Informatik 1

03 – Reduktion | Lexikalische Bindung | Fallunterscheidung

Winter 2026/27

Torsten Grust
Universität Tübingen

Scheme: Reduktion ($\rightsquigarrow$)

Scheme ist gleichzeitig eine **ausdrucksstarke** und **konzeptuell einfache** Programmiersprache. Lasst uns den zweiten Punkt näher beleuchten.

- Kern von Scheme ist die **Reduktion** ($\rightsquigarrow$) von Ausdrücken e.
- Verabredung: $[\![e]\!]$ bezeichnet den **Wert,** der durch Reduktion von Ausdruck e entsteht. Es gilt also: $e \rightsquigarrow [\![e]\!]$.

Sprechweise: Die **Auswertung** $e \rightsquigarrow [\![e]\!]$ **reduziert** Ausdruck e zu seinem Wert $[\![e]\!]$ (auch: $[\![e]\!]$ ist die **Semantik** von e).

1 Scheme: Notation der Reduktionsregeln

Wir definieren jetzt $[\![\cdot]\!]$ durch sechs **Reduktionsregeln.** Das braucht nicht mehr als zwei Slides…

- Wir werden die Klammern $[\![\cdot]\!]^k$ jeweils mit einer Zahl $k \geqslant 0$ annotieren:
  - Die Reduktion von Ausdruck e startet mit $[\![e]\!]^0$.
  - ⚠️ Enthält ein (schon teilweise reduzierter) Ausdruck mehrere Klammern $[\![\cdot]\!]^k$, wird jeweils **die Klammer mit dem maximalen k als nächstes reduziert.**
  - Existieren mehrere solche Klammern $[\![\cdot]\!]^k$ mit maximalem k, kann eine *beliebige* dieser Klammern als nächstes reduziert werden.

Scheme: Reduktionsregeln (1)

Definiere ($\stackrel{\text{def}}{=}$) $[\![e]\!]^k$ je nach Art des Ausdrucks e:

❶ Literal ℓ (1, #t, "abc", #<function:⊗>, ...): [**eval_lit**]

$$[\![\ell]\!]^k \stackrel{\text{def}}{=} \ell$$

---

❷ Lambda-Abstraktion (**lambda** ($v_1 ... v_n$) e): [**eval_λ**]

$$[\![(\textbf{lambda}\ (v_1 ... v_n)\ e)]\!]^k \stackrel{\text{def}}{=} (\textbf{lambda}\ (v_1 ... v_n)\ e)$$

---

❸ Identifier id: [**eval_id**]

$$[\![id]\!]^k \stackrel{\text{def}}{=} [\![e]\!]^k \text{ , falls } (\textbf{define}\ id\ e) \text{ gilt}$$

Scheme: Reduktionsregeln (2)

Definiere $[\![e]\!]^k$ je nach Art des Ausdrucks e:

4 **Applikation** $(f\ e_1\ \ldots\ e_n)$

$$[\![(f\ e_1\ \ldots\ e_n)]\!]^k$$

$$\stackrel{\text{def}}{=} \begin{cases} [\![[\![e_1]\!]^{k+1} \otimes \cdots \otimes [\![e_n]\!]^{k+1}]\!]^k\ , & \text{[apply_prim]} \\ \quad \text{falls } f = \#\text{<function:}\otimes\text{>} & \\ [\![e\{v_1 \to [\![e_1]\!]^{k+2}, \ldots, v_n \to [\![e_n]\!]^{k+2}\}^{k+1}]\!]^k\ , & \text{[apply_}\lambda\text{]} \\ \quad \text{falls } f = (\textbf{lambda}\ (v_1 \ldots v_n)\ e) & \\ [\![([\![f]\!]^{k+1}\ e_1\ \ldots\ e_n)]\!]^k\ , \text{ sonst} & \text{[apply]} \end{cases}$$

- $e\{v_1 \to e_1, \ldots, v_n \to e_n\}$ bedeutet: In Ausdruck e wird Identifier v_i durch Ausdruck e_i ersetzt.
  Beispiel: $(+\ (*\ 4\ y)\ x)\{x \to 2, y \to 10\} = (+\ (*\ 4\ 10)\ 2)$.

Scheme: Beispiele für Reduktion

Reduktion von `(+ 40 2)`:

$[\![(+\ 40\ 2)]\!]^0$	[**apply**]
$= [\![([\![+]\!]^1\ 40\ 2)]\!]^0$	[**eval_id:**+]
$= [\![([\![\#\texttt{<function:+>}]\!]^1\ 40\ 2)]\!]^0$	[**eval_lit:**#<function:+>]
$= [\![(\#\texttt{<function:+>}\ 40\ 2)]\!]^0$	[**apply_prim**]
$= [\![[\![40]\!]^1 + [\![2]\!]^1]\!]^0$	[**eval_lit:**40, **eval_lit:**2]
$= [\![40 + 2]\!]^0$	[Arithmetik: +]
$= [\![42]\!]^0$	[**eval_lit:**42]
$= 42.$	

Scheme: Beispiele für Reduktion

```
(define sqr                          ; ← sei diese Definition
  (lambda (x)                        ;    von sqr gegeben
    (* x x)))
```

Reduktion von (sqr 9):

$[\![(\text{sqr } 9)]\!]^0$	[**apply**]
$= [\![([\![\text{sqr}]\!]^1\ 9)]\!]^0$	[**eval_id:**sqr]
$= [\![([\![(\text{lambda } (x)\ (* \ x\ x))]\!]^1\ 9)]\!]^0$	[**eval_λ**]
$= [\![((\text{lambda } (x)\ (* \ x\ x))\ 9)]\!]^0$	[**apply_λ**]
$= [\![(* \ x\ x)\{x \to [\![9]\!]^2\}^1]\!]^0$	[**eval_lit:**9,{·→·}]
$= [\![(* \ 9\ 9)]\!]^0$	…

Scheme: Beispiele für Reduktion

… $[\![(* \ 9 \ 9)]\!]^0$	[**apply**]
$= [\![([\![*]\!]^1 \ 9 \ 9)]\!]^0$	[**eval_id:***]
$= [\![([\![\#\texttt{<function:*>}]\!]^1 \ 9 \ 9)]\!]^0$	[**eval_lit:**#<function:*>]
$= [\![(\#\texttt{<function:*>} \ 9 \ 9)]\!]^0$	[**apply_prim**]
$= [\![[\![9]\!]^1 * [\![9]\!]^1]\!]^0$	[2 × **eval_lit:**9]⊠
$= [\![9 * 9]\!]^0$	[Arithmetik: *]
$= [\![81]\!]^0$	[**eval_lit:**81]
$= 81.$	

(⊠ Erneute Auswertung von $[\![9]\!]$ ist der Einfachheit unserer Regeln geschuldet. Scheme wertet $[\![9]\!]$ nur einmal aus.)

2 Lexikalische Bindung

Q: Bezeichnen diese beiden Lambda-Abstraktionen eigentlich die gleiche Funktion?

```
(lambda (x) (* x x))        (lambda (r) (* r r))
```

A: Ja! Die Wahl des Parameternamens `x` bzw. `r` hat keinen Einfluss auf die Semantik der Lambda-Abstraktionen. Beide implementieren die `sqr`-Funktion.

☆ Diese Unabhängigkeit von der Wahl des Identifiers ist wünschenswert. Das hat aber Konsequenzen für Reduktionsregel **apply_λ**, wenn Argumente e_i für Parameter v_i eingesetzt werden.

DrRacket: Reduktionen im Stepper

```
; π
(: pi real)
(define pi 3.14159265)

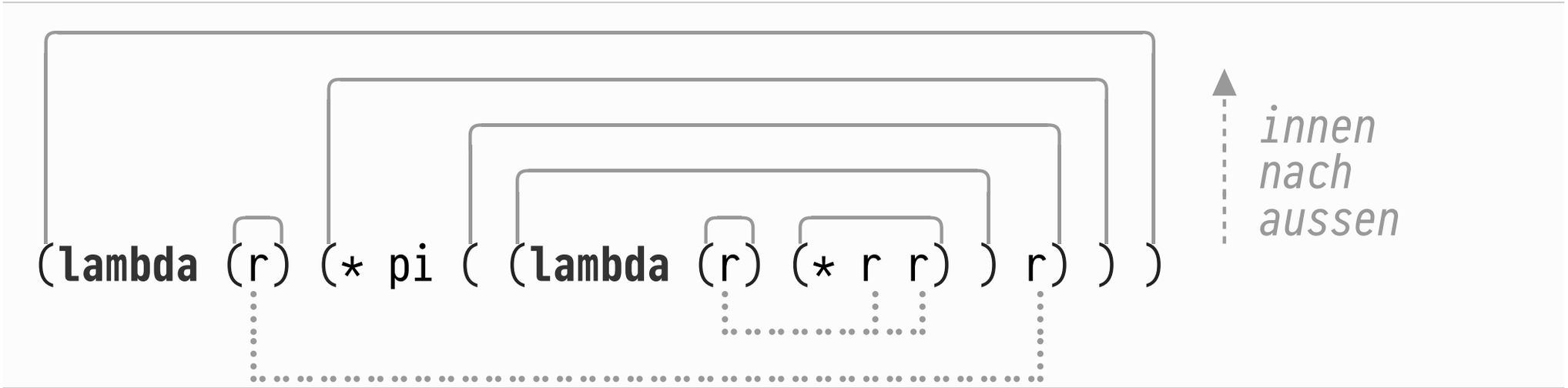
; Fläche des Kreises mit Radius r (πr²)
(: circle-area (real -> real))
(define circle-area
  (lambda (r)
    (* pi
       ((lambda (□) (* □ □)) r))))
                  ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
                       sqr
```

- Jetzt: Reduziere Applikation `(circle-area 10)` im **Stepper** mittels Button [Stepper ▶|] .
- Die Wahl des Identifiers □ darf die Auswertung *nicht* beeinflussen.

Scheme: Prinzip der Lexikalischen Bindung

Das **bindende Vorkommen** (≡ Definition) eines Identifiers *x* kann im Programmtext systematisch bestimmt werden:

- Suche von *x* aus von “innen nach aussen” bis zum ersten
  1. **lambda** (... *x* ...) oder
  2. (**define** *x* ...).



3 ┆ A Mystery…

Hmm… Was berechnet die folgende Funktion mystery?

```
; … ?
(: mystery (… … -> …))
(define mystery
    (lambda (x₁ x₂)
      (/ (+ x₁
            x₂
            (abs (- x₁ x₂)))
         2)))
```

- Der Body von mystery in mathematischer Notation:

$$\frac{x_1 + x_2 + |x_1 - x_2|}{2}$$

Fallunterscheidungen

Ein kritischer Blick 🤔 und Testen zeigt: (mystery x_1 x_2) ≡ (maximum x_1 x_2).

Funktion maximum realisiert eine **Fallunterscheidung,** für die die Mathematik spezifische Notationen kennt. Etwa:

$$\text{maximum}(x_1,x_2) = \begin{cases} x_1 & \textit{falls } x_1 \geqslant x_2 \quad \leftarrow \textbf{Zweig (branch)} \\ x_2 & \textit{sonst} \quad \leftarrow \textbf{Zweig} \end{cases}$$

Test

Die geschweifte Klammer { symbolisiert eine **binäre Fallunterscheidung** (auch: **two-way conditional**).

Scheme: Prädikate

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

```
(: =          (number number   -> boolean))
(: <          (number number   -> boolean))  ; >, <=, >=
(: string=?   (string string   -> boolean))  ; string>?, ...
(: boolean=?  (boolean boolean -> boolean))
(: zero?      (number          -> boolean))
(: odd?       (number          -> boolean))  ; even?
(: positive?  (number          -> boolean))  ; negative?
                    ⋮
```

- `(string>? s₁ s₂)` vergleicht s_1 und s_2 *lexikographisch*:

```
                  ┌─────── > ──────┐
(string>? "Star_Wars" "Star_Trek") ↠ #t  ; "W" > "T"
          └─────── = ──────┘
```

Scheme: Fallunterscheidung

Spezialform zur Fallunterscheidung [(*n*+1)-way conditional]:

```
(cond
  (t₁   e₁)
  (t₂   e₂)
    ⋮
  (tₙ   eₙ)
  (else eₙ₊₁) )  [← optional]
```

Mathematik:

$$\begin{cases} e_1 & \text{falls } t_1 \\ e_2 & \text{falls } t_2 \\ \vdots & \vdots \\ e_n & \text{falls } t_n \\ e_{n+1} & \text{sonst} \end{cases}$$

1. Führe **Tests** in Reihenfolge t_1, t_2, … durch.
2. Ergibt Test t_i dabei #t, werte **Zweig** e_i aus. Der Wert von e_i ist das Ergebnis der Fallunterscheidung.
3. Falls auch t_n zu #f auswertet, dann …

$$\begin{cases} \text{ist } e_{n+1} \text{ das Ergebnis} & \text{falls } \textbf{else}\text{-Zweig existiert} \\ \text{Programmabbruch } \lightning & \text{sonst} \end{cases}$$

Scheme: Signatur enum

Die Signatur enum lässt genau einen der aufgezählten n Werte (Ausdrücke) e_i zu:

```
(enum e1 e2 ... en)
```

Beispiele:

- Signatur für Wochentage (dargestellt durch Werte der Signatur string):

```
(enum "Mo" "Di" "Mi" "Do" "Fr" "Sa" "So")
```

- Die beiden folgenden Signaturen sind äquivalent (≡):

```
boolean ≡ (enum #t #f)
```

Scheme: Reduktion von cond

Neue **Reduktionsregeln für cond** (ergänzt ?...? oben):

5 Fallunterscheidung $(\textbf{cond}\ (t_1\ e_1)\ (t_2\ e_2)\ \ldots)$

- $[\![(\textbf{cond}\ (\text{else}\ e_1))]\!]^k \quad \stackrel{\text{def}}{=} [\![e_1]\!]^k \quad [\textbf{cond_else}]$
- $[\![(\textbf{cond}\ (\#\text{t}\ e_1)\ (t_2\ e_2)\ \ldots)]\!]^k \stackrel{\text{def}}{=} [\![e_1]\!]^k \quad ⚐ \quad [\textbf{cond_\#t}]$
- $[\![(\textbf{cond}\ (\#\text{f}\ e_1)\ (t_2\ e_2)\ \ldots)]\!]^k \quad ⚑ \quad [\textbf{cond_\#f}]$
  $\stackrel{\text{def}}{=} [\![(\textbf{cond}\ (t_2\ e_2)\ \ldots)]\!]^k$

- $[\![(\textbf{cond})]\!]^k \quad \stackrel{\text{def}}{=}$ Abbruch ↯ $\quad [\textbf{cond_fail}]$

- $[\![(\textbf{cond}\ (t_1\ e_1)\ (t_2\ e_2)\ \ldots)]\!]^k \quad [\textbf{cond}]$
  $\stackrel{\text{def}}{=} [\![(\textbf{cond}\ ([\![t_1]\!]^{k+1}\ e_1)\ (t_2\ e_2)\ \ldots)]\!]^k$

- **NB:** cond ist “faul” bei der Auswertung:
  - ⚐: t_2, e_2, t_3, e_3, … nie ausgewertet.
  - ⚑: e_1 nie ausgewertet (“*der Test t_1 beschützt e_1*”).

Scheme: Binäre Fallunterscheidung (Syntaktischer Zucker **if**)

Einige Konstellationen (auch: **Idiome**) von Scheme-Konstrukten treten so häufig auf, dass sie eine eigene Abkürzung verdienen:

Fallunterscheidung mit genau zwei Zweigen (Abkürzung **if**):

```
(if t1          (cond (t1   e1)
    e1     ≡          (else e2))
    e2)
```

- `(if ... ... ...)` ist *keine* eigene Spezialform, sondern kürzt **cond** lediglich ab ⇒ `if` ist genau so faul wie **cond**.
- Solche Abkürzungen sind auch als **syntaktischer Zucker** 🍭 (auch: **syntactic sugar** oder **derived form**) bekannt.

Scheme: Komplexe Prädikate (Syntaktischer Zucker **or**, **and**)

or und and konstruieren **komplexe Prädikate** (Tests mit Ergebnis der Signatur boolean) aus einfachen Tests t_i.

- or und and sind als syntaktischer Zucker 🍭 definiert:

```
; Trifft mindestens einer der n Tests t_i zu?
(or t_1 t_2 ... t_n)   ≡  (if t_1 #t (or t_2 ... t_n))
(or)                   ≡  #f

; Treffen alle n Tests t_i zu?
(and t_1 t_2 ... t_n)  ≡  (if t_1 (and t_2 ... t_n) #f)
(and)                  ≡  #t
```

- **Beispiel:** (and (>= h 0) (< h 12)) testet, ob h ∈ [0,12). (👆 Training: Expandiere (and (>= h 0) (< h 12)) bis jeglicher syntaktischer Zucker entfernt ist.)