Kurs 1612 “Konzepte imperativer Programmierung”

Musterlösung zur Klausur am 02.03.2002

Lösung 1 (6 Punkte)

Der Kern des Programms `perfekteZahl` besteht aus zwei ineinander geschachtelten **`for`**-Schleifen. Die äußere der beiden Schleifen läuft über den gegebenen Zahlenbereich 1 bis 1000. In der inneren Schleife werden sukzessiv alle Teiler `i` der gerade aktuellen Zahl `n` ermittelt und aufsummiert. Ist diese Summe gleich `n`, dann ist `n` eine perfekte Zahl.

```
program perfekteZahl (input, output);
{ berechnet alle "perfekten Zahlen" zwischen 1 und 1000 }

  const
  MINIMAL = 1;
  MAXIMAL = 1000;

  type
  tNatZahl = 0..maxint;
  tLoesungsbereich = MINIMAL..MAXIMAL;

  var
  n : tLoesungsbereich;
  i,
  Summe : tNatZahl;

begin
  writeln;
  write ('perfekte Zahlen: ');
  { alle Zahlen zwischen 1 und 1000 untersuchen }
  for n := MINIMAL to MAXIMAL do
  begin
    Summe := 0;
    { alle Teiler von n zwischen 1 und n/2 summieren }
    for i := 1 to (n div 2) do
      if n mod i = 0 then
        Summe := Summe + i;
    if n = Summe then
      write (n,', ')
  end
end. { perfekteZahl }
```

Lösung 2 (8 Punkte)

Wir erstellen die Reise, indem wir in der Variablen `hier` die jeweils aktuelle Stadt festhalten. Die von `hier` aus nächste Stadt wird anhand der Entfernungen in der durch `hier` indizierten Matrixzeile ausgesucht, wobei das Feld `besucht` angibt, welche Städte bereits besucht worden sind. Auch hier geben wir wieder ein Rahmenprogramm an, welches den Test der Prozedur `Reise` ermöglicht:

```
program Aufgabe (input, output);
{ Testprogramm fuer die Prozedur Reise }

  const
  MAXORTSZAHL = 10;

  type
  tOrtsIndex = 1..MAXORTSZAHL;
  tEntfernung = 0..maxint;
  tEntfMatrix = array [tOrtsIndex, tOrtsIndex] of tEntfernung;

  var
  EntfMatrix : tEntfMatrix;
  i,
  j : tOrtsIndex;

  procedure Reise (var inMat : tEntfMatrix);
  { berechnet eine kurze Reise aus der Entfernungsmatrix
    inMat }

    type
    tBoolFeld = array [tOrtsIndex] of boolean;
    { zur Markierung bereits besuchter Staedte }

    var
    besucht : tBoolFeld;
    Gesamtstrecke : tEntfernung;
    hier, { gibt die aktuelle Stadt an }
    Spalte,
    i : tOrtsIndex;
    Einzelstrecke : tEntfernung;
    { bisher gefundene kuerzeste Entfernung von “hier“ zu einer
      anderen noch nicht besuchten Stadt }
    naechste : tOrtsIndex;
    { die Stadt mit Entfernung “Einzelstrecke“ von der Stadt
      “hier“ }
```

```
  begin { procedure }
    for i := 1 to MAXORTSZAHL do
      besucht [i] := false;
    { initialisiert das Feld besuchter Staedte }

    Gesamtstrecke := 0;
    hier := 1;
    besucht [hier] := true;
    writeln ('Reise: ');
    writeln (hier);

    { es sind noch MAXORTSZAHL - 1 Staedte zu besuchen }
    for i := 1 to MAXORTSZAHL - 1 do
    begin
      Einzelstrecke := maxint;
      { suche die naechstgelegene, noch nicht besuchte Stadt }
      for Spalte := 1 to MAXORTSZAHL do
        if not besucht [Spalte] then
          if inMat [hier, Spalte] < Einzelstrecke then
          begin
            Einzelstrecke := inMat [hier, Spalte];
            naechste := Spalte
          end;
      writeln (naechste);
      Gesamtstrecke := Gesamtstrecke + Einzelstrecke;
      besucht [naechste] := true;
      hier := naechste
    end;
    writeln (Gesamtstrecke, ' km')
  end; { Reise }

begin
  for i := 1 to MAXORTSZAHL do
    for j := i to MAXORTSZAHL do
    begin
      { Einlesen einer symmetrischen Entfernungsmatrix: }
      write ('Strecke von ', i, 'nach ', j, ': ');
      readln (EntfMatrix [i, j]);
      EntfMatrix [j, i] := EntfMatrix [i, j];
    end;
  Reise (EntfMatrix)
end. { Aufgabe }
```

Lösung 3 (5 + 5 Punkte)

a)

```
function finden (
            inWert : integer;
            inRefListe : tRefListe) : tRefListe;
{ sucht den Wert inWert in der Liste, auf deren Anfang
  inRefListe zeigt. Wird der Wert gefunden, liefert die
  Funktion einen Zeiger auf das Element zurueck,
  ansonsten nil }

  var
  refLauf : tRefListe;

begin
  gefunden := false;
  refLauf := inRefListe;
  while ((refLauf <> nil) and (not gefunden)) do
      if refLauf^.info = inWert then
        gefunden := true
      else
        refLauf := refLauf^.next;
  finden := refLauf
end; { einfuegen }
```

b)

Platzhalter	Ausdruck, Bedingung, Anweisung oder Anweisungsfolge
(1)	`RefLauf2^.info`
(2)	`ioRefM1`
(3)	`RefWert = ioRefM1`

(4)	`begin`
	`  ioRefM1 := ioRefM1^.next;`
	`  dispose(RefWert)`
	`end`

(5)	`RefLauf1 := ioRefM1;`
	`while RefLauf1^.next <> RefWert do`
	`  RefLauf1 := RefLauf1^.next;`
	`RefLauf1^.next := RefLauf1^.next^.next;`
	`dispose(RefWert)`

Lösung 4 (10 Punkte)

Das Problem in dieser Aufgabe ist, dass der Baum durchlaufen werden muss, bis der Vaterknoten des Elementes gefunden ist, auf das inRefZ zeigt. Am einfachsten wird solch ein Baumdurchlauf als rekursive Prozedur implementiert.

```
procedure Trennen (
            inRefWurzel,
            inRefZ : tRefBinBaum);
{ trennt den Teilbaum, auf dessen Wurzel inRefZ zeigt,
  vom Baum ab, auf dessen Wurzel inRefWurzel zeigt; zeigen
  inRefWurzel und inRefZ auf denselben Knoten, geschieht
  nichts }

begin
  if inRefWurzel <> inRefZ then
  begin
    if inRefWurzel^.links = inRefZ then
      inRefWurzel^.links := nil
    else
      if inRefWurzel^.links <> nil then
        Trennen (inRefWurzel^.links, inRefZ);
    if inRefWurzel^.rechts = inRefZ then
      inRefWurzel^.rechts := nil
    else
      if inRefWurzel^.rechts <> nil then
        Trennen (inRefWurzel^.rechts, inRefZ)
  end
end; { Trennen }
```

Lösung 5 (2 + 6 Punkte)

a) Eine vollständige Anweisungsüberdeckung wird durch den Pfad

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

erreicht.

Ein Testdatum zu diesem Pfad ist: ((1, 1), 1)

b) Folgender Pfad umgeht die Ausführung der while-Schleife:

$(n_{start}, n_{init}, n_{while}, n_{tail}, n_{final})$

Der zugehörige assoziierte Testfall ist: T_1={((a, 0), 1) | a aus {1, 2, 3, ...}}

Zu genau einem Schleifendurchlauf gehören die beiden folgenden Pfade mit ihren assoziierten Testfällen:

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

Der zugehörige assoziierte Testfall ist: T_2={((a, 1), a) | a aus {1, 2, 3, ...}}

$(n_{start}, n_{init}, n_{while}, n_{if}, n_{do}, n_{while}, n_{tail}, n_{final})$

Zu diesem Pfad existiert kein Testdatum: T_3={ }

Die while-Schleife wird genau zweimal durchlaufen bei den folgenden vier Pfaden und ihren assoziierten Testfällen:

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

Der zugehörige assoziierte Testfall ist: T_4={((a, 3), a^3) | a aus {1, 2, 3, ...}}

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

Zu diesem Pfad existiert kein Testdatum: T_5={ }

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

Der zugehörige assoziierte Testfall ist: T_6={((a, 2), a^2) | a aus {1, 2, 3, ...}}

$(n_{start}, n_{init}, n_{while}, n_{if}, n_{do}, n_{while}, n_{if}, n_{do}, n_{while}, n_{tail}, n_{final})$

Zu diesem Pfad existiert kein Testdatum: T_7={ }

Lösung 6 (6 + 3 Punkte)

a) Um zu zeigen, dass

$$\mathtt{INV} \equiv \mathtt{sum} = (z*z - 2*z)/4 \wedge (z \leq X+2)$$

eine Invariante der **while**-Schleife ist, müssen wir die Prämisse der **while**-Regel zeigen, also die Gültigkeit der 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. Wir beginnen mit (3) und erhalten

$$R \equiv \mathtt{sum} = ((z+2)^2 - 2*(z+2))/4 \wedge (z+2 \leq X+2)$$

über das Zuweisungsaxiom. Wir wenden das Zuweisungsaxiom noch einmal an und erhalten als Vorbedingung in (2) das Prädikat

$$\mathtt{sum} + z = ((z+2)^2 - 2*(z+2))/4 \wedge (z+2 \leq X+2)$$

$$\Leftrightarrow \quad \mathtt{sum} = (z^2 - 2z)/4 \wedge z \leq X$$

Wegen

$$\mathtt{INV} \wedge z < X+2$$

$$\Leftrightarrow \quad \mathtt{sum} = (z^2 - 2z)/4 \wedge z \leq X+2 \wedge z < (X+2)$$

$$\Rightarrow \quad \mathtt{sum} = (z^2-2z)/4 \wedge z < (X+2)$$

$$\Rightarrow \quad \mathtt{sum} = (z^2-2z)/4 \wedge z \leq X \text{ (da X und z immer gerade sind)}$$

folgt über die zweite Konsequenzregel die Gültigkeit von (1).
Damit ist gezeigt, dass INV eine Invariante der **while**-Schleife ist.

---

b) Wir zeigen zunächst die Gültigkeit von folgender Programmformel (1):

```
{ P ≡ X ≥ 0 ∧ X gerade }
sum := 0;
z := 2;
{ INV }
```

Gemäß der Sequenzregel müssen wir ein Prädikat S finden, so dass

```
{ P ≡ X ≥ 0 ∧ X gerade}
sum := 0
{ S }
```

und

```
{ S }
z := 2
{ INV }
```

gültige Programmformeln sind. Durch Anwenden des Zuweisungsaxioms erhalten wir

$S \equiv sum = (2^2 - 2*2)/4 \wedge 2 \leq X+2.$

Durch Anwendung des Zuweisungsaxioms auf die Zuweisung `sum := 0` erhalten wir aus S:

$0 = (2^2 - 2*2)/4 \wedge 2 \leq X+2$

$\Leftrightarrow \quad 2 \leq X+2$

$\Leftrightarrow \quad X \geq 0$

Da $P \Rightarrow X \geq 0$, folgt über die zweite Konsequenzregel die Gültigkeit von (1) .

Wir zeigen jetzt, dass Q aus INV und der nicht erfüllten Schleifenbedingung folgt:

$$
\begin{aligned}
& INV \wedge z \geq X+2 \\
\Leftrightarrow \quad & sum = (z^2-2*z)/4 \wedge z=X+2 \\
\Rightarrow \quad & sum = ((X+2)^2-2*(X+2))/4 \\
\Rightarrow \quad & sum = (X^2+4*X+4-2*X-4)/4 \\
\Rightarrow \quad & sum = (X^2+2*X)/4 \equiv Q
\end{aligned}
$$

Nach Anwendung der ersten Konsequenzregel, der Sequenzregel und der Zusammensetzungsregel ist damit die partielle Korrektheit des Programms `Aufgabe6` gezeigt.

Lösung 7 (6 Punkte)

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

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

und wählen

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

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

Vor einem Schleifendurchlauf ist

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

Nach einem Durchlauf gilt

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

Die Funktionswerte in Schleifendurchlaufreihenfolge bilden 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

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

gilt. Da

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

ist, folgt diese Forderung unmittelbar aus der Schleifenbedingung. Damit ist w3) gezeigt.

Lösung 8 (5 + 6 Punkte)

a)

wahr	falsch
B, D	A, C, E

b) **Aussage A ist falsch**, da eine Schleife, die stets terminiert, sehr wohl unerwünschte Ergebnisse liefern, d.h. Fehler enthalten kann. Diese Fehler kann man z.B. mit einem boundary-interior Test aufdecken.

Aussage D ist wahr.
Die Gültigkeit der Programmformel

`{x = 4} x := x + 1 {x = 5}`

kann ohne Probleme nachgewiesen werden. Aus

`false` $\Rightarrow$ `x = 4`

und der zweiten Konsequenzregel folgt unmittelbar die Gültigkeit der Programmformel

`{false} x := x + 1 {x = 5}`.

Aussage E ist falsch.
Offensichtlich ist `{true}` eine Invariante für *jede* while-Schleife, da für jede Schleife `{true}` sowohl vor als auch nach Ausführung jedes Schleifendurchlaufs erfüllt ist. Wäre Aussage E wahr, so hieße dies, dass *jede* while-Schleife dieselbe Semantik haben müsste. Da dies offensichtlich nicht der Fall ist, kann Aussage E nur falsch sein.