FernUniversität
-Gesamthochschule- in Hagen

Bitte hier unbedingt Matrikelnummer und Adresse eintragen, sonst keine Bearbeitung möglich.

Postanschrift: FernUniversität D - 58084 Hagen

Name, Vorname

Straße, Nr.

PLZ, Wohnort

FERNUNIVERSITÄT
- Gesamthochschule -
EINGANG

INF

FERNUNIVERSITÄT
Gesamthochschule
D-58084 Hagen

Fachbereich Informatik

Kurs: 1613 "Einführung in die imperative Programmierung"

Hauptklausur am 2. März 2002

Hörerstatus:

( ) Vollzeitstudent
( ) Teilzeitstudent
( ) Zweithörer
( ) Gasthörer

Klausurort:

( ) Berlin
( ) Bochum
( ) Frankfurt
( ) Hamburg
( ) Karlsruhe
( ) Kassel
( ) Köln
( ) München
( ) Bregenz
( ) Wien
( ) . . . .

Aufgabe	1	2	3	4	5	Summe
erreichbare Punktzahl	6	8	10	10	8	42
bearbeitet						
erreichte Punktzahl						

Datum: ____________________ Korrektur: ____________________

©2000 FernUniversität - Gesamthochschule in Hagen

Kurs 1613 “Einführung in die imperative Programmierung”

Klausur am 02.03.2002

Wintersemester 2001/2002
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 14),
   - 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 oben auf jedem von Ihnen beschriebenen Blatt links Ihren Namen und 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 finden Sie im Anschluß an die Aufgabenstellung.
6. Es sind maximal 42 Punkte erreichbar. Sie haben die Klausur sicher dann bestanden, wenn Sie mindestens 21 Punkte erreicht haben.
7. Die Klausurdauer beträgt **zwei Stunden**.

Wir wünschen Ihnen bei der Bearbeitung der Klausur viel Erfolg!

Aufgabe 1 (6 Punkte)

Eine natürliche Zahl n heißt *perfekt*, wenn n als Summe aller natürlicher Zahlen i ($1 \leq i \leq \frac{n}{2}$), durch die n ohne Rest teilbar ist, dargestellt werden kann. Die Zahl 28 ist beispielsweise eine perfekte Zahl. Es ist 28 = 1 + 2 + 4 + 7 + 14 und 1, 2, 4, 7 und 14 sind genau alle Zahlen, durch die 28 ohne Rest teilbar ist. Schreiben Sie ein Programm `perfekteZahl`, das alle perfekten Zahlen zwischen 1 und 1000 auf dem Bildschirm ausgibt.

Name: ______________________________ Matrikelnummer: _______________

Aufgabe 2 (8 Punkte)

Ein Handlungsreisender muss `MAXORTSZAHL` Städte besuchen (`MAXORTSZAHL > 1`) und möchte dafür eine Reise planen, bei der jede Stadt genau einmal besucht wird. Aus Sparsamkeits- und Umweltschutzgründen sollte die Reise möglichst kurz sein.

Es soll eine PASCAL-Prozedur `Reise` entwickelt werden, die eine Reise ermittelt und die Ortsnummern in der Reihenfolge der Reise sowie die insgesamt zu fahrenden Kilometer ausgibt. Dazu wird der Prozedur eine Entfernungsmatrix übergeben, in der das Element in der i-ten Zeile und j-ten Spalte die Entfernung zwischen der Stadt i und der Stadt j angibt. Die Entfernungen zwischen den Städten und die Länge der Reise sind immer positive `integer`-Zahlen kleiner als `maxint`.

Zur Bestimmung einer kurzen Reise wird folgendes Verfahren benutzt: Ausgehend von Stadt 1 wird diejenige Stadt bestimmt, welche die kürzeste Entfernung von Stadt 1 besitzt. Von dieser Stadt aus wird wiederum die nächstgelegene Stadt ermittelt und von der aus wieder die nächstgelegene usw., bis alle Städte besucht worden sind. Die Information, welche Städte schon besucht sind, wird in einem booleschen Feld abgelegt.

Als Beispiel für `MAXORTSZAHL = 3` geben wir die folgende Entfernungsmatrix und die resultierende Prozedurausgabe an:

Entfernungsmatrix:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} 1\ 2\ 3 \\ \begin{bmatrix} 0 & 7 & 4 \\ 7 & 0 & 6 \\ 4 & 6 & 0 \end{bmatrix} \end{array}$$

Prozedurausgabe:

```
Reise:
1
3
2
10 km
```

Ihre Aufgabe besteht darin, unter Benutzung der Konstantendefinition und Typdefinitionen

```
const
MAXORTSZAHL = 10;

type
tOrtsIndex = 1..MAXORTSZAHL;
tEntfernung = 0..maxint;
tEntfMatrix = array [tOrtsIndex, tOrtsIndex] of tEntfernung;
```

untenstehende Prozedur so zu ergänzen, dass sie obiges Verfahren implementiert.

Kurs 1613 "Einführung in die imperative Programmierung"

Klausur am 02.03.2002

Name: ______________________________ Matrikelnummer: ________________

```
procedure Reise (var inMat : tEntfMatrix);
{ berechnet eine kurze Reise aus der Entfernungsmatrix
  inMat }

  type
  tBoolFeld = array [tOrtsIndex] of boolean;
  { zur Markierung bereits besuchter Staedte }

  var
  besucht : tBoolFeld;
  Gesamtstrecke : tEntfernung;
  hier, { gibt die aktuelle Stadt an }
  naechste, { gibt die naechste zu besuchende Stadt an }
  i : tOrtsIndex;
  { ...setzen Sie hier evtl. weitere Deklarationen ein }

begin { procedure }
  for i := 1 to MAXORTSZAHL do
    besucht [i] := false;
  { initialisiert das Feld besuchter Staedte }

  Gesamtstrecke := 0;
  hier := 1;
  besucht [hier] := true;
  writeln ('Reise: ');
  writeln (hier);

  { es sind noch MAXORTSZAHL - 1 Staedte zu besuchen }
  for i := 1 to MAXORTSZAHL - 1 do
  begin

    Komplettieren Sie den Schleifenrumpf von Reise, indem
    Sie die hier erforderlichen Anweisungen nach dem
    untenstehenden Buchstaben A angeben.

  end;
  writeln (Gesamtstrecke, ' km')
end; { Reise }
```

A:

Kurs 1613 “Einführung in die imperative Programmierung”

Klausur am 02.03.2002

Name: ______________________________ Matrikelnummer: ________________

Aufgabe 3 (5 + 5 Punkte)

Zwei Mengen `M1` und `M2` mit ganzen Zahlen seien durch einfach-verkettete lineare Listen realisiert. Die Zahlen innerhalb der Liste sind nicht sortiert. Die Prozedur `Minus(M1, M2)` entfernt alle Zahlen aus der Liste `M1`, die ebenfalls in der Liste `M2` enthalten sind. Das Entfernen dieser Zahlen aus der Liste `M1` soll ausschließlich durch Ändern der Verkettung geschehen.

Implementieren Sie die Prozedur `Minus`, indem Sie die Teilaufgaben a) und b) bearbeiten. Gehen Sie dabei von den angegebenen Typen und Prozedurköpfen aus.

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

a) Implementieren Sie eine Funktion `finden`, die einen Suchwert in einer Liste finden soll. Der zu suchende Wert wird in dem Parameter `inWert` und die zu durchsuchende Liste in dem Parameter `inRefListe` an die Funktion übergeben. Wird der Suchwert in der Liste gefunden, so soll die Funktion einen Zeiger auf dieses Element zurückliefern, ansonsten gibt die Funktion den Wert **`nil`** zurück.

```
function finden (
            inWert : integer;
            inRefListe : tRefListe) : tRefListe;
{ sucht den Wert inWert in der Liste, auf deren Anfang
  inRefListe zeigt. Wird der Wert gefunden, liefert die
  Funktion einen Zeiger auf dieses Element zurueck,
  ansonsten nil. }
```

b) Implementieren Sie die Prozedur `Minus` unter Benutzung der Funktion `finden`. Vervollständigen Sie dazu untenstehende Prozedur, indem Sie für die Platzhalter (1) bis (5) in der Prozedur einen Ausdruck, eine Bedingung, eine Anweisung oder eine Anweisungsfolge einsetzen.

```
procedure Minus (
            var ioRefM1 : tRefListe;
                inRefM2 : tRefListe);
{ entfernt nur durch Aendern der Verkettung alle Elemente
  aus M1, deren Werte auch in M2 vorkommen. }

  var
  RefLauf1,
  RefLauf2: tRefListe;
```

Kurs 1613 "Einführung in die imperative Programmierung"

Klausur am 02.03.2002

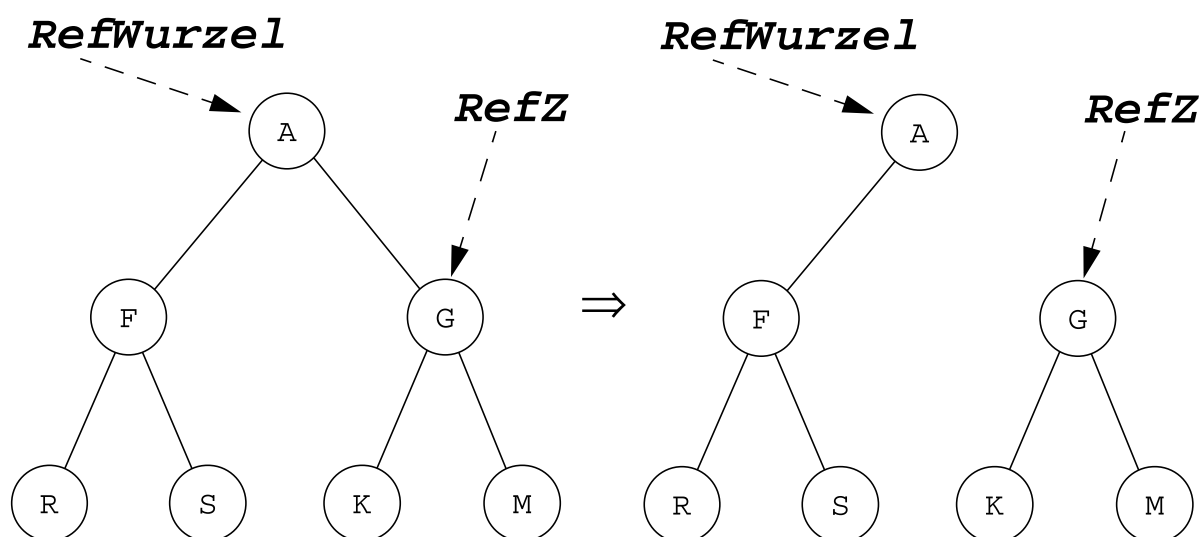
Name: ______________________________ Matrikelnummer: ________________

```
begin
  RefLauf2 := inRefM2;
  while RefLauf2 <> nil do
  begin
    RefWert := finden((1), (2));
    if RefWert <> nil then { Suchwert in ioRefM1 gefunden? }
      if (3) then
        { Sonderfall: Suchwert ist erstes Element in ioRefM1 }
        (4)
      else { Normalfall }
      begin
        (5)
      end
    end;
    RefLauf2 := RefLauf2^.next;
  end { while }
end; { Minus }
```

Platzhalter	Ausdruck, Bedingung, Anweisung oder Anweisungsfolge
(1)	
(2)	
(3)	
(4)	
(5)	

Aufgabe 4 (10 Punkte)

Gegeben sei ein (nicht sortierter) binärer Baum und ein Zeiger `RefZ` auf einen Knoten, der im Baum enthalten ist. Ihre Aufgabe ist es, eine Prozedur `Trennen` zu implementieren, die den Teilbaum, auf dessen Wurzel `RefZ` zeigt, vom Baum abtrennt. Zeigt `RefZ` auf die Wurzel des Baumes, dann ist nichts zu tun. Die folgende Abbildung veranschaulicht das Prinzip.



Gehen Sie von den folgenden Typdefinitionen und dem angegebenen Prozedurkopf aus.

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

procedure Trennen (
            inRefWurzel,
            inRefZ : tRefBinBaum);
{ trennt den Teilbaum, auf dessen Wurzel inRefZ zeigt,
  vom Baum ab, auf dessen Wurzel inRefWurzel zeigt; zeigen
  inRefWurzel und inRefZ auf denselben Knoten, geschieht
  nichts }
```

Name: ______________________________ Matrikelnummer: ________________

Aufgabe 5 (2 + 6 Punkte)

Die Funktion `Hoch` berechnet die `n`-te Potenz einer `integer`-Zahl, die größer 0 ist. Beispielsweise ist die 4-te Potenz von 6 gleich 6 * 6 * 6 * 6 = 1296; also `Hoch` (6,4) = 1296. Sie können annehmen, dass `Zahl` > 0 und `n>=0` gilt.

Gehen Sie dabei von folgender Typdefinition aus:

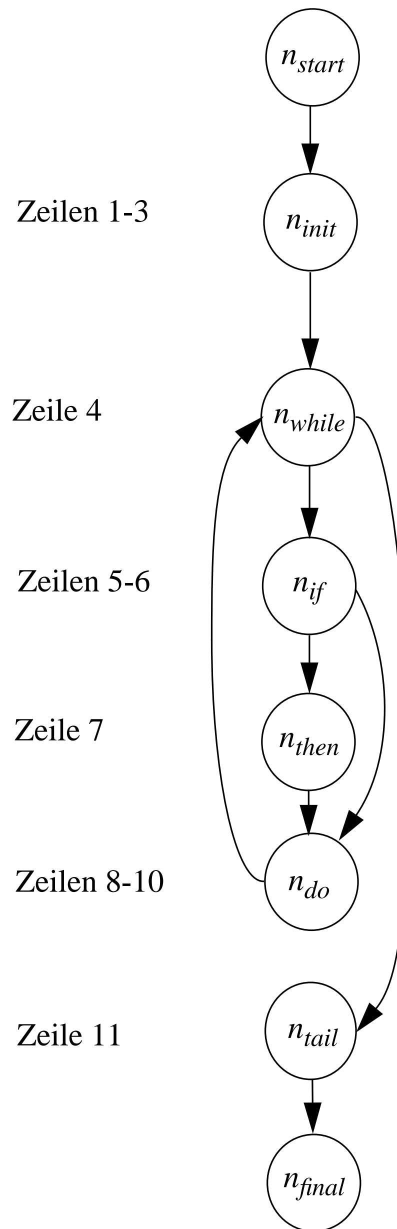
```
type
tNatZahl = 0..maxint;
tNatZahlPlus = 1..maxint;

function Hoch (Zahl : tNatZahlPlus;
               n : tNatZahl): tNatZahlPlus;
{berechnet fuer n >= 0 die n-te Potenz von Zahl > 0}

  var
  Temp1,
  Temp2 : tNatZahlPlus;
  Temp3 : tNatZahl;

begin
1   Temp1 := 1;
2   Temp2 := Zahl;
3   Temp3 := n;
4   while Temp3 > 0 do
5   begin
6     if odd(Temp3)then
7       Temp1 := Temp1*Temp2;
8     Temp2 := sqr(Temp2);
9     Temp3 := Temp3 div 2
10  end;
11  Hoch := Temp1
end; {Hoch}
```

Kurs 1613 "Einführung in die imperative Programmierung"

Klausur am 02.03.2002

Name: ______________________________ Matrikelnummer: ________________

Wir geben zusätzlich den kompaktifizierten Kontrollflußgraphen zu `Hoch` an:



Kompaktifizierter Kontrollflußgraph von `Hoch`

Kurs 1613 “Einführung in die imperative Programmierung”

Klausur am 02.03.2002

---

a) Geben Sie eine Knotenfolge an, die eine vollständige Anweisungsüberdeckung für die Funktion `Hoch` bildet. Wie lautet ein Testdatum zu dieser Knotenfolge?

b) Es soll ein boundary-interior Test der Funktion durchgeführt werden. Geben Sie für jede der drei Testklassen einen passenden Pfad und die dazugehörigen assoziativen Testfälle an. Wählen Sie für die interior-Klasse n=2, d.h. betrachten Sie Testdaten, die die Schleife genau zweimal durchlaufen.

Kurs 1613 “Einführung in die imperative Programmierung”

Klausur am 02.03.2002

Name: ______________________________ Matrikelnummer: ________________

Kurs 1613 “Einführung in die imperative Programmierung”

Klausur am 02.03.2002

Name: ______________________________ Matrikelnummer: ________________

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.