**Kurs 1612 "Konzepte imperativer Programmierung"**

Musterlösung zur Klausur am 06.04.2002

---

## Lösung 1 (5+3 Punkte)

a) Da Effizienzbetrachtungen bei der Lösung der Aufgabe keine Rolle spielen, wählen wir einen einfachen aber ineffizienten Algorithmus mit zwei ineinander verschachtelten **for**-Schleifen. Da die **for**-Schleifen nicht abbrechen, wenn ein Element gefunden wurde, das nicht den Wert 0 hat, kann man sich einer Zwischenvariablen (hier: `UngleichNull`) bedienen, um das Ergebnis bis zum Ende der **for**-Schleifen zu speichern.

```
function UntereDreiecksmatrix (
            var inMatrix : tMatrix) : boolean;
{ liefert den Wert true, falls inMatrix eine Untere
  Dreiecksmatrix ist, sonst false }

  var
  Zeile,
  Spalte : tZeilenSpalten;
  UngleichNull : boolean;

begin
  UngleichNull := false;
  for Zeile := 1 to ZEILENSPALTENMAX - 1 do
    for Spalte := Zeile + 1 to ZEILENSPALTENMAX do
      UngleichNull := UngleichNull or
                      (inMatrix [Zeile, Spalte] <> 0);
  UntereDreiecksmatrix := not UngleichNull
end; { UntereDreiecksmatrix }
```

b) Bei der Berechnung der Determinante wird zunächst geprüft, ob es sich bei der übergebenen Matrix um eine untere Dreiecksmatrix handelt. Ist dies der Fall, wird die Determinante durch Multiplikation der Werte aller Elemente auf der Hauptdiagonalen ermittelt.

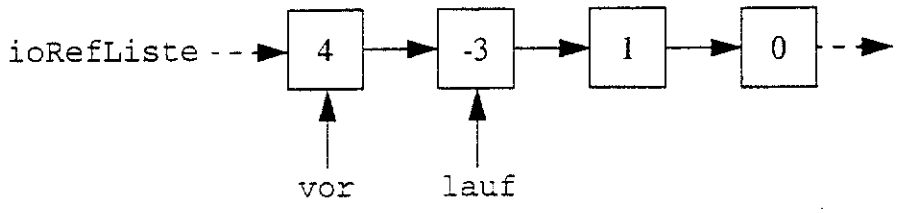
```
function Determinante (var inMatrix : tMatrix) : integer;
{ berechnet die Determinante der quadratischen Matrix
  inMatrix durch Multiplikation der Werte der
  Hauptdiagonalen, falls es sich um eine Untere
  Dreiecksmatrix handelt, sonst durch Aufruf der Funktion
  DeterminanteAllgemein }

  var
  diagonal : tZeilenSpalten;
  det : integer;

begin
  if not UntereDreiecksmatrix (inMatrix) then
    det := DeterminanteAllgemein (inMatrix)
```

```
  else
  begin
    det := 1;
    for diagonal := 1 to ZEILENSPALTENMAX do
      det := det * inMatrix [diagonal, diagonal]
  end;
  Determinante := det
end; { Determinante }
```

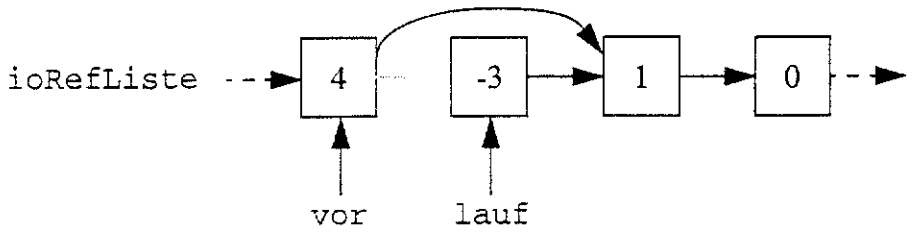
---

## Lösung 2 (9 Punkte)

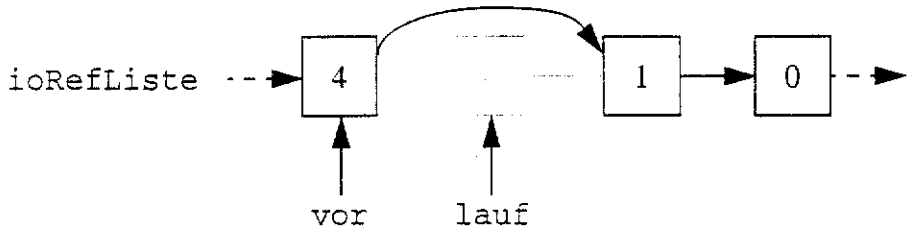
a) Die Liste wird mit zwei Zeigern `vor` und `lauf` bis zum Ende durchlaufen. Dabei zeigt `lauf` auf das jeweils aktuelle Element und `vor` auf den entsprechenden Vorgänger. Gelöscht wird nur dann, wenn die `info`-Komponente von `lauf` echt kleiner Null ist.
Die folgenden Grafiken veranschaulichen das Löschen eines Listenelements *innerhalb* einer Liste.
Ausgangssituation: `lauf` zeigt auf ein zu löschendes Listenelement.




Im ersten Schritt wird der Nachfolger von `vor` auf das übernächste Element gesetzt.
**`vor^.next := lauf^.next`**:



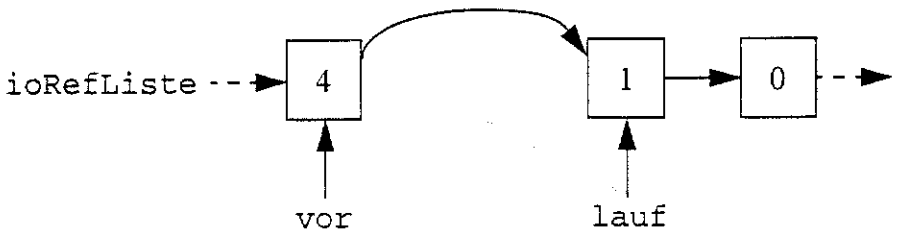

Nun kann das Element mit der ungeraden Ziffer gelöscht werden.
**`dispose(lauf)`**:




Als letztes wird der Zeiger `lauf` auf den Nachfolger von `vor` gesetzt.

---

**lauf := vor^.next:**




Muss das erste Listenelement gelöscht werden, wird der Anfangszeiger `ioRefListe` aktualisiert. `vor` zeigt zu diesem Zeitpunkt noch auf `nil`.

```
procedure LoescheNegative(var ioRefListe: tRefListe);
{ loescht alle negativen Zahlen aus ioRefListe }
  var
  vor,
  lauf: tRefListe;

begin
  if ioRefListe <> nil then
  begin
    vor := nil;
    lauf := ioRefListe;
    while lauf <> nil do
      if lauf^.info < 0 then  { Ist die Zahl negativ? }
        if lauf = ioRefListe then
        begin
          { am Anfang loeschen }
          ioRefListe := ioRefListe^.next;
          dispose(lauf);
          lauf := ioRefListe
        end
        else
        begin
          { innerhalb der Liste loeschen }
          vor^.next := lauf^.next;
          dispose(lauf);
          lauf := vor^.next
        end { if }
      else
      begin
       vor := lauf;
       lauf := lauf^.next
      end
  end { if }
end; { LoescheNegative }
```

## Lösung 3 (6 Punkte)

Mit den folgenden Anweisungen funktioniert das Programm.

| Platzhalter | Anweisung |
|---|---|
| (1) | `ioRefAnfang := elem` |
| (2) | `elem^.next := nil` |
| (3) | `elem^.next := ioRefAnfang` |
| (4) | `ioRefAnfang := elem` |
| (5) | `ende := true` |
| (6) | `lauf := lauf^.next` |

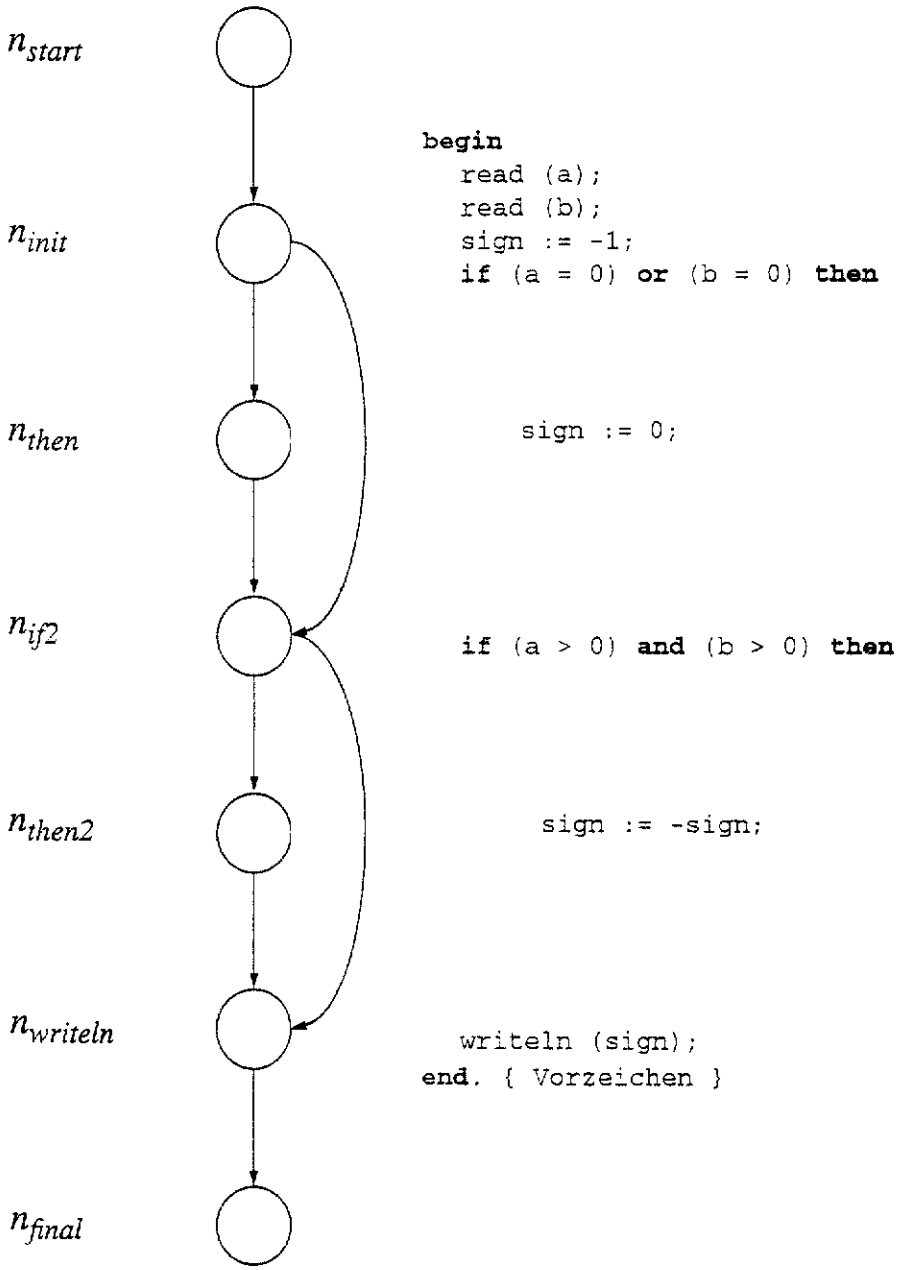
---

## Lösung 4 (10 Punkte)

```
procedure SetzeWerte(inRefWurzel: tRefBinBaum);

  var
  wertLinks,
  wertRechts: integer;

begin
  if (inRefWurzel^.links <> nil) or
     (inRefWurzel^.rechts <> nil) then     { kein Blatt }
  begin

    if inRefWurzel^.links <> nil then
    begin
      SetzeWerte(inRefWurzel^.links);
      wertLinks := inRefWurzel^.links^.info
    end
    else
      wertLinks := 0;

    if inRefWurzel^.rechts <> nil then
    begin
      SetzeWerte(inRefWurzel^.rechts);
      wertRechts := inRefWurzel^.rechts^.info
    end
    else
      wertRechts := 0;

    inRefWurzel^.info := wertLinks + wertRechts
  end
end;
```

## Lösung 5 (4 + 5 Punkte)

a) Wir geben zunächst den kompakten Kontrollflußgraphen zum Programm `Vorzeichen` an:




Den Pfad, der die Ausführung sämtlicher Anweisungen (also beider **then**-Zweige) beinhaltet, können wir sofort angeben:

$$(n_{start}, n_{init}, n_{then}, n_{if2}, n_{then2}, n_{writeln}, n_{final})$$

---

Der mit diesem Pfad assoziierte Testfall ist leer, da es keine Testdaten gibt, deren Eingabedaten zu diesem Pfaddurchlauf führen. Also benötigt man zwei Pfade für eine vollständige Anweisungsüberdeckung, z.B.:

$(n_{start}, n_{init}, n_{then}, n_{if2}, n_{writeln}, n_{final})$

und

$(n_{start}, n_{init}, n_{if2}, n_{then2}, n_{writeln}, n_{final})$

Die dazugehörigen assoziativen Testfälle sind

$$T_1 = \{((a, b), 0) \in \mathbb{Z}^2 \times \mathbb{Z} \mid a = 0 \vee b = 0\} \text{ und}$$
$$T_2 = \{((a, b), 1) \in \mathbb{Z}^2 \times \mathbb{Z} \mid a > 0 \wedge b > 0\}.$$

Als Testdaten wählen wir z.B. ((0,0),0) und ((4,5),1). Das vom Programm gelieferte Ergebnis stimmt für beide Testdaten mit dem erwarteten überein.

b) Für eine Zweigüberdeckung reicht es aus, zusätzlich noch folgenden Pfad zu betrachten:

$(n_{start}, n_{init}, n_{if2}, n_{writeln}, n_{final})$.

Dazu dürfen weder a noch b gleich Null sein, und es dürfen nicht beide größer als Null sein. Der assoziierte Testfall ist dann:

$$T_3 = T_{3,1} \cup T_{3,2} \cup T_{3,3} \text{ mit } T_{3,1} = \{((a, b), -1) \in \mathbb{Z}^2 \times \mathbb{Z} \mid a < 0 \wedge b > 0\},$$
$$T_{3,2} = \{((a, b), -1) \in \mathbb{Z}^2 \times \mathbb{Z} \mid a > 0 \wedge b < 0\},$$
$$T_{3,3} = \{((a, b), 1) \in \mathbb{Z}^2 \times \mathbb{Z} \mid a < 0 \wedge b < 0\}.$$

Da für jeden Wert aus $T_3$ die beiden **then**-Zweige übersprungen werden, wird stets der zuvor an `sign` zugewiesene Wert `-1` ausgegeben. Wird ein Testdatum aus $T_{3,3}$ gewählt, so fällt ein Fehler im Programm auf, andernfalls aber nicht. Es ist also möglich, den Fehler mit Hilfe der Zweigüberdeckung zu finden, wenn die Testdaten (zufällig) geeignet gewählt werden. Wir wählen beispielsweise zusätzlich zu (0, 0) aus a) das Paar (-5, -1).

---

Der mit diesem Pfad assoziierte Testfall ist

$$T_1 = \{((a, b), 0) \in \mathbb{Z}^2 \times \mathbb{Z} \mid a > 0 \wedge b = 0 \vee a = 0 \wedge b > 0\}.$$

Bei einem Durchlauf des Programms unter der Eingabe (dem Eingabedatum) `(0, 0)` wird zunächst `sign` auf `-1` gesetzt, dann wird der **`then`**-Zweig der ersten **`if`**-Anweisung ausgeführt, also `sign` auf `0` gesetzt, und schließlich im **`then`**-Zweig der zweiten **`if`**-Anweisung `sign` auf `-sign=0` gesetzt. Wir erhalten folglich das gewünschte Ausgabedatum.

b) Für eine Zweigüberdeckung reicht es aus, zusätzlich noch folgenden Pfad zu betrachten:

$$(n_{start}, n_{init}, n_{if2}, n_{writeln}, n_{final})$$

Dazu dürfen weder a noch b gleich Null sein, und es dürfen nicht beide größer als Null sein. Der assoziierte Testfall ist dann:

$$T_2 = T_{2,1} \cup T_{2,2} \cup T_{2,3} \text{ mit } T_{2,1} = \{((a, b), -1) \in \mathbb{Z}^2 \times \mathbb{Z} \mid a < 0 \wedge b > 0\},$$
$$T_{2,2} = \{((a, b), -1) \in \mathbb{Z}^2 \times \mathbb{Z} \mid a > 0 \wedge b < 0\},$$
$$T_{2,3} = \{((a, b), 1) \in \mathbb{Z}^2 \times \mathbb{Z} \mid a < 0 \wedge b < 0\}.$$

Da für jeden Wert aus $T_2$ die beiden **`then`**-Zweige übersprungen werden, wird stets der zuvor an `sign` zugewiesene Wert `-1` ausgegeben. Wird ein Testdatum aus $T_{2,3}$ gewählt, so fällt ein Fehler im Programm auf, andernfalls aber nicht. Es ist also möglich, den Fehler mit Hilfe der Zweigüberdeckung zu finden, wenn die Eingabedaten (zufällig) geeignet gewählt werden. Wir wählen beispielsweise zusätzlich zu (0, 0) aus a) das Paar (-5, -1).

---

## Lösung 6 (6 + 3 Punkte)

a) Um zu zeigen, dass

```
INV ≡ (prod = FAKTOR1*(FAKTOR2-b)) ∧ (b ≥ 0)
```

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

```
{ INV ∧ b ≥ 1 }
begin
  prod := prod + FAKTOR1;    (1)
  b := b - 1
end;
{ INV }
```

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

```
{ INV ∧ b ≥ 1 }
prod := prod + FAKTOR1;      (2)
{ R }
```

und

```
{ R }
b := b - 1                   (3)
{ INV }
```

gelten. Wir beginnen mit (3) und erhalten

```
R ≡ (prod = FAKTOR1*(FAKTOR2 - b + 1)) ∧ (b - 1 ≥ 0)
```

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

```
    (prod + FAKTOR1 = FAKTOR1*(FAKTOR2-b+1)) ∧ (b - 1 ≥ 0)
⇔   (prod + FAKTOR1 = FAKTOR1*(FAKTOR2-b) + FAKTOR1) ∧ b ≥ 1
⇔   (prod = FAKTOR1 * (FAKTOR2 - b)) ∧ b ≥ 1
```

Wegen

```
    INV ∧ b ≥ 1
⇔   (prod = FAKTOR1*(FAKTOR2-b)) ∧ b ≥ 0 ∧ b ≥ 1
⇒   (prod = FAKTOR1*(FAKTOR2-b)) ∧ (b ≥ 1)
```

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 ≡ (FAKTOR1≥0) ∧ (FAKTOR2≥0) }
prod := 0;
b := FAKTOR2;
{ INV }
```

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

```
{ P ≡ (FAKTOR1≥0) ∧ (FAKTOR2≥0) }
prod := 0;
{ S }
```

und

```
{ S }
b := FAKTOR2;
{ INV }
```

gültige Programmformeln sind. Durch Anwenden des Zuweisungsaxioms erhalten wir

```
S ≡ prod = 0 ∧ FAKTOR2≥0.
```

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

```
     0=0 ∧ FAKTOR2≥0
⇔    FAKTOR2 ≥ 0
```

Da `P ⇒ FAKTOR2≥0`, folgt über die zweite Konsequenzregel die Gültigkeit von (1).

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

```
     INV ∧ 0 ≥ b
⇔    (prod = FAKTOR1*(FAKTOR2-b)) ∧ b = 0
⇒    prod = FAKTOR1*(FAKTOR2-0)
⇒    prod = FAKTOR1*FAKTOR2 ≡ Q
```

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

---

## Lösung 7 ( 5 Punkte)

Im Schleifenrumpf und der Schleifenabbruchbedingung kommen die Variablen `prod` und `b` sowie die Konstante `FAKTOR1` vor. Wir suchen also eine dreistellige Terminierungsfunktion

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

und wählen

```
τ(prod, b, FAKTOR1) := b.
```

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

Vor einem Schleifendurchlauf ist

```
τ(prod, b, FAKTOR1) = b.
```

Nach einem Durchlauf gilt

```
τ(prod, b, FAKTOR1) = b-1 < b.
```

Die Funktionswerte in Schleifendurchlaufreihenfolge bilden somit eine streng monoton fallende Folge. Somit ist w2) erfüllt.
Um w3) zu zeigen, wählen wir `t*=1`. Wir müssen nachweisen, dass vor Ausführung des Schleifenrumpfes immer

```
τ(prod, b, FAKTOR1) = b ≥ t*
```

gilt. Diese Eigenschaft folgt unmittelbar aus der Schleifenbedingung $b \geq 1$. Damit ist w3) gezeigt.

---

## Lösung 8 (2+2+3+3 Punkte)

a) Um ein Programm verifizieren zu können, benötigt man eine mathematisch präzise Programmspezifikation und eine mathematisch präzise Beschreibung der Semantik der verwendeten Programmiersprachenanweisungen, also eine formale Semantik der verwendeten Programmiersprache.

b) Ist die Programmformel $\{P \wedge B\}$ S $\{P\}$ gültig und ist P ein aussagekräftiges Prädikat, dann beschreibt die Programmformel die Semantik des Schleifenrumpfes.

c) Die Wirkung einer while-Schleife ist abhängig von der Wirkung des Schleifenrumpfes. Da in der Programmformel $\{P\}$ **while** B **do** S $\{P \wedge \neg B\}$ nichts über die Wirkung von S ausgesagt ist, reicht die Programmformel für eine präzise Definition der Wirkung einer while-Schleife nicht aus.

d) Sei $\{P\}$ S $\{Q\}$ eine Programmspezifikation. Dann ist jedes Programm S' bezüglich dieser Spezifikation partiell korrekt, wenn eine gültige Programmformel
$\{P'\}$ S' $\{Q'\}$ mit P => P' und Q' => Q existiert.

Q' => Q bedeutet, dass S' Ergebnisse liefert, für die Q' und damit auch Q wahr wird. S' produziert damit „zulässige" Ergebnisse..

P => P' bedeutet anschaulich, dass S' sogar für mehr Eingaben als in der Originalspezifikation gefordert ein „zulässiges" Ergebnis (d.h. ein Ergebnis, für das Q wahr wird) liefert.