Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Wintersemester 2008/2009
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,
   - 6 Aufgaben (Seite 2 - Seite 21),
   - 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 die 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! (Sollten Sie mehr als eine Lösung zu einer Aufgabe abgeben, so wird nur eine davon korrigiert – und nicht notwendig die bessere.)
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, benutzen Sie **keinen Bleistift**!) sind **keine weiteren Hilfsmittel** zugelassen. Die Muss-Regeln des Programmierstils finden Sie im Anschluss an die Aufgabenstellung.
6. Es sind maximal 32 Punkte erreichbar. Sie haben die Klausur sicher dann bestanden, wenn Sie mindestens 16 Punkte erreicht haben.

Wir wünschen Ihnen bei der Bearbeitung der Klausur viel Erfolg!

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Aufgabe 1 (3+1 Punkte)

Es sind folgende Konstanten- und Typdefinitionen für ein Feld von Integer-Zahlen gegeben:

```
const MAXINDEX = 10;
type  tIndex = 1..MAXINDEX;
      tFeld = array [tIndex] of integer;
```

a) Schreiben Sie eine Prozedur, die ein Feld vom Typ `tFeld` übergegeben bekommt und sowohl das Minimum als auch das Maximum aller Werte im Feld bestimmt und zurückgibt.
Durchlaufen Sie dabei das Feld genau einmal mit einer For-Schleife!
Verwenden Sie weiterhin folgenden Prozedurkopf:

```
procedure FeldMinMax(inFeld: tFeld; var outMin, outMax: integer);
{ Ermittelt den kleinsten sowie den größten aller Werte in inFeld
und gibt diese in den Ausgabeparametern outMin bzw. outMax zurück.}
```

b) Ist der Einsatz einer For-Schleife hier sinnvoll? Begründen Sie Ihre Antwort.

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

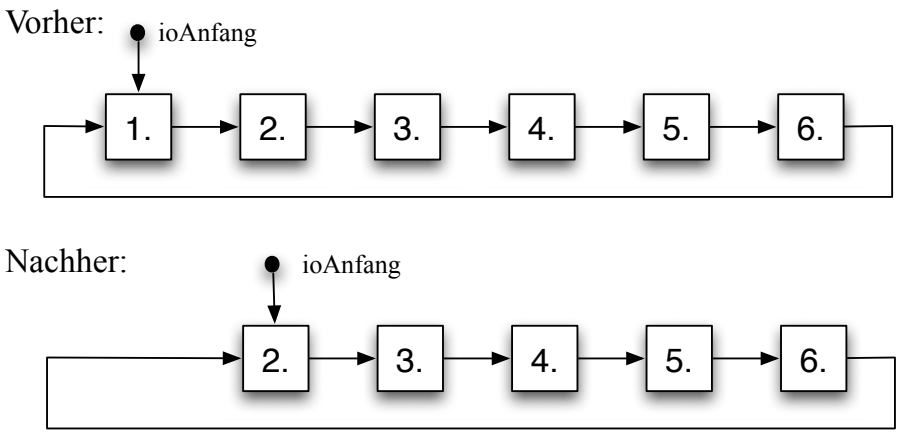
Aufgabe 2 (5 Punkte)

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

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

Aus Elementen dieses Typs kann man neben den aus dem Kurs bekannten linearen Listen auch *zyklische* lineare Listen bilden, indem der `next`-Zeiger des letzten Elements nicht den Wert **`nil`** besitzt, sondern wieder auf das erste Listenelement verweist. Als erstes Element eines solchen „Rings“ betrachten wir dabei das Element, auf das der Anfangszeiger verweist. Eine leere Liste wird unverändert durch einen Anfangszeiger mit dem Wert **`nil`** dargestellt.

Gesucht ist eine Prozedur, die das erste Element aus einer solchen zyklischen Liste löscht, wie in der folgenden Abbildung am Beispiel einer sechselementigen Liste skizziert:



Implementieren Sie die Prozedur. Sie dürfen dabei voraussetzen, dass die Liste nicht leer ist, dass also `ioAnfang <> nil` gilt. Tipp: Überlegen Sie jedoch vor der Implementierung, ob es noch weitere Sonderfälle zu behandeln gibt.

Verwenden Sie folgenden Prozedurkopf:

```
procedure LoescheErstesElement(var ioAnfang: tRefElement);
{ Löscht das erste Element aus der nicht leeren zyklischen Liste
  mit Anfangszeiger ioAnfang. }
```

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Aufgabe 3 (6 Punkte)

Im Folgenden geben wir die zur Darstellung von binären Bäumen verwendeten Typdefinitionen vor sowie drei Funktionen, die das Maximum aller Werte in einem *binären Suchbaum* (sortierter Binärbaum) bestimmen sollen. Dabei setzen wir voraus, dass der Suchbaum *nicht leer* ist, so dass das Maximum immer definiert ist.

```
type
  tRefKnoten = ^tKnoten;
  tKnoten = record
              info: integer;
              links,
              rechts: tRefKnoten
            end;

function suchbaumMax1(inWurzel: tRefKnoten): integer;
begin
  if inWurzel^.rechts = nil then
    suchbaumMax1 := inWurzel^.info
  else
    suchbaumMax1 := suchbaumMax1(inWurzel^.rechts)
end;

function suchbaumMax2(inWurzel: tRefKnoten): integer;
var maxL, maxR, max: integer;
begin
  max := inWurzel^.info;
  if (inWurzel^.links <> nil) then
  begin
    maxL := suchbaumMax2(inWurzel^.links);
    if maxL > max then
      max := maxL;
  end;
  if (inWurzel^.rechts <> nil) then
  begin
    maxR := suchbaumMax2(inWurzel^.rechts);
    if maxR > max then
      max := maxR;
  end;
  suchbaumMax2 := max;
end;
```

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Name: _____ Matrikelnummer: _____

```
function suchbaumMax3(inWurzel: tRefKnoten): integer;
var lauf: tRefKnoten;
begin
  lauf := inWurzel;
  while lauf^.rechts <> nil do
    lauf := lauf^.rechts;
  suchbaumMax3 := lauf^.info
end;
```

- Geben Sie zu jeder der drei Funktionen an, ob sie das Maximum eines nicht leeren Suchbaumes korrekt ermittelt.
- Falls eine oder mehrere der Funktionen das Maximum nicht korrekt bestimmen, erläutern Sie jeweils, wo der Fehler liegt.
- Falls mehrere der Funktionen das Maximum korrekt bestimmen, beschreiben Sie *kurz*, worin sich ihre Ansätze unterscheiden und welcher Ansatz zu bevorzugen ist.

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Aufgabe 4 (4+2 Punkte)

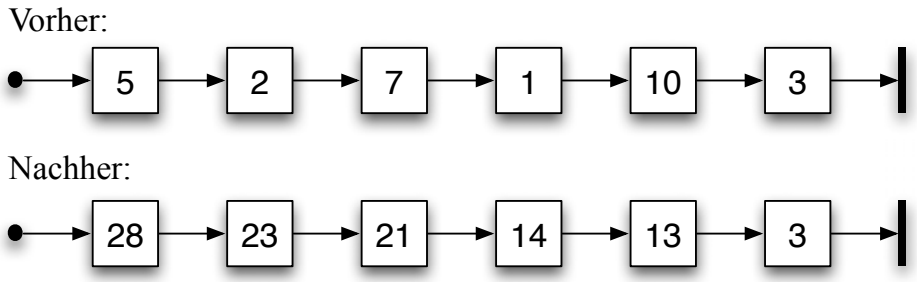
Gegeben seien folgende Typvereinbarungen für Elemente einer linearen Liste ganzer Zahlen:

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

a) Schreiben Sie eine *rekursive* Prozedur `rueckwaertsAddieren`, die eine solche ihr übergebene Liste wie folgt verändert:

- — Das letzte Listenelement bleibt unverändert.
- — Zum Wert des vorletzten Listenelements wird der Wert des letzten Listenelements addiert.
- — Anschließend wird zum Wert des drittletzten Listenelements der im vorhergehenden Schritt bereits neu berechnete Wert des vorletzten Listenelements addiert u.s.w.
- — Allgemein wird zum Wert eines jeden Listenelements der Wert seines Nachfolgers addiert, nachdem diese Operation bereits für den Nachfolger erfolgt ist. (Die Liste wird also rückwärts abgearbeitet.)

Die folgende Abbildung zeigt beispielhaft die Veränderung an einer der Prozedur übergebenen Liste:



Sie dürfen voraussetzen, dass die *Liste nicht leer* ist.

Verwenden Sie folgenden Prozedurkopf und ergänzen Sie dabei an den mit **??** gekennzeichneten Stellen die Übergabeart des Parameters:

```
procedure rueckwaertsAddieren(?? ??Anfang: tRefElement);
{Addiert den Wert des letzten Listenelements zum Wert des
 vorletzten, den neuen Wert des vorletzten zum Wert drittletzten
 etc.}
```

b) Handelt es sich hier um eine sinnvolle Anwendung der Rekursion? Begründen Sie Ihre Antwort.

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Name: _____ Matrikelnummer: _____

Aufgabe 5 Black-Box-Test (2+2 Punkte)

Die Prozedur aus Aufgabe 1, also eine Prozedur zur Bestimmung des Maximums und des Minimums aller Werte eines einzugebenden Feldes, soll einem *funktionsorientierten Test* unterzogen werden. Bekannt seien die Konstanten-[1] und Typdefinitionen sowie der Prozedurkopf:

```
const MAXINDEX = 3;
type  tIndex = 1..MAXINDEX;
      tFeld = array [tIndex] of integer;

procedure FeldMinMax(inFeld: tFeld; var outMin, outMax: integer);
{ Ermittelt den kleinsten sowie den größten aller Werte in inFeld und
gibt diese in den Ausgabeparametern outMin bzw. outMax zurück. }
```

Die Menge $A := \{ (min, max) \mid min \leq max, min \in \mathbb{Z}, max \in \mathbb{Z} \}$ der zulässigen Ausgaben soll nun in *Ausgabeäquivalenzklassen* zerlegt werden. Wir geben im Folgenden 6 mögliche Ausgabeäquivalenzklassen vor:

$A_1 := \{ (min, max) \mid min < max, max < 0 \}$

$A_2 := \{ (min, max) \mid min < max, max = 0 \}$

$A_3 := \{ (min, max) \mid min = max, max < 0 \}$

$A_4 := \{ (min, max) \mid min = max, max = 0 \}$

$A_5 := \{ (min, max) \mid 0 < min, min < max \}$

$A_6 := \{ (min, max) \mid 0 = min, min < max \}$

(Als Teilmenge von A gilt für jede Klasse A_i weiterhin $min, max \in \mathbb{Z}$, was wir aus Gründen der Übersichtlichkeit nicht mehr in jeder Menge explizit mit angeben.)

a) A_1 bis A_6 allein bilden noch keine Zerlegung von A. Nennen Sie die noch fehlenden Klassen.

b) Geben Sie zu den Klassen A_4 und A_5 jeweils ein Testdatum an. Notieren Sie dabei ein Array vom Typ `tFeld` als ein Tripel in eckigen Klammern. ([–7,2,0] stehe beispielsweise für das Array a mit a[1] = –7, a[2] = 2 und a[3] = 0.)

---

1. Den Wert der Konstanten MAXINDEX haben wir in dieser Aufgabe niedriger als in Aufgabe 1 angesetzt, um Ihnen Schreibarbeit bei Teil b) zu sparen.

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Aufgabe 6 White-Box-Test (2+3+2 Punkte)

Gegeben seien folgende Vereinbarungen:

```
const
  FELDMAX = 4;
type
  tFeld = array[1..FELDMAX] of integer;

function enthaeltNull(inFeld: tFeld): boolean;
{ Gibt genau dann true zurueck, wenn inFeld eine 0 enthält. }
var gefunden: boolean;
    i: integer;


1   begin
2     gefunden := false;
3     i := 1;
4     repeat
5       i := i + 1;
6       if inFeld[i] = 0 then
7         gefunden := true;
8     until gefunden or (i = FELDMAX);
9     enthaeltNull := gefunden;
10  end;
```

Die Funktion `enthaeltNull` ist möglicherweise fehlerhaft und soll einem Boundary-Interior-Pfadtest unterzogen werden.

a) Komplettieren Sie den auf der folgenden Seite abgebildeten Kontrollflussgraphen, indem Sie die Programmzeilen 1 bis 10 der oben angegebenen Funktion `enthaeltNull` den Knoten des Graphen zuordnen. (Es genügt, wenn Sie die Zeilennummern rechts neben die Knoten schreiben.)

b) Als nächstes sind die Pfadklassen für einen Boundary-Interior-Pfadtest zu bilden. Da eine Repeat-Schleife immer mindestens einmal durchlaufen wird, betrachten wir nur die Boundary-Klasse sowie die Interior-Klasse für n=2. Geben Sie zu jeder der beiden Klassen jeweils alle Pfade an, die sie umfasst (unabhängig davon, ob die Pfade ausführbar[1] sind).

c) Geben Sie zu jedem Pfad (der *Boundary*- und der *Interior-Klasse*) an, ob er ausführbar ist oder nicht. Beachten Sie dabei, dass `FELDMAX` = 4 gilt. Begründen Sie Ihre Antwort.

---

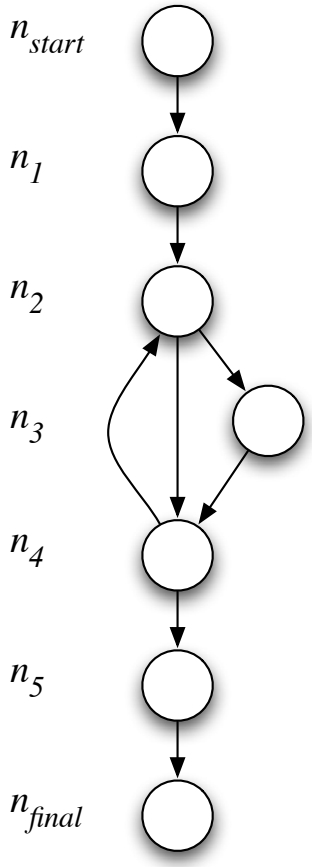
1. Ein Pfad ist ausführbar, genau dann wenn sein assoziierter Testfall nicht die leere Menge ist, wenn also Testdaten existieren, deren Eingabedaten zum Durchlauf des Pfades führen.

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Name: _____ Matrikelnummer: _____

Der Kontrollflussgraph[1]



1. Um die Zuordnung nicht teilweise vorweg zu nehmen und Ihnen Schreibarbeit in Teil b) zu ersparen, haben wir alle inneren Knoten nur mit kurzen, numerierten Knotennamen versehen. Diese Namen müssen Sie nicht ändern.

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Kurs 1613 „Einführung in die imperative Programmierung“

Nachklausur am 28.03.2009

Name: _____ Matrikelnummer: _____

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.