Informatik I

2. Erste Schritte in Scheme

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

21. Oktober 2010

Informatik I

21. Oktober 2010 — 2. Erste Schritte in Scheme

2.1 Erste Schritte in Scheme

- Allgemeines
- Ausdrücke und Auswertung

Programmiersprachen

Ada, Basic, C, C++, C♯, Cobol, Curry, Fortran, Go, Gödel, HAL, Haskell, Java, Lisp, Lua, Mercury, Miranda, ML, OCaml, Pascal, Perl, PHP, Python, Prolog, Ruby, OCaml, Scheme, Shakespeare, Smalltalk, Visual Basic, u.v.m.

Wir lernen hier Scheme, einen Dialekt von Lisp. Genauer gesagt lernen wir einen speziell für die Lehre entwickelten Dialekt von Scheme.

Später lernen wir noch Python (es sei denn zwischenzeitlich bebt es erd und flutet es sint).

Die Programmiersprache Scheme . . .

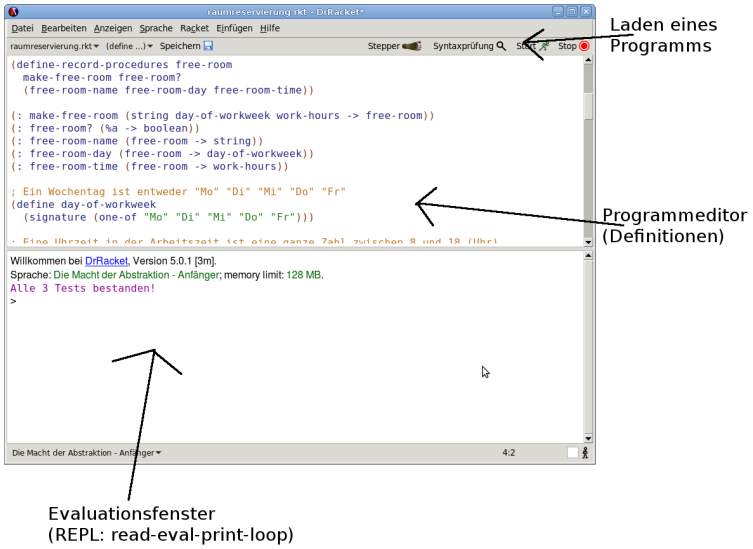
- . . . wurde 1975 von Gerald Jay Sussman and Guy Lewis Steele Jr. am Massachusetts Institute of Technology entwickelt.
- Die aktuelle (September 2007) Beschreibung ist: R6RS *Revised*^{6} *Report on the Algorithmic Language Scheme*.

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 (manche besser, andere schlechter).

DrRacket: Die Programmierumgebung

http://www.racket-lang.org/



Sprache: Syntax

Erinnerung:

- Es gibt verschiedene Programmiersprachen, aber sie alle sind formale Sprachen, d.h., sie sind exakt, durch strikte Regeln, definiert. Das unterscheidet sie von natürlichen Sprachen wie Deutsch oder Italienisch.
- So wie Sätze in natürlicher Sprache aus Wörtern und Satzzeichen gemäß einer bestimmten Grammatik zusammengefügt werden, so werden Programme in einer Programmiersprache aus Grundbausteinen unter Verwendung von Kombinationsmitteln zusammengefügt.

Hier geht es um Syntax, ein Hauptaspekt jeglicher Sprache, ob natürliche Sprache, Programmiersprache, Musiknotation etc.
In der Informatik spricht man sehr viel von Syntax!

Sprache: Semantik

Über Algorithmen hieß es:

- Die Bedeutung jedes Einzelschritts ist eindeutig festgelegt.

Ich sagte: „. . . wie auch immer er notiert sein mag“. Jetzt, da wir uns mit der Notation konkret beschäftigen, gilt die Ausage unvermindert.

Hier geht es um Semantik, ebenfalls ein Hauptaspekt jeglicher Sprache. Von Semantik wird in der Informatik ebenfalls viel gesprochen! Semantik ist oft schwierig zu fassen.

Grundbausteine

Zeichen mit fester Bedeutung

- Literale (Konstanten), z.B. für Zahlen oder Strings:
  `42 -17 2/3 3.1415926535 "Banane"`
- Vordefinierte Namen (primitive Operatoren), z.B. für arithmetische Operationen:
  `+ - * /`

Zeichen mit frei wählbarer Bedeutung

Namen (Bezeichner, Identifier, Variablen)

`x y banane`

Aus diesen bilden wir Ausdrücke.

Bildung von Ausdrücken

- Ein Literal ist ein Ausdruck.
- Eine Variable ist ein Ausdruck.
- Die (Funktions-)Anwendung, Applikation eines vordefinierten Namens auf Ausdrücke (Operanden) ist ein Ausdruck:
  (⟨*operator*⟩ ⟨*operand*⟩ ... ⟨*operand*⟩)
  `(+ 17 4)` `(* x (+ 17 4))`

Auswertung

- Ausdrücke (⟨*expression*⟩) haben einen Wert, sie können ausgewertet werden.
- Jeder Ausdruck beschreibt einen Berechnungsprozess zur Ermittlung seines Wertes (Auswertung). Start der Auswertung durch Eingabe in das REPL-Fenster.

Auswertung: Beispiele

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
```

Auswertung: Beispiele (Forts.)

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
```


2.2 Programme

- Definitionen
- Abstraktionen
- Erste Konstruktionsanleitungen
- Formale Semantik

Programme

Wieso kamen bei den Beispielen für Auswertung keine Variablen vor?
Bevor wir diese Frage beantworten, gehen wir nochmals eine Stufe höher.

Programm

Ein Programm ist eine Folge von Formen. Formen können sein

- Definitionen
- Ausdrücke

Bemerkungen:

- Ich verwende das Wort „Form“ trotzdem weiter auch in seiner allgemeinsprachlichen Bedeutung.
- Ein Kommentar beginnt mit dem Zeichen ; und endet mit dem Zeilenende.
- Leerzeichen, Zeilenumbrüche und Kommentare sind Trennzeichen ohne Bedeutung.

Definitionen

Eine Definition ist eine Spezialform eingeleitet durch das Schlüsselwort `define`:
(`define` ⟨*variable*⟩ ⟨*expression*⟩)

- 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 können keine Variablennamen sein.
- Variablennamen können keine Trennzeichen enthalten.

Bemerkung: Dies war die Definition von „Definition“. Ich verwende das Wort „Definition“ trotzdem weiter auch in seiner allgemeinsprachlichen Bedeutung.

Definitionen: Beispiele

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


Variablen in Ausdrücken

Wie könnte der Wert eines Ausdrucks definiert sein, in dem eine Variable vorkommt? Wenn die Variable vorher definiert wurde, ist das nicht sehr schwierig. Z.B.:

Quadrieren

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

Man sagt: Ausdruck `(* x x)` enthält die freie Variable `x` (die im Programm definiert ist).
Aber irgendwie ist das noch nicht befriedigend. Man müsste doch sagen können: „Für beliebiges `x`, gib mir `(* x x)`." Schließlich soll `x` veränderlich sein.

Abstraktionen

Abstraktion von `x` führt zu dem Lambda-Ausdruck (Abstraktion, Prozedur, Funktion)

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

- Man sagt: eine Abstraktion ist ein parametrisierter Ausdruck. Die Variable `x` ist die gebundene Variable des Lambda-Ausdrucks.
- Der Ausdruck `(* x x)` ist der Rumpf des Lambda-Ausdrucks.

Was tut man mit einem Lambda-Ausdruck?

So wie einen vordefinierten Namen (z.B. $+$) auch, kann man einen Lambda-Ausdruck anwenden (applizieren). Die Applikation setzt den Operanden für die gebundene Variable ein:

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

Was tut man mit einem Lambda-Ausdruck? (Forts.)

Ein Lambda-Ausdruck ist ein Ausdruck. Deshalb kann man selbstverständlich einen Namen als den Lambda-Ausdruck definieren:

```
   (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)
=> ...
=> 16
```


Auswertung der Funktionsanwendung

Zur Auswertung von

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

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

Der Rückgabewert bestimmt sich wie folgt:

1. Ist v_0 primitiver Operator, so wird er auf $v_1, \ldots, v_n$ angewendet.
2. Ist $v_0 =$ `(lambda (`$x_1 \ldots x_n$`)` 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.

Fläche eines Kreises

Aufgabe: Fläche eines Kreises

Eingabe: Radius r des Kreises ($r \geq 0$)

Ausgabe: Fläche πr^2 des Kreises

Definiere hierfür eine Prozedur.

Wir haben oben schon `pi` und `square` definiert.

Fläche eines Kreises II

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

```
; Fläche eines Kreises berechnen
(: circle-area (number -> number))
```

Daraus ergibt sich folgendes Gerüst für die Definition:

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

Testfälle:

Fläche eines Kreises III

Nun vervollständigen wir das Gerüst unter Verwendung von `pi` und `square`:

```
; Fläche eines Kreises berechnen
(: circle-area (number -> number))
(define circle-area
  (lambda (radius)
    (* pi (square radius))))
; Testfälle
(check-expect (circle-area 0) 0)
(check-within (circle-area 1) 3.14159 1e-5)
(check-within (circle-area 2) 12.56637 1e-5)
```


Das Parkplatzproblem

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

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

Signatur 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)
```


Das Parkplatzproblem II

Eingabe: $n, r \in \mathbb{N}$, 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)
```


Testfälle

- Ein Testfall besteht aus der Anwendung der zu schreibenden Prozedur auf gewisse Eingaben, sowie der erwarteten Ausgabe. DrRacket unterstützt Testfälle durch
  `(check-expect` ⟨*expression*⟩ ⟨*expression*⟩`)`
- Die Ausgabe soll „von Hand“ separat berechnet werden!
  Das geht natürlich nur für kleine Eingaben, und manchmal auch nur ungefähr. Deshalb bietet DrRacket folgendes an:
  `(check-within` ⟨*expression*⟩ ⟨*expression*⟩ ⟨*expression*⟩`)`
  Damit kann getestet werden, ob der Wert eines Ausdrucks in einem bestimmten Bereich liegt.
- Es sollten Testfälle bereitgestellt werden für
  - Randfälle (im Parkplatzproblem $r = 2n$ und $r = 4n$);
  - Standardfälle;
  - Fehlerfälle.

Testfälle II

- Ein nicht bestandener Test soll folgende Konsequenzen haben:
  - Korrektur des Programms;
  - ggf. Korrektur des Testfalls nach gründlichem Nachdenken.
- Er sollte keinesfalls folgende Konsequenzen haben:
  - Entfernen des Testfalls;
  - einfache Anpassung des Testfalls an die tatsächlich berechnete Ausgabe.

Konstruktionsanleitung 1 (Konstruktion von Prozeduren)

(Erste Annäherung)

Kurzbeschreibung	Schreibe eine einzeilige Kurzbeschreibung.
Signatur	Wähle einen Namen und schreibe die Signatur für die Prozedur. Verwende dafür die Form (: ⟨*name*⟩ ⟨*signature*⟩).
Gerüst	Leite aus der Signatur 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 (Signatur vor Ausführung)

Schreibe — vor der Programmierung des Prozedurrumpfes — eine Kurzbeschreibung der Aufgabe als Kommentar, sowie eine Signatur ins Programm.

Mantra #2 (Testfälle)

Schreibe Testfälle vor dem Schreiben der Definition.

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)
    (* (circle-area radius) height)))
; Testfall
(check-within (cylinder-volume 1 1) 3.14159 1e-5)
(check-within (cylinder-volume 2 1) 12,56636 1e-4)
(check-within (cylinder-volume 1 4) 12,56636 1e-4)
```

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)
=> (* (* 3.141... ((lambda (x) (* x x)) 5)) 4)
=> (* (* 3.141... (* 5 5)) 4)
=> (* (* 3.141... 25) 4)
=> (* 78.53... 4)
=> 314.1...
```


Mantra

Mantra #3 (Strukturerhaltung)

Versuche, das Programm wie das Problem zu strukturieren.

„Untermantras“:

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.

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?

Freie, gebundene, und bindende Variablen

- Ein Vorkommen einer Variable in einem Ausdruck heißt frei, falls keine umschließende Bindung existiert. Beispiele:

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

- Ein Vorkommen einer Variable in einem Ausdruck heißt gebunden, falls eine umschließende Bindung existiert. Beispiele:

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

- Andernfalls, d.h., wenn das Vorkommen direkt hinter dem `lambda` steht, heißt das Vorkommen bindend.

Lexikalische Bindung

- Im Folgenden kennzeichnen wir Vorkommen durch Superskripte:

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

  Die Vorkommen ^{1} und ^{2} von `x` sind bindend, und die Vorkommen, ^{3} und ^{4} sind gebunden.
- Doch welches gebundene Vorkommen bezieht sich auf welche Bindung?
  Es gilt die lexikalische Bindung: Eine gebundenes Vorkommen 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“).

Lexikalische Bindung II

- Äquivalenter Ausdruck durch konsistente Umbenennung eines bindenden Vorkommens und aller gebundenen Vorkommen zu dieser Bindung:

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

  wird zu

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


Berechnungsregeln des Substitutionsmodells

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

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

  werden 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 $v_0 =$ `(lambda (`$x_1 \ldots x_n$`)` 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. Andernfalls: Laufzeitfehler!

Animation des Substitutionsmodells

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

Zusammenfassung

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