Kurs 1612 "Konzepte imperativer Programmierung"

Klausur am 06.04.2002

Wintersemester 2001/2002
Hinweise zur Bearbeitung der Klausur
zum Kurs 1612 "Konzepte imperativer Programmierung"

Wir begrüßen Sie zur Klausur "Konzepte imperativer Programmierung". Lesen Sie sich diese Hinweise vollständig und aufmerksam durch, bevor Sie mit der Bearbeitung der Aufgaben beginnen:

1. Prüfen Sie die Vollständigkeit Ihrer Unterlagen. Die Klausur umfasst:
   - 2 Deckblätter,
   - 1 Formblatt für eine Bescheinigung für das Finanzamt,
   - diese Hinweise zur Bearbeitung,
   - 8 Aufgaben (Seite 2 - Seite 30),
   - die Muß-Regeln des Programmierstils,
   - die Hoare-Regeln
   - die Definition der Terminierungsfunktion.
2. Füllen Sie, **bevor** Sie mit der Bearbeitung der Aufgaben beginnen, folgende Seiten des Klausurexemplares aus:
   a) BEIDE Deckblätter mit Namen, Anschrift sowie Matrikelnummer. **Markieren Sie vor der Abgabe auf beiden Deckblättern die von Ihnen bearbeiteten Aufgaben.**
   b) Falls Sie eine Teilnahmebescheinigung für das Finanzamt wünschen, füllen Sie bitte das entsprechende Formblatt aus.

   Nur wenn Sie beide Deckblätter vollständig ausgefüllt haben, können wir Ihre Klausur korrigieren!
3. Schreiben Sie Ihre Lösungen auf den freien Teil der Seite unterhalb der Aufgabe bzw. auf die leeren Folgeseiten. Sollte dies nicht möglich sein, so vermerken Sie, auf welcher Seite die Lösung zu finden ist. Streichen Sie ungültige Lösungen deutlich durch.
4. Schreiben Sie auf jedem von Ihnen beschriebenen Blatt oben links Ihren Namen und oben rechts Ihre Matrikelnummer. Wenn Sie weitere eigene Blätter benutzt haben, heften Sie auch diese, mit Name und Matrikelnummer versehen, an Ihr Klausurexemplar. Nur dann werden auch Lösungen außerhalb Ihres Klausurexemplares gewertet!
5. Neben unbeschriebenem Konzeptpapier und Schreibzeug (Füller oder Kugelschreiber) sind **keine** weiteren Hilfsmittel zugelassen. Die Muß-Regeln des Programmierstils, die Tabelle mit den im Kurs verwendeten Hoare-Regeln und die Definition der Terminierungsfunktion finden Sie im Anschluß an die Aufgabenstellung.
6. Es sind maximal 66 Punkte erreichbar. Sie haben die Klausur sicher dann bestanden, wenn Sie mindestens 33 Punkte erreicht haben.

Wir wünschen Ihnen bei der Bearbeitung der Klausur viel Erfolg!

Kurs 1612 "Konzepte imperativer Programmierung"

Klausur am 06.04.2002

Aufgabe 1 (5 + 3 Punkte)

Jeder reellen quadratischen Matrix kann man eindeutig eine reelle Zahl, die sogenannte Determinante, zuordnen, die bei Rang- und Inversenbestimmungen von Matrizen oder Lösungen linearer Gleichungssysteme eine Rolle spielt. Die Berechnung der Determinanten ist normalerweise ziemlich aufwändig, für einige Spezialfälle jedoch einfach und schnell durchführbar. Ein solcher Spezialfall liegt beispielsweise dann vor, wenn es sich bei der zu untersuchenden Matrix um eine Untere Dreiecksmatrix handelt. Eine solche Matrix ist folgendermaßen definiert:

Eine Untere Dreiecksmatrix *Mat* ist eine n×n-Matrix mit der Eigenschaft, dass alle Elemente oberhalb der Hauptdiagonalen den Wert 0 haben. Mit anderen Worten, bezeichne *Mat*`[i, j]` das Element in Zeile `i` und Spalte `j` der Matrix *Mat*, dann gilt *Mat*`[i, j] = 0` für $1 \le i < j \le n$. Die folgende 5×5-Matrix *M* ist eine Untere Dreiecksmatrix.

$$M = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 8 & 5 & 0 & 0 & 0 \\ 9 & 12 & 1 & 0 & 0 \\ 3 & 17 & 19 & 2 & 0 \\ 9 & 2 & 5 & 8 & 10 \end{bmatrix}$$

Die Determinante einer Unteren Dreiecksmatrix ist nun einfach das Produkt aller Werte auf der Hauptdiagonalen:

$$\det(\mathrm{Mat}) = \prod_{i=1}^{n} \mathrm{Mat}[i, i]$$

Für die oben dargestellte Matrix *M* ist det(M) = 3·5·1·2·10 = 300.

Ihre Aufgabe ist es, ausgehend von einer bereits implementierten (aber sehr aufwändigen) Funktion zur Berechnung der Determinante einer <u>integer</u> n×n-Matrix (n>1), die Berechnung für den Fall einer Unteren Dreiecksmatrix zu beschleunigen. Bearbeiten Sie dazu die Teilaufgaben a) und b).

a) Implementieren Sie die Funktion `UntereDreiecksmatrix`, mit der festgestellt werden kann, ob eine als Parameter übergebene Matrix eine Untere Dreiecksmatrix ist. Gehen Sie dabei von den unten angegebenen Typdefinitionen und dem angegebenen Funktionskopf für `UntereDreiecksmatrix` aus. Beachten Sie, dass Effizienz hier keine Rolle spielt. Wählen Sie also einen einfach zu implementierenden Algorithmus.

b) Implementieren Sie die Funktion `Determinante`, die folgendermaßen arbeitet: Handelt es sich bei der als Parameter übergebenen Matrix um eine Untere Dreiecksmatrix, wird die Determinante durch einfache Multiplikation der Elemente der Hauptdiagonalen berechnet, sonst durch Aufruf der Funktion `DeterminanteAllgemein`, die Sie als gegeben voraussetzen können, also nicht implementieren sollen. Haben Sie a) nicht bearbeitet, können Sie die Implementierung von `UntereDreiecksmatrix` voraussetzen.

Kurs 1612 "Konzepte imperativer Programmierung"

Klausur am 06.04.2002

Name: _____ Matrikelnummer: _____

```
const
ZEILENSPALTENMAX = ..;

type
tZeilenSpalten = 1..ZEILENSPALTENMAX;
tMatrix = array [tZeilenSpalten, tZeilenSpalten] of integer;

function DeterminanteAllgemein (
          var inMatrix : tMatrix) : integer;
{ ermittelt die Determinante der quadratischen Matrix
  inMatrix }

{ Diese Funktion sollen Sie nicht implementieren, sondern
  als gegeben voraussetzen! }

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

{ Teilaufgabe a) }

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 }

{ Teilaufgabe b) }
```

Kurs 1612 "Konzepte imperativer Programmierung"

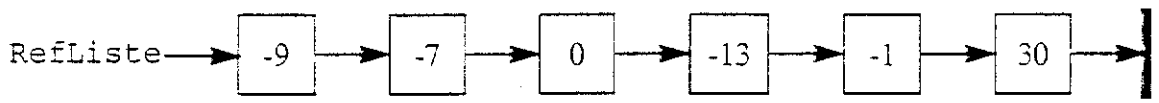
Klausur am 06.04.2002

Aufgabe 2 (9 Punkte)

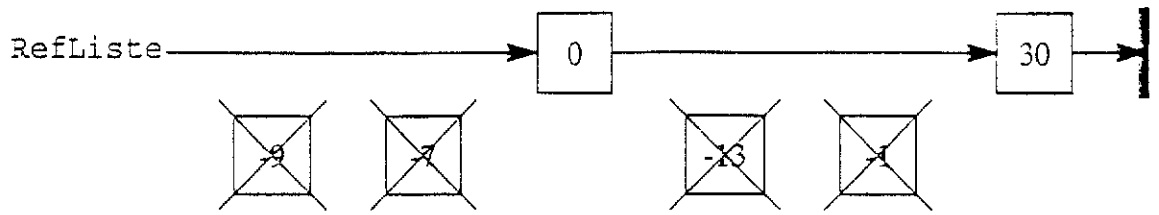
Gegeben ist folgende Definition einer linearen Liste:

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

Implementieren Sie eine Prozedur `LoescheNegative`, die alle Zahlen echt kleiner Null aus einer Liste mit einem gegebenen Anfangszeiger vom Typ `tRefListe` löscht. Sie können davon ausgehen, dass die übergebene Liste nicht leer ist. Ein Beispiel:
Vor dem Aufruf:



Danach:



Kurs 1612 "Konzepte imperativer Programmierung"

Klausur am 06.04.2002

Aufgabe 3 (6 Punkte)

Gegeben ist folgende Definition einer linearen Liste:

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

Es soll eine Prozedur `SortiertEinfuegen` implementiert werden, die einen Wert in eine aufsteigend sortierte Liste so einfügt, dass die Sortierung erhalten bleibt. Betrachten Sie dazu folgende lückenhafte Prozedur:

```
procedure SortiertEinfuegen(
                  inWert: integer;
              var ioRefAnfang: tRefListe);

  var
  lauf,
  elem: tRefListe;
  ende: boolean;

begin
  new(elem);
  elem^.info := inWert;
  if ioRefAnfang = NIL then              (* Sonderfall: Leere Liste *)
  begin
    (1);
    (2)
  end
  else
    if inWert < ioRefAnfang^.info then(* Sonderfall: Anfuegen am An-
                                          fang *)
    begin
      (3);
      (4)
    end
    else (* Normalfall *)
```

```
    begin
      ende := false;
      lauf := ioRefAnfang;
      while not ende and (lauf^.next <> NIL) do
        if lauf^.next^.info > inWert then
          (5)
        else
          (6); (* Ende von while *)
      elem^.next := lauf^.next;
      lauf^.next := elem
    end;
end;
```

Für jeden der Platzhalter (1), (2), (3), (4), (5) und (6) ist genau eine Anweisung so anzugeben, dass die Prozedur `SortiertEinfügen` ihre Aufgabe korrekt erfüllt.

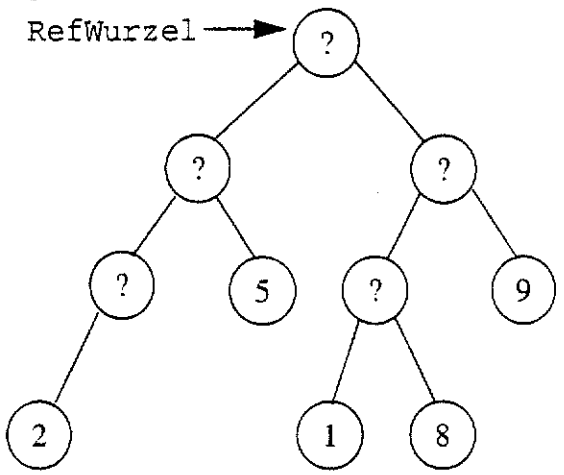
Kurs 1612 "Konzepte imperativer Programmierung"

Klausur am 06.04.2002

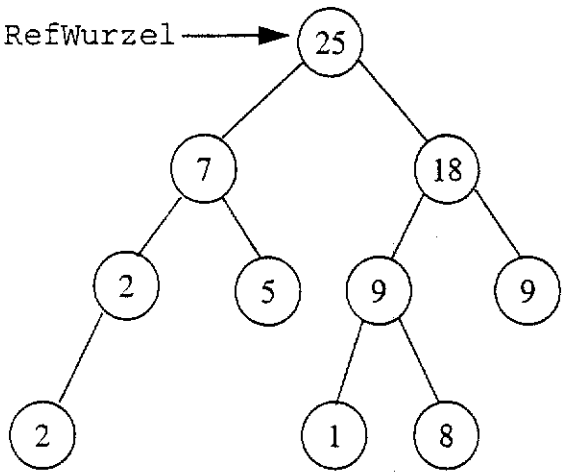
Aufgabe 4 (10 Punkte)

Gegeben sei ein binärer Baum, dessen Blätter vordefinierte Werte besitzen. Schreiben Sie eine rekursive Prozedur `SetzeWerte`, welche die inneren Knoten des Baumes wie folgt setzt: Hat ein innerer Knoten sowohl einen rechten als auch einen linken Nachfolger, so ergibt sich sein Wert aus der Summe der Werte der beiden Nachfolger. Besitzt ein innerer Knoten genau einen Nachfolger, so wird ihm der Wert des Nachfolgers zugewiesen (Daraus ergibt sich, dass der leere (Teil-)Baum den „Knotenwert" 0 besitzt).

Beispiel:



Vor dem Aufruf von `SetzeWerte(RefWurzel)`
(? = undefiniert)



Danach

Verwenden Sie folgende Typdefinition und folgenden Prozedurkopf, ergänzen Sie auch die Parameterübergabeart:

```
type
  tRefBinBaum = ^tBinBaum;
  tBinBaum = record
               info : integer;
               links,
               rechts: tRefBinBaum
             end;


procedure SetzeWerte(?? ??RefWurzel: tRefBinBaum);

var WertLinkerNachfolger, WertRechterNachfolger: integer;
```

Hinweis: Verwenden Sie die Durchlaufreihenfolge, mit der zuerst die Teilbäume und dann die Wurzel bearbeitet werden.

Kurs 1612 "Konzepte imperativer Programmierung"

Klausur am 06.04.2002

Aufgabe 5 (4 + 5 Punkte)

Betrachten Sie die folgende zweistellige Funktion, die für ganze Zahlen a,b aus $\mathbb{Z}$ definiert ist:

$$\text{sign}(a,b) = \begin{cases} 1 \text{ , falls } a \cdot b > 0 \\ 0 \text{ , falls } a \cdot b = 0 \\ -1 \text{ , falls } a \cdot b < 0 \end{cases}$$

Als Implementierung von sign(a,b) bieten wir das folgende Programm an:

```
program Vorzeichen (input, output);
{ das Programm soll sign(a*b) berechnen }

  var
  a,
  b,
  sign: integer;

begin
  read (a);
  read (b);
  sign := -1;
  if (a = 0) or (b = 0) then
    sign := 0;
  if (a > 0) and (b > 0) then
    sign := - sign;
  writeln(sign)
end. { Vorzeichen }
```

a) Bestimmen Sie zunächst den zu dem Programm `Vorzeichen` gehörenden kompakten Kontrollflußgraphen. Geben Sie anschließend den Pfad für eine vollständige Anweisungsüberdeckung an. Bestimmen Sie den mit diesem Pfad assoziierten Testfall und wählen Sie ein Testdatum zu diesem Testfall aus.

b) Ergänzen Sie die für eine vollständige Zweigüberdeckung notwendigen Testfälle. Geben Sie dazu wiederum zunächst geeignete Pfade an und bestimmen Sie deren assoziierten Testfälle. Wählen Sie auch hier zu jedem Testfall ein Testdatum aus. Ist es möglich, den Fehler im Programm mit diesem Testverfahren zu finden? Falls ja, begründen Sie ihre Antwort.

Kurs 1612 "Konzepte imperativer Programmierung"

Klausur am 06.04.2002

Aufgabe 6 (6 + 3 Punkte)

Das folgende Programm `Aufgabe6` berechnet das Produkt von `FAKTOR1` und `FAKTOR2` ohne Verwendung der vordefinierten Operation *.

Zeigen Sie, dass das Programm `Aufgabe6` bezüglich der Spezifikation
`{P ≡ (FAKTOR1≥0) ∧ (FAKTOR2≥0)} Aufgabe6 {Q ≡ prod=FAKTOR1*FAKTOR2}` partiell korrekt ist, indem Sie die Aufgabenteile a) und b) bearbeiten.

```
program Aufgabe6;

  type
  tNatZahl = 0..MAXINT;

  const
  FAKTOR1 = ?;
  FAKTOR2 = ?;

  var
  prod,
  b : tNatZahl;

  begin
    prod := 0;
    b := FAKTOR2;
    while b ≥ 1 do
    begin
      prod := prod + FAKTOR1;
      b := b - 1
    end
  end.
```

a) Zeigen Sie, dass

$$INV \equiv (prod=FAKTOR1*(FAKTOR2-b)) \wedge (b \geq 0)$$

eine Invariante der while-Schleife des Programms `Aufgabe6` ist. Geben Sie die verwendeten Hoare-Regeln explizit an.

b) Benutzen Sie die Invariante `INV` zum Nachweis der partiellen Korrektheit des Programms `Aufgabe6`. Geben Sie auch hier die verwendeten Hoare-Regeln explizit an.

Aufgabe 7 (5 Punkte)

Zeigen Sie, dass das Programm `Aufgabe6` terminiert, indem Sie eine Funktion τ angeben und beweisen, dass τ eine Terminierungsfunktion der **while**-Schleife ist.

Kurs 1612 "Konzepte imperativer Programmierung"

Klausur am 06.04.2002

Aufgabe 8 (2+2+3+3 Punkte)

a) Um ein Programm verifizieren zu können, benötigt man zwei Voraussetzungen. Welche sind das?

b) Betrachten Sie die Semantikdefintion der while-Schleife:

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{P \wedge \neg B\}}$$

Nehmen wir an $\{P \wedge B\}\ S\ \{P\}$ ist eine gültige Programmformel und P ein aussagekräftiges Prädikat. Was sagt dann die Programmformel anschaulich aus?

c) Begründen Sie, warum die Programmformel $\{P\}$ `while` B `do` S $\{P \wedge \neg B\}$ allein nicht ausreicht, um die Semantik der while-Schleife zu definieren.

d) Wann ist ein Programm S' partiell korrekt bezüglich einer Programmspezifikation $\{P\}\ S\ \{Q\}$? Geben Sie nicht nur die formale Definition an, sondern erläutern Sie diese auch anschaulich.

Zusammenfassung der Muß-Regeln

1. Selbstdefinierte Konstantenbezeichner bestehen nur aus Großbuchstaben. Bezeichner von Standardkonstanten wie z.B. `maxint` sind also ausgenommen
2. Typbezeichnern wird ein `t` vorangestellt. Bezeichner von Zeigertypen beginnen mit `tRef`. Bezeichner formaler Parameter beginnen mit `in`, `io` oder `out`.
3. Jede Anweisung beginnt in einer neuen Zeile; **`begin`** und **`end`** stehen jeweils in einer eigenen Zeile
4. Anweisungsfolgen werden zwischen **`begin`** und **`end`** um eine konstante Anzahl von 2 - 4 Stellen eingerückt. **`begin`** und **`end`** stehen linksbündig unter der zugehörigen Kontrollanweisung, sie werden nicht weiter eingerückt.
5. Anweisungsteile von Kontrollanweisungen werden genauso eingerückt.
6. Im Programmkopf wird die Aufgabe beschrieben, die das Programm löst.
7. Jeder Funktions- und Prozedurkopf enthält eine knappe Aufgabenbeschreibung als Kommentar. Ggf. werden zusätzlich die Parameter kommentiert.
8. Die Parameter werden sortiert nach der Übergabeart: Eingangs-, Änderungs- und Ausgangsparameter.
9. Die Übergabeart jedes Parameters wird durch Voranstellen von `in`, `io` oder `out` vor den Parameternamen gekennzeichnet.
10. Das Layout von Funktionen und Prozeduren entspricht dem von Programmen.
11. Jede von einer Funktion oder Prozedur benutzte bzw. manipulierte Variable wird als Parameter übergeben. Es werden keine globalen Variablen manipuliert. Einzige Ausnahme sind Modul-lokale Variablen, die in den Parameterlisten der exportierten Prozeduren und Funktionen des Moduls nicht auftauchen, selbst wenn sie von diesen geändert werden.
12. Jeder nicht von der Prozedur veränderte Parameter wird als Wertparameter übergeben. Lediglich Felder können auch anstatt als Wertparameter als Referenzparameter übergeben werden, um den Speicherplatz für die Kopie und den Kopiervorgang zu sparen. Der Feldbezeichner beginnt aber stets mit dem Präfix `in`, wenn das Feld nicht verändert wird.
13. Funktionsprozeduren werden wie Funktionen im mathematischen Sinne benutzt, d.h. sie besitzen nur Wertparameter. Wie bei Prozeduren ist eine Ausnahme nur bei Feldern erlaubt, um zusätzlichen Speicherplatz und Kopieraufwand zu vermeiden.
14. Wertparameter werden nicht als lokale Variable mißbraucht.
15. Die Schlüsselworte **`unit`**, **`interface`** und **`implementation`** werden ebenso wie **`begin`** und **`end`** des Initialisierungsteils linksbündig positioniert. Nach dem Schlüsselwort **`unit`** folgt ein Kommentar, der die Aufgabe beschreibt, welche die Unit löst.
16. Für die Schnittstelle gelten dieselben Programmierstilregeln wie für ein Programm. Dies betrifft Layout und Kommentare. Nach dem Schlüsselwort **`interface`** folgt im Normalfall kein Kommentar.
17. Für den Implementationsteil gelten dieselben Programmierstilregeln wie für ein Programm. Nach dem Schlüsselwort **`implementation`** folgt nur dann ein Kommentar, wenn die Realisierung einer Erläuterung bedarf (z.B. wegen komplizierter Datenstrukturen und/oder Algorithmen).
18. In Programmen oder Moduln, die andere Moduln importieren („benutzen"), wird das Schlüsselwort **`uses`** auf dieselbe Position eingerückt wie die Schlüsselworte **`const`**, **`type`**, **`var`** usw.
19. Die Laufvariable wird innerhalb einer **`for`**-Anweisung nicht manipuliert.
20. Die Grundsätze der strukturierten Programmierung sind strikt zu befolgen.

Hoare-Regeln

1. Konsequenzregel 1

$$\frac{\{P\}\ S\ \{R\},\ R \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Konsequenzregel 2

$$\frac{P \Rightarrow R,\ \{R\}\ S\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

2. Nullaxiom

$$\{P\}\ \{P\}$$

3. Zuweisungaxiom

$$\{P\langle x \leftarrow expr\rangle\}\ \texttt{x:=expr}\ \{P\}$$

4. Sequenzregel

$$\frac{\{P\}\ S_1\ \{R\},\ \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1;\ S_2\ \{Q\}}$$

5. Zusammensetzungsregel

$$\frac{\{P\}\ S\ \{Q\}}{\{P\}\ \textbf{begin}\ S\ \textbf{end}\ \{Q\}}$$

6. Bedingungsregel 1

$$\frac{\{P \wedge B\}\ S_1\ \{Q\},\ \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

Bedingungsregel 2

$$\frac{\{P \wedge B\}\ S\ \{Q\},\ (P \wedge \neg B) \Rightarrow Q}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \{Q\}}$$

7. **while**-Regel

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{P \wedge \neg B\}}$$

8. **repeat**-Regel

$$\frac{\{P\}\ S\ \{Q\},\ (Q \wedge \neg B) \Rightarrow P}{\{P\}\ \textbf{repeat}\ S\ \textbf{until}\ B\ \{Q \wedge B\}}$$