Kurs 1612 "Konzepte imperativer Programmierung"

Musterlösung zur Nachklausur am 08.04.2006

Aufgabe 1

a) Idee: Die Zahl wird so lange durch 10 dividiert, bis sie einstellig ist. Die gesuchte Ziffernzahl ist um eins größer als die Anzahl dieser Divisionen.

```
function countZiff (inZahl : tNatZahl): tNatZahl;
{gibt die Stellenanzahl der inZahl zurück}

  var
  i,
  j : tNatZahl;

begin
  i := inZahl;
  j := 1;

  while i >= 10 do
  begin
    j := j + 1;
    i := i div 10
  end;

  countZiff := j
end;
```

b)

```
function potenziere (inZahl,
                     inPotenz: tNatZahl): tNatZahl;
{berechnet die n-te Potenz einer Zahl}

  var
  i,
  ZwischErg : tNatZahl;

begin
  ZwischErg := 1;
  for i := 1 to inPotenz do
    ZwischErg := ZwischErg * inZahl;

  potenziere := ZwischErg
end;
```

---

c) In der Funktion `isArmstrong` ist die Funktion `potenziere` an der Stelle (2) oder (3) aufzurufen.
Eine geeignete Programmzeile ist:
`ZwischErg := potenziere(LetzteZiffer, AnzDerStellen).`

d)

```
program ArmstrongZahlen;

  var
  i,
  x,
  y : tNatZahl;

begin
  readln(x);
  readln(y);
  for i := x to y do
  begin
    if isArmstrong(i) then
      writeln(i,' ist eine Armstrong-Zahl.');
  end {for}
end;
```

Aufgabe 2

a)

```
function kommtVor(inRefListe:tRefListe; inWert:integer):
                                          boolean;
{Rückgabe true, falls die übergebene Liste ein Element mit
 info-Komponente inWert enthält, sonst false}
  var lauf: tRefListe;

begin
  if inRefListe<>nil then
  begin
    lauf := inRefListe;
    while (lauf^.next <> nil) and (lauf^.info <> inWert) do
      lauf := lauf^.next;
    { Schleife terminiert, wenn ein Vorkommen gefunden wurde
     oder das letzte Element erreicht ist.
     Genau im ersten Fall ist das Ergebnis true: }
    kommtVor := (lauf^.info=inWert)
```

```
  end
  else {inRefListe=nil}
    kommtVor:=false
end;
```

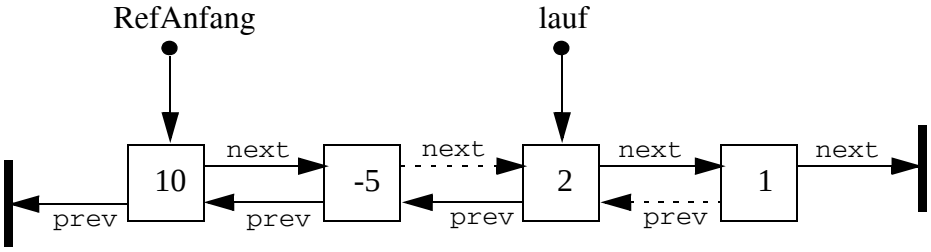
b) Da vorausgesetzt werden durfte, dass genau ein Element mit Wert 0 in der Liste vorkommt, entfällt der Sonderfall der leeren Liste. Wir prüfen daher als erstes, ob die Liste mit 0 beginnt, denn dann ist gar nichts zu tun. Andernfalls suchen wir zunächst das 0-Element, genauer: dessen Vorgänger. Außerdem müssen wir in einer zweiten Schleife noch das letzte Element suchen, da dessen next-Zeiger auch geändert werden muss (siehe Skizze in Aufgabenstellung). Nachdem nun Zeiger auf das erste Element, das letzte Element und den Vorgänger des 0-Elements vorliegen, kann die Verkettungsänderung durch Verändern von drei Zeigern vorgenommen werden.

```
procedure Tausche(var ioRefAnfang: tRefListe);
{ siehe Aufgabenstellung }
var lauf, ende: tRefListe;
begin
  if ioRefAnfang^.info <> 0 then
  { Nur wenn 0 nicht das erste Element ist, muss ueberhaupt
    etwas getan werden }
  begin
    lauf := ioRefAnfang;
    while lauf^.next^.info <> 0 do
      lauf := lauf^.next;
    { lauf zeigt jetzt auf den Vorgaenger des 0-Elements}
    ende := lauf^.next;
    while ende^.next <> nil do
      ende := ende^.next;
    { ende zeigt jetzt auf das letzte Listenelement }
    { Jetzt koennen die Zeiger geaendert werden : }
    ende^.next := ioRefAnfang;
    ioRefAnfang := lauf^.next;
    lauf^.next := nil
  end
end;
```

Aufgabe 3

a) Da in der doppelt verketteten Liste jedes Element auch wieder eine Referenz auf seinen Vorgänger hat, ist es zum Entfernen nicht nötig, den Vorgänger des zu entfernenden Elementes zu suchen, sondern es kann direkt das zu löschende Element gesucht werden.

Die folgende Skizze zeigt die Liste aus der Aufgabenstellung und einen Laufzeiger, mit dem das Element mit info-Wert „2“ gesucht und gefunden wurde:

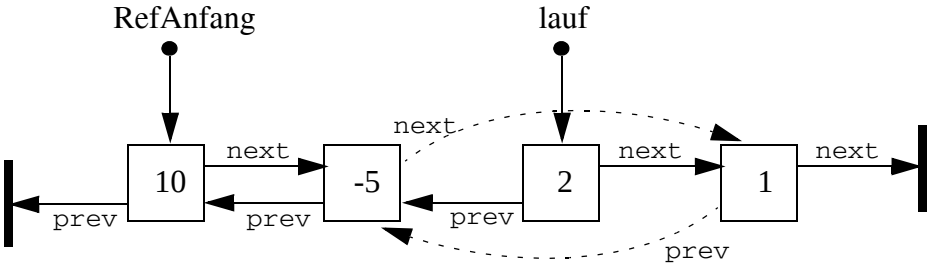


Zum Entfernen müssen die beiden Zeiger geändert werden, die auf dieses Element „2“ verweisen, in der Skizze oben gepunktet dargestellt, also der `next`-Zeiger des Vorgängers und der `prev`-Zeiger des Nachfolgers:

`lauf^.prev^.next := lauf^.next;` und

`lauf^.next^.prev := lauf^.prev;`

Die Situation nach dieser Verkettungsänderung wird in folgender Skizze dargestellt:



Jetzt muss nur noch das Element, auf das `lauf` verweist, gelöscht werden:
`dispose(lauf).`

<u>Sonderfälle</u>:
Der Sonderfall, dass das Element in der Liste nicht vorkommt, wurde in der Aufgabenstellung ausgeschlossen, ebenso wie der Spezialfall der leeren Liste. Damit verbleiben noch folgende zwei Sonderfälle, die zu behandeln sind:

- Das zu löschende Element kann das letzte sein. In diesem Fall hat es keinen Nachfolger und es entfällt das Ändern von `lauf^.next^.prev`. Mit anderen Worten: `lauf^.next^.prev` muss nur geändert werden, falls `lauf^.next<>`**`nil`** ist.
- Das zu löschende Element kann das erste sein. In diesem Fall entfällt das Ändern von `lauf^.prev^.next` und es muss noch der Anfangszeiger `ioRefAnfang` auf das zweite Element `lauf^.next` gesetzt werden.

```
procedure loeschen(var ioRefAnfang: tRefDVListe;
                   inSuch:integer);
{Loescht das erste Listenelement mit Info inSuch. Es muss ein
solches Element existieren!}

var lauf: tRefDVListe;

begin
  {Zu loeschendes Element suchen}
  lauf:=ioRefAnfang;
  while lauf^.info <> inSuch do
    lauf:=lauf^.next;

  {Wenn es Nachfolger gibt, dessen prev aktualisieren.}
  if lauf^.next<>nil then
    lauf^.next^.prev := lauf^.prev;

  {Wenn es Vorgänger gibt, dessen next aktualisieren.}
  if lauf^.prev<>nil then
    lauf^.prev^.next := lauf^.next
  else {Erstes Elem. gelöscht => Anfangszeiger ändern!}
    ioRefAnfang:=lauf^.next;

  {Nun noch das Element selbst löschen}
  dispose(lauf)
end;
```

b) Die Typen `tRefDVListe/tDVListe` sowie `tRefBinBaum/tBinBaum` beschreiben keine Listen oder Bäume, sondern lediglich *Elemente* solcher Datenstrukturen (genauer: `tDVListe` bzw. `tBinBaum` beschreiben die Elemente, `tRefDVListe` bzw. `tRefBinBaum` beschreiben Zeiger auf solche Elemente). Die Elemente eines Binärbaums und einer doppelt verketteten Liste sind gleich aufgebaut: Sie bestehen hier aus einem Integer und zwei Zeigern. Die eigentliche Struktur, z.B. eine Listen- oder Baumstruktur, wird erst durch geeignete Verkettung der einzelnen Elemente zur Laufzeit gebildet und kommt nicht in den Typen zum Ausdruck. Somit stellen diese Typen auch nicht sicher, dass eine Datenstruktur, die der Prozedur `loeschen` aus Teil a) übergeben wird, immer tatsächlich eine korrekte doppelt verkettete Liste ist.

Aufgabe 4

a) Wir geben im Folgenden eine mögliche Implementierung an, wobei wir einen Postorder-Durchlauf absolvieren, d.h. zuerst jeweils die beiden Teilbäume spiegeln und erst dann die jeweilige Wurzel bearbeiten. Die Spiegelung funktioniert aber auch mit Pre- oder Inorder-Durchläufen.
Da ein Baum auch leer sein kann und wir somit den Sonderfall des leeren Baumes ohnehin berücksichtigen müssen, bietet sich diese Prüfung auch gleich als Rekursionsabbruch an, d.h. wir prüfen nicht extra, ob `links` bzw. `rechts` ungleich **`nil`** ist, bevor wir einen rekursiven Subaufruf tätigen.

```
procedure spiegeln(inRefWurzel: tRefBinBaum);
{Spiegelt einen binären Baum durch Vertauschen der
 Nachfolger eines jeden Knotens.}
var merker:tRefBinBaum;

begin
  {Für leeren Baum ist nichts zu tun.}
  if inRefWurzel<>nil then
  begin
    {Spiegeln beider Teilbaeume:}
    spiegeln(inRefWurzel^.links);
    spiegeln(inRefWurzel^.rechts);

    {Nun Spiegelung durch Vertauschen
     beider Teilbäume komplettieren:}
    merker:=inRefWurzel^.links;
    inRefWurzel^.links:=inRefWurzel^.rechts;
    inRefWurzel^.rechts:=merker;
  end;
end;
```

b) Es handelt sich um eine sinnvolle Anwendung der Rekursion, da jeder Knoten des Baumes bearbeitet werden muss und somit der gesamte Baum einmal durchlaufen werden muss. Um bei einem solchen Durchlauf nach Abarbeitung eines Teilbaums wieder zum Vorgängerknoten zurückspringen zu können, wird ein Stapel benötigt, der bei einer iterativen Lösung explizit zu implementieren wäre, während bei der Rekursion der Laufzeitstapel diese Aufgabe übernimmt.

Aufgabe 5

Ein Testfall ist eine Menge von Testdaten, wobei ein Testdatum wiederum ein Paar ist, welches zum Ersten die Eingabedaten (hier: Eingabe für Parameter inZahl) und zum Zweiten die bei dieser Eingabe laut Spezifikation *erwartete* Ausgabe enthält.

Für Boundary- und Interior-Test geben wird exemplarisch zwei verschiedene Notationsmöglichkeiten (eine formale und eine mit natürlichsprachiger Umschreibung der Quersumme) an.

Abweisender Test:

$$T_0 = \{ (e, e) \mid e \in \{0, ..., 9\} \}$$

Boundary-Test:

$$\begin{aligned} T_1 &= \{ (10z+e, z+e) \mid z \in \{1, ..., 9\} \wedge e \in \{0, ..., 9\} \} \\ &= \{ (e, a) \mid e \in \{10, ..., 99\} \wedge \\ &\quad \text{a ist die Summe der zwei Ziffern von e} \} \end{aligned}$$

Interior-Test (2 Durchläufe)

$$\begin{aligned} T_2 &= \{ (100h+10z+e, h+z+e) \mid h \in \{1, ..., 9\} \wedge z,e \in \{0, ..., 9\} \} \\ &= \{ (e, a) \mid e \in \{100, ..., 999\} \wedge \\ &\quad \text{a ist die Summe der drei Ziffern von e} \} \end{aligned}$$

Aufgabe 6

a) Um zu zeigen, dass

$$INV \equiv sum = (z^2 - 2z)/4 \wedge (z \leq X+2) \wedge X_g \wedge z_g$$

eine Invariante der **while**-Schleife ist, müssen wir die Prämisse der **while**-Regel zeigen, also die Gültigkeit der folgenden Programmformel:

```
{ INV ∧ z < (X+2)}
begin
  sum := sum + z;        (1)
  z := z + 2
end;
{ INV }
```

Gemäß der Zusammensetzungsregel und der Sequenzregel ist (1) gültig, wenn wir ein Prädikat R finden, so dass die Programmformeln

```
{ INV ∧ z < (X+2)}
sum := sum + z           (2)
{ R }
```

und

```
{ R }
z := z + 2                (3)
{ INV }
```

gelten. Mit dem Zuweisungsaxiom ermitteln wir ein Prädikat R, für das Programmformel (3) gültig ist:

$$R \equiv sum = ((z+2)^2 - 2(z+2))/4 \wedge (z+2 \leq X+2) \wedge X_g \wedge (z+2)_g$$

Mit diesem R als Nachbedingung ermitteln wir wiederum über das Zuweisungsaxiom eine Vorbedingung R′, so dass {R′} sum := sum + z {R} eine gültige Programmformel ist:

$$R' \equiv sum + z = ((z+2)^2 - 2(z+2))/4 \wedge (z+2 \leq X+2) \wedge X_g \wedge (z+2)_g$$

$$\Leftrightarrow \quad sum + z = (z^2+4z+4-2z-4)/4 \wedge (z \leq X) \wedge X_g \wedge z_g$$

$$\Leftrightarrow \quad sum + z = (z^2 + 2z)/4 \wedge (z \leq X) \wedge X_g \wedge z_g$$

$$\Leftrightarrow \quad sum = (z^2 - 2z)/4 \wedge (z \leq X) \wedge X_g \wedge z_g$$

Wegen

$$INV \wedge z < X+2$$

$$\Leftrightarrow \quad sum = (z^2 - 2z)/4 \wedge z \leq X+2 \wedge z < X+2 \wedge X_g \wedge z_g$$

$$\Rightarrow \quad sum = (z^2 - 2z)/4 \wedge z < X+2 \wedge X_g \wedge z_g$$

$$\Rightarrow \quad sum = (z^2 - 2z)/4 \wedge z \leq X \wedge X_g \wedge z_g$$

(es gilt $z < X+2 \wedge X_g \wedge z_g \Rightarrow z \leq X$,
da z=X+1 dadurch ausgeschlossen ist, dass sowohl X als auch z gerade sind.)

$$\Leftrightarrow \quad R'$$

folgt über die zweite Konsequenzregel die Gültigkeit von (2), womit auch insgesamt die Gültigkeit von (1) gezeigt ist. INV ist also eine Invariante der **while**-Schleife ist.

b) Gesucht ist ein Prädikat P, so dass

```
{ P }
sum := 0;       (PF_b)
z := 2
{ INV }
```

eine gültige Programmformel ist. Laut Sequenzregel ist diese gültig, wenn es ein Prädikat S gibt, so dass

```
{ P }
sum := 0        (4)
{ S }
```

und

```
{ S }
z := 2          (5)
{ INV }
```

gültige Programmformeln sind. Durch Anwenden des Zuweisungsaxioms auf (5) erhalten wir

$$S \equiv \text{sum} = (2^2 - 2*2)/4 \wedge 2 \leq X+2 \wedge X_g \wedge 2_g \ ,$$

so dass (5) gültig ist. Durch anschließende Anwendung des Zuweisungsaxioms auf (4) erhalten wir ein P, so dass auch (4) und somit (PF_b) gültig werden:

$$\begin{aligned} P \equiv\ & 0 = (2^2 - 2*2)/4 \wedge 2 \leq X+2 \wedge X_g \wedge 2_g \\ \Leftrightarrow\ & 2 \leq X+2 \wedge X_g \\ \Leftrightarrow\ & X \geq 0 \wedge X_g \end{aligned}$$

Das Programm `Aufgabe6` kann also nur unter der Voraussetzung die gewünschte Ausgabe liefern, dass die Eingabe X eine nicht-negative gerade Zahl ist.

c) Die Gültigkeit der folgenden Programmformeln dürfen wir laut Aufgabenstellung als bekannt voraussetzen (und haben wir in den Teilen a) und b) auch bewiesen):

```
{ P }
sum := 0;       (PFb)
z := 2
{ INV }
```

sowie

```
{ INV }
Swhile
{ INV ∧ z ≥ (X+2) }
```

(wobei S_{while} kurz für die gesamte While-Schleife stehe).

Nach der Sequenzregel und der Zusammensetzungsregel ist dann insbesondere auch folgende Programmformel gültig:

```
{ P }
begin
  sum := 0;
  z := 2;            (6)
  Swhile
end
{ INV ∧ z ≥ (X+2) }
```

oder kurz:

```
{ P } Aufgabe6 { INV ∧ z ≥ (X+2) }
```

Zum Nachweis der partiellen Korrektheit muss die Gültigkeit von

`{ P } Aufgabe6 { Q }`

gezeigt werden, was mit Hilfe der Konsequenzregel 1 aus der Gültigkeit von (6) gefolgert werden kann, falls zusätzlich die Implikation

$\mathrm{INV} \wedge z \geq (X+2) \Rightarrow Q$

gilt. Wir müssen also nur noch zeigen, dass diese Implikation gilt:

$$
\begin{aligned}
& \mathrm{INV} \wedge z \geq (X+2) \\
\Leftrightarrow\ & \mathrm{sum} = (z^2 - 2z)/4 \wedge (z \leq X+2) \wedge (z \geq X+2) \wedge X_g \wedge z_g \\
\Rightarrow\ & \mathrm{sum} = (z^2 - 2z)/4 \wedge (z = X+2) \\
\Leftrightarrow\ & \mathrm{sum} = ((X+2)^2 - 2(X+2))/4 \\
\Leftrightarrow\ & \mathrm{sum} = (X^2 + 4X + 4 - 2X - 4)/4 \\
\Leftrightarrow\ & \mathrm{sum} = (X^2 + 2X)/4 \\
\Leftrightarrow\ & Q
\end{aligned}
$$

Somit ist das Programm `Aufgabe6` partiell korrekt bezüglich der Spezifikation `{P} S {Q}` mit der gegebenen Nachbedingung `Q` und der in b) ermittelten Vorbedingung `P`.

Aufgabe 7

Im Schleifenrumpf und der Schleifenabbruchbedingung kommen die Variablen `sum` und `z` sowie die Konstante `X` vor. Wir suchen also eine dreistellige Terminierungsfunktion

$$\tau: \mathbb{Z}^3 \rightarrow \mathbb{Z}$$

und wählen

$$\tau(\mathrm{sum}, z, X) := X+2-z.$$

Damit erfüllt τ die Eigenschaft w1).
Es ist noch zu zeigen, dass τ auch die Eigenschaften w2) und w3) erfüllt.

Da `z` in jedem Schleifendurchlauf echt inkrementiert wird und `X+2` konstant ist, wird die Summe `X+2-z` (und somit der Funktionswert von τ) mit jedem Schleifendurchlauf echt kleiner:

$$\tau(\mathrm{sum}+z, z+2, X) = X+2-z-2 = X-z < X+2-z = \tau(\mathrm{sum}, z, X).$$

Die Funktionswerte in Schleifendurchlaufreihenfolge bilden also eine streng monoton fallende Folge. Somit ist w2) erfüllt.

Um w3) zu zeigen, wählen wir `t*=0` und müssen nachweisen, dass vor Ausführung des Schleifenrumpfes immer

---

$$\tau(\text{sum}, z, X) = X+2-z \geq t^*$$

gilt. Diese Ungleichung folgt direkt aus der Schleifeneintrittsbedingung:

$$\begin{aligned} & z < (X+2) \\ \Leftrightarrow\ & X+2-z > 0 \\ \Rightarrow\ & X+2-z \geq 0 \end{aligned}$$

Damit ist w3) gezeigt.

<u>Anmerkung</u>: Weder zum Beweis von w2) noch zum Beweis von w3) haben wir eine t-geeignete Invariante benötigt. Wir haben somit gezeigt, dass die Schleife in jedem Fall terminiert, unabhängig vom Rest des Programms (den Initialisierungen vor der Schleife) und unabhängig von der Vorbedingung P des Programms, die in Aufgabe 7 ermittelt werden sollte.

Aufgabe 8

a) Gemeinsamkeit:
- Beide Verfahren versuchen, die Fehlerfreiheit von Programmen sicherzustellen.

Unterschiede:
- Verifikation ist im Gegensatz zum Testen ein mathematisch-formales Verfahren.
- Ist die Verifikation erfolgreich, ist die Korrektheit (Fehlerfreiheit) des Programms sichergestellt. Dagegen garantiert erfolgreiches Testen noch nicht die Fehlerfreiheit. (Testen kann nur die *Existenz von Fehlern* beweisen, während Verifikation die *Fehlerfreiheit* nachweist.)

b) Ein binärer Suchbaum vom Typ `tRefBinBaum` ist ein Baum, der nach seinen info-Werten sortiert ist, d.h. es gilt für jeden Knoten, dass alle Knoten des linken Teilbaums kleinere[1] Werte als die Wurzel enthalten und alle Knoten des rechten Teilbaums größere Werte. Man kann also einfach entscheiden, ob der aktuelle Knoten den Suchwert enthält, und falls nicht, in welchem der beiden Teilbäume weiterzusuchen ist. Es wird nur ein einziger Pfad (der sog. Suchpfad) durchlaufen, der entweder an einem Knoten mit dem Suchwert oder einem Blatt endet. Ein solcher Pfaddurchlauf ist ganz einfach iterativ zu implementieren, eine solche iterative Implementierung finden Sie im Kurstext im Kapitel zu binären Bäumen. Da diese iterative Lösung ohne Stapel auskommt und mit konstantem Speicherbedarf arbeitet, ist sie einer rekursiven Lösung eindeutig vorzuziehen.

   Es ist also nicht automatisch die Rekursion sinnvoll, nur weil hier ein Baum betrachtet wird. Der Suchbaum heißt übrigens gerade deshalb *Such*baum, weil darin effizient gesucht werden kann: Es muss nicht jedes Element des Baumes betrachtet werden, sondern es wird nur ein einziger Pfad durchlaufen. Bei einem balancierten Baum erhielten wir so eine Laufzeit logarithmischer Größenordnung.

c) Nein.
   Die totale Korrektheit eines Programms besagt nur, dass das Programm für die Vorbedin-

1. Falls im Suchbaum Duplikate (also gleiche Werte) zugelassen werden, muss eindeutig festgelegt sein, ob diese immer im linken oder immer im rechten Teilbaum zu suchen sind.

gung erfüllende Eingabewerte terminiert und ein korrektes Ergebnis liefert. Über das Programmverhalten bei anderen Eingabewerten kann keine Aussage gemacht werden.

d) Aus der Programmformel können wir folgendes ablesen:
Falls die Abbruchbedingung `B` vor der Ausführung des Schleifenrumpfes nicht erfüllt ist, so ist sie nach dem Durchlauf erfüllt.

Wir können daher über die Anzahl der Schleifendurchläufe folgende Aussagen treffen:

- Falls vor Ausführung der Schleife (also vor dem ersten Durchlauf) $\neg$`B` gilt, so wird die Schleife genau einen Durchlauf absolvieren, denn nach dem ersten Durchlauf wird die Abbruchbedingung erfüllt sein.
- Falls vor Ausführung der Schleife dagegen `B` gilt, so absolviert die Schleife mindestens einen, maximal jedoch zwei Durchläufe: Darüber, ob nach dem ersten Durchlauf `B` oder $\neg$`B` gilt, können wir keine Aussage ableiten, aber im ersten Fall wird die Schleife sofort terminieren (da `B` ja die Abbruchbedingung ist), im anderen Fall (wenn also die Abbruchbedingung `B` noch nicht erfüllt ist) wird die Schleife noch einen zweiten Durchlauf absolvieren. Die Programmformel sagt aus, dass dann nach dem zweiten Durchlauf die Abbruchbedingung erfüllt sein wird.

Zusammengefasst kann man also folgende Aussage treffen: Die Schleife wird entweder einen oder zwei Durchläufe absolvieren. In jedem Fall wird sie terminieren.