

Scheme-Crash-Kurs

Christoph Hetz
hetz@informatik.uni-tuebingen.de

April 2006

Funktionale Programmierung

- Funktionen/Prozeduren sind „Werte erster Ordnung“
 - Funktionen ...
 - ... können als Parameter übergeben werden
 - ... können Rückgabewerte anderer Funktionen sein
 - ... können andere Namen zugewiesen werden
- Zuweisungen sind verboten oder selten
 - ⇒ „korrektere Programme“
- Keine Schleifen, nur Rekursion

Scheme

- „Die wahrscheinlich kleinste Programmiersprache der Welt“
- Lisp Dialekt (“besonders saubere Teilmenge”)
(\implies basiert auf Lambda-Kalkül)
- Sehr einfache Syntax
- Call-by-value
- Die meisten Programmierparadigmen ausdrückbar:
funktional, imperativ, objektorientiert, regelbasiert, ...
- Automatische Speicherverwaltung

R⁵RS

- „Revised⁵ Report on the Algorithmic Language Scheme”
- Definiert und beschreibt die Sprache:
Semantik, Syntax, Konzepte, Ausdrücke, Programmstruktur,
Sprachumfang, Datenstrukturen, ...
- Sehr dünn
- <http://www.schemers.org/Documents/Standards/R5RS/>

scsh

- „Scheme Shell“
- Unix Shell, auf Scheme48 basierend
- elegantes Modulsystem
- zum Schreiben von Shell-Skripten und Systemanwendungen
- <http://www.scsh.net>

XEmacs

- Einstellungen für die Benutzung von scsh mit dem XEmacs
In die Datei `~/.xemacs/init.el` einfügen:

```
(require 'cmuscheme48)
(setq scheme-program-name "scsh")
```

- Starten von scsh innerhalb XEmacs:
M-x run-scheme RET
- Nützliche Tastenkombinationen:
C-c l lädt eine Datei in scsh
C-c e schickt den Ausdruck, in dem der Cursor steht, an scsh
C-c r schickt die momentan markierte Region an scsh
C-c z wechselt zum scsh-Buffer

scsh interaktiv

```
Welcome to scsh 0.6.6 (King Conan)
Type ,? for help.
>
```

- REPL = Read Eval Print Loop
- scsh stellt R⁵RS-Funktionalität bereit
- Weitere Funktionen durch das Öffnen von Modulen mit `,open <modulname>`
- Übersicht über interaktive Befehle mit `,help`.

Syntax von Scheme

- Variablen: foo, bar-baz, bla-blubber!?, +, ...
scsh: Groß-/Kleinschreibung beachten!
- Literale: 23, #\A, #t, #f, ...
- Alles andere sind *Kombinationen*: ($\langle operator \rangle \langle operands \rangle^*$)
Die Art der Operation läßt sich an der Form von $\langle operator \rangle$ erkennen.
- Vollständig geklammerte Präfixnotation
(\implies keine Präzedenzregeln)
- Alle Klammern rund
- Alle Klammern signifikant
- Kommentare mit ; bis zum Zeilenende

Literale

```
> 23
```

```
23
```

```
> #t
```

```
#t
```

```
> #f
```

```
#f
```

```
> "The sky is blue."
```

```
"The sky is blue."
```

```
> #\A
```

```
#\A
```

Prozeduraufrufe

(⟨operator⟩ ⟨operands⟩*)

```
> +  
#{Procedure}  
> (+ 23 19)  
42  
> (* (+ 23 19) 5)  
210  
> (< 23 42)  
#t
```

*+, * und < sind auch nur Namen!*

Definitionen

```
> (define king-kong 14)
> king-kong
14
> (define godzilla 27)
> (* king-kong godzilla)
378
```

```
> (define tmp -)
> (define - *)
> (define * tmp)
> (* (- 5 5) (- 4 4))
9
```

Prozeduren selber schreiben

`(lambda <formals> <body>)`

```
> (lambda (x) (+ x 1))  
#{Procedure}
```

⇒ Prozeduren sind Werte (erster Ordnung)

```
> ((lambda (x) (+ x 1)) 17)  
18
```

⇒ Operator ist ein Ausdruck

Prozedurale Programmierung

```
(define pi 3.14169265)

(define square
  (lambda (x)
    (* x x)))

(define circle-area
  (lambda (radius)
    (* pi (square radius))))

(define cylinder-volume
  (lambda (radius height)
    (* (circle-area radius) height)))
```

Prozedurale Programmierung

```
> (cylinder-volume 2 4)
50.2671
> (cylinder-volume 0.5 18)
14.1376
```

Prozedurfabriken

```
(define make-adder  
  (lambda (n)  
    (lambda (x)  
      (+ n x))))
```

```
> (define add-1 (make-adder 1))  
> (add-1 5)  
6  
> (define add-5 (make-adder 5))  
> (add-5 5)  
10
```

Syntaktischer Zucker

```
(define (circle-area radius)
  (* pi (square radius)))

(define (cylinder-volume radius height)
  (* (circle-area radius) height))
```

wird übersetzt nach:

```
(define circle-area
  (lambda (radius)
    (* pi (square radius))))

(define cylinder-volume
  (lambda (radius height)
    (* (circle-area radius) height)))
```

Conditionals

```
> (= 1 4)
#f
> (= 4 4)
#t
> (if #t 1 2)
1
> (if #f 1 2)
2
```

(if *<predicate>* *<consequent>* *<alternative>*)

and **und** or

```
> (and #t #f)
#f
> (or #t #f)
#t
> (and 1 2)
2
> (or 1 2)
1
```

Prädikate

Prozeduren, die entweder #t oder #f liefern:

```
> (zero? 5)
#f
> (zero? 0)
#t
> (number? #t)
#f
> (boolean? #t)
#t
> (= 23 23)
#t
```

Rekursion

```
(define (fac n)
  (if (zero? n)
      1
      (* n (fac (- n 1)))))
```

```
(define (fac n)
  (or (and (zero? n) 1)
      (* n (fac (- n 1)))))
```

```
> (fac 5)
120
```

Verschachtelte Conditionals

```
(define (fib n)
  (cond
    ((= n 0)
     0)
    ((= n 1)
     1)
    (else (+ (fib (- n 2))
              (fib (- n 1))))))
```

```
> (fib 10)
55
```

Paare

```
> (cons 23 42)
'(23 . 42)
> (cons #t 42)
'(#t . 42)
> (cons (cons 1 2) 3)
'((1 . 2) . 3)
> (cons "function" +)
("function" . #{Procedure})
```

Paare

```
> (define p1 (cons 23 42))
> (pair? p1)
#t
> (car p1)
23
> (cdr p1)
42
> (define p2 (cons (cons 1 2) 3))
> (car p2)
'(1 . 2)
> (car (car p2))
1
```

Folgen durch Paare

```
> (cons (cons 1 2) 3)  
'((1 . 2) . 3)
```

```
> (cons 1 (cons 2 3))  
'(1 2 . 3)
```

Warum nicht (1 . (2 . 3))?

Die Punkt-Klammer-Zap-Regel

„Wenn ein Punkt von einer Klammer auf gefolgt wäre, so wird in der externen Repräsentation der Punkt und das Klammernpaar weggelassen.“

$$(1 \ . \ (2 \ . \ 3)) \implies (1 \ 2 \ . \ 3)$$

Folgen durch Paare

```
> (define folge (cons 1 (cons 2 (cons 3 4))))  
> (car folge)  
1  
> (car (cdr folge))  
2  
> (car (cdr (cdr folge)))  
3
```

Für Element n der Folge:
($n - 1$)-mal `cdr` anwenden, dann `car`.
Leider nicht für das letzte Element!
⇒ Listen

Die leere Liste

```
> '()  
'()  
> (list? '())  
#t  
> (null? '())  
#t  
> (pair? '())  
#f
```

Listen

```
> (cons 1 (cons 2 (cons 3 '())))  
'(1 2 3)
```

```
> (list 1 2 3)  
'(1 2 3)
```

```
> (list? (list 1 2 3))  
#t  
> (pair? (list 1 2 3))  
#t
```

Listen sind eine induktive Menge

- Die leere Liste '()' ist eine Liste.

```
> (list? '())  
#t
```

- Falls l eine Liste und v ein beliebiger Wert ist, so ist das Paar mit v als `car` und l als `cdr` ebenfalls eine Liste.

```
> (list? (cons v l))  
#t
```

- Nichts sonst ist eine Liste.

Prozeduren auf Listen

```
(define (list-length list)
  (if (null? list)
      0
      (+ 1 (list-length (cdr list)))))
```

```
> (list-length (list 1 2 3 4 5))
5
```

Prozeduren auf Listen

```
(define (concatenate list-1 list-2)
  (if (null? list-1)
      list-2
      (cons (car list-1)
            (concatenate (cdr list-1) list-2))))
```

```
> (concatenate (list 1 2 3) (list 4 5 6))
'(1 2 3 4 5 6)
```

Prozeduren auf Listen

```
> (map even? (list 1 7 3 8 5 2))  
'(#f #f #f #t #f #t)  
> (map (lambda (n) (+ n 1)) (list 1 7 3 8 5 2))  
'(2 8 4 9 6 3)
```

```
> (filter even? (list 1 7 3 8 5 2))  
'(8 2)
```

Lokale Variablen

```
(let ((v1 e1) ... (vn en)) b)
```

```
> (let ((x 23)
        (y 42))
    (+ x y))
```

```
65
```

```
> x
```

```
Error: undefined variable
```

```
  x
```

```
(package user)
```

Lokale Variablen

- let funktioniert parallel

```
> (let ((x 5)
        (y (+ x 17)))
      y)
Error: undefined variable x
```

- let* funktioniert sequentiell

```
> (let* ((x 5)
         (y (+ x 17)))
       y)
22
```

- letrec für Rekursion: siehe R⁵RS.

Symbole sind Werte, die einen Namen haben

```
> 'foo
'foo
> (symbol? 'foo)
#t
> (equal? 'foo 'bar)
#f
> (equal? 'foo 'foo)
#t
> (equal? (cons 1 2) (cons 1 2))
#t
```

Symbole haben dieselbe Syntax wie Variablennamen,
sind aber keine!

quote vermittelt zwischen Repräsentation und Literalen

```
> (cons 1 2)
'(1 . 2)
> (quote (1 . 2))
'(1 . 2)
> (quote (1 2 3))
'(1 2 3)
> (quote 5)
5
> (quote #f)
#f
```

'*x* ist eine Abkürzung für (quote *x*)

- Paare:

```
> '(1 . 2)
'(1 . 2)
```

- Listen:

```
> '(1 2 3 4 5)
'(1 2 3 4 5)
```

- Symbole:

```
> 'foo
'foo
```

Zuweisungen

- Zuweisungen verändern Bindungen:

```
> (define x 23)
> (set! x 42)
> x
42
```

- call-by-value

```
> (define (f x)
  (set! x 4711))
> (f x)
> x
42
```

Mutatoren

- Mutatoren verändern Datenstrukturen:

```
> (define p1 (cons 23 42))  
> (set-car! p1 27)  
> p1  
(27 . 42)  
> (set-cdr! p1 7)  
> p1  
(27 . 7)
```

Paare und Prozeduren werden als Zeiger übergeben

```
> (define p (cons 23 42))  
> (define (f p)  
    (set-car! p 27))  
> (f p)  
> p  
'(27 . 42)
```

eq? vergleicht Zeiger

```
> (define p1 (cons 23 42))
> (define p2 (cons 23 42))
> (define p3 p1)
> (equal? p1 p2)
#t
> (eq? p1 p2)
#f
> (eq? p1 p3)
#t
```

Ausgabe

```
> (display "The sky is blue")
The sky is blue> (begin
                  (display "The sky is blue")
                  (newline))

The sky is blue
> (define p1 (cons 1 2))
> (begin
   (display p1)
   (newline))
(1 . 2)
```

SRFI 9: Records

- Um Records benutzen zu können, muss das Modul `srfi-9` geöffnet werden:

```
> ,open srfi-9
```

- Die Definition eines Records für Paare sieht so aus:

```
> (define-record-type :pare ; Name des Record-Typs
  (kons x y) ; Konstruktor
  pare? ; Prädikat
  (x kar set-kar!) ; Selektor, Mutator
  (y kdr)) ; Selektor
```

SRFI 9: Records

```
> (pare? (kons 1 2))  
#t  
> (pare? (cons 1 2))  
#f  
> (kar (kons 1 2))  
1  
> (kdr (kons 1 2))  
2  
> (let ((k (kons 1 2)))  
      (set-kar! k 3)  
      (kar k))  
3
```

Concurrent Programming

- Modul für spawn

```
> ,open threads
```

- Modul für make-lock, obtain-lock und release-lock

```
> ,open locks
```

Nachschlagen mit Suchfunktion

- Programm `help-desk`, kommt mit DrScheme-Implementierung
- <http://www.drscheme.org>
- zum Nachschlagen im R⁵RS
- Ideal für Scheme-Einsteiger

- Material auf Homepage

Links

Alle wichtigen Infos und Links auf

<http://www-pu.informatik.uni-tuebingen.de/cp-2006>