Kurs 1612 “Konzepte imperativer Programmierung”

Musterlösung zur Klausur am 04.03.2000

Lösung 1 (6 + 5 + 3 + 4 = 18 Punkte)

a)

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

  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;
```

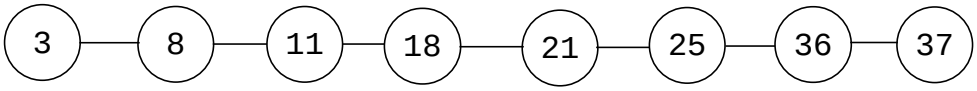
Lösung 2 (10 + 4 = 14 Punkte)

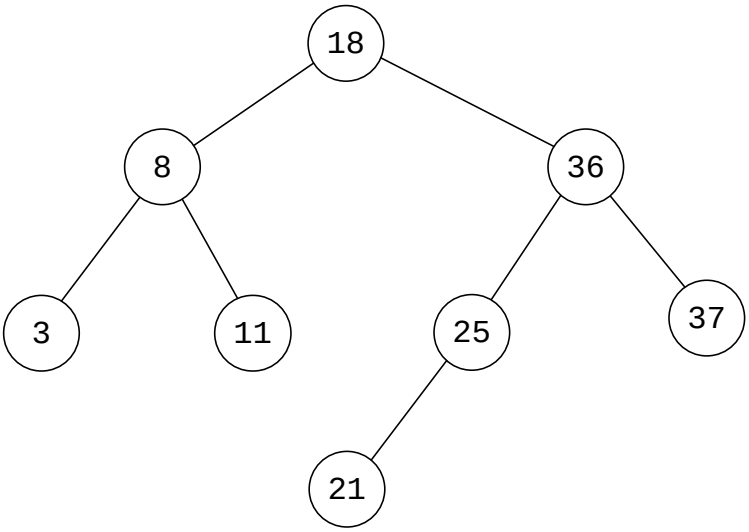
a)

```
function countLeaf(inRefWurzel : tRefBinBaum ): tNatZahl;
{die Funktion liefert die Anzahl der Blätter des inRefWurzel}
  var
  Anzahl : tNatZahl;

begin
  if inRefWurzel = nil then
    Anzahl := 0
  else
  begin
    if (inRefWurzel^.links = nil) and
                              (inRefWurzel^.rechts = nil) then
      Anzahl := 1
    else
      Anzahl := countLeaf(inRefWurzel^.links) +
```

```
                countLeaf(inRefWurzel^.rechts);

    end;
    countLeaf := Anzahl
  end; {Blaetter}
```

b)





Lösung 3 (9 + 9 = 18 Punkte)

Die Schwierigkeit besteht im Aufbau der Liste, da ja am Ende eingefügt werden muß. In der vorgeschlagenen Version wird ein Zeiger auf das Ende der Liste benutzt. (Die Prozedur arbeitet auch, wenn überhaupt kein Element eingegeben wird.) Wir zeigen die Lösung von Teil a) und b) zusammen.

```
program Zahleninversion (input, output);
{ erzeugt eine lineare Liste aus einer Folge einzugebender
  ganzer Zahlen und invertiert diese Liste mit einem Stapel. }

  uses
  Stapel;
```

```
type
tRefListe = ^tListe;
tListe = record
           info : integer;
           next : tRefListe
         end;

var
Zahl : integer;
Liste,
ZgListe : tRefListe;

{ Loesung von Teil a) der Aufgabe }

procedure Zahlenliste (var outListe : tRefListe );
{ liest integer-Zahlen von der Tastatur ein und speichert
  sie in der Reihenfolge der Eingabe in der linearen Liste
  outListe; Null beendet die Eingabe }

  var
  Zahl : integer;
  AnfListe,
  EndListe,
  hilf : tRefListe;

begin
  AnfListe := nil;
  writeln ('Bitte natuerliche Zahlen eingeben, 0=Ende!');
  read (Zahl);
  while Zahl <> 0 do
  begin
    new (hilf);
    hilf^.info := Zahl;
    hilf^.next := nil;
    if AnfListe = nil then
    begin
      { erste Zahl, Zeiger initialisieren }
      AnfListe := hilf;
      EndListe := hilf
    end
    else
    begin
      { zweite und weitere Zahlen }
      EndListe^.next := hilf;
      EndListe := hilf
```

```
      end;
      readln (Zahl)
    end; { while }
    outListe := AnfListe
  end; { Zahlenliste }

begin { ab hier wieder Teil b) der Aufgabe }
  { Zahlen einlesen und Liste aufbauen: }
  Zahlenliste (Liste);
  { Werte auf den Stapel legen: }
  ZgListe := Liste;
  while ZgListe <> nil do
  begin
    Stapel.push (ZgListe^.info);
    ZgListe := ZgListe^.next
  end;
  { Zahlen in invertierter Reihenfolge ausgeben: }
  writeln ('Die invertierte Liste der Zahlen ist: ');
  while not Stapel.isEmpty do
  begin
    Stapel.top (Zahl);
    Stapel.pop;
    write (Zahl)
  end
end. { Zahleninversion }
```

Lösung 4 (6 Punkte)

Das Wort "acbcbccac" gehört zu der Sprache L(G), die von der BNF-Grammatik G erzeugt wird. Dabei ist eine Ableitung für "acbcbccac"

<start> → <A><B>c<B><C><A>
→ a<A><B>c<B><C><A>
→ ac<B>c<B><C><A>
→ acbc<B><C><A>
→ acbcb<C><A>
→ acbcb<A><C><A>
→ acbcbc<C><A>
→ acbcbcc<A>
→ acbcbcca<A>
→ acbcbccac.

Lösung 5 (9 + 8 = 17 Punkte)

a) Um zu zeigen, daß
INV {$2y = i^2 + i + 2$}
eine Invariante der while-Schleife ist, müssen wir die Prämisse der while-Regel zeigen, also die Gültigkeit der Programmformel

```
{INV ∧ i < x}

  begin
    i := i + 1;          (1)
    y := y + i
  end

{INV }
```

Nach der Zusammensetzungsregel ist (1) gültig, wenn

```
{INV ∧ i < x}

  i := i + 1;            (2)
  y := y + i;

{INV}
```

eine gültige Programmformel ist.
Gemäß der Sequenzregel ist (2) gültig, wenn wir ein Prädikat R finden, so dass die Programmformeln

```
{INV ∧ i < x}

  i := i + 1;            (3)

{R}
```

und

```
{R}

  y := y + i;            (4)

{INV}
```

---

gelten.
Wir beginnen mit (4) und erhalten über das Zuweisungsaxiom:

$R \equiv 2(y + i) = i^2 + i + 2.$

Wir wenden das Zuweisungsaxiom noch einmal an und erhalten als Vorbedingung in (3) das Prädikat.

$$
\begin{aligned}
& 2(y + (i + 1)) = (i + 1)^2 + (i + 1) + 2 \\
\Leftrightarrow\ & 2y + 2i + 2 = i^2 + 2i + 1 + i + 1 + 2 \\
\Leftrightarrow\ & 2y + 2i + 2 = i^2 + 3i + 4 \\
\Leftrightarrow\ & 2y = i^2 + i + 2 \\
\Leftrightarrow\ & \ \mathrm{INV}
\end{aligned}
$$

Aus der zweiten Konsequenzregel folgt $\{INV \wedge i < x\} \Rightarrow \{INV\}$ damit ist (1) gültig. Es war zu zeigen, daß INV <u>eine</u> Invariante der while-Schleife ist.

b) Die Vorschläge A und C führen nicht zur Vorbedingung und der Vorschlag D führt nicht auf die Nachbedingung.
Wir wählen B mit $INV' \equiv 2y = i^2 + i + 2 \wedge i \leq x.$ Offensichtlich ist INV’ eine Invariante, da $2y = i^2 + i + 2$ wie in a) gezeigt eine Invariante ist und vor und nach jedem Schleifendurchlauf $i \leq x$ gilt. Die Gültigkeit von

$\{INV'\}$

```
while i < x do
begin
  i := i + 1;
  y := y + i
end
```

$\{INV' \wedge x \leq i\}$

setzen wir voraus. Gemäß der Sequenzregel müssen wir nun das Prädikat S finden, so daß

$\{P \equiv x > 0\}$

```
i := 1
```

$\{S\}$

und

$\{S\}$

```
  y := 2
{INV'}
```

gilt.

Die Anwendung des Zuweisungsaxioms liefert:

$S \equiv 2 * 2 = i^2 + i + 2 \wedge i \leq x$

Nochmaliges Anwenden des Zuweisungsaxioms ergibt:

$4 = 1 + 1 + 2 \wedge 1 \leq x \Leftrightarrow 4 = 4 \wedge 1 \leq x \Leftrightarrow 0 < x \Leftrightarrow P.$

Somit ist die Programmformel (1) gültig.

Nun bleibt noch zu zeigen, daß Q aus INV' und der nicht erfüllten Schleifenbedingung folgt:

$$
\begin{aligned}
& INV' \wedge x \leq i \\
\Leftrightarrow\ & 2y = i^2 + i + 2 \wedge i \leq x \wedge x \leq i \\
\Leftrightarrow\ & 2y = i^2 + i + 2 \wedge x = i \\
\Leftrightarrow\ & y = \frac{1}{2} x^2 + \frac{1}{2} x + 1
\end{aligned}
$$

Folglich ist INV' eine geeignete Invariante.

Lösung 6 (10 Punkte)

Wir wählen gemäß der Aufgabenstellung
$INV \equiv x \geq 0 \wedge i \geq 0$.
Eigenschaften:

w_1) Im Schleifenrumpf und der Schleifenabbruchbedingung kommen die Variablen y und i sowie die Konstante x vor. Wir wählen also eine dreistellige Terminierungsfunktion

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

mit

$$\tau(x, y, i) = x - i.$$

---

w_2) Bei einem Schleifendurchlauf wird i um den Wert 1 erhöht. Gilt vor dem Schleifendurchlauf

$$\tau = x - i,$$

so gilt nach dem Durchlauf

$$\tau = x - (i + 1) = x - i - 1$$

Damit fällt τ bei jedem Durchlauf fällt und w2) ist bewiesen.

w_3) Wir wählen $t^* = 1$ und beweisen, dass vor Ausführung des Schleifenrumpfes stets

$$\tau(x, y, i) = x - i \geq t^* = 1$$

gilt.

Dazu betrachten wir:

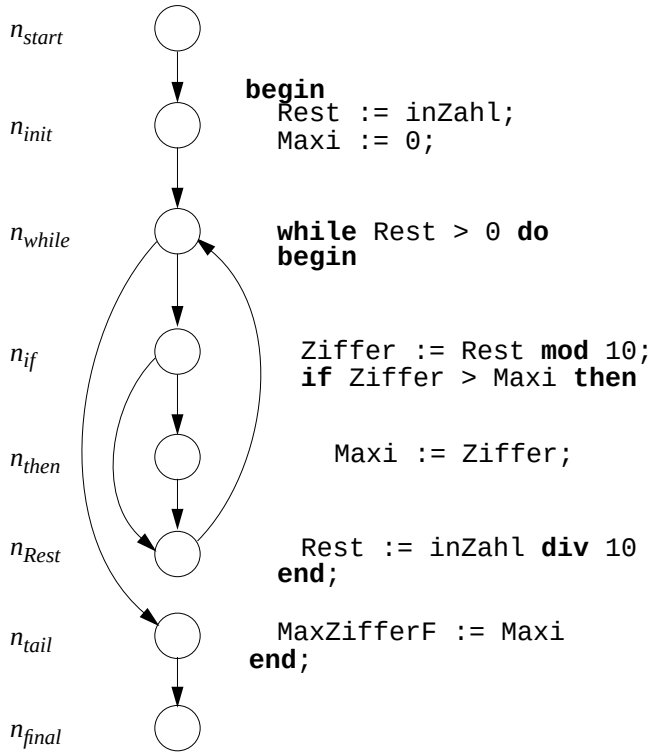
```
{INV ∧ B}
<=> x ≥ 0 ∧ i ≥ 0 ∧ i < x
 => i < x
 => x - i > 0 <=> x - i ≥ 1
```

Damit ist

$$(x, y, i) \geq 1 = t^*.$$

Lösung 7 (6 + 3 + 5 + 3 = 17 Punkte)

a) Der kompakte Kontrollflußgraph für die Prozedur `MaxZifferF` kann folgendermaßen gezeichnet werden:



b) Eine Zweigüberdeckung von 100% erreichen wir bereits mit dem Pfad:

$$(n_{start}, n_{init}, n_{while}, n_{if}, n_{then}, n_{Rest}, n_{while}, n_{if}, n_{Rest}, n_{while}, n_{tail}, n_{final}).$$

Zu diesem Pfad können wir folgenden Testfall T_1 angeben:

$$T_1 = \{(n, m) \in \mathbb{N}^2 \mid 10 < n < 100,\ n=10*a_1+a_0 \text{ und } a_1,a_0 \in \{1,..,9\} \text{ mit } a_1 \leq a_0,\ m = a_0\},$$

mit dem erwarteten Ergebnis `m`. Als Testdatum kann hieraus z.B. das Zahlenpaar (11,1) gewählt werden.

---

c) Für den boundary-interior Pfadtest ergeben sich folgende Testfälle, Pfade und Testdaten:

Testfall	Pfad	Testdatum
$T_1 = \{(n, m) \in \mathbb{N}^2 \mid n = 0 \text{ und } m = 0\}$	$(n_{start}, n_{init}, n_{while}, n_{tail}, n_{final})$	(0, 0)
$T_2 = \{(n, m) \in \mathbb{N}^2 \mid 0 < n < 10$ und $m = n\}$	$(n_{start}, n_{init}, n_{while}, n_{if}, n_{then}, n_{Rest}, n_{while}, n_{tail}, n_{final})$	(1, 1)
$T_3 = \{\}$	$(n_{start}, n_{init}, n_{while}, n_{if}, n_{Rest}, n_{while}, n_{tail}, n_{final})$	nicht ausführbar
$T_4 = \{(n, m) \in \mathbb{N}^2 \mid 10 < n < 100, n=10*a_1+a_0$ und $a_1,a_0 \in \{1,..,9\}$ mit $a_1 \leq a_0, m = a_0\}$	$(n_{start}, n_{init}, n_{while}, n_{if}, n_{then}, n_{Rest}, n_{while}, n_{if}, n_{Rest}, n_{while}, n_{tail}, n_{final})$	(11, 1)
$T_5 = \{(n, m) \in \mathbb{N}^2 \mid 10 < n < 100, n=10*a_1+a_0$ und $a_1,a_0 \in \{1,..,9\}$ mit $a_1 > a_0, m = a_1\}$	$(n_{start}, n_{init}, n_{while}, n_{if}, n_{then}, n_{Rest}, n_{while}, n_{if}, n_{then}, n_{Rest}, n_{while}, n_{tail}, n_{final})$	(21, 2)
$T_6 = \{(n, m) \in \mathbb{N}^2 \mid 10 < n < 100, n=10*a_1+a_0$ und $a_1,a_0 \in \{0,..,9\}$ mit $a_1 > a_0 = 0, m = a_1\}$	$(n_{start}, n_{init}, n_{while}, n_{if}, n_{Rest}, n_{while}, n_{if}, n_{then}, n_{Rest}, n_{while}, n_{tail}, n_{final})$	(10, 1)
$T_7 = \{\}$	$(n_{start}, n_{init}, n_{while}, n_{if}, n_{Rest}, n_{while}, n_{if}, n_{Rest}, n_{while}, n_{tail}, n_{final})$	nicht ausführbar

Anmerkung: Die Angabe der Testfälle ist nach der Aufgabenstellung nicht verpflichtend.

d) Der Fehler wird bereits vom Zweigüberdeckungstest durch die Eingabe der 11 aufgedeckt. Die fehlerhafte Anweisung `Rest := inZahl` **`div`** `10` setzt die Variable `Rest` immer wieder auf den Wert 1, was zu einer Endlosschleife führt. Diese Anweisung müßte `Rest := Rest` **`div`** `10` lauten!