

Informatik 1

01 – Scheme: Ausdrücke, Auswertung und Abstraktion

Winter 2026/27

Torsten Grust
Universität Tübingen

1 | DrRacket

Willkommen! Wir werden in dieser *Informatik 1* die Programmiersprache **Scheme** einsetzen.

- Programmierumgebung für Scheme: **DrRacket**, frei verfügbar für macOS , Linux , Windows 
- Download: <http://tiny.cc/Racket>
(Version 9.3)

1. [Help ▶ Deutsche Benutzeroberfläche für Racket](#) 
2. Racket neu starten
3. [Sprache ▶ Sprache auswählen...: SdP! – Anfänger ▶ OK](#) 
4. [Sprache ▶ Teachpack hinzufügen ▶ image.rkt ▶ OK](#) 

DrRacket: Definition + Interaktion



2 | Scheme: Funktionsanwendung nutzt Präfixnotation

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>
$\lfloor x \rfloor$	<code>(floor x)</code>
9^2	<code>(expt 9 2)</code>
$3!$	<code>(! 3)</code>

Allgemein: Anwendung einer Funktion f auf n Argumente via `(f argument1 argument2 ... argument $n$ )`.

Scheme: Ausdrücke und Auswertung

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

```
(+ 40 2)  $\rightsquigarrow$  42
```

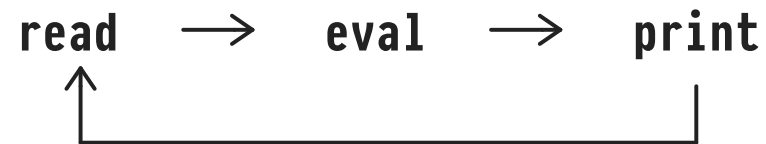
```
(odd? 42)  $\rightsquigarrow$  #f
```

☆ $\rightsquigarrow$: Durch **Auswertung** (auch: **Reduktion** oder **Evaluation**) führt Scheme Programme aus

DrRacket: Interaktionsfenster $\equiv$ REPL

DrRackets **Interaktionsfenster** realisiert einen interaktiven **read-eval-print loop** (REPL):

1. Liest Ausdruck (**read**),
2. wertet diesen mittels $\rightsquigarrow$ aus (**eval**),
3. gibt den resultierenden Wert aus (**print**).



3 | Scheme: Literale

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

Literal	
#t #f	<i>true</i> , <i>false</i> , Wahrheitswerte
"Zelda" "x" ""	Zeichenketten (auch: <i>Strings</i>)
0 1904 -42 007	} exakte Zahlen
0.42 3.1415 -273.15	
1/2 3/4 -1/10	
#i1.4142 #i-1.0	inexakte Zahlen
	Bilder

4 | Scheme: Zusammengesetzte Ausdrücke

Die Auswertung **zusammengesetzter Ausdrücke** vollzieht $\rightsquigarrow$ in mehreren Schritten (*Steps* **1**, **2**, ...), von “innen nach aussen”. Die Auswertung stoppt, sobald keine weitere Reduktion mehr möglich ist:

$$\begin{array}{ccccccc}
 (+ \underbrace{(+ \ 20 \ 20)}_{\substack{\nwarrow \\ \text{parallele Reduktion wäre OK}}} \ (+ \ 1 \ 1)) & \xrightarrow{\mathbf{1}} & (+ \ 40 \ \underbrace{(+ \ 1 \ 1)}) & \xrightarrow{\mathbf{2}} & \underbrace{(+ \ 40 \ 2)} & \xrightarrow{\mathbf{3}} & 42
 \end{array}$$

$\underbrace{\hspace{1cm}}$ $\equiv$ wird als nächstes reduziert

Scheme: Zusammengesetzte Ausdrücke

Beispiel: Auswertung des zusammengesetzten mathematischen Ausdrucks $2 - 1 + 3$.

- Die Mathematik benötigt neben “Punkt-vor-Strich” weitere Konventionen, die Zweideutigkeiten im Ausdruck auflösen:

$$\begin{array}{l}
 \text{1. Rechts-tief geklammert: } \overset{\boxed{2}}{2} \overset{\cdot}{\cdot} \overset{\cdot}{\cdot} - (\overset{\boxed{1}}{1} \overset{\cdot}{\cdot} \overset{\cdot}{\cdot} + 3) = -2 \\
 \text{2. Links-tief geklammert: } (\underset{\cdot}{\cdot} \underset{\cdot}{\cdot} \underset{\cdot}{\cdot} \overset{\boxed{1}}{2} - \underset{\cdot}{\cdot} \underset{\cdot}{\cdot} \underset{\cdot}{\cdot} \overset{\boxed{1}}{1}) + \underset{\cdot}{\cdot} \underset{\cdot}{\cdot} \underset{\cdot}{\cdot} \overset{\boxed{2}}{3} = 4
 \end{array}$$

Scheme ist explizit und unterscheidet Varianten 1. und 2. ...

Scheme: Arithmetik mit exakten vs. inexakten Zahlen

1. Exakte Zahlen:

`(+ 0.1 0.2)`

$\rightsquigarrow$ `0.3`

2. Inexakte Zahlen:

`(+ #i0.1 #i0.2)`

$\rightsquigarrow$ `#i0.3000000000000000000004`



Schemes interne Darstellung von inexakten Zahlen `#i...` ist kompakt und Arithmetik ist sehr effizient, kann aber zu Ungenauigkeiten führen.

Insbesondere können Tests auf Gleichheit unerwartet ausfallen: `(= (+ #i0.1 #i0.2) 0.3)` $\rightsquigarrow$ `#f.` ☹️

5 | Scheme: Namen und Bindungen

Ein Wert kann an einen **Namen** (auch: **Identifizier**) **gebunden** werden:

<code>(define <i>id</i> <i>e</i>)</code>	<i>id</i> : Identifizier, <i>e</i> : Ausdruck
---	---

- Solche Bindungen erlauben die konsistente Wiederverwendung und dienen der Selbstdokumentation von Programmen.
- **Achtung:** `(define ...)` ist eine sogenannte **Spezialform**. Insbesondere besitzt diese Spezialform *keinen* Wert, sondern einen **Effekt**: der Name *id* wird an den Wert von Ausdruck *e* **gebunden**.

DrRacket: Komplexere Programme im Definitionsfenster

Sobald Programme komplexer werden (und mehrere Bindungen mittels `define` enthalten), gehören diese in DrRackets oberes

Definitionsfenster:

1. Programm im Definitionsfenster konstruieren.
2. Gleichwertig: `Racket ▶ Start`  | `Button [Start`  `]`  | `⌘R`
3. Ausgabe der Werte der reduzierten Ausdrücke in REPL.
Alle gebundenen Namen sind auch in der REPL verfügbar.

Das Programm im Definitionsfenster kann mittels

- `Datei ▶ Definitionen speichern`  (`⌘S`)

in einem Racket-File (Dateiendung `.rkt`) gespeichert werden.

Scheme: Namen

Namen (Identifier) können in Scheme fast beliebig gewählt werden, solange

1. die Zeichen () [] { } " , ' ` ; # | \ nicht vorkommen,
2. der Name nicht einem numerischen Literal gleicht,
3. kein Whitespace (*Space*, *Tab*, *Return*) enthalten ist, und
4. der Name nicht bereits an einen Wert gebunden ist
(⚠ Groß-/Kleinschreibung ist in Namen *nicht* relevant).

- **Beispiel:** Damit ist `euro->us$` ein gültiger Identifier.
`Euro->US$` ist als Identifier identisch.

6 | Scheme: Ausdrücke mit “Löchern” □

- Ausdruck mit “Loch” □:

Setze Wert ein, erhalte auswertbaren Ausdruck

([.]
* □ (* 135 minutes-in-a-day))

- Scheme markiert solche “Löcher” mittels **lambda**:

(**lambda** (□) (* □ (* 135 minutes-in-a-day)))



(**lambda** (days) (* days (* 135 minutes-in-a-day)))

Scheme: Lambda-Abstraktion

Eine **Lambda-Abstraktion** (auch: **Funktion**) erlaubt die Formulierung eines Ausdrucks e , in dem mittels **Parametern** p_i von konkreten Werten **abstrahiert** wird:

	Parameter	
	⏟	
(lambda	$(p_1 \ p_2 \ \dots)$	$e)$
	↑	
	Rumpf , enthält Vorkommen von $p_1, p_2, \dots$	

(**lambda** ...) ist eine **Spezialform**. Der Wert der Lambda-Abstraktion ist **#<function>**. (Darauf kommen wir zurück.)

Scheme: Lambda? λ ? (*What the heck?*)

“In Russell and Whitehead's “Principia Mathematica” the notation for the function f with $f(y) = 2y+1$ is $2\hat{y}+1$. [Alonzo] Church originally intended to use the notation $\hat{y}.2y+1$. The typesetter could not position the hat on top of the y and placed it in front of it, resulting in $\wedge y.2y+1$. Then another typesetter changed it to $\lambda y.2y+1$. ”

— Henk Barendregt, *The Impact of Lambda Calculus in Logic and Computer Science* (1997)

7 | Scheme: Funktionsanwendung (Applikation)

Die **Anwendung** (auch: **Applikation**) einer Lambda-Abstraktion führt zur Ersetzung (unten: **1**) aller Vorkommen der Parameter p_i im Rumpf e durch die angegebenen **Argumente**.

Die Anwendung von Funktion f auf Argument x wird — wie gewohnt — mittels **Juxtaposition** notiert: $(f\ x)$.

		Funktion	Argument
		$\overbrace{((\text{lambda (days)} (* \text{ days } (* 135 \text{ minutes-in-a-day})))}$	$\downarrow$ 365)
1	$\rightsquigarrow$	$(* 365 (* 135 \text{ minutes-in-a-day}))$	
2	$\rightsquigarrow$	$(* 365 (* 135 1440))$	
3	$\rightsquigarrow$	$(* 365 194400)$	
4	$\rightsquigarrow$	70956000	

Beispiel: Das Parkplatz-Puzzle

Erinnert euch: Anzahl der PKWs $P(n,r)$ auf einem Parkplatz mit n Fahrzeugen und r Rädern: $P(n,r) = (r - 2 \times n) / 2$.

- Eine mögliche Definition der Funktion P in Scheme:

```
(define parking-lot-cars
  (lambda (n r)
    (/ (- r (* 2 n))
       2)))
```

- Applikation der Funktion in der REPL:

```
(parking-lot-cars 6 16)      ; 6 Fahrzeuge mit 16 Rädern
↪ 2                         ; 2 Autos + 4 Motorräder
```

8 | Scheme: Kommentare

☹ Der Code von `parking-lot-cars` allein lässt (bspw. für ein Team-Mitglied) Fragen offen:

- Was genau ist `n`?
- Was genau bezeichnet Parameter `r`?
- Wie ist das Ergebnis der Funktion zu interpretieren?

Kommentare annotieren Programme, um solche Fragen zu klären.

In Scheme leitet ein Semikolon (`;`) einen **Kommentar** ein, der bis zum Zeilenende reicht und vom System bei der Auswertung vollkommen ignoriert wird.

☆ Funktionsdefinitionen sollte eine ein- bis zweizeilige **Kurzbeschreibung** als Kommentar vorangestellt werden.

Kommentare...

“Programs must be written for people to read, and only incidentally for machines to execute.”

— H. Abelson and G. Sussman, *The Structure and Interpretation of Computer Programs*

~~*“Real programmers don't comment their code. If it was hard to write, it should be hard to understand.”*~~

— Tom van Vleck

9 | Scheme: Signaturen

Eine **Signatur** prüft, ob ein Name *id* an einen Wert einer angegebenen Sorte *sig* gebunden wird:

```
(: id sig)
```

(: ...) ist eine Spezialform mit Effekt: **Signaturüberprüfung**.

Racket kennt eine Reihe eingebauter Signaturen:

Signatur <i>sig</i>	akzeptierte Werte
<code>natural</code>	$\mathbb{N}_0$ (natürliche Zahlen mit 0)
<code>integer</code>	$\mathbb{Z}$ (ganze Zahlen)
<code>rational</code>	$\mathbb{Q}$ (rationale Zahlen, Brüche)
<code>real</code>	$\mathbb{R}$ (reelle Zahlen)
<code>number</code>	$\mathbb{C}$ (komplexe Zahlen, umfasst $\mathbb{N}_0$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$)
<code>boolean</code>	<code>#t</code> , <code>#f</code> (Wahrheitswerte, Booleans)
<code>string</code>	Zeichenketten
<code>image</code>	Bilder

Scheme: Signaturen für Funktionen

Funktionssignaturen der Form `(... -> ...)` legen Signaturen sowohl für die Parameter $p_1, p_2, \dots, p_n$ als auch für den Resultatwert einer Funktion fest:

```
(sig1 sig2 ... sign -> sig-result)
```

- Signaturen werden *bei jeder Anwendung* der zugehörigen Funktion erneut überprüft:
 - Passt das *i*te Argument zur Signatur *sig_i* und
 - passt das Resultat der Funktion zu *sig-result*?



Signaturverletzungen werden von DrRacket protokolliert, aber das Programm wird weiter ausgeführt. ☹️

10 | Scheme: Testfälle

Testfälle dokumentieren das erwartete Ergebnis eines Ausdrucks nach Auswertung:

(check-expect e_1 e_2) ; *ergibt e_1 den erwarteten Wert e_2 ?*

- **(check-expect ...)** ist **Spezialform** mit folgendem Effekt:
 1. Werte Ausdruck e_1 aus,
 2. teste ob der erhaltene Wert unserer Erwartung (= Wert des Ausdrucks e_2) entspricht,
 3. protokolliere positive/negative Testergebnisse.



Stelle Funktionsdefinitionen direkt mehrere Testfälle für *interessante* Argumente (bspw. Randfälle) voran.

11 | Scheme: Konstruktionsanleitung für Funktionen

☆ Zusammen ergeben diese Elemente von Scheme eine erste **Konstruktionsanleitung für eine Funktion f** :

```
; ...  
(:  $f$  (... -> ...))  
(check-expect ( $f$  ...) ...)  
:  
(check-within ( $f$  ...) ...)  
(define  $f$   
  (lambda ( $p_1$   $p_2$  ...)  
    ...))
```

- 1 Kurzbeschreibung (Kommentar)
- 2 Funktionssignatur
- 3 Testfälle für f
- 4 Definition von f
- 5 Funktionsrumpf programmieren

- Im Laufe des Semesters werden wir diese Konstruktionsanleitung systematisch verfeinern.