x_1 + x_2 \geq \max\{x_1, x_2\}$

```

; Fuer alle natuerlichen Zahlen x1,x2 gilt: x1 + x2 ≥ max(x1,x2)
(check-property
  (for-all ((x1 natural)
            (x2 natural))
    (>= (+ x1 x2) (max x1 x2))))

```

Konstruktion von Funktionen, die bestimmte gesetzte Daten konsumiert.

- Welche Record-Componenten sind relevant für Funktionen?

→ Schablone:

```

(: sith? (character -> boolean))
(define sith?
  (lambda (c)
    ... (character-jedi? c)
    ... (character-force c) ...))

```

Konstruktion von Funktionen, die zusammengesetzte Daten konstruieren

- Der konstruktor muss aufgerufen werden

→ Schablone:

```

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

```

- Konkrete Beispiele:

```

; Könnte Charakter c ein Sith sein?
(: sith? (character -> boolean))
(check-expect (sith? yoda) #f)
(check-expect (sith? r2d2) #f)
(define sith?
  (lambda (c)

```

```
(and (not (character-jedi? c))
      (> (character-force c) 0)))

; Bilde den Charakter c zum Jedi aus (sofern c überhaupt Macht
; besitzt)
(: train-jedi (character -> character))

(check-expect (train-jedi luke) (make-character "Luke_Skywalker"
#t 50))
(check-expect (train-jedi r2d2) r2d2)

(define train-jedi
  (lambda (c)
    (make-character (character-name c)
                    (> (character-force c) 0)
                    (* 2 (character-force c)))))
```