Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Wintersemester 2000/2001
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 30),
   - 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 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, die Tabelle mit den im Kurs verwendeten Hoare-Regeln und die Definition der Terminierungsfunktion finden Sie im Anschluß an die Aufgabenstellung.
6. Sie haben die Klausur sicher dann bestanden, wenn Sie mindestens 50 % der Punkte erreicht haben.

Wir wünschen Ihnen bei der Bearbeitung der Klausur viel Erfolg!

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Aufgabe 1 (4+4 Punkte)

Gegeben sei folgende Typdefinition eines Arrays:

```
const
Max = ?;

type
tIndex = 1..Max;
tFeld = array [tIndex] of integer;
```

a) Schreiben Sie eine Prozedur `VerschiebeZyklisch`, die die Werte eines übergebenen Feldes vom Typ `tFeld` um eine Position nach rechts verschiebt. Dabei soll der Wert des letzten Elements zum Wert des neuen ersten Elements werden.
Beispiel:
Vor Aufruf der Prozedur :

9	7	5	1	3	·····	4	8
1	2	3	4	5	·····	MAX	

Danach :

8	9	7	5	1	·····	4
1	2	3	4	5	·····	MAX

Verwenden Sie folgenden Prozeduraufruf:

```
procedure VerschiebeZyklisch (var ioFeld : tFeld);
```

Sie dürfen <u>kein</u> Hilfsarray zur Lösung der Aufgabe benutzen.

b) Benutzen Sie die Prozedur aus a), um eine Prozedur `VerschiebeBis` zu implementieren, die die Werte eines übergebenes Feldes solange nach rechts verschiebt, bis ein übergebener Elementwert an erster Stelle steht. Falls dieser Wert nicht im Feld vorkommt oder der erste Wert ist, soll das Feld unverändert zurückgegeben werden.
Verwenden Sie folgenden Prozeduraufruf:

```
procedure VerschiebeBis (inWert: integer; var ioFeld : tFeld);
```

Dabei können Sie `VerschiebeZyklisch` als in dem Programmkontext bekannt voraussetzen.

Hinweis: Eine einfache Lösung besteht aus zwei Schritten: Im ersten Schritt wird festgestellt, ob `inWert` in dem Array vorkommt. Kommt `inWert` vor, so wird im zweiten Schritt wie verlangt verschoben.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Aufgabe 2 (6+4 Punkte)

Gegeben sei folgende Typdeklaration einer linearen Liste:

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

a) Schreiben Sie eine Prozedur `suchen`, die einen Zeiger auf das letzte Vorkommen einer Zahl in einer als Parameter übergebenen Liste zurückgibt. Sie können davon ausgehen, dass der Suchwert mindestens einmal in der Liste vorhanden ist. Verwenden Sie folgenden Prozedurkopf :

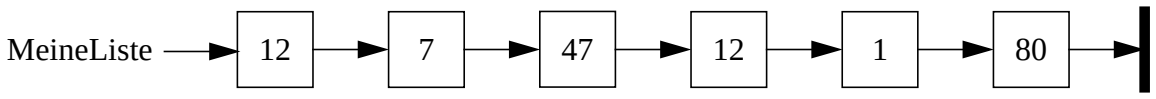
```
procedure suchen( inSuchwert: integer; inRefAnfang: tRefListe;
                  var outPos: tRefListe);
```

b) Schreiben Sie mit Hilfe der Prozedur `suchen` , die Sie als bekannt voraussetzen können, auch wenn Sie den Teil a) nicht bearbeitet haben, eine Prozedur `loeschen`, die das letzte Vorkommen einer Zahl in einer als Parameter übergebenen Liste löscht. Sie können davon ausgehen, dass der Löschwert mindestens einmal in der Liste vorhanden ist. Verwenden Sie folgenden Prozedurkopf und wählen Sie die passenden Parameterübergabearten:

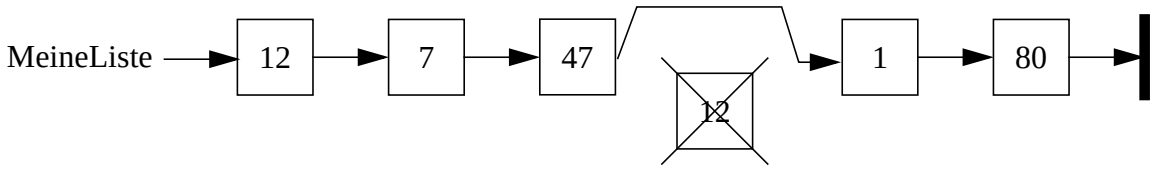
```
procedure loeschen( ?? ??Suchwert: integer; ?? ??RefAnf: tRefLi-
                                                          ste);
```

Beispiel:

Vor dem Aufruf von `loeschen(12, MeineListe)`:



Danach:



Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Aufgabe 3 (10 Punkte)

Gegeben ist folgende Definition einer linearen Liste:

```
type
tRefListe = ^tListe;
tListe = record
           info : integer;
           next : tRefZahl
         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, neu, vor: TRefListe;

begin
  new(neu);
  neu^.info := inWert;
  if ioRefAnfang = NIL then                (* Leere Liste *)
  begin
    neu^.next := NIL;
    ioRefAnfang := neu
  end
  else
  begin
    vor := ioRefAnfang;
    lauf := ioRefAnfang^.next;
    if vor^.info>inWert then               (* Neues Element wird erstes
    begin                                    Element *)
      (1);
      (2)
    end
    else
      if lauf = NIL then             (* Die Liste besteht nur aus einem
      begin                            Element. Das neue Element wird
        (3);                           das zweite *)
        (4)
      end
      else
      begin
```

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

```
        while (lauf^.info < inWert) AND (lauf^.next <> NIL) do
        begin     (* Liste besteht aus mindestens zwei Elementen.
                      Einfügeposition suchen *)
          (5);
          (6)
        end;
        if lauf^.info >= inWert then (* Innerhalb der Liste einfügen *)
        begin
          (7);
          (8)
        end
        else (* An das Ende der Liste einfügen *)
        begin
          (9);
          (10)
        end
      end
  end
end;
```

Folgende zehn Anweisungen müssen so in die Prozedur eingefügt werden, dass sie ihre Aufgabe korrekt erfüllt. Schreiben Sie dazu zu jeder Anweisung die entsprechende Position in folgende Tabelle. Einige Anweisungen werden zweimal benötigt und kommen daher doppelt vor:

Anweisung	Position
`neu^.next := NIL`	
`neu^.next := lauf`	
`neu^.next := vor`	
`neu^.next := NIL`	
`vor^.next := neu`	
`vor := vor^.next`	
`vor^.next := neu`	
`lauf^.next := neu`	
`lauf := lauf^.next`	
`ioRefAnfang := neu`	

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Aufgabe 4 (6+6+2 Punkte)

Gegeben sei folgende Typdefinition eines binären Suchbaumes:

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

a) Schreiben Sie eine rekursive Funktion `Hoehe`, die die Höhe eines übergebenen Suchbaumes bestimmt. Verwenden Sie folgenden Funktionskopf, ergänzen Sie auch die Parameterübergabeart:

```
function Hoehe(?? ??Wurzel: tRefBinBaum):integer;
```

Hinweis: Die Höhe eines Suchbaumes ist gleich der Anzahl der Knoten auf dem längsten Suchpfad im Suchbaum. Die Höhe des leeren Baumes ist 0.

b) Einen binären Baum nennt man *balanciert*, wenn für jeden Knoten gilt, dass sich die Höhen seiner beiden Teilbäume um höchstens 1 unterscheiden. Schreiben Sie eine rekursive Prozedur `Pruefen`, die feststellt, ob ein binärer Baum balanciert ist oder nicht. Dazu dürfen Sie die Funktion `Hoehe` aus a) verwenden, auch wenn Sie diesen Aufgabenteil nicht bearbeitet haben. Benutzen Sie folgenden Prozedurkopf:

```
procedure Pruefen(inWurzel: tRefBinBaum; var ioBalanciert:boolean);
```

Hinweis: Die Prozedur besteht im Wesentlichen aus einem Durchlauf durch den Baum, bei dem für jeden Knoten die Differenz der Höhen seiner Teilbäume überprüft wird.

c) Welchen Wert muß `ioBalanciert` beim ersten Aufruf besitzen?

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Aufgabe 5 (9+3 Punkte)

Folgendes Programm bestimmt die Summe der natürlichen Zahlen von 1 bis N.

```
program Aufgabe5;

  const
  N = ?;

  var
  i,
  sum : integer;

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

a) Zeigen Sie, dass folgendes Prädikat eine Invariante der while-Schleife ist:

$$\texttt{INV} \equiv \quad \texttt{sum} = \frac{i \cdot (i+1)}{2} \quad \wedge \; i \leq N$$

Geben Sie dazu die verwendeten Hoare-Regeln explizit an.

b) Wie lautet eine formale Spezifikation des Programms `Aufgabe5`, so dass `Aufgabe5` bezüglich dieser partiell korrekt ist?

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Aufgabe 6 (3+5 Punkte)

Zeigen Sie, dass das Programm `Aufgabe5` terminiert.

a) Beschreiben Sie kurz anschaulich, warum das Programm terminiert. Benutzen Sie dazu nicht mehr als 50 Worte.

b) Beweisen Sie jetzt formal die Terminierung, indem Sie eine Funktion τ angeben und beweisen, dass τ eine Terminierungsfunktion der **while**-Schleife ist.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Aufgabe 7 (8 Punkte)

Gegeben sei die BNF-Grammatik G = (N, T, P, S) mit :

```
N = {<start>, <A>, <B>},
T = {a, b},
S = <start>,
P = {<start> ::= a<start>b | <A><B>,
   <A>  ::= ε | a<A>,
   <B> ::= ε | b<B>}.
```

Genau eines der folgenden Worte gehört zu der von der BNF-Grammatik G erzeugten Sprache L(G). Bestimmen Sie das Wort und geben Sie eine Ableitung an:

a) bbbbaaa

b) aaabba

c) aaabbbb

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Aufgabe 8 (3 +6+5 Punkte)

Die Funktion `Hoch` berechnet (ziemlich umständlich) die `n`-te Potenz einer `integer` -Zahl , die größer 0 ist. Beispielsweise ist die 2-te Potenz von 6 gleich 36 und damit `Hoch` (6,2) = 36. Sie können annehmen, dass `Zahl` > 0 und `n>=0` gilt.

Gehen Sie dabei von folgender Typdefinition aus:

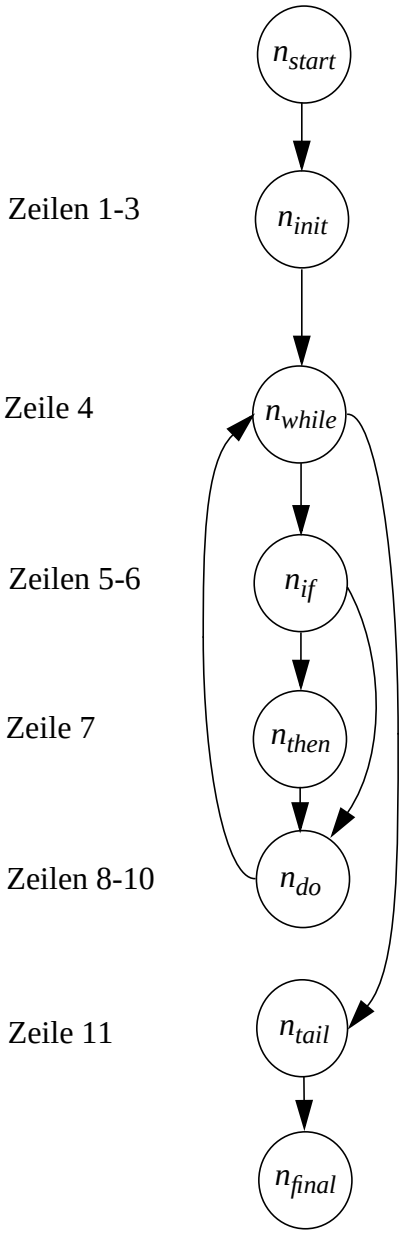
```
type
tNatZahl = 0..maxint;
tNatZahlPlus = 1..maxint;

function Hoch (Zahl : tNatZahlPlus;
                n : tNatZahl): tNatZahlPlus;
{berechnet die n-te Potenz von Zahl > 0
fuer n >= 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 := Temp2 * Temp2;
9     Temp3 := Temp3 div 2
10  end;
11  Hoch := Temp1
end; {Hoch}
```

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

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



Kompaktifizierter Kontrollflußgraph von `Hoch`

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

---

a) Geben Sie <u>einen</u> Pfad an, mit dem eine vollständige Zweigüberdeckung der Funktion `Hoch` erreicht wird.

b) Geben Sie die *du*-Menge (`Temp3`, n_{do}) und dazugehörige Teilpfade an.

c) Welche der folgenden Aussagen sind richtig?

- A. Ist eine vollständige Anweisungsüberdeckung erreicht, ist auch zugleich eine vollständige Zweigüberdeckung erreicht.

- B. Bei einer vollständigen Zweigüberdeckung sind die Testfälle so gewählt, dass jede Schleife mindestens einmal durchlaufen wird.

- C. Beim boundary-interior Test konzentriert man sich auf die Fehlersuche im Zusammenhang mit Schleifen.

- D. Ohne Spezifikation des Programms ist kein kontrollfluß- oder datenflußbezogener Test sinnvoll.

- E. Ist die Terminierung eines Programmes bewiesen, deckt ein boundary interior Test keinen Fehler mehr auf.

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

Name: _____ Matrikelnummer: _____

Kurs 1612 “Konzepte imperativer Programmierung”

Klausur am 03.03.2001

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