Listenverarbeitung

Hansruedi Schneider

Definiert man analog zum Mengenbegriff die Aneinanderreihung von beliebigen Objekten zu einem neuen Ganzen als Liste erhält man den einfachsten Datentyp mit dem sämtliche Datenstrukturen modelliert werden können. Viele Programmiersprachen sind über dem Fundament der Listenstruktur aufgebaut. Allen voran LISP, ihr "Kind" LOGO aber auch LUA! Die Modellierung einer Datenstruktur in einer dieser Sprachen geschieht dann über ein beliebig kompliziertes Gebilde von Listen die selber wieder Listen als Elemente enthalten können. Somit ist einleuchtend, dass das wichtigste Werkzeug im Umgang mit Listen die Rekursion ist.

In dem von TI-Basic angebotenen Datentyp der Liste müssen die Elemente (leider) von einfachem Typ sein: Zahlen oder Zeichenketten. Das nachfolgende Beispiel zeigt, wie dieser Mangel zu einer schwer verständlichen Fehlermeldung führt:

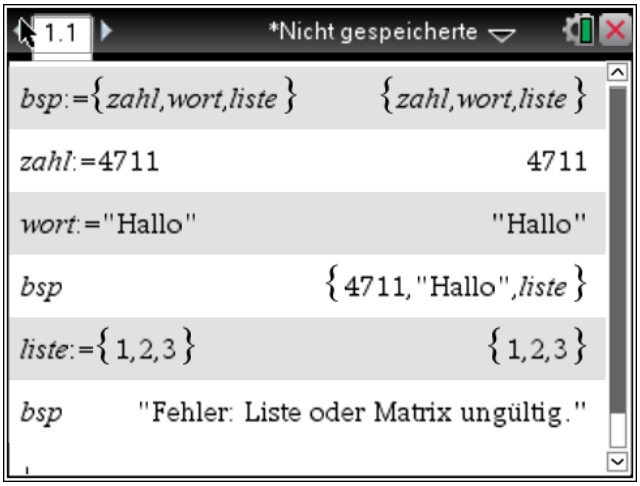

Abb. 1

Im Folgenden werden allein Listen mit endlich vielen natürlichen Zahlen betrachtet. Allein mithilfe weniger Grundbefehle programmieren wir weitere, mächtigere Befehle zur Manipulation unserer eingeschränkten Listen.

Grundbefehle

Im Folgenden bezeichnet *liste* eine Liste in unserem Sinne. Zuerst definieren wir die Funktion **empty**. *empty(liste)* gibt true zurück, wenn die *liste* die leere Liste { } ist. Andernfalls wird false zurückgegeben. Die Funktionen **first** und **last** lesen aus einer Liste ihr erstes bzw. letztes Element.

***butfirst**(liste)* entnimmt einer Kopie von li das erste Element und gibt die Restliste als Funktionswert zurück. Analog: ***butlast**(liste)* entfernt in der Kopie von liste das letzte Element. Das Verknüpfen von zwei Objekten zu einer neuen Liste geschieht mit **join**. Die beiden Parameter von join können wahlweise eine Zahl oder eine Liste sein.

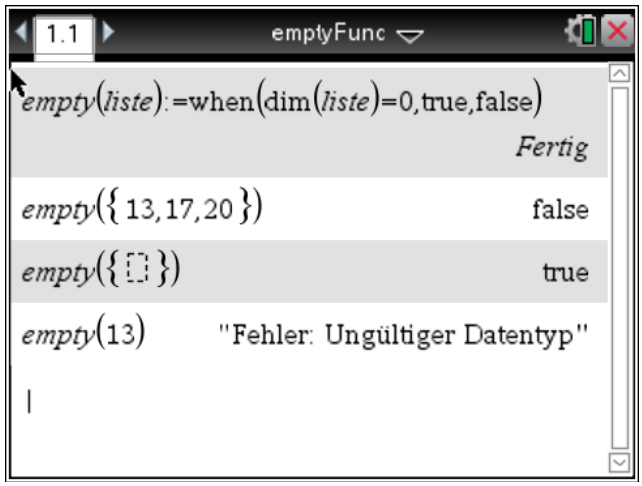

Abb.2

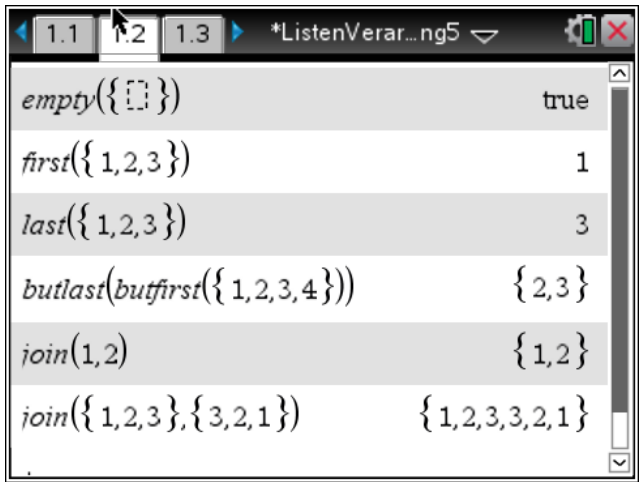

Abb.3

```
Define last(liste)=
Func
Local res
  If not empty(liste) Then
    res:=liste[dim(liste)]
  Else
    res:="Liste ist leer!"
  EndIf
  Return res
EndFunc
```

Abb.4

```
Define butfirst(liste)=
Func
  Local res,k
  If empty(liste) Then
    res:=liste
  Else
    res:=newList(dim(liste)-1)
    For k,1,dim(res)
      res[k]:=liste[k+1]
    EndFor
  EndIf
  Return res
EndFunc
```

Abb.5

```
Define join(l1,l2)=
Func
  If getType(l1)≠"LIST" Then
    l1:={l1}
  EndIf
  If getType(l2)≠"LIST" Then
    l2:={l2}
  EndIf
  Return augment(l1,l2)
EndFunc
```

Abb.6

Alle nun vorgestellten Funktionen zur Listenverarbeitung werden nun ausschliesslich mit mithilfe der vorgestellten Funktionen empty, first, last, butfirst, butlast und join programmiert.

Funktionen für erweiterten Zugriff auf Listen

Die bool'sche Funktion member(x,liste) soll feststellen, ob die Zahl x in der Liste liste enthalten ist. member lässt sich direkt in einer Calculatorseite definieren:

```
member(x,liste):=
  when(empty(liste),false,when(x=first(liste),
                               true,member(x,butfirst(liste))))
```

member arbeitet rekursiv: Ist die Liste leer, befindet sich das Element nicht in der Liste und es wird false zurückgegeben ("Rekursionsanker 1"). Andernfalls hat die Liste ein erstes Element. Ist dieses gleich dem gesuchten Element wird true zurückgegeben ("Rekursionsanker 2"). Andernfalls wenden wir das bis hierhin vorgestellte Verfahren für das gesuchte Element und die Liste ohne ihr erstes Element an (Rekursion). Da die Liste bei jedem Aufruf um ein Element kleiner wird, braucht die Funktion höchsten soviel Berechnungsschritte, wie die ursprüngliche Liste Elemente hat.

Die Definition in der Calculatorseite ist unübersichtlich. Es empfiehlt sich die Funktionen im Programmeditor zu schreiben. An Stelle der when-Funktionen treten if-Anweisungen.

Weiter verwenden wir eine lokale Variable res, in welcher das Resultat berechnet wird und in der letzten Zeile jeweils mit return res zurückgegeben wird.

first und last "lesen" das erste bzw. letzte Element einer Liste. Wir programmieren nun die Verallgemeinerung von first und last welche es erlaubt ein Listenelement aufgrund seiner Nummer (=Index) herauszulesen:

item(nr,liste) gibt den Wert des Elements mit Index nr zurück. Falls nr ein ungültiger Index ist wird die Zeichenkette "<nr>.Listenelement existiert nicht!".

```
Define item(nr,liste)=
Func
  Local res
  If empty(liste) or nr<1 Then
    res:=string(nr)&". Listenelement existiert nicht!"
  ElseIf nr=1 Then
    res:=first(liste)
  Else
    res:=item(nr-1,butfirst(liste))
  EndIf
  Return res
EndFunc
```

Abb.7

Die Funktion member prüft nur, ob ein Element in der Liste ist. Mit demselben Aufwand könnte an Stelle des Rückgabewerts true der Index des Elements (1,2,3,..,n) und bei false der Wert n+1 zurückgegeben werden. Dies realisieren wir mit der Hilfsfunktion pos(x,liste). Den Wert Listenlänge+1 bei nicht Vorkommen zurückzugeben ist unpraktisch. Besser ist es, den Rückgabewert 0 zu verwenden

itemnr(x,liste) gibt den Index des ersten Vorkommens von x in liste zurück. Falls x nicht vorkommt wird 0 zurückgegeben.

```
Define pos(x,liste)=
Func
  Local res
  If empty(liste) or first(liste)=x Then
    res:=1
  Else
    res:=1+pos(x,butfirst(liste))
  EndIf
  Return res
EndFunc
```

Abb.8

```
Define itemnr(x,liste)=
Func
  Local res
  res:=pos(x,liste)
  If res>dim(liste) Then
    res:=0
  EndIf
  Return res
EndFunc
```

Abb.9

Hat man ein Element mit der Funktion itemnr in einer Liste gefunden, möchte man dieses allenfalls entfernen und die Liste ohne das Element mit dem ermittelten Index zurückgeben. Diese "Tätigkeit" übernimmt die Funktion butItem(nr,liste). butItem ist die Verallgemeinerung von butfirst bzw. butlast:

butfirst(liste) entspricht butitem(1,liste), butlast(liste) entspricht butitem(dim(liste),liste).

Zumindest butlast müsste nicht zwingend programmiert werden.

```
Define butitem(nr,liste)=
Func
  Local res
  If empty(liste) or nr<1 Then
    res:=liste
  ElseIf nr=1 Then
    res:=butfirst(liste)
  Else
    res:=join(first(liste),
      butitem(nr-1,butfirst(liste)))
  EndIf
  Return res
EndFunc
```

Abb.10

Mengenoperationen

Nach Cantor ist eine Menge eine Zusammenfassung von wohlunterscheidbaren Dingen zu einem neuen Ganzen. Demnach ist eine Liste in unserem Sinne eine Menge, wenn jedes Element genau einmal enthalten ist. Wir benötigen deshalb zuerst eine Funktion, welche aus einer Liste eine Menge macht:

menge(liste) entfernt aus einer Kopie von liste die "doppelt" vorkommenden Elemente. Wiederum wird das Problem mittels Rekursion gelöst:

Ist *liste* leer, so wird die leere Liste zurückgegeben (Rekursionsanker). Andernfalls wird geprüft, ob *first(liste)* in *butfirst(liste)* liegt. Wenn ja, kommt *first(liste)* im Rest nochmals vor und es wird deshalb *menge(butfirst(liste))* zurückgegeben (Rekursion 1). Andernfalls müssen wir *first(liste)* für die Rückgabe beachten, wir geben deshalb *join(first(liste), menge(butfirst(liste)))* zurück (Rekursion 2).

```
Define menge(liste)=
Func
  Local res
  If empty(liste) Then
    res:=liste
  ElseIf member(first(liste),butfirst(liste)) Then
    res:=menge(butfirst(liste))
  Else
    res:=join(first(liste),menge(butfirst(liste)))
  EndIf
  Return res
EndFunc
```

Abb.11

Für die nun folgenden Funktionen zur Implementation der Mengenoperationen setzen wir voraus, dass die beteiligten Operanden Mengen sind. Die Vereinigung von zwei Mengen kann ganz einfach in einer Calculatorseite vereinbart werden:

union(mg1,mg2):=menge(join(mg1,mg2))

Zur Berechnung des Durchschnitts bzw. der Differenz zweier Mengen nehmen wir wieder die bewährte Rekursion zur Hilfe:

```
Define section(menge1,menge2)=
Func
  Local res
  If empty(menge1) or empty(menge2) Then
    res:={}
  ElseIf member(first(menge1),menge2) Then
    res:=join(first(menge1),section(menge2,butfirst(menge1)))
  Else
    res:=section(menge2,butfirst(menge1))
  EndIf
  Return res
EndFunc
Define difference(menge1,menge2)=
Func
  Local res
  If empty(menge1) Then
    res:=menge1
  ElseIf member(first(menge1),menge2) Then
    res:=difference(butfirst(menge1),menge2)
  Else
    res:=join(first(menge1),difference(butfirst(menge1),menge2))
  EndIf
  Return res
EndFunc
```

Abb.12

Abschliessend zum Exkurs über Mengen geben wir die Funktion zum Prüfen der Gleichheit von zwei Mengen :

```
mgleich(a,b):=
        empty(difference(union(a,b),section(a,b)))
```

Kategorien

Die vorgestellten Funktionen für Mengen lassen leicht sich auf unsere Listen natürlicher Zahlen erweitern. Wir bezeichnen im folgenden eine ungeordnete Liste von natürlichen Zahlen als Kategorie, wohl wissend, dass dieser Begriff weit über unsere Listen hinausgeht. In einer Kategorie darf ein Element mehrfach vorkommen. Weiter: Zwei Kategorien sind gleich, wenn sie in allen Elementen und deren Vielfachheiten übereinstimmen. Die folgenden Beispiele sollen die Verallgemeinerung von *union, section* und *difference* für Kategorien veranschaulichen:

vereinigung({1,2,1,3,2},{3,2,3})={1,1,2,2,2,3,3,3}
durchschnitt({1,2,1,3,2},{3,2,3})={2,3}
differenz({1,2,1,3,2},{3,2,3})={1,2,1}
differenz({3,2,3},{1,2,1,3,2},)={3}

Die Vereinigung zweier Kategorien kann allein mit der Funktion join realisiert werden. Für *durchschnitt* und *differenz* benötigen wir die vorgängig vorgestellten Funktionen *itemnr* und *butitem*.

```
Define durchschnitt(kat1,kat2)=
Func
  Local res,nr
  If empty(kat1) or empty(kat2) Then
    res:={}
  Else
    nr:=itemnr(first(kat1),kat2)
    If nr=0 Then
      res:=durchschnitt(butfirst(kat1),kat2)
    Else
      res:=join(first(kat1),durchschnitt(butfirst(kat1),butitem(nr,kat2)))
    EndIf
  EndIf
  Return res
EndFunc
Define differenz(kat1,kat2)=
Func
  Local nr,res
  If empty(kat1) Then
    res:={}
  Else
    nr:=itemnr(first(kat1),kat2)
    If nr=0 Then
      res:=join(first(kat1),differenz(butfirst(kat1),kat2))
    Else
      res:=differenz(butfirst(kat1),butitem(nr,kat2))
    EndIf
  EndIf
  Return res
EndFunc
```

Abb.13

Die Gleichheit zweier Kategorien kann ebenfalls mit den vorgestellten Funktionen geprüft werden:

```
kgleich(a,b):=
    empty(differenz(differenz(vereinigung(a,b),
                durchschnitt(a,b)),durchschnitt(a,b)))
```

Erklärung: Gilt a=b, so ist durchschnitt(a,b)=a=b. vereinigung(a,b) enthält jedes Element von a(=b) mit doppelter Vielfachheit. "Subtrahiert" man zweimal durchschnitt(a,b), also a bzw. b, so bleibt die leere Menge zurück. Es ist leicht einsehbar, dass für a verschieden von b der Term nicht leer sein kann.

kgleich ist nicht sehr effizient! Besser verwendet man eine eigens für diesen Zweck programmierte Funktion:

```
Define equals(liste1,liste2)=
Func
  Local res,nr
  If empty(liste1) and empty(liste2) Then
    res:=true
  ElseIf not empty(liste1) and not empty(liste2) Then
    nr:=itemnr(first(liste1),liste2)
    If nr>0 Then
      res:=first(liste1)=item(nr,liste2) and
                        equals(butfirst(liste1),butitem(nr,liste2))
    Else
      res:=false
    EndIf
  Else
    res:=false
  EndIf
  Return res
EndFunc
```

Abb.14

Quellen:

Helmut Schauer: *LOGO jenseits der Turtle*, Springer (Kapitel 5: Listenverarbeitung)

Autor:

Hansruedi Schneider, St.Gallen (Ch)
hansruedi.schneider@ksbg.ch