Kurs 1613 "Einführung in die imperative Programmierung"

Klausur am 03.03.2007

Wintersemester 2006/2007
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 19),
   - 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.

   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 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 27 Punkte erreichbar. Sie haben die Klausur sicher dann bestanden, wenn Sie mindestens 14 Punkte erreicht haben.

Wir wünschen Ihnen bei der Bearbeitung der Klausur viel Erfolg!

Kurs 1613 "Einführung in die imperative Programmierung"

Klausur am 03.03.2007

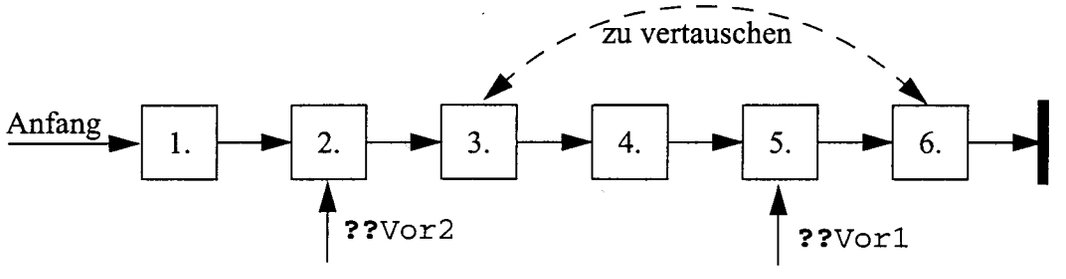
Aufgabe 1 (5 Punkte)

Gegeben seien folgende Typvereinbarungen für Elemente einer linearen Liste von Integer-Zahlen:

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

Schreiben Sie eine Prozedur, die zwei Elemente einer nicht-leeren Liste durch Änderung der Verkettung vertauscht. Die Prozedur bekommt zu diesem Zweck zwei Zeiger **??**Vor1 und **??**Vor2 übergeben, die jeweils auf ein Listenelement, jedoch nicht das letzte Listenelement zeigen. Es sind dann die Nachfolger der durch **??**Vor1 und **??**Vor2 referenzierten Elemente zu vertauschen.

Beispiel:



Benutzen Sie folgenden Prozedurkopf und ergänzen Sie dabei an den mit **??** gekennzeichneten Stellen die Übergabeart der Parameter:

```
procedure tauschen(?? ??Vor1, ?? ??Vor2:tRefElement);
{Vertauscht zwei Elemente einer Liste durch Änderung der Verkettung.
In ??Vor1 und ??Vor2 werden dabei Referenzen auf die Vorgängerelemente
der zu vertauschenden Elemente übergeben.
Vorbedingung:
??Vor1 und ??Vor2 zeigen jeweils auf ein existierendes Element derselben Liste, jedoch nicht auf das letzte.}
```

Kurs 1613 "Einführung in die imperative Programmierung"

Klausur am 03.03.2007

Aufgabe 2 (4 Punkte)

Gegeben seien folgende Typvereinbarungen für ein Array von Integer-Zahlen:

```
const MAX = ?;

type
  tIndex = 1..MAX;
  tIntFeld = array[tIndex] of integer;
```

Schreiben Sie eine Funktion, welche die Werte eines solchen Arrays abwechselnd addiert und subtrahiert und das Ergebnis zurückgibt.

Ein Beispiel:

Index i	1	2	3	4	5	6	7
Wert Feld[i]	9	7	4	3	1	0	-2

Für das in obiger Tabelle dargestellte Feld (bei MAX=7) berechnet sich die Summe wie folgt:

```
AltSumme := 9 - 7 + 4 - 3 + 1 - 0 + (-2) = 2
```

Sie können davon ausgehen, dass MAX > 1 gilt.

Benutzen Sie folgenden Funktionskopf:

```
function AltSumme(inFeld: tIntFeld): integer;
```

Kurs 1613 "Einführung in die imperative Programmierung"

Klausur am 03.03.2007

Aufgabe 3 (7 Punkte)

Es soll ein bestimmtes Sortierverfahren (eine einfache Variation von Quicksort) zum Sortieren von Arrays implementiert werden: Zum Sortieren dient eine *rekursive* Prozedur `sortiere`, die ihrerseits in jedem Rekursionsschritt auf eine Hilfsprozedur `trenne` zurückgreift.

Grob gesagt trennt die Prozedur `trenne` einen Array-Ausschnitt zwischen zwei Indizes (`inVon` und `inBis`) logisch in drei Teile auf: Ein „Trenn-Element", dessen Index in einem Ausgabeparameter `outTE` an die Prozedur `sortiere` zurückgegeben wird, einen Ausschnitt links des „Trenn-Elements", dessen Elemente sämtlich kleiner oder gleich dem „Trenn-Element", jedoch untereinander im Allgemeinen noch unsortiert sind, und einen Ausschnitt rechts des „Trenn-Elements", dessen Elemente sämtlich größer als das „Trenn-Element", jedoch untereinander ebenfalls im Allgemeinen unsortiert sind. Im Extremfall kann die Trennstelle `outTE` mit `inVon` oder `inBis` übereinstimmen, also der linke bzw. rechte Ausschnitt leer sein.
Das Sortierverfahren sieht nun vor, einen zu sortierenden Arrayausschnitt mittels `trenne` aufzuspalten und die dabei entstehenden noch unsortierten Teile zu sortieren.

Im Folgenden sind die benutzten Typvereinbarungen sowie die Prozedurköpfe von `trenne` und `sortiere` samt Spezifikation in Form eines Kommentars gegeben:

```
const MAX=10;
type
  tIndex=1..MAX;
  tIntFeld=array[tIndex] of integer;

procedure trenne(inVon, inBis:tIndex;
                 var ioFeld:tIntFeld;
                 var outTE: tIndex);
{Vorbedingung: inVon < inBis
 Arrangiert die Elemente von ioFeld im Ausschnitt von inVon bis inBis
 um und gibt einen Feldindex outTE (TE steht für Trenn-Element) aus, so
 dass folgende Nachbedingungen erfüllt sind:
 inVon <= outTE <= inBis
 Für alle i mit inVon <= i < outTE gilt ioFeld[i] <= ioFeld[outTE].
 Für alle j mit outTE < j <= inBis gilt ioFeld[j] > ioFeld[outTE].}

procedure sortiere(inVon, inBis:tIndex; var ioFeld:tIntFeld);
{Sortiert den Ausschnitt des Feldes ioFeld von inVon bis inBis
 (jeweils inklusive) aufsteigend, sofern inVon <= inBis gilt.
 Um das gesamte Feld zu sortieren, ist inVon=1 und inBis=MAX
 zu übergeben.}
```

Implementieren Sie nur die Prozedur `sortiere`. Die Prozedur `trenne` können Sie als gegeben voraussetzen.

Kurs 1613 "Einführung in die imperative Programmierung"

Klausur am 03.03.2007

Aufgabe 4 (7 Punkte)

Gegeben seien folgende Typvereinbarungen für binäre Bäume, deren Knoten natürliche Zahlen enthalten:

```
type
  tNatZahl= 0..maxint;
  tRefBinBaum = ^tBinBaum;
  tBinBaum =  record
                info : tNatZahl;
                links : tRefBinBaum;
                rechts : tRefBinBaum
              end;
```

Gesucht ist eine Funktion zur Erkennung, ob ein gegebener Binärbaum unsortiert ist. Man erkennt einen unsortierten Baum, sobald man darin einen Knoten findet, der einen linken Nachfolger mit größerem Wert oder einen rechten Nachfolger mit kleinerem Wert aufweist.

Da ein unsortierter Baum in der Regel sehr viele solcher Knoten aufweist und ein vollständiger Baumdurchlauf bei größeren Bäumen recht aufwendig ist, soll nur ein einzelner Pfad durch den Baum durchlaufen werden. Findet sich auf diesem Pfad ein solcher Knoten, wird der Baum als unsortiert erkannt. Andernfalls wird keine Aussage darüber getroffen, ob er sortiert oder unsortiert ist.

Hierzu soll folgender Algorithmus in Form einer *iterativen* Funktion implementiert werden:

1. Betrachte zunächst die Wurzel. Ist der Baum leer, gib `false` zurück.
2. Sei K der gerade betrachtete Knoten. Hat K einen linken Nachfolger, dessen `info`-Wert größer als der von K ist, oder hat K einen rechten Nachfolger, dessen `info`-Wert kleiner als der von K ist, so brich den Durchlauf ab und gib `true` zurück. → unsortiert
3. Andernfalls suche weiter:
   Hat der Knoten keinen Nachfolger, gib `false` zurück.
   Hat der Knoten nur einen Nachfolger, betrachte diesen als nächstes.
   Hat der Knoten zwei Nachfolger, betrachte im Folgenden denjenigen, dessen Wert stärker von dem Wert des aktuellen Knotens abweicht.

Nachfolgend geben wir eine unvollständige Implementierung der Funktion vor.

a) Geben Sie die an der mit { **a)** } markierten Stelle einzufügenden Anweisungen zur Umsetzung des Schritts 2. des oben angegebenen Algorithmus an, d.h. setzen Sie die Variable `gefunden` auf **`true`**, falls der Knoten `lauf^` einen linken Nachfolger mit größerem Wert oder einen rechten Nachfolger mit kleinerem Wert als `lauf^.info` besitzt.

b) Geben Sie die zum Vervollständigen der Funktion an der mit { **b)** } markierten Stelle noch einzufügenden Anweisungen an.

c) Wäre eine rekursive Implementierung der Funktion sinnvoller? Begründen Sie Ihre Antwort.

```
function testeUnsortiert(inRefWurzel: tRefBinBaum):boolean;
{Sucht nach einem Beweis, dass der Baum mit Wurzel inWurzel unsortiert
ist.
Ausgabe true impliziert, dass der Baum nicht sortiert ist.
Ausgabe false sagt nichts über die Sortierung des Baumes aus.
Für den leeren Baum wird false ausgegeben.}

var gefunden,
    hatlinks,
    hatrechts: boolean;
    lauf: tRefBinBaum;

begin
  gefunden:=false;
  {gefunden gibt an, ob ein Knoten mit kleinerem rechten oder
   größerem linken Nachfolger gefunden wurde.}
  lauf:=inRefWurzel;
  while (lauf <> nil) and not gefunden do
  begin
    hatlinks := lauf^.links <> nil;
    hatrechts := lauf^.rechts <> nil;

    {Existiert größerer linker oder kleinerer rechter Nachf.?}
    { a) }

    {Ansonsten weitersuchen}
    if not gefunden then
      if not hatlinks then
        {hat entweder nur rechten Nachfolger oder gar keinen}
        lauf:=lauf^.rechts
      else {hat mindestens linken Nachf.}
        if not hatrechts then {hat *nur* linken Nachf.}
          lauf:=lauf^.links
        else
          {Hat linken und rechten Nachf., wobei gilt:
          lauf^.links^.info <= lauf^.info <= lauf^.rechts^.info}
          { b) }
  end;
  testeUnsortiert := gefunden;
end;
```

Kurs 1613 "Einführung in die imperative Programmierung"

Klausur am 03.03.2007

Aufgabe 5 (4 Punkte)

Die Funktion `MinZiffer` soll die kleinste Ziffer der Dezimaldarstellung der nicht-negativen ganzen Zahl `inZahl` bestimmen.

```
type
tZiffer = 0..9;
tNatZahl = 0..maxint;

function MinZiffer (inZahl:tNatZahl): tZiffer;
{ bestimmt die kleinste Ziffer von inZahl}

  var
  rest : tNatZahl;
  min,
  ziffer: tZiffer;

1   begin
2     rest := inZahl;
3     min := 9;
4     while rest > 0 do                    2
5     begin
6       rest := rest div 10;
7       ziffer := rest mod 10;   2
8       if ziffer < min then
9         min := ziffer;                   min : 2
10    end;
11    MinZiffer := min
12  end; { MinZiffer }
```

a) Wird mit dem Testdatum (20, 0) eine *vollständige Zweigüberdeckung* erreicht? Begründen Sie Ihre Antwort.

b) Geben Sie im Rahmen des Boundary-Interior-Tests den *Testfall* für den *Boundary-Test* an.

→ Test mit 1 Schleifen-durchlauf ✓

Kurs 1613 "Einführung in die imperative Programmierung"

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.

Kurse 1612 und 1613

Anlage zur Klausur am 03.03.2007

Zu Aufgabe 1

Beim Vertauschen der zwei Listenelemente gibt es einen Sonderfall, der in der Aufgabenstellung eigentlich hätte ausgeschlossen werden sollen, nämlich den Fall, dass die beiden zu vertauschenden Elemente direkt aufeinander folgen. Die Behandlung dieses Sonderfalls erfordert zusätzlichen Aufwand.

Damit Ihnen aus unserem Fehler kein Nachteil entsteht, treffen wir folgende Regelung:

- Wurde die Aufgabe wie vorgesehen (d.h. ohne Sonderfallbehandlung) gelöst, können auch die vollen dafür vorgesehenen 5 Punkte erreicht werden.
- Haben Sie den Sonderfall erkannt und gesondert behandelt — also mehr geleistet, als eigentlich vorgesehen — können Sie dafür bis zu 3 Bonuspunkte erhalten.

Abschließend geben wir noch eine Musterlösung an, welche den Sonderfall berücksichtigt:

```
procedure tauschen(inVor1, inVor2:tRefElement);
{Vertauscht zwei Elemente einer Liste durch Änderung der Verket-
tung. In inVor1 und inVor2 werden dabei Referenzen auf die Vor-
gängerelemente der zu vertauschenden Elemente übergeben.
Vorbedingung:
inVor1 und inVor2 zeigen jeweils auf ein existierendes Element
derselben Liste, jedoch nicht auf das letzte.}

var
  Element1, Element2, Nach1, Nach2 : tRefElement;

begin
  {Der Übersichtlichkeit halber setzen wir Hilfszeiger auf die
  zu vertauschenden Elemente sowie ihre Nachfolger:}
  Element1 := inVor1^.next;
  Element2 := inVor2^.next;
  Nach1 := Element1^.next;
  Nach2 := Element2^.next;

  {Zeiger auf die beiden Elemente ändern}
  if inVor1 <> Element2 then
    inVor1^.next := Element2
  else {Sonderfall: Element2 direkt vor Element1}
    Element1^.next := Element2;
  if inVor2 <> Element1 then
    inVor2^.next := Element1
  else {Sonderfall: Element1 direkt vor Element2}
    Element2^.next := Element1;
```

```
  {Nachfolgerzeiger der Elemente ändern}
  if Element2 <> Nach1 then
    Element2^.next := Nach1;
  if Element1 <> Nach2 then
    Element1^.next := Nach2;
end;
```

Zu Aufgabe 2

In der Musterlösung zu Aufgabe 2 sind die Kommentare fehlerhaft: Arrayelemente mit *geradem* Index werden subtrahiert, Elemente mit *ungeradem* Index werden addiert.