2 Erste Schritte in Scheme

Die Programmiersprache Scheme

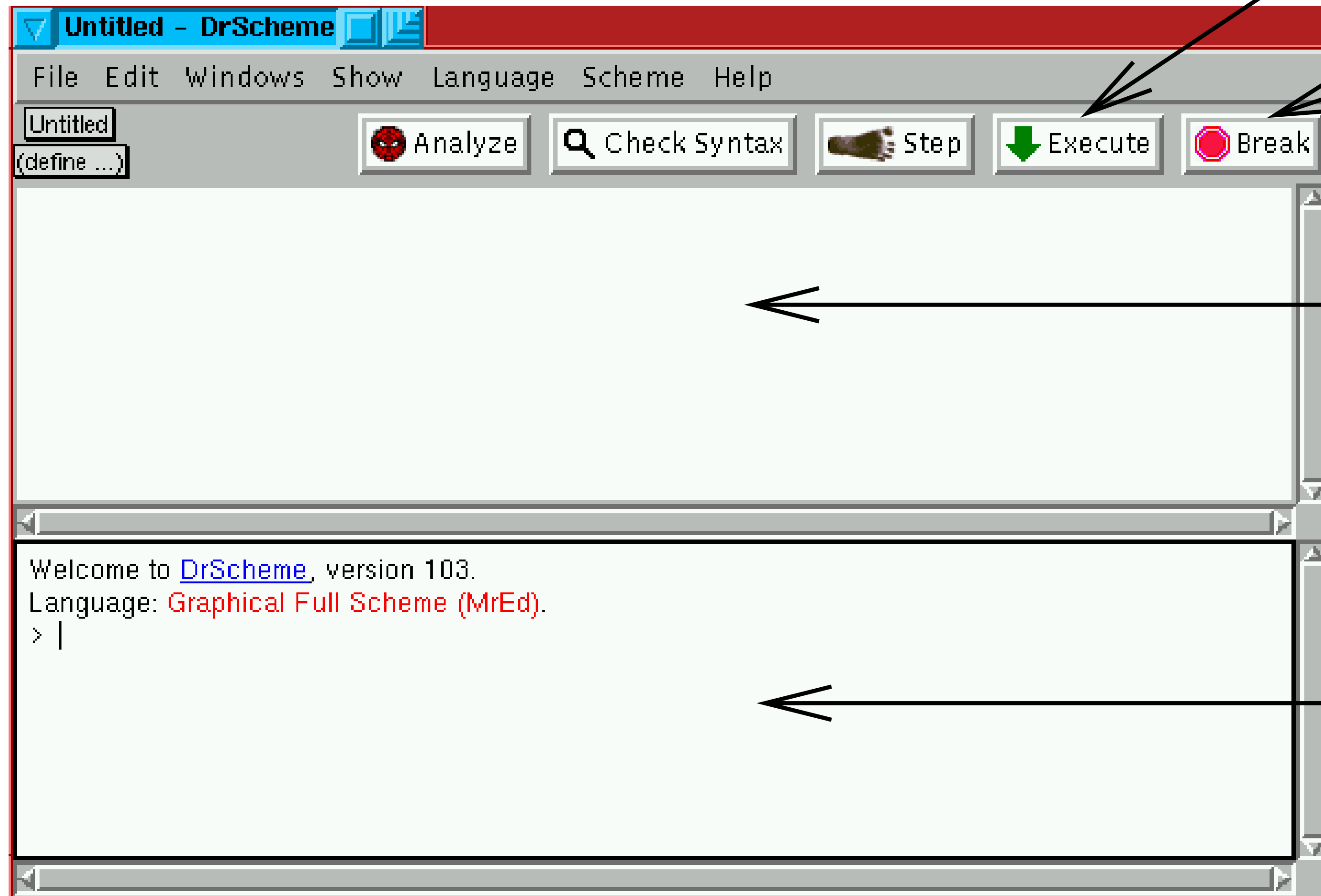
- geboren 1975
- Eltern: Gerald Jay Sussman and Guy Lewis Steele Jr.
- Ort: Massachusetts Institute of Technology
- aktuelle Beschreibung: R5RS (Februar 1998)
  Revised^5 *Report on the Algorithmic Language Scheme*

Scheme ist besonders geeignet zur Ausbildung, denn

- Scheme ist einfach: einmal gelernt, nie wieder vergessen
- Scheme ist klein: die Sprachdefinition umfasst 50 Seiten
- Scheme ist mächtig: alle Programmierkonzepte lassen sich in Scheme demonstrieren

 © 2001 Peter Thiemann

2.1 DrScheme: Die Programmierumgebung



Laden eines Programms

Abbruch eines Programms

Programmeditor

REPL

(read–eval–print–loop)

 © 2001 Peter Thiemann

2.2 Sprache

2.2.1 Aspekte einer Sprache

Syntax Regeln zur Kombination von Zeichen (Bildung von Wörtern, Sätzen, usw)

Semantik Bedeutung; Beziehung Zeichen und bezeichneten Objekten

Pragmatik Beziehung zwischen Zeichen und dem Anwender der Zeichen

2.2.2 Syntax einer Programmiersprache

formale Sprache mit genauer Definition

Literale Zeichen mit fester Bedeutung

Kombinationen zum Zusammensetzen von Zeichen zu grösseren Zeichen

Abstraktionsmittel zum Benennen (Abkürzen) von Zeichen

 © 2001 Peter Thiemann

2.3 Syntax von Scheme

Konventionen

- Ein *Kommentar* beginnt mit dem Zeichen ; und endet mit dem Zeilenende.
- Leerzeichen, Zeilenumbrüche und Kommentare sind Trennzeichen ohne Bedeutung
- Grundlegendes Sprachelement: *Ausdruck* (⟨*expression*⟩)

Literale z.B. für Zahlen

`42   -17   2/3   3.1415926535`

Vordefinierte Namen z.B. für arithmetische Operationen

`+   -   *   /`

Kombinationen *Einzige Form:* (⟨*operator*⟩ ⟨*operand*⟩ . . . ⟨*operand*⟩)

```
(+ 17 4)          (* 2 (+ 17 4))
```


© 2001 Peter Thiemann

2.4 Auswertung

Jeder Ausdruck beschreibt einen *Berechnungsprozess* zur Ermittlung seines Wertes (Auswertung). Start der Auswertung durch Eingabe in das REPL-Fenster.

Konstante

```
42
```

Berechnung von $2 \cdot (17 + 4)$

```
   (* 2 (+ 17 4))
=> (* 2 21)
=> 42
```

Berechnung von $3 + 13 \cdot 3$

```
   (+ 3 (* 13 3))
=> (+3 39)
=> 42
```

Berechnung von $(2 + 2) \cdot (3 + 5) \cdot 30/10/2$

```
   (* (+ 2 2) (/ (* (+ 3 5) (/ 30 10)) 2))
=> (* 4 (/ (* (+ 3 5) (/ 30 10)) 2))
=> (* 4 (/ (* 8 (/ 30 10)) 2))
=> (* 4 (/ (* 8 3) 2))
=> (* 4 (/ 24 2))
=> (* 4 12)
=> 48
```

 © 2001 Peter Thiemann

2.5 Werte benennen

```
> (define answer 42)
> answer
42
> (define pi (* 4 (atan 1)))
> pi
3.141592653589793
```

Allgemein: `(define` ⟨*variable*⟩ ⟨*expression*⟩`)`

- Das erste Argument von `define` ist der Name einer *Variablen*.
- Das zweite Argument wird ausgewertet und der Wert wird an die Variable *gebunden*.
- Nach der Definition steht der Name der Variable für den Wert. Die Berechnung wird nicht wiederholt.
- Literale sind keine Variablennamen.
- Variablennamen können keine Trennzeichen enthalten.

 © 2001 Peter Thiemann

2.6 Variablen in Ausdrücken

Quadrieren

```
   (define x 4)
   (* x x)
=> (* 4 x)
=> (* 4 4)
=> 16
```

Problem: Ausdruck `(* x x)` enthält *freie Variable* `x`.

Er kann nur einmal verwendet werden, für einen Wert von `x`.

Lösung: *Abstraktion* von `x` führt zu einem parametrisierten Ausdruck, dem *Lambda-Ausdruck*

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

Die Variable `x` ist die *gebundene Variable* des Lambda-Ausdrucks.

Der Ausdruck `(* x x)` ist der *Rumpf* des Lambda-Ausdrucks.

Einzige Operation: Einsetzen eines Wertes für die gebundene Variable.

 © 2001 Peter Thiemann

2.7 Lambda Ausdruck

2.7.1 Verwendung als Operator

```
   ((lambda (x) (* x x)) 4)        ; Einsetzen von 4 für x
=> (* 4 4)                         ; Regel für *
=> 16
```

2.7.2 Verwendung als Wert

```
   (define square
     (lambda (x)
       (* x x)))
   (square 13)                     ; Einsetzen für square
=> ((lambda (x) (* x x)) 13)       ; Einsetzen von 13 für x
=> (* 13 13)                       ; Regel für *
=> 169

   (square 4)                      ; Einsetzen für square, s.o.
```

 © 2001 Peter Thiemann

2.8 Aufgabe: Fläche einer Scheibe

Eingabe: Radius r der Scheibe $(r > 0)$

Ausgabe: Fläche πr^2 der Scheibe

Eingabe und Ausgabe sind Zahlen, daher schreibe

```
; SIGNATUR
; disk-area : number -> number
; DEFINITION
(define disk-area
  (lambda (radius)
    (* pi (square radius))))
; BEISPIEL
(disk-area 2)
; ERGEBNIS
12.566370614359172
```

 © 2001 Peter Thiemann

2.9 Aufgabe: Rauminhalt eines Zylinders

```
; SIGNATUR
; cylinder-volume : number number -> number
; ERKLÄRUNG
; Eingabe: Radius r und Höhe h eines Zylinders
; Ausgabe: Rauminhalt πr^2h des Zylinders
; DEFINITION
(define cylinder-volume
  (lambda (radius height)
    (* (disk-area radius) height)))
; BEISPIEL
(cylinder-volume 1 1)
; ERGEBNIS
3.141592653589793
```

 © 2001 Peter Thiemann

2.10 Berechnungsprozess zu `cylinder-volume`

```
    (cylinder-volume 5 4)
=> ((lambda (radius height) (* (circle-area radius) height)) 5 4)
=> (* (circle-area 5) 4)
=> (* ((lambda (radius) (* pi (square radius))) 5) 4)
=> (* (* pi (square 5)) 4)
=> (* (* 3.141592653589793 ((lambda (x) (* x x)) 5)) 4)
=> (* (* 3.141592653589793 (* 5 5)) 4)
=> (* (* 3.141592653589793 25) 4)
=> (* 78.53981633974483 4)
=> 314.1592653589793
```

 © 2001 Peter Thiemann

2.11 Das Parkplatzproblem

```
; SIGNATUR
; cars-in-parking-lot : number number -> number
; ERKLÄRUNG
; Eingabe: n, r ∈ N, r gerade, 2n ≤ r ≤ 4n
; Ausgabe: ... (s.o.)
; DEFINITION
(define cars-in-parking-lot
  (lambda (nr-of-vehicles nr-of-wheels)
    (/ (- nr-of-vehicles
          (* nr-of-wheels 2))
       2)))
```

 © 2001 Peter Thiemann

MANTRA

Mantra #1 (Strukturerhaltung)
Strukturiere das Programm nach dem Problem

Mantra #2 (Abstraktion)
Schreibe Abstraktionen für Teilprobleme

Mantra #3 (Namen)
Benenne Konstanten

 © 2001 Peter Thiemann

2.12 Definition durch Fallunterscheidung

Viele Funktionen passen nicht in das bisherige Muster:

- $\max(x, y) = \begin{cases} x & x \geq y \\ y & x < y \end{cases}$
- $d(x) = \begin{cases} 1 & x = 0 \\ 0 & x \neq 0 \end{cases}$
- $s(x) = -(y - 1)$ falls $x = 2z + y$ und $z \in \mathbf{Z}$ und $0 \leq y < 2$

Um sie zu definieren, müssen Bedingungen (wie $x = 0$) getestet werden.

 © 2001 Peter Thiemann

2.13 Der Datentyp `boolean` (Wahrheitswerte)

Literale:

```
#t      ; wahr
#f      ; falsch
```

Operationen:

- vom Typ `number number -> boolean`

  `=` `<` `>` `>=` `<=`

- numerische Prädikate vom Typ `number -> boolean`

  `zero?` `positive?` `negative?` `odd?` `even?`

- logische Operationen

  `not` `and` `or`

 © 2001 Peter Thiemann

2.14 Ausdrücke vom Typ `boolean`

```
    #t
=> #t

    (= 17 4)
=> #f

    (>= 17 4)
=> #t

    (odd? (+ 17 4))
=> (odd? 21)
=> #t
```

```
    (define y 1)
    (and (= 5 (+ (* 2 2) y)) (and (<= 0 y) (< y 2)))
=> (and (= 5 (+ 4 y)) (and (<= 0 y) (< y 2)))
=> (and (= 5 (+ 4 1)) (and (<= 0 y) (< y 2)))
=> (and (= 5 5) (and (<= 0 y) (< y 2)))
=> (and #t (and (<= 0 y) (< y 2)))
=> (and (<= 0 y) (< y 2))
=> (and (<= 0 1) (< y 2))
=> (and #t (< y 2))
=> (< y 2)
=> (< 1 2)
=> #t
```


© 2001 Peter Thiemann

2.15 Bedingte Ausdrücke

Format: `(if` ⟨*expression*⟩ ⟨*expression*⟩ ⟨*expression*⟩`)`

1. Ausdruck: Bedingung
2. Ausdruck: auszuwerten, falls Bedingung wahr
3. Ausdruck: auszuwerten, falls Bedingung falsch

```
(if (not (zero? x)) (/ 1 x) 0)
```

Auswertung:

```
   (define x 7)
   (if (not (zero? x)) (/ 1 x) 0)
=> (if (not (zero? 7)) (/ 1 x) 0)
=> (if (not #f) (/ 1 x) 0)
=> (if #t (/ 1 x) 0)
=> (/ 1 x)
=> (/ 1 7)
=> 1/7
```

 © 2001 Peter Thiemann

2.16 Bedingte Ausdrücke in Funktionen

- $\max(x,y) = \begin{cases} x & x \geq y \\ y & x < y \end{cases}$

```
(define maximum
  (lambda (x y)
    (if (>= x y) x y)))
```

- $d(x) = \begin{cases} 1 & x = 0 \\ 0 & x \neq 0 \end{cases}$

```
(define dirac
  (lambda (x)
    (if (zero? x) 1 0)))
```

- $s(x) = -(y-1)$ falls $x = 2z + y$ und $z \in \mathbf{Z}$ und $0 \leq y < 2$

  . . . später

Beispiel

```
   (maximum 4711 1024)
=> ((lambda (x y) (if (>= x y) x y)) 4711 1024)
=> (if (>= 4711 1024) 4711 1024)
=> (if #t 4711 1024)
=> 4711
```

 © 2001 Peter Thiemann

2.17 Wiederholte Berechnungen: Der let-Ausdruck

```
(define square-sum
  (lambda (x y)
    (* (+ x y) (+ x y))))
```

wiederholt die Auswertung von `(+ x y)`.
Verbesserung durch benanntes Zwischenergebnis:

```
(let ((sum (+ x y))) (* sum sum))
```

Format: `(let ((`$\langle$*variable*$\rangle$ $\langle$*expression*$\rangle_1$`))` $\langle$*expression*$\rangle_2$`)`

- erster Ausdruck $\langle$*expression*$\rangle_1$ wird ausgewertet
- Wert des ersten Ausdrucks wird für Variable im zweiten Ausdruck eingesetzt
- zweiter Ausdruck $\langle$*expression*$\rangle_2$ wird ausgewertet und liefert Wert des gesamten Ausdrucks

$\Rightarrow$ $\langle$*variable*$\rangle$ ist *lokale Variable*, die nur in $\langle$*expression*$\rangle_2$ bekannt ist!

 © 2001 Peter Thiemann

2.18 Wiederholte Berechnungen vermeiden

```
(define square-sum
  (lambda (x y)
    (let ((sum (+ x y)))
      (* sum sum))))
```

```
   (square-sum 4 3)
=> ((lambda (x y) (let ((sum (+ x y))) (* sum sum))) 4 3)
=> (let ((sum (+ 4 3))) (* sum sum))
=> (let ((sum 7)) (* sum sum))
=> (* 7 7)
=> 49
```

 © 2001 Peter Thiemann

2.19 Let-Ausdrücke sind überflüssig

. . . aber bequem!

⇒ *syntaktischer Zucker*

- erleichtert das Programmieren
- kann aber durch Kombination anderer Ausdrücke beschrieben werden

Alternative Definition von let

(let ((x $\langle$*expression*$\rangle_1$)) $\langle$*expression*$\rangle_2$)

$\equiv$

((lambda (x) $\langle$*expression*$\rangle_2$) $\langle$*expression*$\rangle_1$)

 © 2001 Peter Thiemann

2.20 Let*-Ausdrücke

Ein geschachtelter let-Ausdruck

```
(let ((x (- a b)))
  (let ((y (- b c)))
    (let ((z (- c d)))
      (+ (* x y) (* y z) (* z x)))))
```

kann zur besseren Lesbarkeit als let*-Ausdruck geschrieben werden:

```
(let* ((x (- a b))
       (y (- b c))
       (z (- c d)))
  (+ (* x y) (* y z) (* z x)))
```

 © 2001 Peter Thiemann

2.21 Zusammenfassung: Berechnungsmodell für Scheme

- bestimmt den Wert eines Ausdrucks
- beruht auf Einsetzen von Werten für Variable
- $\Rightarrow$ *Substitutionsmodell*
- Berechnungsregeln für jede Art von Ausdruck
- Format:
  1. Auswerten von Teilausdrücken
  2. Ersetzungsschritt

 © 2001 Peter Thiemann

2.22 Zusammenfassung: Berechnungsregeln

- Ein Literal ist ein Wert.
- Ein Lambda-Ausdruck ist ein Wert.
- Eine Variable wird durch ihren definierenden Wert ersetzt.
- Zur Berechnung des Werts einer Kombination

  `(`$\langle$*operator*$\rangle$ $\langle$*operand*$\rangle_1$ ... $\langle$*operand*$\rangle_n$`)`

  wird zuerst der Wert v_0 von $\langle$*operator*$\rangle$, sowie die Werte $v_1, \ldots, v_n$ der Operanden bestimmt.

  1. Ist v_0 primitiver Operator, so wird er auf $v_1, \ldots, v_n$ angewendet.
  2. Ist $v_0 = (\mathtt{lambda}(\mathtt{x_1} \ldots \mathtt{x_n})\mathtt{e})$, so wird in e jedes freie Vorkommen von x_1 durch v_1, x_2 durch v_2 usw. ersetzt und der Wert des entstehenden Ausdrucks ermittelt.

- Zur Berechnung des Werts von `(if` $\langle$*test*$\rangle$ $\langle$*consequent*$\rangle$ $\langle$*alternate*$\rangle$`)` wird zuerst der Wert v von $\langle$*test*$\rangle$ bestimmt.

  1. Ist $v =$ `#t`, so wird der Wert von $\langle$*consequent*$\rangle$ ermittelt.
  2. Ist $v =$ `#f`, so wird der Wert von $\langle$*alternate*$\rangle$ ermittelt.

 © 2001 Peter Thiemann