Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Wintersemester 2009/2010
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 22),
   - 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 30 Punkte erreichbar. Sie haben die Klausur sicher dann bestanden, wenn Sie mindestens 15 Punkte erreicht haben.

Wir wünschen Ihnen bei der Bearbeitung der Klausur viel Erfolg!

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Aufgabe 1 (5 Punkte)

Gegeben ist folgende Typdefinition für eine lineare Liste:

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

Es soll eine Prozedur `NachVorn` implementiert werden, die ein Element mit einem als Parameter übergebenen Info-Wert (vom Typ `integer`) sucht und dieses an den Anfang der Liste *nur durch Ändern der Verkettung* einfügt.
Es darf dabei vorausgesetzt werden, dass ein solches Element in der Liste vorkommt!

Benutzen Sie untenstehenden Prozedurkopf und ergänzen Sie auch die Parameterübergabearten.

```
procedure NachVorn( ?? ??Wert: integer;
                    ?? ??RefAnfang: tRefElement);
```

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Aufgabe 2 (6 Punkte)

Römische Zahlen werden nach einem Additionssystem gebildet: Sie bestehen aus einer Folge von römischen Ziffern, deren Werte im Allgemeinen addiert, in gewissen Fällen auch subtrahiert werden.
Die folgende Tabelle zeigt die zulässigen Ziffern sowie ihre Werte:

Ziffer	I	V	X	L	C	D	M
Wert	1	5	10	50	100	500	1000

Wir betrachten römische Zahlen, die nach folgenden Regeln aufgebaut sind[1]:

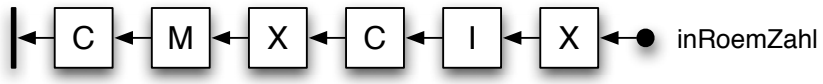
- Die einzelnen Ziffern einer römischen Zahl sind nach ihrem Wert absteigend sortiert, d.h. auf eine Ziffer folgt entweder eine gleichwertige (außer V, L, oder D) oder niederwertige Ziffer – mit folgender Ausnahme:
- Subtraktionsregel: Es ist erlaubt, maximal eine kleinere Ziffer vor eine größere Ziffer zu schreiben (genauer: Vor einem X oder V darf ein I stehen, vor einem C oder L darf ein X stehen und vor einem M oder D darf ein C stehen). In diesem Fall wird der Wert der vorangestellten kleineren Ziffer vom Wert der nachfolgenden Ziffer subtrahiert, in allen anderen Fällen werden die Ziffern addiert.

  Anmerkung: Es dürfen explizit nicht mehrere kleinere Ziffern vor einer größeren stehen, denn dann wäre der Wert der Zahl nicht mehr eindeutig bestimmt. Beispiel: IXL könnte als 50 - (10 - 1) = 41 oder auch als (50 - 10) - 1 = 39 gelesen werden.

Betrachten wir zwei Beispiele zur Ermittlung des Werts einer römischen Zahl:

Römische Zahl	Addition der Ziffernwerte	Gesamtwert
CMXCIX	(-100)+1000+(-10)+100+(-1)+10	999
MCMLXXXIV	1000+(-100)+1000+50+10+10+10+(-1)+5	1984

Ihre Aufgabe ist, eine Pascal-Funktion zu schreiben, die den Wert einer römischen Zahl berechnet. Eine römische Zahl wird dabei als (nicht leere) lineare Liste von römischen Ziffern dargestellt, in welcher die Ziffern von rechts nach links aufgezählt werden. Die folgende Abbildung zeigt beispielhaft die Darstellung der Zahl CMXCIX als solche Liste:



CMXCIX = 999

Hinweis: Die Ziffern sind deshalb von rechts nach links verkettet, da beim Addieren der Werte von rechts nach links die Entscheidung, ob eine Ziffer subtrahiert oder addiert werden muss, direkt im Vergleich mit der gerade zuvor verarbeiteten Ziffer getroffen werden kann und nicht „vorausgeschaut“ werden muss.

1. Es gibt auch andere Schreibweisen für römische Zahlen. Obige Regeln spezifizieren die Syntax der Zahlen nicht vollständig, aber hinreichend, um solche Zahlen lesen und auswerten zu können.

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Name: _____ Matrikelnummer: _____

Für diese Darstellung vereinbaren wir folgende Typen:

```
type  tRoemZiff = (I, V, X, L, C, D, M);
      tRefRoemZahlStelle = ^tRoemZahlStelle;
      tRoemZahlStelle = record
                          ziffer: tRoemZiff;
                          links:  tRefRoemZahlStelle;
                        end;
```

Weiterhin dürfen Sie folgende Funktion als gegeben annehmen und verwenden:

```
function RoemZiffWert(inZiff: tRoemZiff): integer;
{Gibt den Wert einer römischen Ziffer zurück,
 z.B. RoemZiffWert(L)=50.}
```

Schreiben Sie nun eine iterative Funktion `RoemZahlWert` zur Berechnung des Werts einer römischen Zahl unter Rückgriff auf `RoemZiffWert`. Benutzen Sie folgenden Funktionskopf:

```
function RoemZahlWert(inRoemZahl: tRefRoemZahlStelle): integer;
{Berechnet den Wert der römischen Zahl, auf deren letzte/rechte Stelle
der Zeiger inRoemZahl verweist.
Vorbedingung: inRoemZahl <> nil, d.h. die römische Zahl ist mindestens
einstellig. }
```

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Name: _____ Matrikelnummer: _____

Aufgabe 3 (4 Punkte)

Die Folge der so genannten Fibonacci-Zahlen beginnt mit der Zahl 0, gefolgt von der Zahl 1. Jede weitere Fibonacci-Zahl ergibt sich aus der Summe der beiden ihr vorangehenden Zahlen:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, ...

Für $n \geq 0$ können wir können die n-te Fibonacci-Zahl wie folgt rekursiv definieren:

$$f(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ f(n-1) + f(n-2) & \text{für } n > 1. \end{cases}$$

Gegeben sei weiterhin die folgende Pascal-Typdefinition zur Darstellung natürlicher Zahlen:

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

Schreiben Sie eine *iterative* Pascal-Funktion `Fibonacci`, die eine natürliche Zahl n (des Typs `tNatZahl`) entgegennimmt und dazu entsprechend obiger Definition die n-te Fibonacci-Zahl (ebenfalls vom Typ `tNatZahl`) zurückgibt.

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Name: _____ Matrikelnummer: _____

Aufgabe 4 (2+4 Punkte)

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

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

Einen Binärbaum aus Knoten dieses Typs bezeichnen wir im Folgenden als *Max-Heap*[1], wenn für jeden Knoten gilt, dass keines seiner Kinder einen größeren Info-Wert besitzt als er selbst. Umgekehrt betrachtet: Ein Baum ist *kein* Max-Heap, genau dann wenn er einen Knoten mit „größerem Kind“ enthält (also einen Knoten *K* , dessen linker oder rechter Nachfolgerknoten einen Infowert enthält, der größer als `K^.info` ist).

(Der Name Max-Heap geht auf die Eigenschaft zurück, dass für jeden Teilbaum das Maximum in der Wurzel steht.)

Gesucht ist eine Funktion zur Erkennung, ob ein gegebener Binärbaum ein Max-Heap ist oder nicht.

Die Funktion muss also den Baum durchlaufen, bis entweder ein Gegenbeispiel (Knoten mit „größerem Kind“) gefunden wurde – dann ist der Baum *kein* Max-Heap – oder bis der Baum vollständig durchlaufen wurde, ohne ein Gegenbeispiel zu finden – dann *ist* der Baum ein Max-Heap.

a) Für einen Baumdurchlauf gibt es bekanntlich verschiedene mögliche Durchlaufreihenfolgen, insbesondere die Hauptreihenfolge (preorder) und die symmetrische Reihenfolge (inorder). Welche dieser beiden Durchlaufreihenfolgen ist für diese Aufgabe vorzuziehen? Begründen Sie Ihre Antwort! (Eine Antwort ohne Begründung wird nicht bewertet.)

b) Implementieren Sie nun eine *rekursive* Funktion, die oben beschriebene Aufgabe erfüllt. Auf der folgenden Seite geben wir bereits den Anfang der Funktion vor, führen Sie sie geeignet fort.
(Die Wahl der Durchlaufreihenfolge wird in Teil b) nicht erneut bewertet. Wenn Sie Teil a) nicht bearbeitet haben, wählen Sie irgendeine der beiden Durchlaufreihenfolgen.)

1. Der Begriff „Heap“ wird auch noch in anderen Bedeutungen verwendet, bindend für diese Aufgabe ist daher explizit die Begriffserklärung in der Aufgabenstellung.

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Name: _____ Matrikelnummer: _____

```
function istMaxHeap(inRefWurzel: tRefKnoten): Boolean;
{ Gibt true zurück, genau dann wenn der Baum ein Max-Heap ist. }
var linksGroesser, rechtsGroesser: Boolean;
begin
  if inRefWurzel = nil then
    {Der leere Baum ist ein (leerer) Max-Heap, da er keinen Knoten
     mit größerem Kind enthält.}
    istMaxHeap := true
  else
  begin
    {Untersuchung des Wurzelknotens und seiner Kinder}
    if inRefWurzel^.links = nil then
      linksGroesser := false
    else
      linksGroesser := inRefWurzel^.links^.info > inRefWurzel^.info;
    if inRefWurzel^.rechts = nil then
      rechtsGroesser := false
    else
      rechtsGroesser := inRefWurzel^.rechts^.info > inRefWurzel^.info;

    {Führen Sie hier die Funktion geeignet weiter!}
```

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Aufgabe 5 Black-Box-Test (2+1+2 Punkte)

Wir betrachten eine Firma, die ihren Mitarbeitern je nach Beschäftigungszeit ein unterschiedlich hohes Weihnachtsgeld auszahlt: Mitarbeiter, die weniger als drei Jahre bei der Firma beschäftigt sind, erhalten ein Weihnachtsgeld in Höhe von 50% ihres Monatslohns. Ab dem dritten Beschäftigungsjahr erhält ein Mitarbeiter 75%, ab dem sechsten Jahr volle 100% seines Monatslohns als Weihnachtsgeld.

Zur Berechnung des Weihnachtsgelds wurde eine Pascal-Funktion geschrieben. Deren Funktionskopf sowie die benötigten Konstanten- und Typdefinitionen sind wie folgt gegeben:

```
const FEHLERCODE = -1.0;
type  tJahre = 0..maxint;

function Weihnachtsgeld ( inBeschaeftigungszeit: tJahre;
                          inMonatslohn: real;
                        ): real;
```

Die Funktion `Weihnachtsgeld` soll bei Eingabe der Beschäftigungszeit sowie des Monatslohns eines Mitarbeiters die Höhe des an diesen Mitarbeiter auszuzahlenden Weihnachtsgelds ausgeben, also das Produkt des eingegebenen Monatslohns und des von der eingegebenen Beschäftigungszeit abhängigen Faktors (s.o.).

Dass keine negativen Zahlen als Beschäftigungszeit eingegeben werden können, haben die Entwickler mittels des Ausschnittsdatentyps `tJahre` sichergestellt. Eine entsprechende Maßnahme für den Monatslohn war leider nicht möglich, da Pascal für Fließkommazahlen keine Ausschnittsdatentypen erlaubt. Für den Parameter `inMonatslohn` wurde vielmehr der Typ `real` gewählt, welcher auch negative Eingaben zulässt. Bei Eingabe eines negativen Monatslohns soll der Wert `FEHLERCODE` (an Stelle des oben beschriebenen Produkts) zurückgegeben werden, da negative Löhne als ungültige Eingaben angesehen werden. Unentgeltiche Arbeit (d.h. ein Monatslohn von 0.0) sei aber zulässig.

Es sollen nun Testfälle für einen funktionsorientierten Test der Funktion `Weihnachtsgeld` gebildet werden.

a) Beim Betrachten der oben spezifizierten Funktionalität liegt die Idee nahe, dass zumindest vier Fälle zu testen sind: Ungültige Eingaben sowie Eingaben für jede der drei „Weihnachtsgeld-Stufen“. Präzisieren Sie diese Idee, indem Sie zunächst entsprechend vier Eingabeäquivalenzklassen bilden.

b) Die Eingabeäquivalenzklassen können nun zu den gesuchten vier Testfällen vervollständigt werden. Nennen Sie stellvertretend einen der vier Testfälle.

c) Könnte man dieselben vier Testfälle auch mit Hilfe von vier Ausgabeäquivalenzklassen (statt Eingabeäquivalenzklassen) herleiten?
Begründen Sie Ihre Antwort! (Eine unbegründete Antwort wird nicht bewertet.)

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Name: _____ Matrikelnummer: _____

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

Gegeben seien folgende Vereinbarungen:

```
    const
      FELDMAX = 4;

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

1   function enthaeltNull(inFeld: tFeld): boolean;
    { Gibt genau dann true zurueck, wenn inFeld eine 0 enthält. }
2   var gefunden: boolean;
3       i: integer;
4   begin
5     gefunden := false;
6     i := 0;
7     repeat
8       i := i + 1;
9       if (inFeld[i] = 0) then
10        gefunden := true;
11    until gefunden or (i = FELDMAX);
12    enthaeltNull := not gefunden;
13  end;
```

a) Angestrebt wird ein Boundary-Interior-Pfadtest. Da die Schleife maximal `FELDMAX` Durchläufe absolvieren kann und sich erfahrungsgemäß bei Schleifenabbrüchen nach einer maximalen Durchlaufzahl leicht Fehler einschleichen, soll neben einem „normalen“ Interior-Test mit weniger als `FELDMAX` Schleifendurchläufen (wir wählen hier genau zwei Durchläufe) noch ein zweiter Interior-Test mit genau `FELDMAX` (hier also vier) Schleifendurchläufen absolviert werden.
Nennen Sie – sofern möglich – für den keinmaligen, einmaligen, zweimaligen und viermaligen Schleifendurchlauf jeweils ein Testdatum.
Beim Formulieren der Testdaten stellen Sie eine Ausprägung eines Arrays vom Typ `tFeld` bitte als ein Quadrupel in eckigen Klammern dar. [2, 0, 4, 5] stehe beispielsweise für ein Array a mit a[1] = 2, a[2] = 0, a[3] = 4 und a[4] = 5.

b) Umfasst ein beliebiger Boundary-Interior-Pfadtest[1] für die Funktion `enthaeltNull` zwangsläufig eine vollständige einfache Bedingungsüberdeckung? Begründen Sie Ihre Antwort!

1. d.h. für beliebige Testdaten und beliebige Schleifendurchlaufzahl beim Interior-Test

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

Name: _____ Matrikelnummer: _____

Kurs 1613 „Einführung in die imperative Programmierung“

Hauptklausur am 06.02.2010

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.
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. Pascal-Funktionen 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.