Kurs 1612 “Konzepte imperativer Programmierung”

Musterlösung zur Klausur am 01.03.2003

Aufgabe 1

a)

```
procedure InitBoolFeld(inWert: boolean; var ioFeld: tBoolFeld);
{ Initialisiert ioFeld mit inWert }
var i: tBereich;
begin
  for i := 1 to GRAD do
    ioFeld[i] := inWert
end;
```

b)

```
function TesteBoolFeld(inFeld: tBoolFeld):boolean;
{Liefert true, wenn alle Elemente von inFeld true sind }
var i: integer;
    hilf: boolean;
begin
   i:=0;
   hilf := true;
   while hilf and (i<GRAD) do
  {Die Schleife endet, wenn ein false-Wert gefunden oder das
    Ende des Felds erreicht wird }
   begin
      i:=i+1;
      hilf := inFeld[i]
   end;
   TesteBoolFeld := hilf
end;
```

c)

```
function IstHalb(inMatrix: tMatrix):boolean;
{Testet, ob inMatrix halblateinisch ist }
var boolFeldZ, boolFeldS : tBoolFeld;
    i,j: integer;
    hilf : boolean;
begin
  hilf := true;
  i:=1;
  while hilf and (i<=GRAD) do
  {Die Schleife wird beendet, wenn eine Zeile gefunden wird,
    die nicht der halblatein-Bedingung genuegt oder alle Zeilen
  untersucht worden sind }
```

```
  begin
     InitBoolFeld(false, boolFeldZ);
     for j:= 1 to GRAD do
        boolFeldZ[inMatrix[i,j]] := true;
     {Alle in der aktuellen Zeile vorkommenden Werte wurde in
      dem boolschen Feld gespeichert }
     hilf := TesteBoolFeld(boolFeldZ);
      {Es wurde getestet, ob die i.Zeile die halblatein
      Bedingung erfüllt }
     i:=i+1;
  end;
  IstHalb := hilf;
end;
```

Aufgabe 2

```
procedure Tausche(var ioRefListe: tRefListe);
{s. Aufgabenstellung}

var such,
    vor,
    nach : tRefListe;

begin
  vor := NIL;
  such := ioRefListe;
  while such^.info <> 0 do
  begin
    vor := such;
    such := such^.next
  end;
  nach := such^.next^.next;
  {Jetzt sind alle Zeiger positioniert. such zeigt auf das
  Element mit dem Wert 0, vor auf dessen Vorgaenger oder auf nil,
  nach auf den Nachfolger von such^.next}

  if vor=NIL then
  {D.h. es muss am Anfang getauscht werden }
    ioRefListe := such^.next
  else
    vor^.next := such^.next;
```

```
  such^.next^.next := such;
{such^.next ist ungleich nil, da nach Voraussetzung 0 in der
Liste nicht als letztes Element vorkommt }
  such^.next := nach
end;
```

Aufgabe 3

.

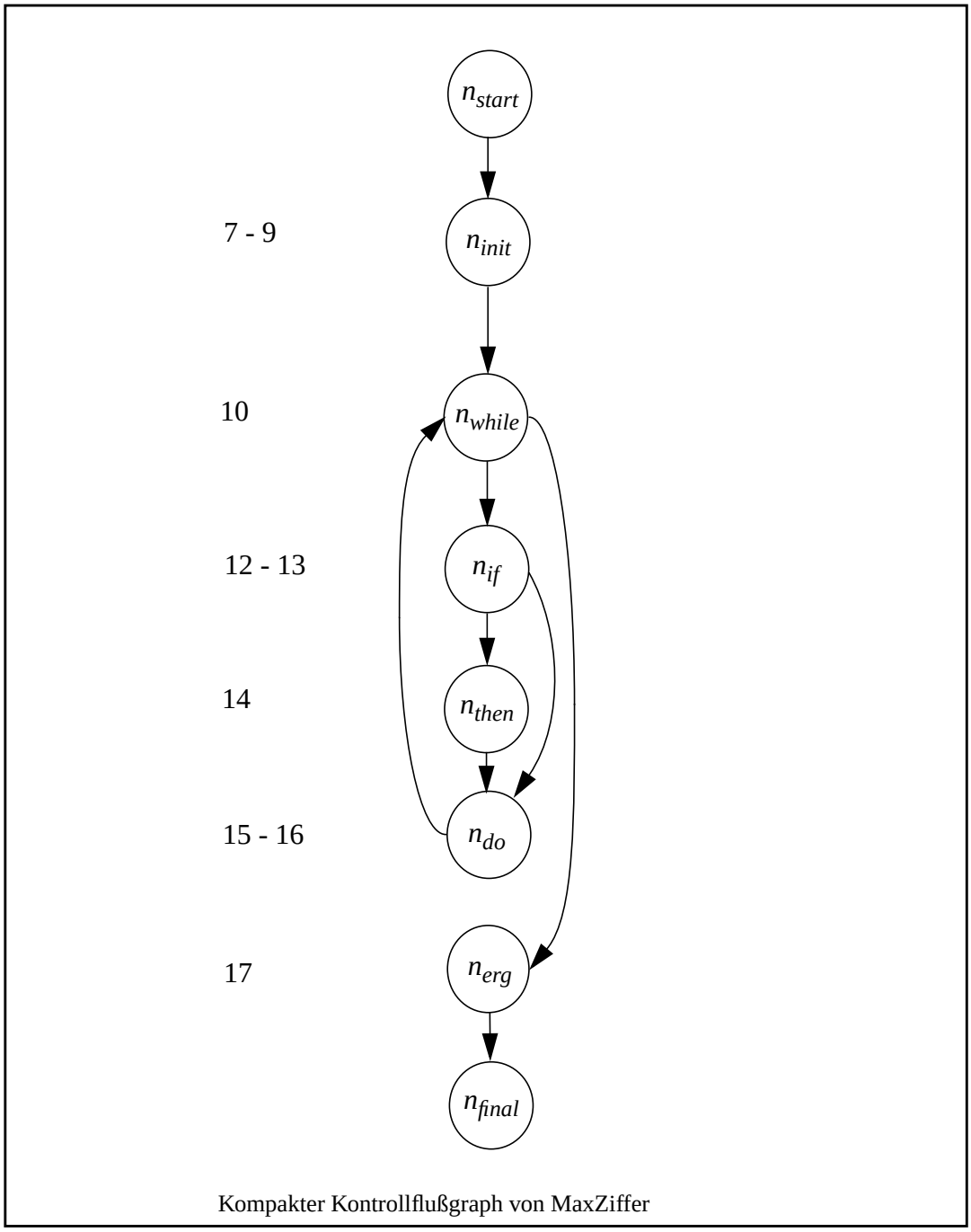
Platzhalter	Anweisung
(1)	ioRefAnfang := elem;
(2)	elem^.next := NIL;
(3)	elem^.next := ioRefAnfang;
(4)	ioRefAnfang := elem;
(5)	ende := true;
(6)	lauf := lauf^.next;
(7)	elem^.next :=lauf^.next
(8)	lauf^.next := elem;

Aufgabe 4

```
procedure HRZwei(inRefWurzel:tRefBinBaum; var ioDrucken :boole-
an);
{Gibt jeden zweiten Knotenwert der Hauptreihenfolge aus }
begin
  if inRefWurzel<>NIL then
  { Nur wenn der Baum nicht leer ist... }
  begin
    if ioDrucken then
    { Muss der Wert ausgegeben werden? }
      writeln(inRefWurzel^.info);
    ioDrucken := not ioDrucken;
    HRZwei(inRefWurzel^.links, ioDrucken);
    HRZwei(inRefWurzel^.rechts, ioDrucken)
  end;
end;
```

Aufgabe 5

a)



Kompakter Kontrollflußgraph von MaxZiffer

b) Folgender Pfad **umgeht** die Schleife: (n_{start} , n_{init} , ,n_{while} , n_{erg} , n_{final}). Das einzig mögliche Testdatum hierfür ist (0,0). Die tatsächliche Ausgabe ist 2. Der Fehler wird gefunden.

Der Pfad (n_{start} , n_{init} , n_{while} , n_{if} , n_{then} ,n_{do} ,n_{while} , n_{erg} , n_{final}) durchläuft die while-Schleife **einmal**. Ein mögliches Testdatum ist (5,5). Der Fehler wird nicht gefunden.

Folgender Pfad gehört zu einem **zweimaligem** Durchlauf:
(n_{start} , n_{init} , n_{while} , n_{if} , n_{then} ,n_{do} ,n_{while} ,n_{if} , n_{then} ,n_{do} ,n_{while} ,n_{erg} , n_{final})
Ein mögliches Testdatum ist:
(43,4)
Der Fehler wird nicht gefunden, denn 4 ist auch das tatsächliche Ergebnis.

c) Der Fehler wird beim boundary interior- Test auf jeden Fall gefunden, da es nur ein Testdatum gibt, mit welchem die Schleife umgangen wird. Dieses Testdatum führt wie oben gezeigt zum Auffinden des Fehlers.

Damit das Programm fehlerfrei funktioniert, muß die Zeile 9 lauten `maxi := 0;`

Aufgabe 6

a) Um zu zeigen, dass

```
INV ≡ (fak=(i-1)!) ∧ (ZAHL ≥ (i-1))
```

eine Invariante der while-Schleife ist, muss die Gültigkeit folgender Programmformel (1) bewiesen werden:

```
{(fak=(i-1)!) ∧ (ZAHL ≥ (i-1))}
  while ZAHL ≥ i do
  begin
    fak := fak * i;
    i := i + 1
  end
{(fak=(i-1)!) ∧ (ZAHL ≥ (i-1))∧ (ZAHL < (i-1)) }
```

Dazu muss die Prämisse der while-Regel gezeigt werden, also die Gültigkeit von

```
{ (fak=(i-1)!) ∧ (ZAHL ≥ (i-1))∧(ZAHL ≥ i) }
begin
  fak := fak * i;
  i := i + 1
end
{(fak=(i-1)!) ∧ (ZAHL ≥ (i-1))}
```

Nach dem *Zuweisungsaxiom* sind

```
{(fak=i!) ∧ (ZAHL ≥ i)}
i := i + 1
{(fak=(i-1)!) ∧ (ZAHL ≥ (i-1))}
```

und

```
{(fak*i=i!) ∧ (ZAHL ≥ i)}
fak := fak * i
{(fak=i!) ∧ (ZAHL ≥ i)}
```

gültige Programmformeln.

Nach der *Sequenzregel* ist daher (2)

```
{(fak*i=i!) ∧ (ZAHL ≥ i)}
fak := fak * i;
i := i + 1
{(fak=(i-1)!) ∧ (ZAHL ≥ (i-1))}
```

eine gültige Programmformel.

Die *Zusammensetzungsregel* garantiert dann die Gültigkeit von (3)

```
{(fak*i=i!) ∧ (ZAHL ≥ i)}
begin
  fak := fak * i;
  i := i + 1
end
{(fak=(i-1)!) ∧ (ZAHL ≥ (i-1))}
```

Da `fak*i=i! <=> fak=(i-1)!` gilt folgende Implikation:

```
(fak=(i-1)!) ∧ (ZAHL ≥ (i-1))∧(ZAHL ≥ i) =>
(fak*i=i!) ∧ (ZAHL ≥ i)
```

Die Anwendung der *Konsequenzregel 2* auf (3) ergibt die Gültigkeit von (4):

```
{ (fak=(i-1)!) ∧ (ZAHL ≥ (i-1))∧(ZAHL ≥ i) }
begin
  fak := fak * i;
  i := i + 1
end
{(fak=(i-1)!) ∧ (ZAHL ≥ (i-1))}
```

Letztlich ergibt sich aus (4) und Anwendung der *while-Regel* die Gültigkeit von (1).

b) Zuerst zeigt man die Gültigkeit von (1)

```
{P ≡ (ZAHL≥0)}
fak := 1;
i := 1
{(fak=(i-1)!) ∧ (ZAHL ≥ (i-1)) }
```

Nach dem Zuweisungsaxiom sind die folgenden zwei Programmformeln gültig:

```
{(fak=0!)∧ (ZAHL ≥ 0)}
i := 1
{(fak=(i-1)!) ∧ (ZAHL ≥ (i-1)) }
```

und

```
{(1=0!)∧(ZAHL ≥ 0)}
fak := 1
{(fak=0!)∧ (ZAHL ≥ 0)}
```

Nach Sequenzregel ist

```
{(1=0!)∧(ZAHL ≥ 0)}
fak := 1
i := 1
{(fak=(i-1)!) ∧ (ZAHL ≥ (i-1)) }
```

Offensichtlich gilt

$P \equiv (ZAHL \geq 0) \Leftrightarrow true \wedge (ZAHL \geq 0) \Leftrightarrow (1=0!) \wedge (ZAHL \geq 0)$

Damit ist die Gültigkeit von (1) bewiesen.

Bleibt noch zu zeigen, dass aus der Invarianten und der negierten Schleifenbedingung Q folgt:

$$
\begin{aligned}
& (fak=(i-1)!) \wedge (ZAHL \geq (i-1)) \wedge (ZAHL < i) \\
\Leftrightarrow\ & (fak=(i-1)!) \wedge ZAHL=i-1 \\
\Rightarrow\ & fak=ZAHL!
\end{aligned}
$$

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

Aufgabe 7

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

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

und wählen

$$\tau(fak,\ i,\ ZAHL) := i.$$

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

Vor einem Schleifendurchlauf ist

$$\tau(fak,\ i,\ ZAHL) = i.$$

Nach einem Durchlauf gilt

$$\tau(fak,\ i,\ ZAHL) = i+1 > i.$$

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

$$\tau(fak,\ i,\ ZAHL) = i \Leftarrow ZAHL = t^*$$

gilt. Dies folgt aber sofort aus der Schleifenbedingung.

Aufgabe 8

a) Nein.
Ein partiell korrektes Programm ist nur dann total korrekt, wenn es terminiert. Schleifen sind nur ein Grund für eine Nichtterminierung. Bei Verwendung von Prozeduren bzw. Funktionen kann es beispielsweise zu zyklischen Aufrufen oder zu Endlosrekursionen kommen.

b) Nein.
Die totale Korrektheit eines Programms besagt nur, dass das Programm für die Vorbedingung erfüllende Eingabewerte terminiert und ein korrektes Ergebnis liefert. Über das Programmverhalten bei anderen Eingabewerten kann keine Aussage gemacht werden.

c) Um ein Programm verifizieren zu können, benötigt man eine mathematisch präzise Programmspezifikation und eine mathematisch präzise Beschreibung der Wirkung der im Programm vorkommenden Anweisungen und damit letztlich eine formale Semantik der verwendeten Programmiersprache.

d) Ist beim Erreichen der Schleife das Prädikat $\{INV \wedge B\}$ wahr, so liegt eine Endlosschleife vor, denn nach Abarbeitung des Schleifenrumpfs ist B immer noch wahr. Ist das Prädikat nicht wahr, so wird die Schleife umgangen. Die Schleife wird daher keinmal oder unendlich oft durchlaufen.