Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Wintersemester 2003/2004
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 34),
   - 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 64 Punkte erreichbar. Sie haben die Klausur sicher dann bestanden, wenn Sie mindestens 32 Punkte erreicht haben.

Wir wünschen Ihnen bei der Bearbeitung der Klausur viel Erfolg!

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Aufgabe 1 (7 Punkte)

Ein Handlungsreisender muß `MAXORTSZAHL` Städte besuchen (`MAXORTSZAHL > 1`) und möchte dafür eine Rundreise planen, während derer jede Stadt genau einmal besucht wird und die wieder in der Stadt endet, in der sie begonnen hat. Aus Sparsamkeits- und Umweltschutzgründen sollte die Reise möglichst kurz sein.

Schreiben Sie hierfür eine PASCAL-Prozedur `Rundreise`, welche als Eingabe eine Entfernungsmatrix erhält, eine Rundreise ermittelt und die Ortsnummern in der Reihenfolge der Rundreise sowie die insgesamt zu fahrenden Kilometer ausgibt. Dabei wird folgendes Verfahren benutzt: Ausgehend von der Stadt 1 wird jeweils zur nächstgelegenen Stadt gefahren, bis alle Städte besucht sind. Zum Schluß muß dann wieder zur Stadt 1 zurückgekehrt werden. Die Entfernungen zwischen den Städten sind in der übergebenen Matrix abgespeichert. Die Information, ob eine Stadt schon besucht ist, soll in einem booleschen Feld abgelegt werden.

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

Entfernungsmatrix:

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

Prozedurausgabe:

```
Rundreise:
1
3
2
1
17 km
```

Sie können davon ausgehen, daß die Länge der Rundreise und die einzelnen Entfernungen zwischen den Städten immer positive `integer`-Zahlen kleiner als `maxint` sind. Benutzen Sie die folgenden Konstanten- und Typvereinbarungen und ergänzen Sie die teilweise angegebene Prozedur:

```
const
MAXORTSZAHL = 10;

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

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

  type
  tBoolFeld = array [tOrtsIndex] of boolean;
```

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

```
    var
    besucht : tBoolFeld;
    Gesamtstrecke : tEntfernung;
    i,
    hier,
    naechste : tOrtsIndex;
    { ...setzen Sie hier evtl. weitere Deklarationen ein }

  begin
    { Initialisierung des Feldes besuchter Staedte }
    for i := 1 to MAXORTSZAHL do
      besucht [i] := false;
    { ... komplettieren Sie Rundreise ab hier! }
```

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Aufgabe 2 (6 Punkte)

Gegeben ist folgende Definition einer linearen Liste:

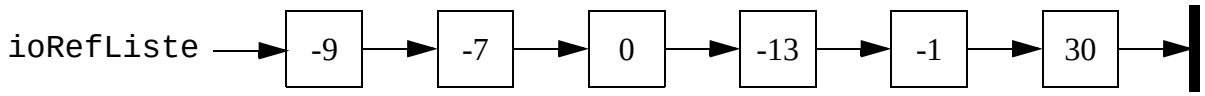
```
type
tRefListe = ^tListe;
tListe = record
           info : integer;
           next : tRefListe
         end;
```

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

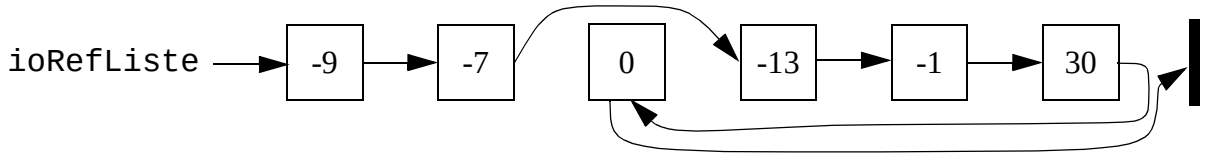
Verwenden Sie folgenden Prozedurkopf:

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

Ein Beispiel:
Vor dem Aufruf:



Danach:



Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

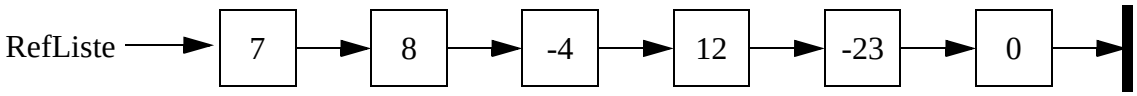
Klausur am 06.03.2004

Aufgabe 3 (8 Punkte)

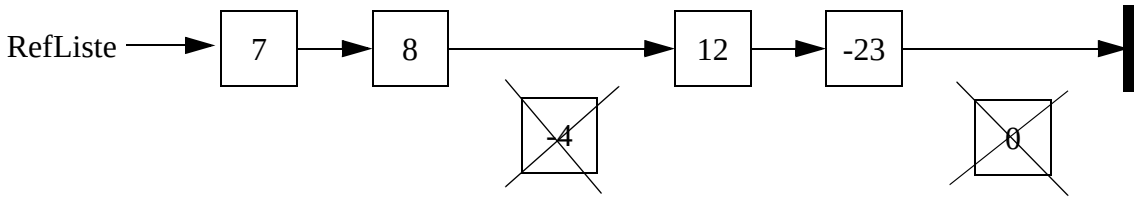
Gegeben sei eine einfach verkettete Liste mit den üblichen Typdefinitionen. Implementieren Sie eine Prozedur `DreifachPosLoeschen`, die als Parameter den Zeiger auf den Anfang der Liste erhält. Die Prozedur soll die Listenelemente an den Positionen drei, sechs, neun usw. löschen, falls die Liste entsprechend viele Elemente besitzt.

Beispiel:

ursprüngliche Liste:



nach Ausführung von **`DreifachPosLoeschen`:**



Gehen Sie bei der Implementation von folgenden Typdefinitionen und der Deklaration des Prozedurkopfes von `DreifachPosLoeschen` aus.

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

procedure DreifachPosLoeschen (inRefAnfang : tRefListe);
{ loescht das dritte, sechste, neunte usw. Element aus der
  Liste, auf deren Anfang inRefAnfang zeigt }
```

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

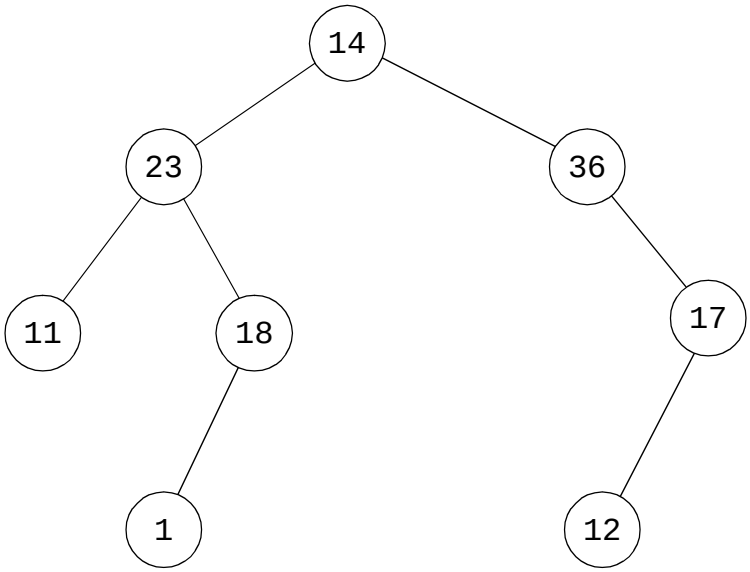
Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Aufgabe 4 (8+2 Punkte)

Eine **rekursive** Funktion `Vorkommen` soll in einem binären Baum die Anzahl aller Knoten bestimmen, bei denen der Wert der `info`-Komponente größer oder gleich einem übergebenen Wert ist. Wird der Funktion `Vorkommen` ein Zeiger auf die Wurzel des Baumes der folgenden Abbildung und der Wert 14 übergeben, so liefert sie als Ergebnis den Wert 5.



a) Ihre Aufgabe ist es, die Funktion `Vorkommen` zu implementieren. Gehen Sie dabei von den folgenden Typdefinitionen und der Funktionsdeklaration aus.

```
type
tNatZahl = 0..maxint;
tRefBinBaum = ^tBinBaum;

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

function Vorkommen (
           inRefWurzel : tRefBinBaum;
           inSuchwert : integer) : tNatZahl;
{ ermittelt die Anzahl aller Knoten des Baumes, auf dessen
  Wurzel inRefWurzel zeigt, bei denen der Wert der info-
  Komponenten groesser oder gleich inSuchwert ist}
```

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

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Aufgabe 5 (3 + 6 Punkte)

Jede natürliche Zahl größer 1 kann als Produkt von Primzahlen dargestellt werden. Für 12 erhalten wir $2 \cdot 2 \cdot 3$ und für 21 ergibt sich $3 \cdot 7$. Eine solche Zerlegung einer natürlichen Zahl in Primfaktoren kann auf einfache Weise ermittelt werden, indem man zunächst versucht, sie so oft wie möglich durch 2 zu kürzen, danach durch 3, dann durch 4 usw. Die Prozedur `Primfaktoren` berechnet alle Primfaktoren eines übergebenen Wertes nach diesem Algorithmus und gibt die gefundenen Primfaktoren auf dem Bildschirm aus.

```
     type
     tNatZahlPlus = 1..maxint;

1.   procedure Primfaktoren (inZahl : tNatZahlPlus);
     { gibt alle Primfaktoren der Zahl inZahl auf dem
       Bildschirm aus }

2.     var
3.     Zahl,
4.     Faktor : tNatZahlPlus;

5.   begin
6.     Zahl := inZahl;
7.     Faktor := 2;
8.     write ('Ergebnis: ');
9.     while Zahl > 1 do
10.      if Zahl mod Faktor = 0 then
11.      begin
12.        write (Faktor, ' ');
13.        Zahl := Zahl div Faktor
14.      end
15.      else
16.        Faktor := Faktor + 1
17.  end; { Primfaktoren }
```

Ihre Aufgabe ist es, die Prozedur zu testen, indem Sie die Teilaufgaben a) und b) bearbeiten.

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

n_{Knoten} ◯ 1 - 17

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

---

b) Geben Sie **alle** Pfade im Kontrollflußgraphen für einen 'boundary interior'-Test (interior-Klasse: n=2) und zu jedem ausführbaren Pfad ein Testdatum an. (Die assoziierten Testfälle brauchen Sie nicht anzugeben.)

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Aufgabe 6 (6 + 3 Punkte)

Das folgende Programm `Aufgabe6` berechnet zu einer natürlichen Zahl N die Summe $N^2 + N$. `MAXINT`-Bereichsüberläufe sollen nicht berücksichtigt werden.

Zeigen Sie, dass das Programm `Aufgabe6` bezüglich der Spezifikation $\{P \equiv (N \geq 0)\}\ S\ \{Q \equiv (sum = N^2 + N)\}$ partiell korrekt ist, indem Sie die Aufgabenteile a) und b) bearbeiten.

```
program Aufgabe6;

  type
  tNatZahl = 0..MAXINT;

  const
  N = ?;

  var
  sum,
  i : tNatZahl;

  begin
    sum := 0;
    i := 0;
    while i<N do
    begin
      i := i + 1;
      sum := sum + 2*i
    end
  end.
```

a) Zeigen Sie, dass

$$INV \equiv (sum = i^2 + i) \wedge (N \geq i)$$

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 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

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 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

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

a) Warum sagt die partielle Korrektheit eines Programms nichts über dessen Terminierung aus?

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) Gegeben sei eine while-Schleife

```
while B do
  S_while
```

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

```
{INV ∧ B}
  S_while
{INV ∧ B}
```

Terminiert die Schleife? Begründen Sie kurz Ihre Antwort?

d) Warum ist es im Allgemeinen schwierig, ein Programm zu verifizieren? Denken Sie nicht nur an die eigentliche Durchführung der Verifikation, sondern auch an die nötigen Voraussetzungen.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 06.03.2004

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.

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