Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Wintersemester 2002/2003
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,
   - 8 Aufgaben (Seite 2 - Seite 32),
   - 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 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, 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 59 Punkte erreichbar. Sie haben die Klausur sicher dann bestanden, wenn Sie mindestens 29Punkte erreicht haben.

Wir wünschen Ihnen bei der Bearbeitung der Klausur viel Erfolg!

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Aufgabe 1 (1 + 2 + 4 Punkte)

Eine quadratische Matrix $(a_{ij})_{i,j:1..N}$ heißt *halblateinisches Quadrat der Ordnung N*, wenn in jeder Zeile jeder der Werte 1..N vorkommt. Ein Beispiel für ein halblateinisches Quadrat der Ordnung 4:

4	3	1	2
3	4	2	1
2	1	3	4
1	2	4	3

Es soll eine Funktion `IstHalb` entwickelt werden, die testet, ob eine übergebene Matrix ein halblateinisches Quadrat ist. Bearbeiten Sie dazu die Aufgabenteile a) bis c) und benutzen Sie die vorgegebenen Typdefinitionen.

```
const GRAD=?? ;
type  tBereich = 1..GRAD;
      tMatrix = array [tBereich, tBereich] of tBereich;
      tBoolFeld = array [tBereich] of boolean;
```

a) Schreiben Sie eine Prozedur `InitBoolFeld`, die ein übergebenes Feld vom Typ `tBoolFeld` mit einem ebenfalls als Parameter übergebenen boolschen Wert initialisiert. Verwenden Sie den folgenden Prozedurkopf:

```
procedure InitBoolFeld(inWert:boolean; var ioFeld: tBoolFeld);
```

b) Schreiben Sie eine Funktion `TesteBoolFeld`, die `true` zurückliefert, wenn alle Elemente des übergebenen Felds `true` sind. Benutzen Sie folgenden Funktionskopf:

```
function TesteBoolFeld(inFeld: tBoolFeld):boolean;
```

c) Schreiben Sie unter Benutzung der Prozedur und Funktion aus a) und b) eine Funktion `IstHalb`, die testet, ob eine übergebene Matrix ein halblateinisches Quadrat ist. Sie können die Prozedur und Funktion aus Aufgabenteil a) und b) als bekannt voraussetzen, auch wenn Sie die Aufgabenteile nicht bearbeitet haben. Benutzen Sie folgenden Funktionskopf:

```
function IstHalb(inMatrix:tMaxtrix):boolean;
```

Anleitung: Für jede Zeile muss kontrolliert werden, ob alle Werte 1..GRAD vorkommen. Benutzen Sie eine Variable vom Typ `tBoolFeld` (z.B. `boolFeld`), um nachzuhalten, welche Werte

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

---

in der aktuell betrachteten Zeile vorhanden sind. `boolFeld[i]=true` bedeutet, dass der Wert i vorkommt. Mit `TesteBoolFeld` kann dann überprüft werden, ob alle Elemente von `boolFeld` true sind.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Aufgabe 2 (6 Punkte)

Gegeben ist folgende Definition einer linearen Liste:

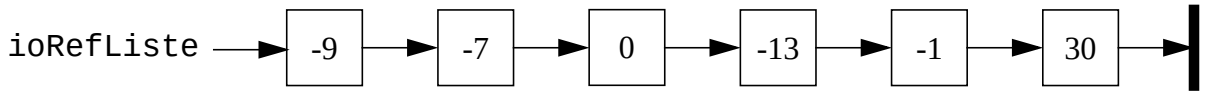
```
type
tRefListe = ^tListe;
tListe = record
           info : integer;
           next : tRefListe
         end;
```

Implementieren Sie eine Prozedur `Tausche`, die in einer übergebenen Liste das Element mit info-Komponente 0 sucht und mit dem Nachfolger nur durch Ändern der Verkettung vertauscht. Sie können davon ausgehen, dass die 0 in der Liste genau einmal, aber nicht als letztes Element vorkommt.

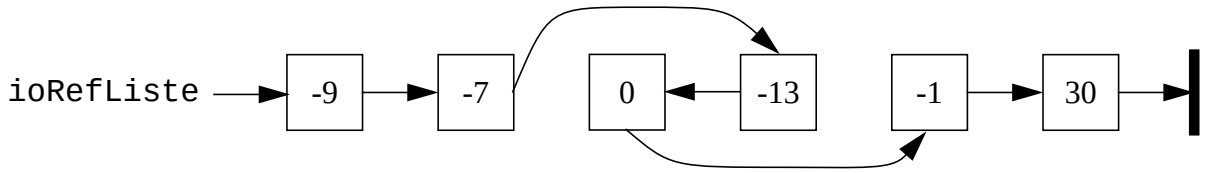
Verwenden Sie folgenden Prozedurkopf:

```
procedure Tausche(var ioRefListe : tRefListe);
```

Ein Beispiel:
Vor dem Aufruf:



Danach:



Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Aufgabe 3 (8 Punkte)

Gegeben ist folgende Definition einer linearen Liste:

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

Es soll eine Prozedur `FuegeEin` implementiert werden, die einen Wert in eine aufsteigend sortierte Liste einfügt, so dass die Sortierung erhalten bleibt. Betrachten Sie dazu folgende lückenhafte Prozedur:

```
procedure FuegeEin(inWert: integer; var ioRefAnfang: tRefListe);

var lauf, elem: tRefListe;
          ende: boolean;

begin
   new(elem);
   elem^.info := inWert;
   if ioRefAnfang = NIL then                  (* Leere Liste *)
   begin
     (1);
     (2)
   end
   else
     if inWert < ioRefAnfang^.info then (* Sonderfall: Einfuegen am
                                                 Anfang *)
     begin
       (3);
       (4)
     end
     else (* Liste nicht leer und nicht am Anfang einfuegen *)
     begin
       ende := false;
       lauf := ioRefAnfang;
       while not ende and (lauf^.next<>NIL) do
         if lauf^.next^.info > inWert then
           (5)
         else
           (6);
       (* Ende der while-Schleife *)
```

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

```
      (7); (* Element einfuegen *)
      (8)
   end
end;
```

Für jeden der Platzhalter (1), (2), (3), (4), (5) ,(6), (7) und (8) ist genau eine Anweisung so anzugeben, dass die Prozedur `FuegeEin` ihre Aufgabe korrekt erfüllt.

Platzhalter	Anweisung
(1)	
(2)	
(3)	
(4)	
(5)	
(6)	
(7)	
(8)	

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

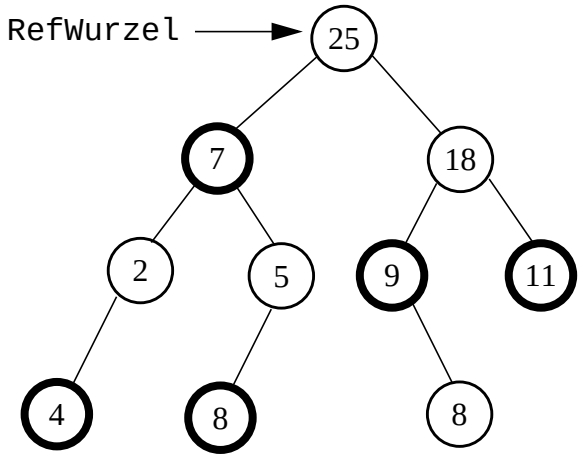
Aufgabe 4 (8 Punkte)

Gegeben sei ein binärer Baum, der folgender Typdefinition genügt:

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

Schreiben Sie eine rekursive Prozedur `HRZwei`, die den Knotenwert von jedem zweiten Knoten in Hauptreihenfolge (preorder-Durchlauf) auf dem Bildschirm ausgibt.

Beispiel:




```
Die Prozedur soll die Zahlen 7,4,8,9,11
in dieser Reihenfolge ausgeben.
```

Verwenden Sie folgenden Prozedurkopf:

```
procedure HRZwei(inRefWurzel:tRefBinBaum; var ioDrucken:boolean);
```

<u>Hinweis</u>: Mit Hilfe des boolschen Parameters `ioDrucken` wird auf jeder Rekursionsebene gesteuert, ob der Knotenwert auszugeben ist oder nicht. Die Prozedur wird das erste Mal mit `ioDrucken=false` aufgerufen.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

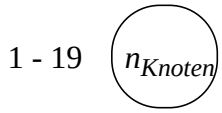
Aufgabe 5 (2 + 4 + 1 Punkte)

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

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

 1   function MaxZiffer (inZahl:tNatZahl): tZiffer;
 2   { bestimmt die größte Ziffer von inZahl}

 3     var
 4     rest : tNatZahl;
 5     maxi,
 6     ziffer: tZiffer;

 7   begin
 8     rest := inZahl;
 9     maxi := 2;
10     while rest > 0 do
11     begin
12       ziffer := rest mod 10;
13       if ziffer > maxi then
14         maxi := ziffer;
15       rest := rest div 10
16     end;
17     MaxZiffer := maxi
18   end; { MaxZiffer }
```

Ihre Aufgabe ist es, einen boundary-interior Test der Funktion `MaxZiffer` durchzuführen, indem Sie die Teilaufgaben a) bis c) bearbeiten.

a) Erstellen Sie den kompakten Kontrollflußgraphen für `MaxZiffer`. Geben Sie dabei zu jedem Knoten an, welche Programmzeilen von diesem Knoten repräsentiert werden. Das soll in folgender Form geschehen:



b) Geben Sie jeweils einen Pfad mit einem zugehörigen Testdatum für den keinmaligen, einmaligen und zweimaligen Schleifendurchlauf an.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

---

c) Kann man den Fehler im Programm mit dem boundary-interior Test in jedem Fall, d.h. unabhängig von der Wahl der Testdaten, finden? Begründen Sie Ihre Antwort!

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Aufgabe 6 (6 + 3 Punkte)

Das folgende Programm `Aufgabe6` berechnet die Fakultät von `ZAHL`. `MAXINT`-Bereichsüberläufe sollen nicht berücksichtigt werden.

Zeigen Sie, dass das Programm `Aufgabe6` bezüglich der Spezifikation
`{P ≡ (ZAHL≥0)} Fakul {Q ≡ (fak = ZAHL!)}` partiell korrekt ist, indem Sie die Aufgabenteile a) und b) bearbeiten.

```
program Aufgabe6;

  type
  tNatZahl = 0..MAXINT;

  const
  ZAHL = ?;

  var
  fak,
  i : tNatZahl;

  begin
    fak := 1;
    i := 1;
    while ZAHL ≥ i do
    begin
      fak := fak * i;
      i := i + 1
    end
  end.
```

a) Zeigen Sie, dass

$$INV \equiv (fak=(i-1)!) \wedge (ZAHL \geq (i-1))$$

eine Invariante der while-Schleife des Programms `Aufgabe6` ist. Geben Sie die verwendeten Hoare-Regeln explizit an.

b) Benutzen Sie die Invariante `INV` zum Nachweis der partiellen Korrektheit des Programms `Aufgabe6`. Geben Sie auch hier die verwendeten Hoare-Regeln explizit an.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

Aufgabe 7 (5 Punkte)

Zeigen Sie, dass das Programm `Aufgabe6` terminiert, indem Sie eine Funktion τ angeben und beweisen, dass τ eine Terminierungsfunktion der **while**-Schleife ist.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Aufgabe 8 (2+2+2+3 Punkte)

a) Terminiert ein partiell korrektes Pascal-Programm, das beliebige Anweisungen, aber keine Schleifen enthält, immer? Begründen Sie kurz Ihre Antwort.

b) Terminiert ein total korrektes Programm für jede mögliche Eingabe (also auch für Eingaben, die nicht der Programmspezifikation entsprechen)? Begründen Sie kurz Ihre Antwort.

c) Welche (formalen) Voraussetzungen müssen mindestens erfüllt sein, damit ein Programm, geschrieben in irgendeiner Programmiersprache, verifiziert werden kann?

d) Gegeben sei eine while-Schleife

```
while B do
  S_while
```

Mit Hilfe einer aussagekräftigen Invariante INV konnte die Gültigkeit folgender Programmformel bewiesen werden:

$$\{INV \wedge B\}\ S_{while}\ \{INV \wedge B\}$$

Wie oft wird die while-Schleife durchlaufen?

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 01.03.2003

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

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

Konsequenzregel 2

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

2. Nullaxiom

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

3. Zuweisungaxiom

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

4. Sequenzregel

$$\frac{\{P\}\ \mathtt{S}_1\ \{R\},\ \{R\}\ \mathtt{S}_2\ \{Q\}}{\{P\}\ \ \mathtt{S}_1;\ \mathtt{S}_2\ \ \{Q\}}$$

5. Zusammensetzungsregel

$$\frac{\{P\}\ \mathtt{S}\ \{Q\}}{\{P\}\ \ \mathbf{begin}\ \mathtt{S}\ \mathbf{end}\ \ \{Q\}}$$

6. Bedingungsregel 1

$$\frac{\{P \wedge B\}\ \mathtt{S}_1\ \{Q\},\ \{P \wedge \neg B\}\ \mathtt{S}_2\ \{Q\}}{\{P\}\ \mathbf{if}\ B\ \mathbf{then}\ \mathtt{S}_1\ \mathbf{else}\ \mathtt{S}_2\ \{Q\}}$$

Bedingungsregel 2

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

7. **while**-Regel

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

8. **repeat**-Regel

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

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 $\tau: \mathbb{Z}^r \to \mathbb{Z}$ 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.

i) Eine Invariante INV, mit der sich die Terminierung einer Schleife beweisen läßt, ist (leider) oft nicht identisch mit einer geeigneten Invarianten, mit der sich die partielle Korrektheit zeigen läßt.
Haben wir eine solche 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

Kurs 1612 “Konzepte imperativer 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.

Kurs 1612 “Konzepte imperativer Programmierung”

Hoare-Regeln

---

1. Konsequenzregel 1

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

Konsequenzregel 2

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

2. Nullaxiom

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

3. Zuweisungaxiom

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

4. Sequenzregel

$$\frac{\{P\}\ \mathtt{S}_1\ \{R\},\ \{R\}\ \mathtt{S}_2\ \{Q\}}{\{P\}\ \ \mathtt{S}_1;\ \mathtt{S}_2\ \ \{Q\}}$$

5. Zusammensetzungsregel

$$\frac{\{P\}\ \mathtt{S}\ \{Q\}}{\{P\}\ \ \mathbf{begin}\ \mathtt{S}\ \mathbf{end}\ \ \{Q\}}$$

6. Bedingungsregel 1

$$\frac{\{P \wedge B\}\ \mathtt{S}_1\ \{Q\},\ \{P \wedge \neg B\}\ \mathtt{S}_2\ \{Q\}}{\{P\}\ \mathbf{if}\ B\ \mathbf{then}\ \mathtt{S}_1\ \mathbf{else}\ \mathtt{S}_2\ \{Q\}}$$

Bedingungsregel 2

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

7. **`while`**-Regel

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

8. **`repeat`**-Regel

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

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.

i) Eine Invariante INV, mit der sich die Terminierung einer Schleife beweisen läßt, ist (leider) oft nicht identisch mit einer geeigneten Invarianten, mit der sich die partielle Korrektheit zeigen läßt.
Haben wir eine solche 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