Kurs 1612 “Konzepte imperativer Programmierung”

Musterlösung zur Klausur am 29.03.2003

Aufgabe 1

a)

```
function ZeilenSumme(inZeile: tBereich; var inMatrix):integer;
{....}
var sum :integer;
    i: tBereich;
begin
  sum := 0;
  { In einer Schleife werden alle Werte der Zeile inZeile
    summiert }
  for i := 1 to N do
    sum := sum + inMatrix[inZeile,i];
  ZeilenSumme := sum
end;
```

b)

```
function MaxZeile(var inMatrix:tMatrix): tBereich;
{....}
var maxIx, {Index des bislang maximalsten Zeilenwertes }
        i: tBereich;
begin
  maxIx := 1;
  for i := 2 to N do
    if ZeilenSumme(i,inMatrix)>ZeilenSumme(maxIx,inMatrix) then
    { Neues Maximum gefunden }
      maxIx := i;
  MaxZeile := maxIx
end;
```

c)

```
function IstGroesser(var inMatrixA,inMatrixB: tMatrix):boolean;
{...}
begin
  IstGroesser :=
      ZeilenSumme(MaxZeile(inMatrixA),inMatrixA) >
      ZeilenSumme(MaxZeile(inMatrixB),inMatrixB)
end;
```

Aufgabe 2

```
procedure Tausche(var ioRefAnfang: tRefListe);
{...}
var lauf, ende: tRefListe;
begin
  if ioRefAnfang^.info <> 0 then
  { Nur wenn 0 nicht das erste Element ist, muss ueberhaupt
    etwas gemacht werden }
  begin
    lauf := ioRefAnfang;
    while lauf^.next^.info <> 0 do
      lauf := lauf^.next;
    ende := lauf^.next;
  { lauf zeigt jetzt auf den Vorgaenger des 0-Elements}
    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

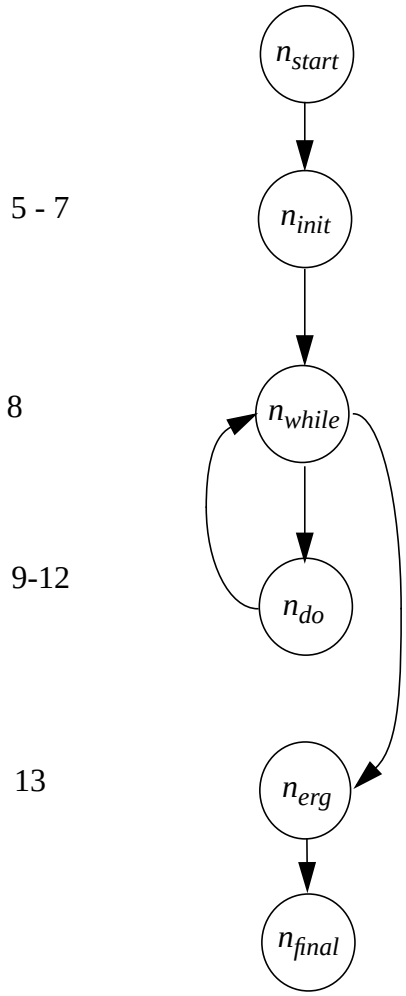
Platzhalter	Anweisung
(1)	ioRefAnfang^.info <> inWert
(2)	gefunden := true
(3)	lauf := lauf^.next
(4)	vor^.next := lauf^.next
(5)	lauf^.next := ioRefAnfang
(6)	ioRefAnfang := lauf

Aufgabe 4

```
function Anzahl(inRefWurzel: tRefBinBaum):tNatZahl;
{...}
begin
  if inRefWurzel=nil then
  { Der leere Baum hat keine inneren Knoten }
    Anzahl := 0
  else
    if (inRefWurzel^.links=nil) and
       (inRefWurzel^.rechts=nil) then
    { Ein einknotiger Baum hat keinen inneren Knoten }
      Anzahl := 0
    else
    { Innerer Knoten liegt vor, die Gesamtanzahl ergibt sich
     aus der Anzahl der inneren Knoten im linken und rechten
    Teilbaum addiert mit 1 fuer die Wurzel}
      Anzahl := 1 + Anzahl(inRefWurzel^.links)+
                    Anzahl(inRefWurzel^.rechts);
end;
```

Aufgabe 5

a)



Kompakter Kontrollflußgraph von Fakultaet

b) Keinmaliger Durchlauf:
(n_{start}, n_{init}, n_{while}, n_{erg}, n_{final})
Testdatum: (0,1)
Testfall: T= {(0,1)}

Einmaliger Durchlauf:
(n_{start}, n_{init}, n_{while}, n_{do}, n_{while},n_{erg}, n_{final})
Testdatum: (1,1)
Testfall: T={(1,1)}

Zweimaliger Durchlauf:
(n_{start}, n_{init}, n_{while}, n_{do}, n_{while}, n_{do}, n_{while},n_{erg}, n_{final})

Testdatum: (2,2)
Testfall: T={(2,2)}

c) Der Fehler wird in jedem Fall gefunden, da die Funktion unabhängig vom Eingabewert immer 0 ausgibt.

Aufgabe 6

a) Zu zeigen ist die Gültigkeit der Prämisse der *while Regel*. Es muss die Programmformel :

```
{(6*sum = 2*i^3+3*i^2+i)∧(ZAHL≥ i)∧(i<ZAHL)}
  begin
    i := i+1;
    sum := sum + i*i                                  (1)
  end
{(6*sum = 2*i^3+3*i^2+i)∧(ZAHL≥ i)}
```

gültig sein.

Nach dem *Zuweisungsaxiom* ist

```
{(6*(sum +i*i) = 2*i^3+3*i^2+i)∧(ZAHL≥ i)}
  sum := sum + i*i;
{(6*sum = 2*i^3+3*i^2+i)∧(ZAHL≥ i)}
```

eine gültige Programmformel.
Ebenfalls nach dem *Zuweisungsaxiom* ist

```
{(6*(sum +(i+1)*(i+1)) = 2*(i+1)^3+3*(i+1)^2+(i+1))∧(ZAHL≥ i+1)}
  i := i + 1;
{(6*(sum +i*i) = 2*i^3+3*i^2+i)∧(ZAHL≥ i)}
```

eine gültige Programmformel. Die Gültigkeit von (2)

```
{(6*(sum +(i+1)*(i+1)) = 2*(i+1)^3+3*(i+1)^2+(i+1))∧(ZAHL≥ i+1)}
  begin
    i := i + 1;
    sum := sum + i*i                                  (2)
  end
{(6*sum = 2*i^3+3*i^2+i)∧(ZAHL≥ i)}
```

ergibt sich aus der Anwendung der *Sequenz*- und der *Zusammensetzungsregel*.
Der erste Term der Vorbedingung von (2) kann folgendermassen umgeformt werden:

$$6(sum +(i+1)*(i+1)) = 2(i+1)^3+3(i+1)^2+(i+1)$$
$$\Leftrightarrow 6sum+6i^2+12i+6 = 2i^3+6i^2+6i+2+3i^2+6i+3+i+1$$
$$\Leftrightarrow 6sum = 2i^3 + 3i^2 + i$$

Da ferner gilt: $(ZAHL \geq i) \wedge (i<ZAHL) \Rightarrow (ZAHL \geq i+1)$
gilt die Implikation

$$(6*sum = 2*i^3+3*i^2+i) \wedge (ZAHL \geq i) \wedge (i<ZAHL)$$
$$\Rightarrow (6*(sum +(i+1)*(i+1)) = 2*(i+1)^3+3*(i+1)^2+(i+1)) \wedge (ZAHL \geq i+1)$$

und die Anwendung der *zweiten Konsequenzregel* garantiert die Gültigkeit von (1).
Damit ist gezeigt, dass INV tatsächlich eine Invariante der while-Schleife ist.

b) Zum Nachweis der partiellen Korrektheit muss noch gezeigt werden:
1. Die Invariante und die negierte Schleifenbedingung implizieren Q:

$$INV \wedge (i \geq ZAHL)$$
$$\Leftrightarrow (6*sum = 2*i^3+3*i^2+i) \wedge (ZAHL \geq i) \wedge (i \geq ZAHL)$$
$$\Leftrightarrow (6*sum = 2*i^3+3*i^2+i) \wedge (ZAHL=i)$$
$$\Rightarrow 6*sum = 2*ZAHL^3+3*ZAHL^2+ZAHL$$
$$\Leftrightarrow Q$$

2. Die folgende Programmformel (3)

$$\{P\}$$
```
  sum := 0;
  i := 0;
```
$$\{(6*sum = 2*i^3+3*i^2+i) \wedge (ZAHL \geq i)\} \qquad (3)$$

ist gültig.

Nach dem *Zuweisungsaxiom* ist

$$\{(6*sum = 2*0^3+3*0^2+0) \wedge (ZAHL \geq 0)\} \Leftrightarrow \{(6*sum=0) \wedge (ZAHL \geq 0)\}$$
```
  i:= 0;
```
$$\{(6*sum = 2*i^3+3*i^2+i) \wedge (ZAHL \geq i)\}$$

gültig.
Ebenfalls nach dem *Zuweisungsaxiom* gültig ist:

$$\{(6*0=0) \wedge (ZAHL \geq 0)\} \Leftrightarrow \{(ZAHL \geq 0)\} \Leftrightarrow \{P\}$$
```
  sum := 0;
```
$$\{(6*sum=0) \wedge (ZAHL \geq 0)\}$$

Die Anwendung der *Sequenzregel* ergibt dann sofort die Gültigkeit von (3).

Aufgabe 7

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

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

und wählen

$$\tau(\texttt{i, sum, ZAHL}) := \texttt{i}.$$

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

Vor einem Schleifendurchlauf ist

$$\tau(\texttt{i, sum, ZAHL}) = \texttt{i}.$$

Nach einem Durchlauf gilt

$$\tau(\texttt{i, sum, ZAHL}) = \texttt{i+1} > \texttt{i}.$$

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

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

$$\tau(\texttt{i, sum, ZAHL}) = \texttt{i} <= \text{ZAHL} = \texttt{t*}$$

gilt. Dies folgt aber sofort aus der Schleifenbedingung B.

Aufgabe 8

a) Ja, S’ kann trotzdem total korrekt sein. Mit Hilfe einer Terminierungsfunktion kann nicht in allen Fällen die Terminierung total korrekter Programme formal nachgewiesen werden. Sollten auch andere Beweisansätze scheitern, kann immer noch keine Aussage bezüglich der totalen Korrektheit von S’ gemacht werden. Erst wenn formal die Terminierung bewiesen oder falsifiziert wurde (z.B. durch Eingabedaten, die zu einer Endlosschleife führen) kann die Frage nach der totalen Korrektheit von S’ endgültig beantwortet werden.

b) Nein, damit ist noch nichts nachgewiesen. Nur wenn man zeigen kann, dass für diese Eingabedaten P wahr ist, hat man bewiesen, dass S’ nicht total korrekt ist. Über das Verhalten eines total korrekten Programms für Eingabedaten, die nicht der Spezifikation genügen, kann keine Aussage gemacht werden.

c) Man sollte den boundary interior Test durchführen, da mit diesem Verfahren explizit Schleifen getestet werden. Ein partiell korrektes Programm ohne Prozeduren oder Funktionen ist nur dann nicht total korrekt, wenn es Endlosschleifen enthält.

d) Wenn S’ für Eingabedaten terminiert, dann ist das Ergebnis korrekt, da S’ partiell korrekt ist. Auf die Angabe des erwarteten Ergebnisses kann daher verzichtet werden.