Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Wintersemester 2007/2008
Hinweise zur Bearbeitung der Klausur
zum Kurs 1613 “Einführung in die imperative Programmierung”

Wir begrüßen Sie zur Klausur "Einführung in die imperative 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,
   - 5 Aufgaben (Seite 2 - Seite 18),
   - die Muss-Regeln des Programmierstils.
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 und belassen Sie es in der Klausur. Sie erhalten es dann zusammen mit der Korrektur abgestempelt zurück.

   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 jedes von Ihnen beschriebene Blatt oben links Ihren Namen und oben rechts Ihre Matrikelnummer. Wenn Sie weitere eigene Blätter benutzt haben, heften Sie auch diese, mit Namen 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 finden Sie im Anschluss an die Aufgabenstellung.
6. Es sind maximal 33 Punkte erreichbar. Sie haben die Klausur sicher dann bestanden, wenn Sie mindestens 16,5 Punkte erreicht haben.

Wir wünschen Ihnen bei der Bearbeitung der Klausur viel Erfolg!

Kurs 1613 "Einführung in die imperative Programmierung"

Nachklausur am 05.04.2008

Aufgabe 1 (3+3 Punkte)

Es soll eine Prozedur zur Berechnung des Produktes zweier Matrizen implementiert werden.

Als $m \times n$-Matrix (für natürliche Zahlen m und n) wird im Folgenden eine Matrix mit m Zeilen und n Spalten (von ganzen Zahlen) bezeichnet. Für eine $m \times n$-Matrix A bezeichne $a_{i,j}$ (für $1 \le i \le m$ und $1 \le j \le n$) den Wert der Matrix A in Zeile i und Spalte j.

Das Produkt $A \cdot B$ einer $l \times m$-Matrix A und einer $m \times n$-Matrix B ist dann eine $l \times n$-Matrix C, deren Komponenten $c_{i,j}$ ($1 \le i \le l$, $1 \le j \le n$) sich wie folgt errechnen:

$$c_{i,j} = \sum_{k=1}^{m} A_{i,k} \cdot B_{k,j}$$

Das heißt, den Wert der Ergebnismatrix C in Zeile i und Spalte j erhält man, indem man für k von 1 bis m jeweils das k-te Element der i-ten Zeile von A mit dem k-ten Element der j-ten Spalte von B multipliziert und dann die Summe aus diesen m Produkten bildet.

Ein Beispiel für die Multiplikation einer 2×3-Matrix mit einer 3×2-Matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 0 & -1 \\ -2 & -3 \\ -4 & -5 \end{bmatrix} = \begin{bmatrix} 1 \cdot 0 + 2 \cdot (-2) + 3 \cdot (-4) & 1 \cdot (-1) + 2 \cdot (-3) + 3 \cdot (-5) \\ 4 \cdot 0 + 5 \cdot (-2) + 6 \cdot (-4) & 4 \cdot (-1) + 5 \cdot (-3) + 6 \cdot (-5) \end{bmatrix} = \begin{bmatrix} -16 & -22 \\ -34 & -49 \end{bmatrix}$$

Zur Darstellung solcher Matrizen seien die folgenden Pascal-Typen[1] gegeben:

```
const
  L = 2;
  M = 3;
  N = 2;

type
  tIndexL = 1..L;
  tIndexN = 1..N;
  tIndexM = 1..M;
  tMatrixLxM = array[tIndexL, tIndexM] of integer; {L x M - Matrix}
  tMatrixMxN = array[tIndexM, tIndexN] of integer; {M x N - Matrix}
  tMatrixLxN = array[tIndexL, tIndexN] of integer; {L x N - Matrix}
```

---

1. Die Werte der Konstanten L, M und N sind nur beispielhaft gewählt. Ihre Implementierungen zu a) und b) sollen für beliebige Werte dieser Konstanten (> 0) funktionieren.

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Name: _____ Matrikelnummer: _____

---

a) Implementieren Sie zunächst eine Hilfsfunktion zur Berechnung der Summe $c_{i,j}$ nach oben abgebildeter Summenformel zu einer $l \times m$-Matrix `inMatL` und einer $m \times n$-Matrix `inMatR` sowie gegebenen Indizes i (Parameter `inI`) und j (`inJ`). Verwenden Sie folgenden Funktionskopf:

```
function ZeileMalSpalte(inMatL: tMatrixLxM; inMatR: tMatrixMxN;
                        inI: tIndexL; inJ:tIndexN): integer;
{Berechnet für 1 <= k <= M jeweils das Produkt des k-ten Elements
 der inI-ten Zeile von inMatL und des k-ten Elements der inJ-ten
 Spalte von inMatR und summiert diese M Produkte.}
```

b) Schreiben Sie nun unter Verwendung der Funktion `ZeileMalSpalte` eine Prozedur zum Berechnen des Matrizenprodukts. Unabhängig von Ihrer Lösung zu a) können Sie annehmen, dass die Funktion `ZeileMalSpalte` wie gewünscht funktioniert.
Verwenden Sie folgenden Prozedurkopf:

```
procedure matrProdukt(inMatL: tMatrixLxM; inMatR: tMatrixMxN;
                      var outMat: tMatrixLxN);
{Berechnet das Matrizenprodukt inMatL * inMatR.}
```

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Name: _____ Matrikelnummer: _____

Kurs 1613 "Einführung in die imperative Programmierung"

Nachklausur am 05.04.2008

Aufgabe 2 (6 Punkte)

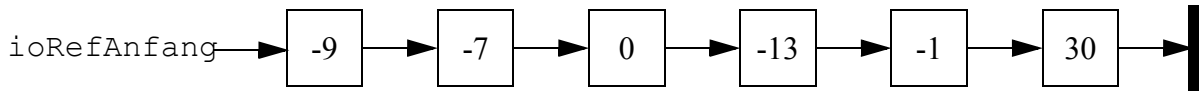
Gegeben sind folgende Typdefinitionen für Elemente einer linearen Liste:

```
type
tRefElement = ^tElement;
tElement = record
             info : integer;
             next : tRefElement
           end;
```

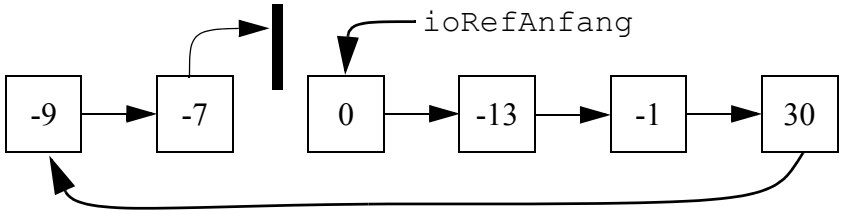
Implementieren Sie eine Prozedur `Tausche`, die in einer übergebenen Liste das Element mit der info-Komponente 0 sucht und die mit diesem Element beginnende Restliste nur durch Ändern der Verkettung aushängt und an den Beginn der Liste einfügt. Sie können davon ausgehen, dass die 0 in der Liste genau einmal vorkommt. Steht die 0 bereits am Anfang, so muss nichts getan werden.

Ein Beispiel:

Vor dem Aufruf:



Danach:



Verwenden Sie folgenden Prozedurkopf:

```
procedure Tausche(var ioRefAnfang : tRefElement);
```

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Name: _____ Matrikelnummer: _____

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Name: _____ Matrikelnummer: _____

Kurs 1613 "Einführung in die imperative Programmierung"

Nachklausur am 05.04.2008

Aufgabe 3 (7 Punkte)

Gegeben seien folgende Typvereinbarungen:

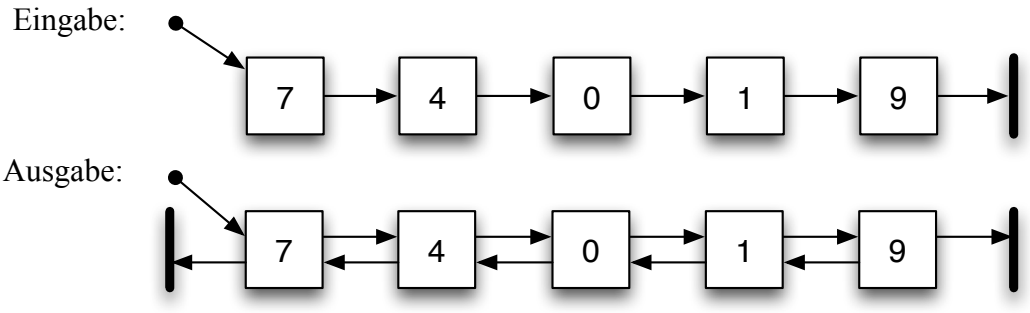
```
type
  tRefElement = ^tElement;
  tElement =  record
                info: integer;
                next: tRefElement
              end;

  tRefDVElement = ^tDVElement;
  tDVElement =  record
                  info: integer;
                  next, prev: tRefDVElement;
                end;
```

Die Typen `tRefElement` und `tElement` dienen zur Bildung einfach verketteter, die Typen `tRefDVElement` sowie `tDVElement` zur Bildung doppelt verketteter linearer Listen von Integer-Zahlen. Eine doppelt verkettete Liste zeichnet sich dadurch aus, dass jedes ihrer Elemente nicht nur einen Zeiger `next` auf das nachfolgende, sondern zusätzlich einen Zeiger `prev` auf das vorhergehende Element besitzt. Analog zum Zeiger `next` des letzten Listenelementes zeigt auch der Zeiger `prev` des ersten Listenelementes auf **`nil`**.

Schreiben Sie eine *iterative* Funktion `DVKopie`, die eine einfach verkettete Liste übergeben bekommt und eine doppelt verkettete Liste ausgibt, welche die gleichen Elemente in gleicher Reihenfolge enthält. Die eingegebene Liste soll unverändert weiter existieren, darf also nicht verändert werden!

Die folgende Abbildung[1] zeigt ein Beispiel:



Verwenden Sie den auf der folgenden Seite bereits vorgegebenen Funktionsrahmen. Bei Bedarf können Sie weitere lokale Variablen deklarieren, nötig zur Lösung der Aufgabe ist das jedoch nicht.

1. Von einem Element nach rechts weisende Pfeile stellen dabei `next`-Zeiger, nach links weisende Pfeile stellen `prev`-Zeiger dar. Die von einem Punkt ausgehenden Pfeile stellen die Anfangszeiger dar, über welche die Listen referenziert werden.

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Name: _____ Matrikelnummer: _____

```
function DVKopie(inRefQuell: tRefElement): tRefDVElement;
{Erzeugt aus einer einzugebenen einfach verketteten Quellliste eine
neue doppelt verkettete Liste, ohne die Quellliste zu verändern.}

var laufQuell: tRefElement; { Laufzeiger für eingegebene Liste }
    anfangKopie,        { Anfangszeiger für neu aufzubauende Liste }
    endeKopie,          { Endzeiger für neu aufzubauende Liste }
    neuesElement:tRefDVElement; { Hilfszeiger zum Erzeugen neuer
                                  Elemente (falls benötigt). }

begin
  { Initialisierung (neue Liste noch leer) }
  anfangKopie := nil;
  endeKopie := nil;
  laufQuell := inRefQuell;
  {Quellliste durchlaufen und eine Kopie jedes Elementes ans Ende der
  neuen Liste anfügen.}
```

... Komplettieren Sie die Funktion ab hier! ...

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Name: _____ Matrikelnummer: _____

Kurs 1613 "Einführung in die imperative Programmierung"

Nachklausur am 05.04.2008

Aufgabe 4 (3+3 Punkte)

Gegeben seien folgende Typdefinitionen für Knoten eines binären Baumes und natürliche Zahlen sowie die Prozedur `whatDoIDo`:

```
type
  tNatZahl = 1..maxint;
  tRefKnoten = ^tKnoten;
  tKnoten = record
              info : integer;
              links,
              rechts: tRefKnoten
            end;

procedure whatDoIDo(inWurzel: tRefKnoten;
                    var outErgebnis:boolean);
var ergL, ergR: boolean;
    Blatt, ZweiNachf: boolean;
begin
  if inWurzel = nil then
    outErgebnis := true
  else
  begin
    whatDoIDo(inWurzel^.links, ergL);
    whatDoIDo(inWurzel^.rechts, ergR);
    Blatt := (inWurzel^.links = nil) and (inWurzel^.rechts = nil);
    ZweiNachf := (inWurzel^.links<>nil) and (inWurzel^.rechts<>nil);
    outErgebnis :=  ergL and ergR
                    and (Blatt or ZweiNachf)
  end
end;
```

a) Beschreiben Sie in ein bis zwei Sätzen, was die Prozedur `whatDoIDo` leistet.

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Name: _____ Matrikelnummer: _____

Die *Höhe* eines binären Baumes ist definiert als die Länge des längsten Pfades von der Wurzel zu einem Blatt. Die Pfadlänge wird dabei gemessen in der Anzahl der Knoten auf dem Pfad.

Wir nehmen an, dass uns eine Funktion `hoehe` zur Verfügung steht, die die Höhe eines beliebigen Binärbaums berechnet. Die Implementierung ist hier nicht relevant, der Funktionskopf lautet:

```
function hoehe (inWurzel: tRefKnoten): tNatZahl;
{ Gibt die Höhe des übergebenen Baumes zurück.
  Der leere Baum hat die Höhe 0. }
```

Die *Vollständigkeit* eines binären Baumes definieren wir (rekursiv) wie folgt:

Ein leerer Baum ist vollständig. Ein nicht-leerer Baum ist vollständig, genau dann wenn seine beiden Teilbäume vollständig sind und die gleiche Höhe haben.

Folgende Prozedur ist eine Erweiterung von `whatDoIDo` und überprüft mit Hilfe der Funktion `hoehe`, ob ein Baum vollständig ist:

```
1 procedure pruefeVollstaendigkeit(inWurzel: tRefKnoten;
2                                  var outErgebnis:boolean);
3 var ergL, ergR: boolean;
4     Blatt, ZweiNachf: boolean;
5 begin
6   if inWurzel = nil then
7     outErgebnis := true
8   else
9   begin
10    pruefeVollstaendigkeit(inWurzel^.links, ergL);
11    pruefeVollstaendigkeit(inWurzel^.rechts, ergR);
12    Blatt := (inWurzel^.links = nil) and (inWurzel^.rechts = nil);
13    ZweiNachf := (inWurzel^.links<>nil) and (inWurzel^.rechts<>nil);
14    outErgebnis :=  ergL and ergR
15                    and (Blatt or ZweiNachf)
16                    and (hoehe(inWurzel^.links)
17                       = hoehe(inWurzel^.rechts));
18  end
19end;
```

b) Geben Sie an, wie sich die Prozedur `pruefeVollstaendigkeit` vereinfachen lässt.
Es genügt, die Änderungen an betroffenen Programmzeilen (mit Zeilennummer) anzugeben bzw. nicht benötigte Programmzeilen zu streichen.

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Name: _____ Matrikelnummer: _____

Kurs 1613 "Einführung in die imperative Programmierung"

Nachklausur am 05.04.2008

Aufgabe 5 (1+3+2+2 Punkte)

Gegeben seien folgende Vereinbarungen:

```
    const MAXWERT=256;
          FELDMAX=5;

    type tFeld = Array[1..FELDMAX] of integer;

    function sucheInFeld(inFeld:tFeld; inZahl:integer): integer;
    { Sucht im Array tFeld nach einem Vorkommen von inZahl.
      inZahl muss zwischen 1 und MAXWERT liegen.
      Rueckgabe: -1, falls inZahl nicht zwischen 1 und MAXWERT,
                  0, falls inZahl gueltig, aber nicht gefunden,
                  Index der Fundstelle im Array (>0) sonst. }
    var i:integer;
1   begin
2     if (inZahl <= 0) or (inZahl > MAXWERT) then
3       sucheInFeld:=-1
4     else
5     begin
6       i:=1;
7       while (i < FELDMAX) and (inFeld[i] <> inZahl) do
8         i:=i+1;
9       sucheInFeld:=i
10    end;
11  end;
```

Die Implementierung ist möglicherweise fehlerhaft und soll getestet werden. Mit Blick auf die Spezifikation im Funktionskopf, die zwei Sonderfälle beschreibt, lassen sich ad-hoc drei Ausgabeäquivalenzklassen bilden: je eine pro Sonderfall und eine dritte für den Normalfall. Zu jeder wählen wir ein Testdatum als Repräsentant, und zwar:

D_1 := (A, 0, -1), D_2 := (A, 10, 0), D_3 := (A, 2, 2)

Der Einfachheit halber setzen wir dabei in allen drei Testdaten dasselbe Array A := [6,2,3,4,1] ein (d.h. es gelte A[1]=6, A[2]=2, A[3]=3, A[4]=4 und A[5]=1).

a) Die ersten beiden Komponenten jedes Testdatums stehen für die beiden Eingaben `inFeld` sowie `inZahl`. Was gibt die dritte Komponente der Testdaten an?

b) Zeichnen Sie den kompakten Kontrollflussgraphen für die Funktion `sucheInFeld`. Geben Sie dabei zu jedem Knoten an, welche Programmzeilen von diesem Knoten repräsentiert werden, indem Sie die Zeilennummern rechts neben den Knoten schreiben. Notieren Sie links den Namen des Knotens.

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Name: _____ Matrikelnummer: _____

Beispiel:

c) Wird mit den drei Testdaten D_1, D_2 und D_3 eine vollständige Zweigüberdeckung erreicht? Begründen Sie Ihre Antwort.

d) Wird mit den drei Testdaten D_1, D_2 und D_3 eine minimale Mehrfachbedingungsüberdeckung erreicht? Begründen Sie Ihre Antwort!

Bei Bedarf können Sie die folgende Tabelle zu Hilfe nehmen:

Prädikate	***Testdaten*** (mit A := [6,2,3,4,1])		
	(A, 0, -1)	(A, 2, 2)	(A, 10, 0)
`inZahl <= 0`			
`inZahl > MAXWERT`			
`if`-Bedingung			
`i < FELDMAX`			
`inFeld[i] <> inZahl`			
`while`-Bedingung			

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

Name: _____ Matrikelnummer: _____

Kurs 1613 “Einführung in die imperative Programmierung”

Nachklausur am 05.04.2008

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.
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 Laufvariable wird innerhalb einer **`for`**-Anweisung nicht manipuliert.
16. Die Grundsätze der strukturierten Programmierung sind strikt zu befolgen.