Scheme - Grundlagen

Scheme ist eine funktionale Programmiersprache. Der Aufruf einer Funktion hat einen Funktionswert zur Folge. Dieser Wert kann – wie auch von JAVA bekannt – void sein, dann hat man die Funktion nur wegen ihres Nebeneffektes ausgewertet[1]. Die grundlegenden Funktionen sind im R[5]RS und R[6]RS[2] beschrieben.

Minimalkonzept

Wir wollen und kommen auch tatsächlich mit nur sehr wenigen Funktionen aus. Zunächst sind vor allem die Funktionen wichtig, die sich mit der Bearbeitung von Listen beschäftigen.

- (**cons** element liste) gibt eine Liste mit dem übergebenen Element als Kopf der Liste zurück[3];

  (cons 67 '(23 6 0 100)) → '(67 23 6 0 100)
- das quote, das Zeichen **'**, verhindert den Auswertungsversuch als Funktion bei einer nachfolgenden Klammer und macht aus einem nachfolgenden Namen ein Symbol

  (quadrat a) sucht in der Auswertungsumgebung eine bekannte Funktion quadrat und wendet sie dann auf den Paramer a an, der ebenso der Auswertungsumgebung bekannt sein muss, während '(quadrat a) einfach eine Liste der beiden Symbole erzeugt,

  '(a b c) entspricht der Anwendung der list-Funktion auf die Symbole, also (**list** 'a 'b 'c)
- (**null?** liste) prüft, ob eine Liste Liste leer ist und gibt den entsprechenden Wahrheitswert (**#t** oder **#f**) zurück;
- (**car** liste) gibt den Kopf der Liste zurück; entspricht (**first** liste)

  (car '(23 6 0 100)) → 23 entsprechend (first '(a b c d)) → a
- (**cdr** liste) gibt die Restliste hinter dem Kopfelement zurück; entspricht (**rest** liste)

  (cdr '(a b c)) → '(b c) entsprechend (rest '(23 6 0 100)) → '(6 0 100)
- (**if** bedingung wenn-fall <ggf. sonst-fall>) ist eine einfache Verzweigung;

  (if (< a b) a b) gibt daher die kleinere beiden Zahlen zurück
- (**cond** (<bedingung1> <wert, wenn sie erfüllt ist> ...) (...) (else ...)) erzeugt eine Verzweigung, bei welcher der Wert zur ersten zutreffenden Bedingung zurück gegeben wird;

  ```
  (cond
    ((< a 0) -1)
    ((b a 0) 1)
    (else 0))
  ```
  liefert also die Vorzeichenfunktion
- (**reverse** liste) gibt die Umkehrliste zur liste zurück;

  (reverse '(23 6 0 100)) → '(100 6 0 23)

---

1 Ein einfaches Beispiel ist die Funktion (display ...), die man allein wegen ihrer Bildschirmausgabe aufruft.

2 Revised 5th bzw. 6th Report of the Algorithmic Language Scheme → Standardisierung für alle Scheme-Versionen

3 Beachten Sie: die ursprüngliche Liste wird dabei – wie in den anderen Fällen – ***nicht*** verändert!

- (**append** liste-1 liste-2) gibt eine Liste zurück, bei der beide verbunden sind;
  (append '(55 12 89) '(23 6 0 100)) → '(55 12 89 23 6 0 100)
- (**define** name wert) führt zu einer globalen Bindung von Variablen und Funktionen, die dauerhaft ist; dadurch „lernt“ das System
  (define quadrat (lambda (a) (* a a))) definiert also die Funktion quadrat in der Auswertungsumgebung
  [Einfachere Schreibweise: (define (quadrat a) (* a a))]
- (**let** ((var-1 wert) (...)) auswertungsteil) führt zu einer lokalen Bindung von Variablen, entspricht also dem define bis auf die fehlende Dauerhaftigkeit; die gebundenen Namen haben nur im Auswertungsteil Bedeutung
  (let ((a 7) (b 8)) (+ a b)) liefert [allerdings hier ohne Sinn] den wert 15 zurück.
- (**let** <name> ((var-1 wert) (...)) auswertungsteil) führt in entsprechender Weise zu einer lokalen Bindung einer Funktion (named let)
  (let loop ((zahl 5))
  (cond
  ((<= zahl 1) 1)
  (else (* zahl (loop (- zahl 1))))))
  liefert eine Anwendung der Fakultätsfunktion mit dem Wert 120

Scheme-Ausdrücke sind flexibel im Typ

Werte und Parameter von Funktionen können bei Scheme grundsätzlich alles sein, neben grundlegenden Typen wie Zahlen, Symbolen, strings und Listen also auch Funktionen selbst, Objekte, Klassen usw.
Erst bei einer Auswertung müssen die Typen passend sein.

Rekursion

Grundlegendes Wiederholungskonzept bei funktionalen Sprachen sind nicht Schleifen, sondern Rekursionen. Dabei unterscheidet Scheme intern genau zwischen normal rekursiven Aufrufen und endrekursiven Aufrufen, da nur bei normal rekursiven Aufrufen die Aufrufkette und die dabei aufgetretenen Inhalte der Variablen auf dem stack abgelegt werden.
Endrekursive Aufrufe kann man daran erkennen, dass zum Zeitpunkt des Aufrufs keine weiteren Aktionen in der aufrufenden Stufe zu erledigen sind. Dann ist kein Rücksprung notwendig und Scheme ersetzt einfach den vorherigen Funktionsaufruf durch den neuen, hinterlässt daher keinen "Müll" auf dem stack[1].
Zur Erläuterung des Unterschieds dienen die nachfolgenden Beispielfunktionen.

Ein Element an eine Liste anhängen: normale rekursive Lösung

Eine normale rekursive Variante muss zunächst bis zum Ende der Liste durchgehen, baut dort eine Liste mit dem neuen Element auf und kann erst dann rekursiv nachklappernd die vorigen Elemente der Liste davor setzen.
Wie immer sind die Abbruchbedingung[2] der Rekursion von zentraler Bedeutung:

- wenn die Liste leer ist, wird eine Liste mit dem Element gebildet

Der verbleibende Fall, also der else-Fall besteht dann im rekursiv nachklappernden

1 Wichtig, weil es ggf. das Problem des stack overflow verhindert.
2 Jede Rekursion ohne funktionierende Abbruchbedingungen führt irgendwann zu einer endlosen Kette von Aufrufen und damit zu einem stack overflow

Neuaufbau des bisherigen Teils der Liste. Es wird jeweils das aktuelle Element an den Kopf des bisher erzielten Ergebnisses gesetzt.
Eine Lösung:

```
;;; ===== ein Element an eine Liste anhängen =====
(define
  (anhaengen element liste)
  (if
   (null? liste)      ; Abbruchfall ->
   (list element)     ; Wert definieren durch Funktion!
   (cons              ; else-Fall: Liste rekonstruieren
    (first liste)
    (anhaengen element (rest liste))
    )
   )
  )

(anhaengen 'g '(a b c d e f))
```

Natürlich bleibt die Reihenfolge bei diesem Verfahren in der restliste nicht aufrechterhalten. Das kann in der Regel aber kein Problem sein.

Eine Liste umkehren: Eine endrekursive Lösung

Eine endrekursive Variante arbeitet mit zwei Listen[1], eine für die Ausgangsliste und eine für die aufzubauende Umkehrliste. Diese steht an dieser Stelle für den bei endrekursiven Lösungen immer notwendigen Akkumulator.
Abbruchbedingung ist wieder die leere Liste, dann wird nur noch direkt die Umkehrliste ausgegeben.

```
;;; ===== umkehrliste =====
(define
  (umkehren liste umkehrliste)
  (if
   (null? liste)      ; Abbruchfall ->
   umkehrliste        ; Wert bekannt!
   (umkehren          ; else-Fall: rueberschieben
    (rest liste)
    (cons (first liste) umkehrliste)
    )
   )
  )

(umkehren '(a b c d e f) '())
```

Eine Aufrufhülle wäre beispielsweise

```
(define (kehre liste) (umkehren liste '()))
```

durch die der sinnvollere Aufruf

```
(kehre '(a b c d e f))
```

möglich wäre.

append selbst geschrieben

Ein schönes Beispiel für normal rekursive Programmierung ist eine selbst geschriebene append[2] – Funktion, die wir hier verbinde nennen wollen.

1 Man braucht also eigentlich für diese Funktion eine Aufrufhülle, die man sich sparen kann, wenn die Funktion nicht direkt aufgerufen werden soll.
2 Übrigens kann append mehr, z.B.: (append '(1 2 39) '(2 3 8) '(2 1 2)) → '(1 2 39 2 3 8 2 1 2). Wie das zu programmieren wäre, können wir auch noch lernen, ist zunächst aber nicht wichtig.

```
;;; ===== verbinde =====
; selbst geschriebenes append
(define
  (verbinde liste-1 liste-2)
  (if
   (null? liste-1)
   liste-2
   (cons (car liste-1) (verbinde (cdr liste-1) liste-2))))

(verbinde '(1 2 3 4) '(a b c))
```

Hier erkennt man gut die grundlegenden Prinzipien rekursiver Programmierung:

- Der rekursive Aufruf muss prinzipiell mit Parametern erfolgen, die näher am Abbruchfall liegen, bei Listen der Abbau durch Bilden der Restliste.
- Im Abbruchfall ist das Problem elementar lösbar.

„Man verbindet zwei Listen, indem man das erste Element vor die Liste setzt, die entsteht, wenn man die Restliste der ersten mit der zweiten Liste verbindet. Ist die erste Liste leer, ist die Verbindung einfach die zweite Liste."

Unterschied zu einer OO Lösung

Bei einer typisch objektorientierten Lösung stellt das Listenobjekt ***liste-1*** eine append-Methode zur Verfügung, der als Parameter das Objekt ***liste-2*** übergeben wird. Durch die Methode wird der Zustand des Objekts ***liste-1*** dauerhaft verändert, es enthält die Elemente von ***liste-2*** zusätzlich.
Funktional behalten beide Listen ihren ursprünglichen Zustand – ob der noch benötigt wird oder nicht – und es wird eine neue, verbundene Liste erzeugt[1].

Verbinde endrekursiv

Die Lösung zeigt, dass nicht alles gleich gut für beide Rekursionsmethoden geeignet ist.

```
;;; ===== verbinde endrekursiv =====
; soll die Reihenfolge erhalten bleiben, ist es uebel; hier ohne Aufrufhuelle
(define
  (verbinde liste-1 liste-2 akku)
  (cond
    ((not (null? liste-1))
     (verbinde (rest liste-1) liste-2 (cons (first liste-1) akku)))
    ((not (null? liste-2))
     (verbinde liste-1 (rest liste-2) (cons (first liste-2) akku)))
    (else
     (reverse akku))))

(verbinde '(1 2 3 4) '(a b c) '())
```

1 Auch die objektorientierte append-Methode könnte ein neues Objekt erzeugen und als Wert zurückgeben, ohne die ursprünglichen Listen zu verändern.