

1 Erste Schritte in Scheme

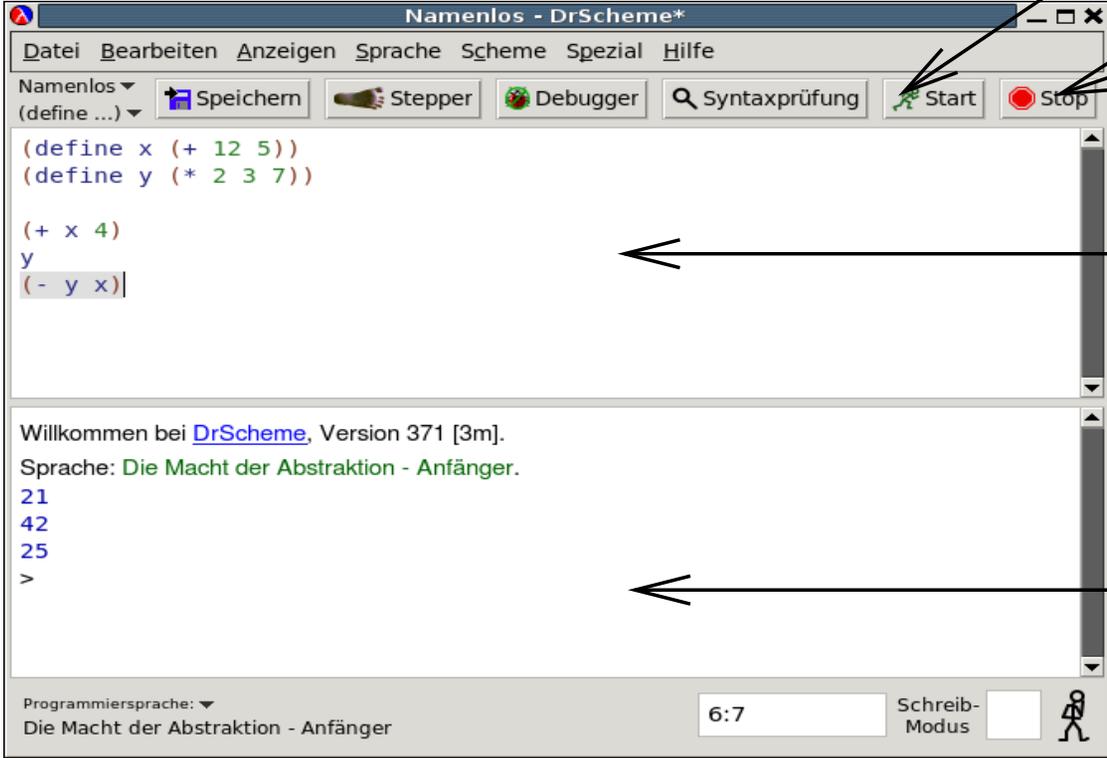
Die Programmiersprache Scheme

- geboren 1975
- Eltern: Gerald Jay Sussman and Guy Lewis Steele Jr.
- Ort: Massachusetts Institute of Technology
- aktuelle Beschreibung: R6RS (September 2007)
Revised⁶ 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 90+70 Seiten
- Scheme ist mächtig: alle Programmierkonzepte lassen sich in Scheme demonstrieren

1.1 DrScheme: Die Programmierumgebung



The screenshot shows the DrScheme IDE window titled "Namenlos - DrScheme*". The interface is divided into several sections:

- Toolbar:** Contains buttons for "Speichern", "Stepper", "Debugger", "Syntaxprüfung", "Start", and "Stop".
- Code Editor:** Displays Scheme code:

```
(define x (+ 12 5))  
(define y (* 2 3 7))  
  
(+ x 4)  
y  
(- y x)
```
- REPL (Interaktionsfenster):** Shows the output of the program:

```
Willkommen bei DrScheme, Version 371 [3m].  
Sprache: Die Macht der Abstraktion - Anfänger.  
21  
42  
25  
>
```
- Status Bar:** Shows "Programmiersprache: Die Macht der Abstraktion - Anfänger", a cursor position of "6:7", and a "Schreib-Modus" button.

Annotations with arrows point to the following elements:

- "Laden eines Programms" points to the "Start" button.
- "Abbruch eines Programms" points to the "Stop" button.
- "Programmeditor (Definitionen)" points to the code editor area.
- "Interaktionsfenster REPL (read-eval-print-loop)" points to the REPL window.

1.2 Sprache

1.2.1 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

1.2.2 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

1.3 Syntax von Scheme, Grundbegriffe

Grundbegriffe

- Ein *Programm* ist eine Folge von *Formen*.
- Formen können sein
 - *Definitionen*
 - *Ausdrücke*
- Ausdrücke ($\langle expression \rangle$) haben einen *Wert*, sie können *ausgewertet* werden.

Konventionen

- Ein *Kommentar* beginnt mit dem Zeichen ; und endet mit dem Zeilenende.
- Leerzeichen, Zeilenumbrüche und Kommentare sind Trennzeichen ohne Bedeutung

1.4 Syntax von Scheme, Ausdrücke

Literale z.B. für Zahlen

42 -17 2/3 3.1415926535

Vordefinierte Namen z.B. für arithmetische Operationen

+ - * /

(Funktions-) Anwendung, Applikation

(*operator* *operand* ... *operand*)

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

1.5 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`

1.6 Werte benennen

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

Allgemein: `(define <variable> <expression>)`

- *Spezialform* eingeleitet durch *Schlüsselwort* `define`.
- Erster Operand: Name einer *Variablen*.
- Zweiter Operand: ein Ausdruck.
- Diese *Bindung* bindet den Namen der Variable an den *Wert des Ausdrucks*. Nun steht der Name der Variable für den Wert. Die Berechnung wird nicht wiederholt.
- Literale sind keine Variablennamen.
- Variablennamen können keine Trennzeichen enthalten.

1.7 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* (*Abstraktion, Prozedur*)

```
(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: Applikation setzt einen Wert für die gebundene Variable ein.

1.8 Lambda Ausdruck und Funktionsanwendung

1.8.1 Verwendung als Operator

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

1.8.2 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.
```

1.9 Auswertung der Funktionsanwendung

Zur Auswertung von

$(\langle operator \rangle \langle operand \rangle_1 \dots \langle operand \rangle_n)$

wird zuerst der Wert v_0 von $\langle operator \rangle$, sowie die Werte v_1, \dots, v_n der Operanden bestimmt. Dies sind die *Argumente* der Funktionsanwendung.

Der *Rückgabewert* bestimmt sich wie folgt.

1. Ist v_0 primitiver Operator, so wird er auf v_1, \dots, v_n angewendet.
2. Ist $v_0 = (\text{lambda } (x_1 \dots x_n) e)$, 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.

1.10 Sorten und Verträge

1.10.1 Aufgabe: Fläche einer Scheibe

Definiere eine Prozedur mit folgenden Eigenschaften

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

Ausgabe: Fläche πr^2 der Scheibe

Aufgabe: Fläche einer Scheibe, I

Eingabe und Ausgabe sind Zahlen, d.h. ihre *Sorte* ist `number`. Der Vorspann der Prozedurdefinition besteht aus *Kurzbeschreibung* und *Vertrag*:

```
; Fläche einer Scheibe berechnen  
(: disk-area (number -> number))
```

Daraus ergibt sich folgendes *Gerüst* für die Definition

```
(define disk-area  
  (lambda (radius)  
    ...))
```

Testfälle:

```
(check-expect (disk-area 0) 0)  
(check-within (disk-area 1) 3.14159 1e-5)  
(check-within (disk-area 2) 12.56637 1e-5)
```

Aufgabe: Fläche einer Scheibe, II

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

Ausgabe: Fläche πr^2 der Scheibe

; Fläche einer Scheibe berechnen

```
(: disk-area (number -> number))
```

```
(define disk-area
```

```
  (lambda (radius)
```

```
    (* pi (square radius))))
```

; Testfälle

```
(check-expect (disk-area 0) 0)
```

```
(check-within (disk-area 1) 3.14159 1e-5)
```

```
(check-within (disk-area 2) 12.56637 1e-5)
```

1.10.2 Das Parkplatzproblem

Eingabe: $n, r \in \mathbf{N}$, r gerade, $2n \leq r \leq 4n$

Ausgabe: $P(n, r) = r/2 - n$

Vertrag und sich daraus ergebendes Gerüst:

; Parkplatzproblem lösen

```
(: cars-in-parking-lot (natural natural -> natural))
```

```
(define cars-in-parking-lot
```

```
  (lambda (nr-of-vehicles nr-of-wheels)
```

```
    ...))
```

Testfälle

```
(check-expect (cars-in-parking-lot 0 0) 0)
```

```
(check-expect (cars-in-parking-lot 1 4) 1)
```

```
(check-expect (cars-in-parking-lot 2 4) 0)
```

1.10.3 Das Parkplatzproblem, II

Eingabe: $n, r \in \mathbf{N}$, r gerade, $2n \leq r \leq 4n$

Ausgabe: $P(n, r) = r/2 - n$

Fertiges Programm durch Einsetzen der Formel

```
; Parkplatzproblem lösen
(: cars-in-parking-lot (natural natural -> natural))
(define cars-in-parking-lot
  (lambda (nr-of-vehicles nr-of-wheels)
    (- (/ nr-of-wheels 2) nr-of-vehicles)))
; Testfälle
(check-expect (cars-in-parking-lot 0 0) 0)
(check-expect (cars-in-parking-lot 1 4) 1)
(check-expect (cars-in-parking-lot 2 4) 0)
```

1.11 Testfälle

- Ein Testfall besteht aus der Anwendung der zu schreibenden Prozedur auf gewisse Eingaben, sowie der erwarteten Ausgabe.
- Die Ausgabe muss „von Hand“ separat berechnet werden!
- Es sollten Testfälle bereitgestellt werden für
 - Randfälle (im Parkplatzproblem $r = 2n$ und $r = 4n$)
 - Standardfälle
 - Fehlerfälle
- DrScheme unterstützt Testfälle durch
 - `(check-expect <expression> <expression>)`
 - `(check-within <expression> <expression> <expression>)`
- Testfälle sollen nie geändert bzw entfernt werden!
- Gefundener Fehler \Rightarrow neuer Testfall!

1.12 Konstruktionsanleitung (Konstruktion von Prozeduren)

(1. Annäherung)

Kurzbeschreibung Schreibe eine einzeilige Kurzbeschreibung.

Vertrag Wähle einen Namen und schreibe den Vertrag für die Prozedur.

Verwende dafür die Form (: *⟨name⟩* *⟨contract⟩*).

Gerüst Leite aus dem Vertrag das Gerüst der Prozedur her.

Testfälle Schreibe einige sinnvolle Testfälle.

Rumpf Vervollständige den Rumpf der Prozedur.

Test Prüfe, dass alle Testfälle erfolgreich ablaufen.

...

MANTRA

Mantra #1 (Vertrag vor Ausführung)

Schreibe —vor der Programmierung des Prozedurrumpfes— eine Kurzbeschreibung der Aufgabe und einen Vertrag als Kommentare ins Programm.

Mantra #2 (Testfälle)

Schreibe Testfälle *vor* dem Schreiben der Definition.

1.13 Zerlegen in Teilprobleme

1.13.1 Aufgabe: Rauminhalt eines Zylinders

Eingabe: Radius r und Höhe h eines Zylinders

Ausgabe: Rauminhalt des Zylinders = Grundfläche * Höhe

```
; Rauminhalt eines Zylinders berechnen
(: cylinder-volume (number number -> number))
(define cylinder-volume
  (lambda (radius height)
    (* (disk-area radius) height)))
; Testfall
(check-within (cylinder-volume 1 1) 3.14159 1e-5)
(check-within (cylinder-volume 2 1) (cylinder-volume 1 4) 1e-5)
```

1.14 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)
=> (* 78.539816339744825 4)
=> 314.1592653589793
```

MANTRA

Mantra #3 (Strukturerhaltung)

Versuche, das Programm wie das Problem zu strukturieren.

Mantra #4 (Abstraktion)

Schreibe eine Abstraktion für jedes Unterproblem.

Mantra #5 (Namen)

Definiere Namen für häufig benutzte Konstanten und verwende diese Namen anstelle der Konstanten, für die sie stehen.

1.15 Formale Semantik: Das Substitutionsmodell

- Formale Definition des Berechnungsprozesses eines Programms.
- Rechenschritt im Substitutionsmodell: *Reduktionsschritt*.
- Berechnungsprozess: *Reduktionssequenz*,
d.h. Folge von Ausdrücken, wobei aufeinanderfolgende Ausdrücke durch einen Reduktionsschritt ineinander übergeführt werden.
- Definition der Semantik:
Lege zu jeder Form fest, ob sie
 - ein Wert ist (d.h., ein Ergebnis) oder ob
 - ein Reduktionsschritt anwendbar ist
 - Wenn ja, wo in der Form?

1.16 Freie und gebundene Variable

- Ein Vorkommen einer Variable in einem Ausdruck heißt *frei*, falls keine umschließende Bindung existiert. Anderenfalls heißt das Vorkommen *gebunden*.
- Beispiele:
x kommt frei vor in

```
x
(* x 5)
(+ 17 (- x y))
((lambda (x) (+ x 1)) (* x x))
```

x kommt gebunden vor in

```
(lambda (x) 42)
((lambda (x) x) (+ y 212))
((lambda (x) (+ x 1)) (* x x))
(lambda (y) (lambda (x) y))
```

1.17 Lexikalische Bindung

```
((lambda (x1)  
  (+ ((lambda (x2) (* x3 3)) 3)  
    (* x4 2))) 14)
```

- Zwei Vorkommen von x , ¹ und ², sind *bindend* und *definieren* x .
- Zwei Vorkommen, ³ und ⁴, *verwenden* x .
- Frage: Welche Verwendung bezieht sich auf welche Bindung?
- Es gilt die *lexikalische Bindung*:
Eine Verwendung bezieht sich **immer** auf das bindende Vorkommen der innersten textlich umschließenden Abstraktion.
- D.h. ³ bezieht sich auf ² und ⁴ bezieht sich auf ¹ (Knopf „Syntaxprüfung“).
- Äquivalenter Ausdruck durch *konsistente Umbenennung* eines bindenden Vorkommens und aller Verwendungen dieser Bindung:

```
((lambda (x1) (+ ((lambda (y2) (* y3 3)) 3) (* x4 2))) 14)
```

1.18 Berechnungsregeln des Substitutionsmodells

- Ein Literal ist ein Wert.
- Ein Lambda-Ausdruck ist ein Wert.
- Eine Variable wird durch ihre Bindung (einen Wert) ersetzt.
- Zur Berechnung des Wertes einer Applikation

$(\langle operator \rangle \langle operand \rangle_1 \dots \langle operand \rangle_n)$

werden zuerst der Wert v_0 von $\langle operator \rangle$, sowie die Werte v_1, \dots, v_n der Operanden bestimmt.

1. Ist v_0 primitiver Operator, so wird er auf v_1, \dots, v_n angewendet.
2. Ist $v_0 = (\text{lambda } (x_1 \dots x_n) e)$, 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.
3. Anderenfalls: Laufzeitfehler!

1.19 Animation des Substitutionsmodells

- Stepper in DrScheme (Barfußknopf).
- Benutzung
 - Programm im Editierfenster
 - Stepper wertet den letzten Ausdruck im Editierfenster Schritt für Schritt aus.

1.20 Zusammenfassung

- Programme
- Ausdrücke und ihre Auswertung (Substitutionsmodell)
- Sorten und Verträge
- Testfälle
- Konstruktionsanleitung für Prozeduren
- Lexikalische Bindung