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^5RS beschrieben [siehe Helpdesk].
Wir kommen 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 Element als Kopf der Liste zurück[2];
  (cons 67 '(23 6 0 100)) → '(67 23 6 0 100)
- (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;
  (car '(23 6 0 100)) → 23
- (cdr liste) gibt die Restliste hinter dem Kopfelement zurück;
  (cdr '(23 6 0 100)) → '(6 0 100)
- (cond (<bedingung1> <wert, wenn sie erfüllt ist> ...) (...) (else ...)) erzeugt eine Verzweigung, bei welcher der Wert zur ersten zutreffenden Bedingung zurück gegeben wird;
- (reverse liste) gibt die Umkehrliste zur liste zurück;
  (reverse '(23 6 0 100)) → '(100 6 0 23)
- (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
- (let <ggf. name> ((var-1 wert) (...)) auswertungsteil) führt zu einer lokalen Bindung von Variablen und Funktionen (named let), entspricht also dem define bis auf die fehlende Dauerhaftigkeit; die gebundenen Namen haben nur im Auswertungsteil Bedeutung
- (if bedingung wenn-fall <ggf. sonst-fall>) ist eine einfache Verzweigung;

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.
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. Endrekursiven Aufrufe kann man daran erkennen, dass zum Zeitpunkt des Aufrufes alle Parameter des Funktionsaufrufes bekannt sind. Da bei ihnen daher kein Rücksprung notwendig ist, ersetzt Scheme hier einfach den vorherigen Funktionsaufruf durch den neuen und hinterlässt keinen Müll auf dem stack[3].
Die nachfolgenden Beispielfunktionen zum Problem des Herausholen des kleinsten (oder auch größten) Elementes einer Liste erläutern den Unterschied.
Es ist dabei das Problem zu lösen, dass die Funktion eigentlich zwei Aufgaben lösen muss, nämlich einmal den Wert zu finden, andererseits ihn aber auch aus der Liste zu entfernen. Während des Durchlaufens der Liste ist aber noch nicht bekannt, welches das

1 Ein einfaches Beispiel ist die Funktion (display ...), die man allein wegen ihrer Bildschirmausgabe aufruft.
2 Beachten Sie: die ursprüngliche Liste wird dabei – wie in den anderen Fällen – ***nicht*** verändert!
3 Wichtig, weil es das Problem des stack overflow verhindert.

kleinste Element ist: Das weiß man erst am Ende der Liste!

normale rekursive Variante

Eine normale rekursive Variante muss also zunächst bis zum Ende durchgehen und kann erst dann rekursiv nachklappernd das kleinste Element nach vorn durchreichen.
Wie immer sind die Abbruchbedingungen[1] der Rekursion von zentraler Bedeutung:

- wenn die Liste leer ist, ist nichts zu tun
- wenn sie nur noch ein Element hat, ist es das kleinste, das durchzureichen ist

Die verbleibenden Fälle einer normal rekursiven Lösung können so beschrieben werden: Wenn das aktuelle Element am Kopf kleiner ist als das zweite, füge das zweite Element hinter dem ersten Element der Liste ein, die aus dem ersten und der restliste der beiden besteht, sonst entsprechend anders herum. Eine Lösung:

```
;;; ===== kleinstes-nach-vorn =====
; bringt das kleinste Element einer Liste zu ihrem Kopf
(define
  (kleinstes-nach-vorn liste)
  (cond
    ((null? liste) liste)          ; sollte nicht auftreten
    ((null? (cdr liste)) liste)    ; am ende angekommen, dies ist das kleinste
    ((< (car liste) (cadr liste))
     (fuege-hinter-dem-ersten-ein
      (cadr liste)
      (kleinstes-nach-vorn (cons (car liste) (cddr liste)))))
    (else
     (fuege-hinter-dem-ersten-ein
      (car liste)
      (kleinstes-nach-vorn (cons (cadr liste) (cddr liste)))))))
;;; ----- fuege-hinter-dem-ersten-ein -----
; Hilfsfunktion
(define
  (fuege-hinter-dem-ersten-ein element liste)
  (cons (car liste) (cons element (cdr liste))))
```

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

endrekursive Variante

Eine endrekursive Variante arbeitet mit zwei Listen[2] und baut das aktuell kleinste Element immer am Kopf der ersten und größere Elemente immer am Kopf der zweiten ein. Die Abbruchbedingung tritt bei einer leeren Liste ein, dann wird die Liste der kleinen Elemente vor die andere gehängt.

```
;;; ===== kleinstes-nach-vorn =====
; bringt das kleinste Element einer Liste zu ihrem Kopf
(define
  (kleinstes-nach-vorn liste)
  (if (null? liste)
      liste
      (hilf-kleinstes-nach-vorn (cdr liste) (cons (car liste) '()) '())))
;;; -----  -----
; Hilfsfunktion
; Vorbedingung: die Liste klein muss mindestens ein Element enthalten!
```

1 Jede Rekursion ohne funktionierende Abbruchbedingungen führt irgendwann zu einer endlosen Kette von Aufrufen und damit zu einem stack overflow

2 man braucht also eigentlich eine Aufrufhülle, die man sich sparen kann, wenn die Funktion nicht direkt aufgerufen werden soll.

```
(define
  (hilf-kleinstes-nach-vorn liste klein gross)
  (cond
    ((null? liste) (append klein gross))
    ((< (car liste) (car klein))
     (hilf-kleinstes-nach-vorn (cdr liste) (cons (car liste) klein) gross))
    (else ; neues kleinstes Element
     (hilf-kleinstes-nach-vorn (cdr liste) klein (cons (car liste) gross)))))
```

append selbst geschrieben

Ein schönes Beispiel für normal rekursive Programmierung ist eine selbst geschriebene append[1] – Funktion, die wir hier verbinde nennen wollen.

```
;;; ===== 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))))
```

Hier erkennt man gut die grundlegenden Prinzipien rekursiver Programmierung:

- Der rekursive Aufruf muss prinzipiell mit Parametern erfolgen, die näher am Abbruchfall liegen, bei Listen ist das in der Regel der Abbau mit dem 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.“

Bewertung des Sortierens durch Auswahl

Insgesamt lässt sich feststellen, dass Sortieren durch Auswahl mit Scheme weniger einfach ist, als das Sortieren durch Einfügen. Eine weiter gehende Bewertung soll hier nicht erfolgen, da sie für unser Thema nicht wichtig ist. Interessant wird eine Bewertung von Sortierverfahren nur dann, wenn man sich mit der Laufzeit bei großen Datenmengen beschäftigt, die man gut für die im Anhang vorliegenden Beispiele durchtesten kann.

Aufgabe:

Untersuchen Sie das Laufzeitverhalten der verschiedenen Sortierverfahren, stellen Sie es grafisch dar und versuchen Sie Erklärungen zu finden!

1 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.

Anhang: Funktionen zu Sortierverfahren

```
;;; ===== sortiere-ein =====
; sortiert durch Einfügen
(define
  (sortiere-ein liste)
  (cond
    ((null? liste) liste)           ; leere Listen kann man nicht sortieren
    ((null? (cdr liste)) liste)     ; ... einem Element braucht man nicht zu ...
    (else
     (einfuegen liste '()))))

;;; ----- einfuegen -----
; Hilfsfunktion fügt alle Elemente in eine sortierte Liste ein
(define
  (einfuegen unsortierte sortierte)
  (cond
    ((null? unsortierte) sortierte)
    (else
     (einfuegen
      (cdr unsortierte)
      (fuege-ein (car unsortierte) sortierte)))))

;;; ----- fuege-ein -----
; Hilfsfunktion fügt ein Element in eine sortierte Liste ein
(define
  (fuege-ein element liste)
  (cond
    ((null? liste) (cons element '()))  ; gehörte nach hinten
    ((< element (car liste))            ; gehört vor das aktuelle in der Liste
     (cons element liste))
    (else
     (cons (car liste) (fuege-ein element (cdr liste))))))

; Testfall:
(sortiere-ein '(34 525 24 3634 13 1 -3 0 756 85 76 4 756 9086 643 -12 75))

;;; ===== sortiere-aus =====
; sortiert durch Auswahl
(define
  (sortiere-aus liste)
  (cond
    ((null? liste) liste)           ; leere Listen kann man nicht sortieren
    ((null? (cdr liste)) liste)     ; ... einem Element braucht man nicht zu ...
    (else
     (let ((temp (kleinstes-nach-vorn liste)))
       (cons (car temp) (sortiere-aus (cdr temp)))))))

;;; ===== kleinstes-nach-vorn =====
; bringt das kleinste Element einer Liste zu ihrem Kopf
(define
  (kleinstes-nach-vorn liste)
  (cond
    ((null? liste) liste)          ; sollte nicht auftreten
    ((null? (cdr liste)) liste)    ; am ende angekommen, dies ist das kleinste
    ((< (car liste) (cadr liste))
     (fuege-hinter-dem-ersten-ein
      (cadr liste)
      (kleinstes-nach-vorn (cons (car liste) (cddr liste)))))
    (else
     (fuege-hinter-dem-ersten-ein
      (car liste)
      (kleinstes-nach-vorn (cons (cadr liste) (cddr liste)))))))
```

```
;;; ----- fuege-hinter-dem-ersten-ein -----
; Hilfsfunktion
(define
  (fuege-hinter-dem-ersten-ein element liste)
  (cons (car liste) (cons element (cdr liste))))

; Testfall:
(sortiere-aus '(34 525 24 3634 13 1 -3 0 756 85 76 4 756 9086 643 -12 75))

;;; ===== bubble =====
; sortieren durch bubble-sort
(define
  (bubble liste)
  (cond
    ((null? liste) liste)           ; leere Listen kann man nicht sortieren
    ((null? (cdr liste)) liste)     ; ... einem Element braucht man nicht zu ...
    (else
     (durchlauf (cdr liste) (cons (car liste) '()) #t))))
;;; ----- durchlauf -----
; Hilfsfunktion
(define
  (durchlauf liste akku ok?)
  (cond
    ((and (null? liste) ok?) (reverse akku))   ; fertig?
    ((null? liste)
     (durchlauf (cdr (reverse akku)) (cons (car (reverse akku)) '()) #t))
    ((> (car akku) (car liste))
     (durchlauf (cons (car akku) (cdr liste)) (cons (car liste) (cdr akku)) #f))
    (else
     (durchlauf (cdr liste) (cons (car liste) akku) ok?))))

; Testfall:
(bubble '(34 525 24 3634 13 1 -3 0 756 85 76 4 756 9086 643 -12 75))

;;; ===== quick =====
; sortieren durch quick-sort
(define
  (quick liste)
  (cond
    ((null? liste) liste)           ; leere Listen kann man nicht sortieren
    ((null? (cdr liste)) liste)     ; ... einem Element braucht man nicht zu ...
    (else
     (spalte-bewertet (car liste) (cdr liste) '() '()))))

;;; ----- spalte -----
; Hilfsfunktion
(define
  (spalte-bewertet spaltelement unbearbeitete kleine grosse)
  (cond
    ((null? unbearbeitete)          ; vollständig gespalten
     (append
      (if (null? kleine) '() (spalte-bewertet (car kleine) (cdr kleine) '()
'()))
      (cons
       spaltelement
       (if (null? grosse) '() (spalte-bewertet (car grosse) (cdr grosse) '()
'()))))))
    ((< (car unbearbeitete) spaltelement)
     (spalte-bewertet spaltelement (cdr unbearbeitete) (cons (car unbearbeitete)
kleine) grosse))
    (else
     (spalte-bewertet spaltelement (cdr unbearbeitete) kleine (cons (car
unbearbeitete) grosse)))))
```

```
; Testfall:
(quick '(34 525 24 3634 13 1 -3 0 756 85 76 4 756 9086 643 -12 75))

;;; ===== merge-sort =====
; Sortieren durch merge-sort
(define
  (merge-sort liste)
  (cond
    ((null? liste) liste)           ; leere Listen kann man nicht sortieren
    ((null? (cdr liste)) liste)     ; ... einem Element braucht man nicht zu ...
    (else
     (let
         ((liste-von-zwei-listen (spalte liste '() '())))
       (mische
        (merge-sort (car liste-von-zwei-listen))
        (merge-sort (cdr liste-von-zwei-listen)))))))
;;; ----- mische -----
; Hilfsfunktion mischt zwei von unten sortierte Listen
(define
  (mische liste-1 liste-2)
  (cond
    ((null? liste-1) liste-2)
    ((null? liste-2) liste-1)
    ((< (car liste-1) (car liste-2))
     (cons
      (car liste-1)
      (mische (cdr liste-1) liste-2)))
    (else
     (cons
      (car liste-2)
      (mische liste-1 (cdr liste-2))))))
;;; ----- mische -----
; Hilfsfunktion spaltet die unsortierte Liste willkürlich in zwei Teillisten
(define
  (spalte liste liste-1 liste-2)
  (cond
    ((null? liste) (cons liste-1 liste-2))
    ((null? (cdr liste)) (cons (cons (car liste) liste-1) liste-2))
    (else
     (spalte (cddr liste) (cons (car liste) liste-1) (cons (cadr liste) liste-
2)))))
(merge-sort '(34 525 24 3634 13 1 -3 0 756 85 76 4 756 9086 643 -12 75))
;;; ==== grosse-daten-liste =====
; zufällige Liste mit sehr viel Daten
;;; ----- generiere -----
; Hilfsfunktion
(define
  (generiere anzahl maximalwert akku)
  (cond
    ((zero? anzahl) akku)
    (else
     (generiere (sub1 anzahl) maximalwert (cons (zufallsgenerator maximalwert)
akku)))))
;;; ----- zufallsgenerator -----
(define
  (zufallsgenerator maximalwert . parameter)
  (make-pseudo-random-generator)
  (cond
    ((null? parameter) (random maximalwert))
    (else
     (make-pseudo-random-generator) ; neu initialisieren
      (random maximalwert))))
```

```
(define
  grosse-daten-liste
  (generiere 100000 1000000 '()))

; Testfall
;(define sortiert (sortiere-ein (generiere 1000 1000000 '())))  sortiert
;(define sortiert (sortiere-aus grosse-daten-liste)) ; sortiert
;(define sortiert (bubble grosse-daten-liste)) ; sortiert
(define sortiert (quick grosse-daten-liste)) ; sortiert
;(define sortiert (merge-sort grosse-daten-liste)) ; sortiert
'fertig
```