FernUniversität in Hagen

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

Postanschrift: FernUniversität D - 58084 Hagen

Name, Vorname

Straße, Nr.

PLZ, Wohnort

INF

FernUniversität in Hagen
Lehrgebiet Software Engineering
D-58084 Hagen

Fachbereich Informatik

Kurs: 1612 "Konzepte imperativer Programmierung"

Hauptklausur am 4. März 2006, $10^{\underline{00}}$ bis $13^{\underline{00}}$ Uhr

Hörerstatus:

( ) Vollzeitstudent
( ) Teilzeitstudent
( ) Zweithörer
( ) Akademiestudent
( ) . . . . . .

Klausurort:

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

Aufgabe	1	2	3	4	5	6	7	Summe
erreichbare Punktzahl	6	7	5	6	6	10	8	48
bearbeitet								
erreichte Punktzahl								

Datum: _____ Korrektur: _____

© 2006 FernUniversität in Hagen

FernUniversität in Hagen

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

Postanschrift: FernUniversität D - 58084 Hagen

Name, Vorname

Straße, Nr.

PLZ, Wohnort

INF

FernUniversität in Hagen
Lehrgebiet Software Engineering
D-58084 Hagen

Fachbereich Informatik

Kurs: 1612 "Konzepte imperativer Programmierung"

Hauptklausur am 4. März 2006, $10^{\underline{00}}$ bis $13^{\underline{00}}$ Uhr

Hörerstatus:

( ) Vollzeitstudent
( ) Teilzeitstudent
( ) Zweithörer
( ) Akademiestudent
( ) . . . . . .

Klausurort:

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

Aufgabe	1	2	3	4	5	6	7	Summe
erreichbare Punktzahl	6	7	5	6	6	10	8	48
bearbeitet								
erreichte Punktzahl								

Datum: _____ Korrektur: _____

© 2006 FernUniversität in Hagen

FernUniversität in Hagen

FACHBEREICH
Informatik
Lehrgebiet
Software Engineering
Prof. Dr. H.-W. Six

FernUniversität in Hagen · 58084 Hagen

Ihr Zeichen
Ihre Nachricht vom
Mein Zeichen
Auskunft erteilt I. Schulz-Gerlach
Telefon 02331 987-2553
Telefax 02331 987-317
E-Mail Kurs1612@FernUni-Hagen.de
Hausanschrift Universitätsstr. 1
58084 Hagen
Datum 04.03.2006

BESCHEINIGUNG
(zur Vorlage beim Finanzamt)

Hiermit wird bescheinigt, dass

Herr / Frau _____

Matrikel-Nr. _____

am 4. März 2006 in der Zeit von 10.00 bis 13.00 Uhr

in _____

an der Klausur zum Kurs 1612 “Konzepte imperativer Programmierung” teilgenommen hat.

Telefonzentrale 02331 987-01
Zentraler Telefaxeingang 02331 987-316
Internet http://www.FernUni-Hagen.de
Buslinien 515
Haltestelle FernUniversität

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Wintersemester 2005/2006
Hinweise zur Bearbeitung der Klausur
zum Kurs 1612 “Konzepte imperativer Programmierung”

Wir begrüßen Sie zur Klausur "Konzepte imperativer 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,
   - 7 Aufgaben (Seite 2 - Seite 26),
   - die Muss-Regeln des Programmierstils,
   - die Hoare-Regeln,
   - die Definition der Terminierungsfunktion.
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, die Tabelle mit den im Kurs verwendeten Hoare-Regeln und die Definition der Terminierungsfunktion finden Sie im Anschluss an die Aufgabenstellung.
6. Es sind maximal 48 Punkte erreichbar. Sie haben die Klausur sicher dann bestanden, wenn Sie mindestens 24 Punkte erreicht haben.

Wir wünschen Ihnen bei der Bearbeitung der Klausur viel Erfolg!

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Aufgabe 1 (5 + 1 Punkte)

a) Schreiben Sie eine PASCAL-Prozedur `rekMax`, die in einem Feld von `integer`-Zahlen das Maximum bestimmt. Der Prozedur liegt ein 'divide and conquer'-Algorithmus zugrunde, d.h. wir teilen das Feld in zwei (möglichst gleichgroße) Hälften und bestimmen in beiden Hälften rekursiv das Maximum. Aus dem Vergleich der zwei Teilmaxima ergibt sich das Maximum des ganzen Feldes. Überlegen Sie zunächst, wann das Verfahren abbrechen soll.

b) Erläutern Sie, ob es sich um eine sinnvolle Anwendung der Rekursion handelt.

Vorgaben zur Lösung:

Betrachten Sie die folgenden Konstanten- und Typdefinitionen sowie den Prozedurkopf von `rekMax` als gegeben. Die Parameter `inVon` und `inBis` geben den zu untersuchenden Ausschnitt des Feldes `inFeld` an. Es muss bei Aufruf der Prozedur stets `inVon` $\leq$ `inBis` gelten. In `outMax` wird schließlich das Maximum der Feldelemente des angegebenen Ausschnitts zurückgegeben.

Der Aufruf der Prozedur `rekMax` im Hauptprogramm lautet folglich `rekMax(UNTEN, OBEN, Feld, maxi)`, wenn `Feld` das zu untersuchende Feld angibt und `maxi` eine Variable für das gesuchte Maximum ist.

```
const
UNTEN = 0;
OBEN = 10;

type
tIndex = UNTEN..OBEN;
tFeld = array [tIndex] of integer;

procedure rekMax (
                  inVon,
                  inBis : tIndex;
              var inFeld : tFeld;
              var outMax : integer);
{ bestimmt rekursiv den maximalen Feldeintrag von inFeld
  ueber ein 'divide and conquer'-Verfahren }
```

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Aufgabe 2 (3 + 4 Punkte)

Gegeben sei folgende Typdeklaration für Elemente einer dynamischen Liste:

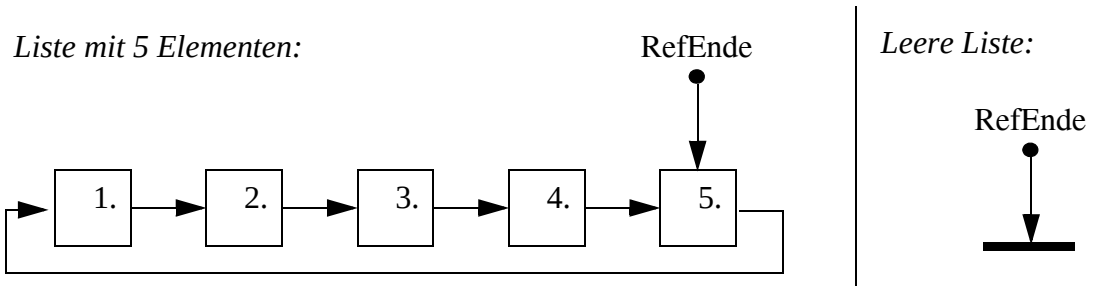
```
type
tRefElement = ^tElement;
tElement =  record
              info: integer;
              next: tRefElement
            end;
```

Damit soll nun eine *zyklische* lineare Liste realisiert werden, indem der next-Zeiger des letzten Elementes nicht auf NIL, sondern auf das erste Element gesetzt wird. Es sei nach wie vor ein ganz bestimmtes Element des so entstehenden Ringes das erste Element, dessen Vorgänger ist dann das letzte Element.

Eine solche Liste soll nicht über einen Anfangs-Zeiger, sondern über einen Ende-Zeiger auf das letzte Element angesprochen werden. Das hat den Vorteil, dass man sowohl das letzte Element als auch das erste Element (Nachfolger des letzten) ohne eine Suche in der Liste direkt ansprechen kann[1].

Natürlich kann auch eine solche zyklische Liste leer sein, dargestellt durch einen NIL-Zeiger.

Die folgende Skizze stellt eine solche Liste mit 5 Elementen sowie eine leere Liste dar, RefEnde ist der Zeiger, über den die gesamte Liste referenziert wird:



Tipp: Hilfreich für die Bearbeitung der Aufgabe könnte es sein, sich vorher auch die einelementige Liste entsprechend zu skizzieren.

a) Implementieren Sie eine Prozedur `appendNotEmpty`, die in eine solche Liste ein neues Element am Ende (also als neues letztes Element hinter dem bisherigen letzten) einfügt. Dabei dürfen Sie davon ausgehen, dass die Liste nicht leer ist.
Benutzen Sie folgenden Prozedurkopf, ergänzen Sie dabei an den mit **??** gekennzeichneten Stellen die Übergabeart.

```
procedure appendNotEmpty(??Zahl:integer; ??RefEnde: tRefElement);
{fügt ??Zahl am Ende der nicht-leeren Liste ein.}
```

1. Mit dieser Datenstruktur könnte man z.B. eine Schlange (first in, first out) realisieren, bei der zum Einfügen weder das Listenende gesucht werden muss noch die Notwendigkeit besteht, zwei Zeiger (für Listenanfang und für Listenende) zu verwalten.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

---

b) Implementieren Sie eine Funktion, die in einer solchen Liste nach einem Element mit einem zu übergebenden Integer-Wert sucht. Als Ergebnis soll eine Referenz auf das gefundene Element zurückgegeben werden, bzw. NIL, falls kein solches Element existiert.

Berücksichtigen Sie, dass die Liste auch leer sein kann.

Benutzen Sie folgenden Funktionskopf:

```
function find(  inSuchwert:integer;
                inRefEnde: tRefElement):tRefElement;
{Sucht ein Element mit Info inSuchwert in der in inRefEnde
übergebenen Liste. Rückgabe von nil, falls nicht gefunden, sonst
Rückgabe einer Referenz auf das gefundene Element.}
```

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

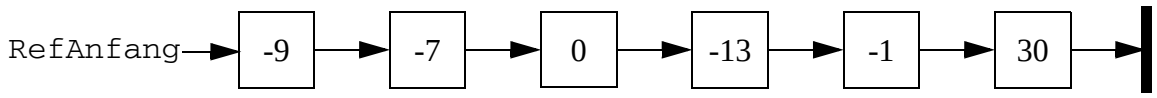
Aufgabe 3 (5 Punkte)

Gegeben ist folgende Definition einer linearen Liste:

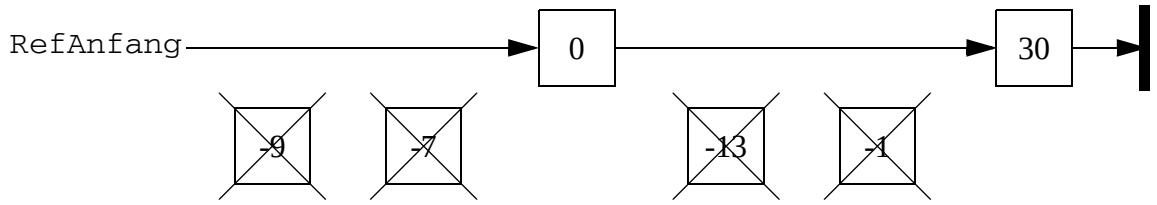
```
type
tRefElement = ^tElement;
tElement = record
           info : integer;
           next : tRefElement
         end;
```

Implementieren Sie eine Prozedur `LoescheNegative`, die alle Zahlen echt kleiner Null aus einer Liste mit einem gegebenen Anfangszeiger vom Typ `tRefElement` löscht.

Ein Beispiel:
Vor dem Aufruf:



Danach:



Zur Vereinfachung dürfen Sie folgende Annahmen voraussetzen:

- Die Liste ist nicht leer.
- Die Liste enthält mindestens ein nicht-negatives Element.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

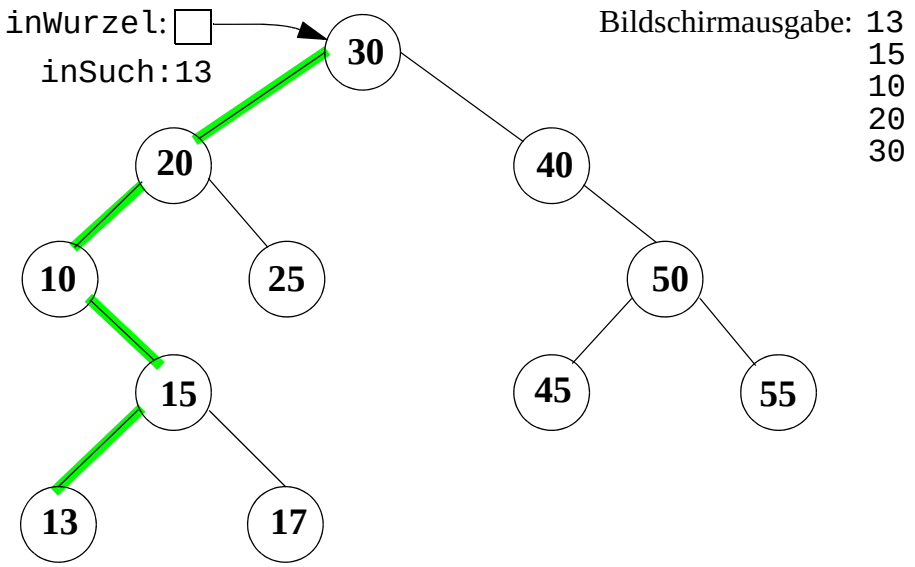
Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Aufgabe 4 (4 + 2 Punkte)

a) Schreiben Sie eine **rekursive** PASCAL-Prozedur `BaumPfad`, die als Eingabe einen Zeiger auf die Wurzel eines binären Suchbaums mit `integer`-Werten sowie einen zu suchenden Wert erhält. Als Ausgabe sollen die gefundenen Werte auf dem Suchpfad in umgekehrter Reihenfolge auf dem Bildschirm ausgegeben werden. Ist der gesuchte Wert nicht im Baum enthalten, so sollen die Werte trotzdem ausgegeben werden. Durchlaufen Sie dazu in Ihrer Prozedur den Suchbaum **rekursiv** anhand des Suchwertes `inSuch` und geben Sie in der Prozedur die gefundenen Werte geeignet aus, wie im folgenden Bild skizziert ist.



Verwenden Sie die unten angegebenen Typdefinitionen sowie den Prozedurkopf von `BaumPfad`!

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

procedure Baumpfad (
            inWurzel : tRefBinBaum;
            inSuch : integer);
{ sucht rekursiv im Suchbaum, auf dessen Wurzel
  inWurzel zeigt, den Wert inSuch und gibt die Werte
  der besuchten Knoten in umgekehrter Reihenfolge auf
  dem Bildschirm aus }
```

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

---

b) Erläutern Sie, ob in diesem Fall die Anwendung der Rekursion sinnvoll ist.
(Eine Ja- oder Nein-Antwort ohne Erläuterung wird nicht bewertet!)

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Aufgabe 5 (6 Punkte)

Gegeben seien folgende Vereinbarungen:

```
const
  FELDMAX=4;

type
  tFeld = Array[1..FELDMAX] of integer;
  tRueckgabe = (null, sortiert, unsortiert);

function pruefeFeld(inFeld:tFeld): tRueckgabe;
{ Prüft, ob inFeld keine Nullen enthält und aufsteigend sortiert
  ist.
  Rueckgabe:  null, falls das Feld mindestens eine Null enthält,
    sortiert, falls das Feld aufsteigend sortiert ist und keine Null
    enthält, unsortiert sonst}
var i:integer;
    isNull:boolean;

begin
  if inFeld[1]=0 then
    pruefeFeld := null
  else
  begin
    isNull := false;
    i := 2;
    while (i<FELDMAX) and
          ((inFeld[i]=0) or (inFeld[i-1]<=inFeld[i])) do
    begin
      if inFeld[i]=0 then
        isNull := true;
      i := i+1
    end; { while }

    if isNull then
      pruefeFeld := null
    else
      if inFeld[i]>inFeld[i-1] then
        pruefeFeld := unsortiert
      else
        pruefeFeld := sortiert;
  end { inFeld[1]<>0 }
end; { pruefeFeld }
```

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

Suchen Sie Testdaten für die Funktion `pruefeFeld`, mit denen eine *minimale Mehrfach-Bedingungsüberdeckung* erreicht wird.

Tragen Sie dazu in der folgenden Tabelle über einer Spalte jeweils ein Eingabedatum ein und werten Sie die vorkommenden atomaren und zusammengesetzen Prädikate für dieses Datum aus. Wählen Sie so lange neue Eingabedaten, bis alle Prädikate vollständig überdeckt sind.

Vervollständigen Sie außerdem jedes Eingabedatum zu einem Testdatum, indem Sie unten das erwartete Ergebnis zu jeder Eingabe angeben. Nennen Sie zusätzlich das tatsächliche Ergebnis zur Eingabe.

Prädikate		`inFeld`					
A)	`inFeld[1]=0`						
B)	`i<FELDMAX`						
C)	`inFeld[i]=0`						
D)	`inFeld[i-1]` `<= inFeld[i]`						
E)	C) **`or`** D)						
F)	B) **`and`** E) *(while-Bedingung)*						
G)	`inFeld[i]=0`						
H)	`isNull`						
I)	`inFeld[i]` `> inFeld[i-1]`						
erwartete Ausgabe							
tatsächliche Ausgabe							

<u>Ausfüllhinweise:</u>

- Um ein Eingabedatum von Typ `tFeld` zu notieren, tragen Sie in die vier Kästchen am Kopf einer Spalte die vier Zahlen ein, mit denen das Feld belegt sein soll.
- In den Zeilen A) bis I) sind pro Spalte folgende Eintragungen möglich:
  T: Das Prädikat wird unter der Eingabe (nur) zu true ausgewertet,
  F: Das Prädikat wird unter der Eingabe (nur) zu false ausgewertet,
  T, F: Das Prädikat wird unter der Eingabe mindestens einmal zu true und mindestens einmal zu false ausgewertet. (Das kann in einer Schleife der Fall sein.)
  —: Das Prädikat wird unter der Eingabe niemals ausgewertet.
- Sie müssen nicht alle sechs Spalten ausnutzen, wenn Sie schon mit weniger Testdaten eine minimale Mehrfach-Bedingungsüberdeckung erreichen.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Aufgabe 6 (10 Punkte)

Das Programm `Potenzieren` berechnet für einen „Eingabewert“ `H` den Wert von 2^H nur durch Addition mit 1, Subtraktion um 1, Zuweisung und Vergleich mit 0:

```
1.  program Potenzieren;
2.  (* berechnet x = 2^H *)

3.    const
4.    H = ..;



5.    var
6.    u,
7.    x,
8.    y : integer;

    { Vorbedingung: P ≡ H ≥ 0 }
9.  begin
10.   y := H;
11.   x := 1;
12.   u := 1;

13.   while y <> 0 do
14.     if u = 0 then
15.     begin
16.       u := x;
17.       y := y - 1
18.     end
19.     else
20.     begin
21.       x := x + 1;
22.       u := u - 1
23.     end

24. end. (* Potenzieren *)
    { Nachbedingung: Q ≡ x = 2^H }
```

Es bezeichne $S_{i,j}$ die Programmzeilen von i bis j des Programms `Potenzieren`. Beweisen Sie die partielle Korrektheit des Programms, indem Sie die Teilaufgaben a) bis c) lösen.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

---

a) Zeigen Sie, dass die Programmformel

$\{ R \wedge y \neq 0 \}$

$S_{14,\,23}$

$\{ R \equiv x + u = 2^{H-y+1} \wedge (y \neq 0 \vee x = u) \}$

eine gültige Programmformel ist. Geben Sie dabei die jeweils verwendeten Hoare-Regeln an. Um den Beweis nicht zu lang werden zu lassen, können Sie davon ausgehen, dass die Programmformel $\{R \wedge y \neq 0 \wedge u \neq 0\} S_{20,\,23} \{R\}$ gültig ist.

b) Warum können Sie aus der Gültigkeit der Programmformel aus Teilaufgabe a) folgern, dass

$\{ R \}$

$S_{13,\,23}$

$\{ R \wedge y = 0 \}$

eine gültige Programmformel ist?

c) Zeigen Sie nun die partielle Korrektheit des Programms `Potenzieren`. Geben Sie auch hier die jeweils verwendeten Hoare-Regeln an. Setzen Sie die Gültigkeit der Programmformel unter b) voraus.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Aufgabe 7 (8 Punkte)

Beantworten Sie die folgenden Fragen und begründen Sie jeweils Ihre Antworten!
(Eine nicht begründete Antwort wie nur „Ja“ oder „Nein“ wird nicht bewertet.)

a) Gegeben sei eine konkrete terminierende While-Schleife `S` mit zugehöriger Terminierungsfunktion τ. Laut Bedingung w3 aus der Definition einer Terminierungsfunktion gibt es demnach zu τ eine untere Schranke $t^* \in \mathbb{Z}$. Ist dann auch t^*-1 eine solche untere Schranke, d.h. ist t^* nicht eindeutig bestimmt?

b) Nehmen Sie Stellung zu folgender Behauptung:
„Um die partielle Korrektheit eines Programms `S'` bezüglich der Spezifikation `{P} S {Q}` nachzuweisen, muss die Gültigkeit der Programmformel `{P} S' {Q}` gezeigt werden.“

c) Sei die Programmformel `{P} S' {Q}` gültig. Ist so sichergestellt, dass das Programm `S'` für jede P-konforme Eingabe[1] eine Q-konforme Ausgabe produzieren wird?

d) Ist für die Ermittlung von Testfällen zu strukturorientierten Tests außer einer vorliegenden Implementierung des Prüflings auch seine Spezifikation erforderlich?

1. Mit „P-konformer Eingabe“ ist dabei eine Eingabe gemeint, für die das Prädikat P wahr wird, „Q-konforme Ausgabe“ ist entsprechend zu lesen.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 04.03.2006

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. 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.

Hoare-Regeln

1. Konsequenzregel 1

$$\frac{\{P\}\ S\ \{R\},\ R \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Konsequenzregel 2

$$\frac{P \Rightarrow R,\ \{R\}\ S\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

2. Nullaxiom

$$\{P\}\ \{P\}$$

3. Zuweisungaxiom

$$\{P\langle x \leftarrow \texttt{expr}\rangle\}\ \texttt{x:=expr}\ \{P\}$$

4. Sequenzregel

$$\frac{\{P\}\ S_1\ \{R\},\ \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1;\ S_2\ \{Q\}}$$

5. Zusammensetzungsregel

$$\frac{\{P\}\ S\ \{Q\}}{\{P\}\ \textbf{begin}\ S\ \textbf{end}\ \{Q\}}$$

6. Bedingungsregel 1

$$\frac{\{P \wedge B\}\ S_1\ \{Q\},\ \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

Bedingungsregel 2

$$\frac{\{P \wedge B\}\ S\ \{Q\},\ (P \wedge \neg B) \Rightarrow Q}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \{Q\}}$$

7. **while**-Regel

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{P \wedge \neg B\}}$$

8. **repeat**-Regel

$$\frac{\{P\}\ S\ \{Q\},\ (Q \wedge \neg B) \Rightarrow P}{\{P\}\ \textbf{repeat}\ S\ \textbf{until}\ B\ \{Q \wedge B\}}$$

Terminierungsfunktion

Definition

Eine Abbildung τ heißt *Terminierungsfunktion* der **`while`**-Schleife **`while`** B **`do`** S, wenn sie die Bedingungen w1) bis w3) erfüllt:

w1) Seien $v_1, ..., v_r$ die in B und S vorkommenden Variablen und Konstanten. Dann ist $\tau: \mathbb{Z}^r \rightarrow \mathbb{Z}$ mit $\tau(v_1, ..., v_r) \in \mathbb{Z}$.

w2) Ist vor einem Schleifendurchlauf $\tau(v_1, ..., v_r) = t$ erfüllt, dann ist nach dem Schleifendurchlauf $\tau(v_1, ..., v_r) < t$ erfüllt.

w3) Es existiert ein Wert $t^* \in \mathbb{Z}$, so dass vor jedem Schleifendurchlauf $\tau(v_1, ..., v_r) \geq t^*$ gilt.

Bemerkung

i) Beachten Sie, dass S nur dann ausgeführt wird, wenn INV $\wedge$ B erfüllt ist. Diese Voraussetzung muss man gewöhnlich ausnutzen, um die Eigenschaften w2) und w3) einer Terminierungsfunktion beweisen zu können.

ii) Eine Invariante INV, mit der sich die Terminierung einer Schleife beweisen lässt, ist (leider) oft nicht identisch mit einer pk-geeigneten Invarianten, mit der sich die partielle Korrektheit zeigen lässt.
Haben wir eine solche t-geeignete Invariante INV gefunden und bewiesen, dass die Schleife unter dieser Invarianten terminiert, müssen wir außerdem noch zeigen, dass INV auch tatsächlich erfüllt ist, wenn die Programmabarbeitung die Schleife erreicht.