Scheme - Grundlagen

Dies soll kein Scheme – Kurs sein, 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 Datentyp[1] 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))))))
```

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!

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
```

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

```
    (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 let[1]

```
(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))
```

Typisch funktional ist auch die nächste Anwendung, bei der die anderen oben angegebenen Fälle geschachtelt ausgesondert werden.

1 siehe auch allgemeine Erläuterungen am Schluss

```
(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)))))
```

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 aber ist (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 (axact->inexact ...) tun oder mit einer inexact - Zahl verknüpfen.
(inexact? (exact->inexact (/ 7 2))) → #t und (inexact? (* 1.0 (/ 7 2))) ebenfalls.

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 LK nicht zu bearbeiten. Wozu auch? Interessant sind eben nur Konstrukte, die man benötigt[1]!
So haben wir Assoziationslisten zur Speicherung von Graphen verwendet und dann natürlich auch auf die Funktion assoc zur Bestimmung von Nachfolgeknoten zurück gegriffen.
So 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.

Übersetzungen einiger Definitionen aus dem R5RS:

(let (Bindungen) Körper) ***(let Name (Bindungen) Körper)***

Bindungen sind zweielementige Listen, bei denen im Kopf 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.

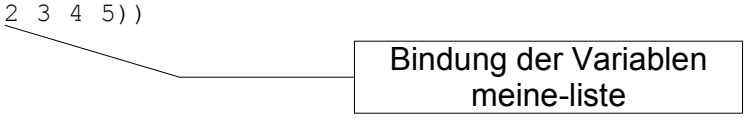
list

(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)
```



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 es denn braucht. Dies ist nämlich eigentlich nicht funktional, sondern prozedural. Fuktional will man in der Regel einen veränderten Wert weiterreichen und dann ist `(cons (car meine-liste) (cons 'zwei (cddr meine-liste)))` das, was man macht.

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!