

Grundlagen von Scheme

Funktionale Programmierung

Bei funktionaler Programmierung wird eine Funktion auf passende Parameter angewendet und ausgewertet. Im Funktionskörper (Auswertungsteil) wird beschrieben, wie der Wert bestimmt wird.

Schachtelungen

Wie auch aus der Mathematik bekannt, können Funktionsaufrufe geschachtelt werden. Die Schachtelung wird eindeutig durch die Klammern gekennzeichnet. Es wird stets erst der innere Ausdruck der Schachtelung ausgewertet.

Beispiel:

```
(define  
  (mittelwert a b)  
  (/ (+ a b) 2)  
)
```

Kopf der Funktion

Schachtelung

In dieser Funktion wird z.B: beim Aufruf von (mittelwert 17 25) den beiden Parametern der Wert 17 bzw. 25 zugewiesen (sie werden an die Werte gebunden, engl. bound to ...).

Bei der Auswertung muss erst der innere Ausdruck berechnet werden, also die Summe von 17 und 25 \rightarrow 42. Dieser Wert ersetzt den Klammerausdruck (+ a b) und nun kann der äußere Ausdruck (/ 42 2) \rightarrow 21 berechnet werden. Dieser wird als Wert der Auswertung zurückgegeben.

Reihungen

Treten innerhalb des Auswertungsteils Reihungen von Ausdrücken auf, dann haben sie nur in ihren Nebeneffekten Bedeutung, der Wert der Funktion wird aber vom letzten Ausdruck bestimmt. Im folgenden Beispiel wird das berechnete Ergebnis kommentiert.

```
(define  
  (mittelwert a b)  
  (display "Mittelwert von ")  
  (display a)  
  (display " UND ")  
  (display B)  
  (display "ist :")  
  (/ (+ a b) 2)  
)
```

Es gibt noch einen Sonderfall einer Reihung, bei welcher der erste Ausdruck als Wert zurückgegeben wird. Sie wird mit der Funktion (begin0 (Ausdruck0) ...) eingeleitet.

Nicht viel Neues

Das ist alles bisher nicht besonders bemerkenswert, da es sich kaum von dem unterscheidet, was wir von JAVA kennen. Bisher bleibt als wesentliche Neuerungen, an die man sich gewöhnen muss:

- die Klammer am Anfang und
- die Angabe des Funktionsnamens auch bei einfachen Funktionen am Anfang des Auswertungsausdrucks (Präfixnotation).

Auch if ist eine Funktion!

Auch Verzweigungen werden als Funktion betrachtet. Sie stellen bei einer funktionalen Sprache keine Verzweigungen im Sinne des Programmablaufes dar, sondern der jeweils einzusetzenden funktionalen Ausdrücke.

Die folgende Funktion gibt also den kleineren von den beiden übergebenen Werten zurück:

```
(define
  (kleinerer-von a b)
  (if
    (< a b)
    a
    b))
```

Mehrfachverzweigung

Scheme stellt zusätzlich eine sehr komfortable Mehrfachverzweigung bereit. Die Funktion heißt *cond* (nach *condition*) und stellt eine Abfolge von Paaren aus einer Bedingung und dem zugehörigen Wert bereit. Es ist aber keine normale Reihung, da diese Abfolge nach dem ersten erzielten Treffer bei der Reihung verlassen wird, denn in dem Fall ist der Wert der Verzweigungsfunktion bekannt. Das letzte Paar stellt den else – Fall dar. Irgendein Wert muss also in jedem Fall zurückgegeben werden können.

Ein Beispiel, bei dem die Typen der Parameter vor einer Auswertung getestet werden:

```
(define
  (plus a b)
  (cond
    ((not (number? a)) (error "a: " a "ist keine Zahl"))
    ((not (number? b)) (error "b: " b "ist keine Zahl"))
    (else
     (+ a b))))
(plus 'hund 2) → a: hund "ist keine Zahl"
(plus 2 'katze) → b: katze "ist keine Zahl"
(plus 2 3) → 5
```

Interessant ist der letzte Fall

```
(plus 'hund 'katze) → a: hund "ist keine Zahl"
```

weil hier besonders deutlich wird, dass nur der erste treffende Wert ausgewertet wird, der Test von b wird daher nicht mehr durchgeführt.

Schematische Darstellung von if und cond

if:

| | |
|-----------|----------------|
| Bedingung | Wert bei true |
| | Wert bei false |

if muss nicht notwendig einen Wert für den false – Fall haben. Wenn das so ist, ist es aber nicht wirklich funktional eingesetzt worden, sondern nur wegen eines Nebeneffektes.

cond

| | |
|-------------|--------------------------------|
| Bedingung 1 | Wert 1 |
| Bedingung 2 | Wert 2 |
| usw. | |
| else | Wert, wenn keiner sonst passte |

Datenstruktur Liste

Die zentrale Datenstruktur in Scheme ist die Liste. '(12 'hund "Wort" 3.2) ist eine Liste – am Anfang das Hochkomma verhindert die Auswertung – und diese Liste enthält in diesem Fall die integer – Zahl 12, das Symbol 'hund, den string "Wort" und die float – Zahl 3.2! Das geht so, welchen Sinn auch immer das haben könnte.

Eine der wichtigsten Fähigkeiten beim Arbeiten mit Scheme ist daher, mit Listen arbeiten zu können. Im Gegensatz zu JAVA sind die Listen in Scheme einfach verkettete Listen, man kann also zu einem Element nicht direkt den Vorgänger bestimmen, sondern nur den Nachfolger.

Die Zugriffsfunktion für den (Zeiger auf den) Listeninhalt am Kopf heißt car, die für dessen Nachfolger cdr. Also lassen sich deutsche Umschreibungen für diese Funktionen angeben:

Ausgabe des ersten Elementes einer Liste

```
(define
  (erstes liste)
  (car liste)
)
```

Ausgabe der Restliste einer Liste

```
(define
  (rest liste)
  (cdr liste)
)
```

car und cdr sind also Standardfunktionen von Scheme

Ausgabe des zweiten Elementes einer Liste

```
(define
  (zweites liste)
  (car (cdr liste))
)
```

typisch ist hier die Schachtelung: Erst die Restliste bestimmen und dann auf deren Kopf zugreifen. "Dann" muss aber davor stehen!

Ausgabe des dritten Elementes einer Liste

```
(define
  (drittes liste)
  (car (cdr (cdr liste)))
)
```

```
(car (cdr (cdr liste)))  
)
```

im Kern geht es darum, dass man Funktionen beliebig schachteln kann

Kurzschreibweisen

Für diese beiden Funktionen gibt es bei Schachtelung von bis zu vier Funktionen Kurzschreibweisen:

```
(cddddr liste) gibt also die Restliste nach dem vierten Element zurück.  
(zweites liste) → (cadr liste)  
(dritte liste) → (caddr liste)  
(cadadr '(1 2 3 4 5 6 7)) → Fehlermeldung
```

Der Zugriff muss zulässig sein. Man muss diese Funktionsbezeichnung von hinten anfangen zu interpretieren, also Restliste bilden → '(2 3 4 5 6 7), davon das erste Element nehmen → 2 und davon soll nach der Vorschrift nun wieder die Restliste gebildet werden. Da die Zahl 2 aber keine Liste darstellt, gibt es eine Fehlermeldung. Zulässig ist:

```
(cadadr '((1 2) (3 4) (5 6 7))) → 4
```

Die Schritte der Auswertung:

```
Restliste      ((3 4) (5 6 7))  
erstes        (3 4)  
davon Restliste (4)  
davon erstes   4
```

n-tes

Für die Ausgabe des n-ten Elementes einer Liste (also eines beliebigen) gibt es eine Standardfunktion in Scheme

```
(define  
  (n-tes liste)  
  (list-ref liste n)  
)
```

Scheme – Grundlagen in Beispielen

Der folgende Textabschnitt soll keinen Scheme – Kurs darstellen, sondern einige Grundlagen an Hand von Beispielen erläutern.

Ich benötige eine Liste der ungeraden natürlichen Zahlen von 1 bis zu irgendeinem Maximalwert.

Hier ist also über den Datentyp¹ und über Rekursion zur Realisierung der Schleife nachzudenken. Scheme – typisch verwenden wir eine Liste und darin enthalten integer – Zahlen. Bei der Rekursion werden wir sinnvollerweise herunter zählen (warum?) und daher mit der Abbruchbedingung (< ... 1) arbeiten. Eine Lösung wäre daher:

```
(define  
  (ungerade-bis wert)  
  (cond  
    ((< wert 1) '())  
    (else  
     (cons wert (ungerade-bis (- wert 2))))))
```

¹ Der R5RS fordert die zu den Prädikaten boolean? pair? symbol? number? char? string? vector? port? procedure? gehörenden Typen.

mit dem Aufruf `(ungerade-bis 13)` erhalten wir `(13 11 9 7 5 3 1)`

Dazu lassen sich mehrere Anmerkungen machen:

1. Die Liste würde man eher wohl anders herum haben wollen.
Das ist kein prinzipielles Problem, man übergibt sie anschließend einfach der Funktion `(reverse ...)`.
2. Was passiert eigentlich, wenn eine gerade Zahl übergeben wird?
Grundsätzlich gibt es wegen der Abbruchbedingung `<` keine Katastrophe.
Allerdings erhält man keine sinnvollen Ergebnisse. Eine Lösung bietet die u.a. Variante.
3. Was passiert eigentlich, wenn eine Zahl kleiner als 1 übergeben wird?
Hier erhält man eigentlich eine sinnvolle Lösung, bestenfalls würde man sich vielleicht noch eine Fehlermeldung wünschen.
4. Was passiert eigentlich, wenn überhaupt keine Zahl übergeben wird, also z.B. ein string?
Gerade in diesem Fall bekommt man eine Fehlermeldung, da der Vergleich mit einem string nicht zulässig ist: Scheme kennt Typen! Man kann alles übergeben, aber wenn eine typgebundene Funktion zugreift, gibt es einen Fehler!
Eine Lösung ist eine Abfragefunktion als erste Bedingung im `cond`, die testet, ob der richtige Typ übergeben wurde. [Dazu siehe unten]

Endrekursion

Vielleicht möchte man ja nicht normal rekursiv arbeiten, sondern endrekursiv. Wir bauen um und erhalten als eine Lösung:

```
(define
  (ungerade wert liste)
  (if
    (< wert 1)
    liste
    (ungerade (- wert 2) (cons wert liste))))
(ungerade 13 '()) → (1 3 5 7 9 11 13)
```

Der zweite Parameter, der beim Aufruf zwingend als leere Liste vorzugeben ist, schreit geradezu nach einer Aufrufhülle. Wir sehen sie uns in zwei Varianten an:

Aufrufhülle mit einer lokalen Funktion (define ...)

```
(define
  (ungerade wert)
  (define
    (ungerade-intern wert liste)
    (if
      (< wert 1)
      liste
      (ungerade-intern (- wert 2) (cons wert liste))))
  (cond
    ((not (number? wert))
     (error "Parameter " wert " ist keine Zahl!"))
    ((not (integer? wert))
     (error "Parameter " wert " ist keine ganze Zahl!"))
    ((not (positive? wert)) (error "Parameter " wert " ist zu klein!"))
    ((even? wert) (error "Parameter " wert " ist gerade!"))
    (else
```

```
(ungerade-intern wert '()))))
```

Aufrufhülle mit einem named let¹

```
(define
  (ungerade wert)
  (cond
    ((not (number? wert))
     (error "Parameter " wert " ist keine Zahl!"))
    ((not (integer? wert))
     (error "Parameter " wert " ist keine ganze Zahl!"))
    ((not (positive? wert)) (error "Parameter " wert " ist zu klein!"))
    ((even? wert) (error "Parameter " wert " ist gerade!"))
    (else
     (let ungerade-intern
       ((wert wert)
        (liste '()))
       (if
        (< wert 1)
        liste
        (ungerade-intern (- wert 2) (cons wert liste))))))))
```

Beide Varianten sind so abgesichert, dass die nachfolgenden Aufrufe mit entsprechenden Fehlermeldungen abbrechen.

```
(ungerade 'hund)
(ungerade 1.3)
(ungerade -13)
(ungerade 10)
```

und nur (ungerade 13) richtig bearbeitet wird.

Die Elemente einer Liste sollen gefiltert werden

Wir wollen die Aufgabe verallgemeinern. Dazu verwenden wir als Parameter eine Funktion, nämlich die Filterfunktion. Sie stellt in unserem Beispiel ein Prädikat dar, also eine Funktion, die nur die Werte #t oder #f zurückliefert. Wir kennzeichnen sie daher Scheme – typisch mit einem Fragezeichen am Ende des Namens.

```
(define
  (alle-ohne filter? liste)
  (cond
    ((null? liste) liste)
    ((filter? (car liste))
     (alle-ohne filter? (cdr liste)))
    (else
     (cons (car liste) (alle-ohne filter? (cdr liste))))))
```

Aufruf:

```
(alle-ohne even? '(654 4 345 36 -16 43 413 543 54 35 32 632 632 -1111))
→ (345 43 413 543 35 -1111)
```

Typisch funktional ist auch die nächste Anwendung, bei der die anderen oben angegebenen Fälle geschachtelt ausgesondert werden. Zunächst definieren wir für dieses Beispiel die benötigten Filterfunktionen²:

¹ siehe auch allgemeine Erläuterungen am Schluss

² Das ist nicht zwingend, die Funktionen können auch als anonyme Funktionen übergeben werden: lambda

```
(define (keine-zahl? par) (not (number? par)))
(define (nicht-ganzzahlig? par) (not (integer? par)))
(define (nicht-positiv? par) (not (positive? par)))
(alle-ohne even?
  (alle-ohne nicht-positiv?
    (alle-ohne nicht-ganzzahlig?
      (alle-ohne keine-zahl?
        '(hund 2 -1 33 2.4 katze 9))))))
→ (33 9)
```

Beachten Sie wiederum: Die Auswertung geht stets von der innersten Schachtelung nach außen. Es ist also vom letzten geschriebenen Ausdruck anzufangen.

Zahlen

Vordefinierte Zahlentypen (number) sind: complex real rational und integer

Es gibt aber wichtige zusätzliche Informationen.

Scheme unterscheidet strikt zwischen exact und inexact.

(exact? 3.5) liefert den Wert #f, daher also (inexact? 3.5) → #t.

(exact? 7) → #t und (inexact? 7) → #f sind einfach zu verstehen.

Wichtig zu wissen ist aber: (exact? (/ 7 2)) → #t.

Scheme rechnet also immer dann exact, wenn es möglich ist!

Der Wert von (/ 7 2) ist daher auch $3\frac{1}{2}$!

Wenn man auf inexacte Werte wechseln will, muss man das ausdrücklich durch die Funktion (exact->inexact ...) tun oder mit einer inexact - Zahl verknüpfen.

```
(inexact? (exact->inexact (/ 7 2))) → #t
```

und ebenfalls

```
(inexact? (* 1.0 (/ 7 2))) → #t
```

Langzahlarithmetik

Langzahlarithmetik ist automatisch verfügbar. So ist es unproblematisch, die optimierte Potenzfunktion modulo zu schreiben, die wir bei RSA verwendet haben.

```
(define
  (schnelle-potenz-modulo basis exponent mod-zahl)
  (cond
    ((zero? exponent) 1)
    ((= exponent 1) (modulo basis mod-zahl))
    ((even? exponent)
     (modulo
      (schnelle-potenz-modulo (* basis basis) (/ exponent 2) mod-zahl)
      mod-zahl))
    (else
     (* basis
        (schnelle-potenz-modulo
         (* basis basis)
         (/ (sub1 exponent) 2)
         mod-zahl))))))
```

Einzig problematisch könnte der normal rekursive Aufruf sein, den man ggf. durch einen endrekursiven ersetzen kann.

Was muss man sonst noch können?

Wie auch JAVA ist das gesamte Scheme – Paket riesig und auch für einen vierstündigen Profilkurs nicht zu bearbeiten. Wozu auch? Interessant sind eben nur Konstrukte, die man im jeweiligen Kontext wirklich benötigt¹!

So werden wir Assoziationslisten zur Speicherung von Graphen verwenden und dann natürlich auch auf die Funktion `assoc` zur Bestimmung von Nachfolgeknoten zurückgreifen.

Bei der Arbeit mit Scheme sollte man sich in jedem Fall mit den grundlegenden Funktionen zu Listen beschäftigen und verstanden haben, wie man mit tiefen Listen (Listen, die Listen enthalten, die...) arbeitet.

¹ Manchmal weiß man aber natürlich nicht, was man benötigt. Meistens aber hat man das andere Problem: Man weiß zwar, was man will, weiß aber nicht, ob die jeweilige Programmiersprache das bereitstellt. Hier hilft nur eine gute Dokumentation der Sprache und die wird von DrScheme zur Verfügung gestellt.

Übersetzungen einiger Definitionen aus dem R5RS:

(let (< Bindungen >) Körper)

(let Name (<Bindungen >) Körper)

<Bindungen> sind Listen mit zwei Elementen, bei denen vorn der Name der Variablen steht und dahinter der Wert, an den sie gebunden wird.

„Named let“ ist eine Variante des Syntaxkonstruktes von let, das ein (allgemeineres als do) Schleifenkonstrukt zur Verfügung stellt und für Rekursion verwendet werden kann. Das Besondere gegenüber dem einfachen let ist die Definition des Namens, der im Körper des let zum rekursiven Aufruf genutzt werden kann. Bei diesem rekursiven Aufruf werden die Variablen in der üblichen Weise jeweils an die neuen Werte gebunden.

list

(list obj ...) liefert eine neue Liste der übergebenen Argumente.

Wie für alle Objekte gilt aber eben: Solange sie diese Liste nicht an eine Variable oder noch ausstehende Auswertung binden, ist sie im selben Moment wieder auf dem Müllhaufen der Geschichte verschwunden!

cdr

Da wir kaum mit wirklichen Paaren arbeiten werden:

(cdr eine-liste) gibt (einen Zeiger auf) die Restliste einer Liste zurück. Wichtig: Die Funktion ändert nicht die ursprüngliche Liste!

```
(define meine-liste (list 1 2 3 4 5))  
(begin  
  (cdr meine-liste)  
  (cdr meine-liste)  
  (cdr meine-liste)  
  meine-liste)
```

Bindung der Variablen
meine-liste

liefert also die Liste (1 2 3 4 5) zurück und jeder einzelne Zwischenwert ist die Liste (2 3 4 5), die aber jedesmal wieder auf den Müll geschmissen wird!

Wenn man wirklich die tatsächliche Bindung einer Variablen an eine Liste verändern will, muss man eine der Funktionen mit dem Ausrufezeichen verwenden, wie z.B. set-cdr! , aber man mache das mit Vorsicht!

(set-cdr! meine-liste 'zwei) liefert nämlich für meine-liste → (1 . zwei) und das war sicher nicht das, was man wollte. Will man das zweite Element neu setzen, muss man

```
(set-car! (cdr meine-liste) 'zwei)
```

verwenden. Das liefert dann (1 zwei 3 4 5)

Wenn man so etwas denn braucht. Dies ist nämlich eigentlich nicht funktional, sondern prozedural. Funktional will man in der Regel einen veränderten Wert weiterreichen und dann ist (cons (car meine-liste) (cons 'zwei (cddr meine-liste))) das, was man machen sollte.

list-ref usw.

append, reverse, list-ref, list-tail sind neben assoc Funktionen, die man sich einmal ansehen sollte.

Trotzdem: Das Besondere an Scheme ist, dass man mit ganz wenigen Grundkonstrukten auskommt!