Kurs 1612 “Konzepte imperativer Programmierung”

Musterlösung zur Nachklausur am 03.04.2004

Aufgabe 1

a) Da Effizienzbetrachtungen bei der Lösung der Aufgabe keine Rolle spielen, wählen wir einen einfachen, aber ineffizienten Algorithmus mit zwei ineinander verschachtelten **for**-Schleifen. Da die **for**-Schleifen nicht abbrechen, wenn ein Element gefunden wurde, das nicht den Wert 0 hat, kann man sich einer Zwischenvariablen (hier: `UngleichNull`) bedienen, um das Ergebnis bis zum Ende der **for**-Schleifen zu speichern.

```
function UntereDreiecksmatrix (
           var inMatrix : tMatrix) : boolean;
{ liefert den Wert true, falls inMatrix eine Untere
  Dreiecksmatrix ist, sonst false }

  var
  Zeile,
  Spalte : tZeilenSpalten;
  UngleichNull : boolean;

begin
  UngleichNull := false;
  for Zeile := 1 to ZEILENSPALTENMAX - 1 do
    for Spalte := Zeile + 1 to ZEILENSPALTENMAX do
      UngleichNull := UngleichNull or
                      (inMatrix [Zeile, Spalte] <> 0);
  UntereDreiecksmatrix := not UngleichNull
end; { UntereDreiecksmatrix }
```

b) Bei der Berechnung der Determinante wird zunächst geprüft, ob es sich bei der übergebenen Matrix um eine untere Dreiecksmatrix handelt. Ist dies der Fall, wird die Determinante durch Multiplikation der Werte aller Elemente auf der Hauptdiagonalen ermittelt.

```
function Determinante (var inMatrix : tMatrix) : real;
{ berechnet die Determinante der quadratischen Matrix
  inMatrix durch Multiplikation der Werte der
  Hauptdiagonalen, falls es sich um eine Untere
  Dreiecksmatrix handelt, sonst durch Aufruf der Funktion
  DeterminanteAllgemein }

  var
  diagonal : tZeilenSpalten;
  det : real;

begin
  if not UntereDreiecksmatrix (inMatrix) then
    det := DeterminanteAllgemein (inMatrix)
```

```
    else
    begin
      det := 1;
      for diagonal := 1 to ZEILENSPALTENMAX do
        det := det * inMatrix [diagonal, diagonal]
    end;
    Determinante := det
  end; { Determinante }
```

Aufgabe 2

Die Liste wird mit den Zeigern `vor`, `lauf` und `nach` durchlaufen. Dabei zeigt `lauf` auf das aktuelle Element und `vor` auf dessen Vorgänger. Soll das aktuelle Element gelöscht werden, zeigt `nach` auf den Nachfolger.

```
procedure EntferneUngerade(var ioRefZahl: tRefListe);
{ loescht alle ungeraden Ziffern aus ioRefZahl }

  var
  vor,
  lauf,
  nach: tRefListe;

begin {Initialisierung von vor und lauf}
  vor := ioRefZahl; {ist stets gerade Ziffer}
  lauf := ioRefzahl^.next; {kann schon nil sein}
  while lauf <> nil do
  begin
    if odd(lauf^.info) then
    begin {Ziffer ungerade}
      nach:=lauf^.next;
      vor^.next := nach;
      dispose(lauf);
      lauf := nach
    end
    else
    begin {Ziffer gerade}
      vor := lauf;
      lauf := lauf^.next
    end {if odd}
  end {while}
end; {EntferneUngerade}
```

Aufgabe 3

```
procedure Vertausche(inPos: tRefListe;
                     var ioRefAnfang: tRefListe);

var lauf, hilf: tRefListe;

begin
  if inPos=ioRefAnfang then   (* Sonderfall: Der Listen-
                              kopf muss getauscht werden *)
  begin
    hilf := ioRefAnfang^.next;              (*1*)
    ioRefAnfang^.next := hilf^.next;        (*2*)
    hilf^.next := ioRefAnfang;              (*3*)
    ioRefAnfang := hilf                     (*4*)
  end
  else
  begin
    lauf := ioRefAnfang;
    while lauf^.next<>inPos do(* Vorgaenger von inPos finden *)
      lauf := lauf^.next;
    {* Normalfall: Vertauschen innerhalb der Liste *}
    lauf^.next := inPos^.next;              (*5*)
    hilf := inPos^.next^.next;              (*6*)
    inPos^.next^.next := inPos;             (*7*)
    inPos^.next := hilf                     (*8*)
  end
end;
```

Aufgabe 4

```
  function zweiSoehne (inRefWurzel : tRefBinBaum) : boolean;
  { gibt den Wert true zurueck, wenn der Baum leer ist oder alle
  inneren Knoten zwei Soehne haben; sonst den Wert false }

  begin
    if inRefWurzel = nil then
      zweiSoehne := true
    else
      if ((inRefWurzel^.links = nil) and
                        (inRefWurzel^.rechts <> nil)) or
```

```
        ((inRefWurzel^.links <> nil) and
                        (inRefWurzel^.rechts = nil)) then
      { der aktuelle Knoten hat genau einen Sohn, es erfolgt
        kein weiterer rekursiver Aufruf }
      zweiSoehne := false
    else
      { untersuche die Soehne des aktuellen Knotens}
      zweiSoehne := zweiSoehne (inRefWurzel^.rechts) and
                    zweiSoehne (inRefWurzel^.links)
end; { zweiSoehne }
```

Aufgabe 5

Abweisender Test:	$T_0 = \{(0, 0)\}$
Boundary Test:	$T_1 = \{(1, 1)\}$
Interior Test (2 Durchläufe):	$T_2 = \{(2,3)\}$

Aufgabe 6

a) Um zu zeigen, dass

```
INV ≡ (sum = i*(i+1)/2) ∧ (i <= N)
```

eine Invariante der Schleife ist, müssen wir die Prämisse der while-Regel zeigen, d.h. die Gültigkeit der Programmformel (1):

```
{(sum = i(i + 1)/2) ∧ (i <= N) ∧ (i < N)}
begin
  i := i + 1;
  sum := sum + i
end
{(sum = i(i + 1)/2) ∧ (i <= N)}
```

Zunächst können wir die Vorbedingung der Programmformel (1) zu

```
{(sum = i(i + 1)/2) ∧ (i < N)}
```

vereinfachen.

Nach dem *Zuweisungsaxiom* sind

```
{(sum + i + 1 = (i + 1)(i + 2)/2) ∧ (i + 1 <= N)}
  i := i + 1
{(sum + i = i(i+1)/2) ∧ (i <= N)}
```

---

und

```
{(sum + i = i(i+1)/2) ∧ (i <= N)}
  sum := sum + i
{(sum = i(i + 1)/2) ∧ (i <= N)}
```

gültige Programmformeln.

Nach der *Sequenzregel* und wegen `i + 1 <= N <=> i < N` ist

```
{(sum + i + 1 = (i + 1)(i + 2)/2) ∧ (i < N)}
  i := i + 1;
  sum := sum + i
{(sum = i(i + 1)/2) ∧ (i <= N)}
```

eine gültige Programmformel.

Aus der *Zusammensetzungsregel* ergibt sich dann die Gültigkeit von

```
{(sum + i + 1 = (i + 1)(i + 2)/2) ∧ (i < N)}
  begin
  i := i + 1;
  sum := sum + i
  end
{(sum = i(i + 1)/2) ∧ (i <= N)}
```

Wenn wir jetzt zeigen können, dass die Äquivalenz (2)

```
   (sum + i + 1 = (i + 1)(i + 2)/2) ∧ (i < N)
<=> (sum = i(i + 1)/2) ∧ (i < N)
```

gilt, haben wir die Gültigkeit von Programmformel (1) bewiesen.

Wir formen um:

$$
\begin{aligned}
& \texttt{sum + i + 1 = (i + 1)(i + 2)/2} \\
\Leftrightarrow\ & \texttt{sum = (}i^2\texttt{ + 2i + i + 2)/2 - (i + 1)} \\
\Leftrightarrow\ & \texttt{sum = (}i^2\texttt{ + i)/2} \\
\Leftrightarrow\ & \texttt{sum = i(i + 1)/2}
\end{aligned}
$$

Damit ist die Äquivalenz (2) und schließlich die Programmformel (1) bewiesen.

b) Wir müssen die Vorbedingung P des Programms aus der Vorbedingung der while-Schleife, d.h. aus der Invarianten INV, und die Nachbedingung Q aus der Nachbedingung der while-Schleife, d.h. aus INV $\wedge$ $\neg$`B`, ableiten.

Kurs 1612 “Konzepte imperativer Programmierung”

Musterlösung zur Nachklausur am 03.04.2004

---

1. Wir „arbeiten uns in Richtung der Vorbedingung Q des Programms vor“. Dazu zeigen wir die Gültigkeit der Programmformel (1)

```
{P ≡ N >= 0}
  i := 0;
  sum := 0;
{INV ≡ (sum = i(i + 1)/2) ∧ (i <= N)}
```

   Wenn wir zeigen können, dass

```
{N >= 0}
  i := 0;                                          (2)
{(0 = i(i + 1)/2) ∧ (i <= N)}
```

   und

```
{(0 = i(i + 1)/2) ∧ (i <= N)
  sum := 0;                                        (3)
{(sum = i(i + 1)/2) ∧ (i <= N)
```

   gültige Programmformeln sind, ist gemäß der *Sequenzregel* auch Programmformel (1) gültig.

   Durch Anwendung des *Zuweisungsaxioms* erhalten wir unmittelbar die Gültigkeit von (3).
   Durch weitere Anwendung des *Zuweisungsaxioms* erhalten wir als Vorbedingung von (2):

```
(0 = 0) ∧ (0 <= N)
```

   Dies ist aber äquivalent zur Vorbedingung `P ≡ N >= 0` des Programms.

2. Wir „arbeiten uns jetzt in Richtung der Nachbedingung Q vor“.
   Wir zeigen, dass INV ∧ ¬B <=> Q.
   Nun ist

```
   INV ∧ ¬B ≡
   (sum = i(i + 1)/2 ∧ (i <= N) ∧ (i >= N)
<=> (sum = i(i + 1)/2) ∧ (i = N)
<=> (sum = N(N + 1)/2) ≡ Q.
```

3. Aus Teil 1. und Teil 2. folgt über die *Zusammensetzungsregel* die partielle Korrektheit des Programms `Aufgabe6`.

Aufgabe 7

In der Schleife kommen die Variablen `i` und `sum` sowie die Konstante `N` vor.
Wir wählen also als Terminierungsfunktion

$\tau : \mathbb{Z}^3 \rightarrow \mathbb{Z}$ und $\tau(i,N,sum) = N-i$.

Offensichtlich erfüllt τ die Eigenschaft w1).

Bei einem Schleifendurchlauf wird i um den Wert 1 erhöht. Gilt vor dem Schleifendurchlauf $\tau = N-i$, so gilt nach dem Schleifendurchlauf $\tau = N-(i+1) = N-i-1$.
Damit bilden die Funktionwerte von τ in Durchlaufreihenfolge eine streng monoton fallende Folge und w2) ist bewiesen.

Um w3) zu zeigen, wählen wir `t*=0` als untere Schranke. Wir müssen beweisen, dass vor Ausführung des Schleifenrumpfes stets $\tau = N-i >= t*=0$ gilt.
Dies folgt aber unmittelbar aus der Schleifenbedingung `i < N`.

Aufgabe 8

a) Ein boundary interior Test kann auch für stets terminierende Schleifen noch Fehler aufdecken, z.B. wenn solche Schleifen nicht das erwartete Ergebnis liefern.

b) Sei {P} S {Q} eine Programmspezifikation. Dann ist jedes Programm S’ bezüglich dieser Spezifikation partiell korrekt, wenn eine gültige Programmformel {P’} S’ {Q’} existiert und P => P’ und Q’ => Q gilt.

c) Die erste und die letzte Aussage sind richtig. Die Implikation P => P’ aus der Definition der partiellen Korrektheit besagt, dass P die stärkere und P’ die schwächere Vorbedingung ist. Es gibt grundsätzlich mindestens so viele Eingaben, die eine schwächere Vorbedingung erfüllen, wie Eingaben, die eine stärkere Vorbedingung erfüllen.

d) Die Semantik einer Schleife kann nicht angegeben werden, ohne dass die Semantik des Schleifenrumpfes bekannt ist. Die Definition der Semantik der while-Schleife stützt sich daher auf der Semantikdefinition ihres Rumpfes ab. Letztere muss bekannt sein und wird als Prämisse des Interferenzschemas formuliert. Die Prämisse zusammen mit der Schleifenbedingung ergibt dann die Semantik der while-Schleife.

e) Gemeinsamkeit:
Beide Verfahren versuchen, die Fehlerfreiheit von Programmen sicher zu stellen.
Unterschiede:
- Verifikation ist im Gegensatz zum Testen ein mathematisch-formales Verfahren.
- Ist die Verifikation erfolgreich, ist die Korrektheit (Fehlerfreiheit) des Programms sichergestellt. Dagegen garantiert erfolgreiches Testen noch nicht die Fehlerfreiheit.