Kurs 1612 “Konzepte imperativer Programmierung”

Musterlösung zur Nachklausur am 01.04.2000

Lösung 1 (6 + 6 = 12 Punkte)

a) Es wird zuerst in einer **`if`**-Bedingung überprüft, ob das Summieren von `Vielfaches` und `inZahl` ohne Bereichsüberschreitung möglich ist. Ist das der Fall, wird `inZahl` solange zu `Vielfaches` aufsummiert, bis `Vielfaches` größer als `MAXZAHL - inZahl` ist. Eine weitere Addition von `inZahl` darf nicht stattfinden, da `Vielfaches` sonst einen Wert außerhalb seines Wertebereiches zugewiesen bekäme.

```
Vielfaches := inZahl;
writeln (Vielfaches);
if Vielfaches <= MAXZAHL - inZahl then
  repeat
    Vielfaches := Vielfaches + inZahl;
    writeln (Vielfaches)
  until Vielfaches > MAXZAHL - inZahl
```

b) In der **`for`**-Schleife iteriert der Index i von 1 bis `MAXZAHL` **`div`** `inZahl`. Vielfaches wird bei jedem Schleifendurchlauf der Wert `i * inZahl`, also ein Vielfaches von `inZahl`, zugewiesen.

```
for i := 1 to MAXZAHL div inZahl do
begin
  Vielfaches := i * inZahl;
  writeln (Vielfaches)
end { for }
```

Lösung 2 (13 + 3 + 4 = 20 Punkte)

a) Wir müssen zunächst die Abbruchbedingung des Verfahrens festlegen: Wir beenden die Rekursion, wenn das zu untersuchende Feldstück nur noch aus einem Element besteht. In diesem Fall ist der dort eingetragene Wert das Maximum des Feldstücks. Im anderen Fall erfolgen zwei rekursive Aufrufe der Prozedur für die beiden "Hälften" des Feldes. Von den zwei ermittelten Teilmaxima wird dann das Maximum bestimmt.

Wir geben zusätzlich wieder ein umgebendes Programm an, welches den Test der Prozedur ermöglicht:

```
program Maximum (input, output);
{ dient dem Test der Prozedur rekMax }

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

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

  var
  Feld : tFeld;
  k : tIndex;
  maxi : 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 }

    var
    schnitt : tIndex;
    max1,
    max2 : integer;

  begin
    if inBis = inVon then
      outMax := inFeld[inVon]
    else
    begin
      schnitt := (inBis + inVon) div 2;
      rekMax (inVon, schnitt, inFeld, max1);
      rekMax (schnitt + 1, inBis, inFeld, max2);
      if max1 > max2 then
        outMax := max1
      else
        outMax := max2
    end { if }
  end; { rekMax }

begin
  writeln ('Bitte 11 integer-Zahlen eingeben!');
  for k := UNTEN to OBEN do
    read (Feld[k]);
  readln;
  rekMax (UNTEN, OBEN, Feld, maxi);
  writeln ('Das Maximum lautet ', maxi)
end. { Maximum }
```

b) Es gibt gute Gründe sowohl für die Übergabe des Feldes (und damit der Feldstücke) als Wertparameter als auch für die Übergabe als Referenzparameter. In jedem Fall funktioniert die Prozedur für beide Übergabearten. Für die Übergabe als Wertparameter spricht, daß das Feld natürlich nicht verändert werden soll, sondern lediglich das Maximum zu bestimmen ist. Für die Übergabe als Referenzparameter spricht, daß bei den rekursiven Aufrufen keine Kopiervorgänge von Feldstücken nötig werden, was Speicherplatz und (Kopier-) Zeit einspart. Wir haben uns für den zweiten Fall entschieden, bezeichnen den Parameter aber trotz Referenzübergabe als `inFeld`, da es sich um einen reinen Eingabeparameter handelt.

c) Dieses Beispiel ist keine sinnvolle Anwendung der Rekursion. Es handelt sich nämlich um ein einfaches lineares Problem (in Abhängigkeit von der Anzahl der Feldelemente), welches durch einen einzigen Felddurchlauf zu lösen wäre. Dabei müßten wir jedes Feldelement genau einmal betrachten. Beim rekursiven Verfahren wird auch jedes Feldelement genau einmal betrachtet werden. Allerdings ergeben sich zusätzlich rekursive Aufrufe, und zwar mehr als Feldelemente vorhanden sind (fast doppelt so viele wie Feldelemente), und jeder Aufruf verursacht Parameterübergaben, Aufteilung in zwei Hälften usw..

Lösung 3 (18 Punkte)

Zur Lösung unseres Problems durchlaufen wir `inListe1` mit den Zeigern `Zeiger1` und `ZeigHilf1` und `inListe2` mit den Zeigern `Zeiger2` und `ZeigHilf2`, wobei stets `Zeiger1^.next=ZeigHilf1` und `Zeiger2^.next=ZeigHilf2` gilt. Dabei ändern wir die Folgezeiger der Listenelemente, bis das Ende von `inListe1` und damit von `inListe2` erreicht ist. Die angegebene Prozedur arbeitet auch für den Fall ungleich langer Listen. Wir geben zusätzlich ein Rahmenprogramm an, welches den Test der Prozedur `ListenMerge` ermöglicht:

```
program Aufgabe (input, output);
{ Testprogramm fuer die Funktion ListenMerge }

  type
  tRefListe = ^tListe;
  tListe = record
             info : integer;
             next : tRefListe
           end;
  tNatZahlPlus = 1..maxint;

  var
  Liste,
  Liste1,
```

```
Liste2 : tRefListe;
Laenge : tNatZahlPlus;

function ListenMerge (
           inListe1,
           inListe2 : tRefListe) : tRefListe;
{ fasst die Elemente der nicht-leeren und gleich langen Listen
  inListe1 und inListe2 mit dem Reissverschlussverfahren durch
  Aenderung der Verkettung zu einer einzigen Liste zusammen
  und gibt einen Zeiger auf den Anfang der Ergebnisliste
  zurück }

  var
  Zeiger1,
  Zeiger2,
  ZeigHilf1,
  ZeigHilf2 : tRefListe;

begin
  Zeiger1 := inListe1;
  Zeiger2 := inListe2;
  ListenMerge := Zeiger1;
  while (Zeiger1 <> nil) and (Zeiger2 <> nil) do
  begin
    ZeigHilf1 := Zeiger1^.next;
    ZeigHilf2 := Zeiger2^.next;
    Zeiger1^.next := Zeiger2;
    Zeiger1 := ZeigHilf1;
    Zeiger2^.next := Zeiger1;
    Zeiger2 := ZeigHilf2
  end
end; { ListenMerge }

procedure ListeAufbauen (
                 inAnz : tNatZahlPlus;
             var ioListe : tRefListe );
{ liest inAnz Zahlen von der Tastatur ein und speichert sie in
  der Reihenfolge der Eingabe in der linearen Liste ioListe }

  var
  Zahl : integer;
  AnfListe,
  EndListe,
  hilf : tRefListe;
  k : tNatZahl;
```

```
  begin
    AnfListe := nil;
    writeln ('Bitte ', inAnz, 'natuerliche Zahlen eingeben:');
    for k := 1 to inAnz do
    begin
      read (Zahl);
      new (hilf);
      hilf^.info := Zahl;
      hilf^.next := nil;
      if AnfListe = nil then
      begin
      { erste Zahl }
        AnfListe := hilf;
        EndListe := hilf
      end
      else
      begin
      { zweite und weitere Zahlen }
        EndListe^.next := hilf;
        EndListe := hilf
      end
    end;
    ioListe := AnfListe
  end; { ListeAufbauen }

begin
  writeln ('Bitte die Listenlaenge eingeben (>0):');
  readln (Laenge);
  ListeAufbauen (Laenge, Liste1);
  ListeAufbauen (Laenge, Liste2);
  Liste := ListenMerge (Liste1, Liste2);
  writeln ('Die "verkettete" Liste lautet:');
  repeat
    write (Liste^.info);
    Liste := Liste^.next
  until Liste = nil
end. { Aufgabe}
```

Lösung 4 (6 Punkte)

Das Wort "`aabcbccacbcacc`"gehört zu der Sprache L(G), die von der BNF-Grammatik G erzeugt wird. Dabei ist eine Ableitung für "`aabcbccacbcacc`"

<start> → <A><B>c<A>c<B><A><C>
→ <A>abc<B>c<A>acbc<B><A>acc
→ <A>aabcbc<B>cacbcacc
→ aabcbccacbcacc

Lösung 5 (9 + 8 = 17 Punkte)

a) Nach der Bedingungsregel 1 ist die Gültigkeit der Programmformel

```
{ P_if ∧ Y > 0 }
begin
  summe := summe + 1;
  i := i + 1
end
{ Q_if }
```

zu zeigen. (Die Gültigkeit des **else**-Zweigs muß laut Aufgabenstellung nicht gezeigt werden). Nach der Zusammensetzungsregel ist also die Gültigkeit der Programmformel

```
{ P_if ∧ Y > 0 }
summe := summe + 1;
i := i + 1
{ Q_if }
```

zu zeigen. Gemäß Sequenzregel ist die obige Programmformel gültig, wenn wir eine Bedingung R angeben können, so daß die beiden Programmformeln

```
{ P_if ∧ Y > 0 }                                    (1)
summe := summe + 1;
{ R }
und

{ R }                                               (2)
i := i + 1
{ Q_if }
```

gültig sind. Wir wenden das Zuweisungsaxiom auf (2) an und erhalten :

$$\{\ R \equiv X = \mathtt{summe} - (\mathtt{i}+1) \wedge (Y > 0 \wedge \mathtt{i}+1 \le Y \vee Y \le 0 \wedge \mathtt{i}+1 \ge Y)\ \}$$

Setzen wir dieses so gewonnene Prädikat R in (1) ein, so erhalten wir mit dem Zuweisungsaxiom die Vorbedingung

$$\{\ P_{then} \equiv X = \mathtt{summe} + 1 - (\mathtt{i}+1) \wedge (Y > 0 \wedge \mathtt{i}+1 \le Y \vee Y \le 0 \wedge \mathtt{i}+1 \ge Y)\ \}$$

Nach der Konsequenzregel 2 ist noch die Implikation $(P_{if} \wedge Y > 0) \Rightarrow P_{then}$ zu zeigen :

$$
\begin{aligned}
& P_{if} \wedge Y > 0 \\
\Leftrightarrow\ & X = summe - i \wedge (Y > 0 \wedge i < Y \vee Y \leq 0 \wedge i > Y) \wedge Y > 0 \\
\Leftrightarrow\ & X = summe - i \wedge (Y > 0 \wedge i < Y \wedge Y > 0) \vee (Y \leq 0 \wedge i > Y \wedge Y > 0) \\
\Leftrightarrow\ & X = summe - i \wedge Y > 0 \wedge i < Y \\
\Leftrightarrow\ & X = summe + 1 - (i + 1) \wedge Y > 0 \wedge i + 1 \leq Y \\
\Rightarrow\ & X = summe + 1 - (i + 1) \wedge (Y > 0 \wedge i + 1 \leq Y \vee Y \leq 0 \wedge i + 1 \geq Y) \\
\Leftrightarrow\ & P_{then}
\end{aligned}
$$

Damit ist die Gültigkeit der Programmformel

```
{ P_if ∧ Y > 0 }
begin
  summe := summe + 1;
  i := i + 1
end
{ Q_if }
```

gezeigt.

b) Nach der **`while`**-Regel ist die Gültigkeit der folgenden Programmformel zu zeigen:

```
{ INV ∧ i ≠ Y }
if Y > 0 then
begin
  summe := summe + 1;
  i := i + 1
end
else
begin
  summe := summe - 1;
  i := i - 1
end
{ INV }
```

Sei weiterhin S_{if} der Programmcode von Zeile 16 bis Zeile 25.

Behauptung A : Es gilt $(INV \wedge i \neq Y) \Leftrightarrow P_{if}$

Beweis :

$$
\begin{aligned}
& INV \wedge i \neq Y \\
\Leftrightarrow\ & summe = X + i \wedge (Y > 0 \wedge i \leq Y \vee Y \leq 0 \wedge i \geq Y) \wedge i \neq Y \\
\Leftrightarrow\ & X = summe - i \wedge (Y > 0 \wedge i < Y \vee Y \leq 0 \wedge i > Y) \\
\Leftrightarrow\ & P_{if}
\end{aligned}
$$

Behauptung B : Es gilt $Q_{if} \Leftrightarrow INV$

Beweis :

$$
\begin{aligned}
& Q_{if} \\
\Leftrightarrow\ & X = summe - i \ \wedge\ (Y > 0 \ \wedge\ i \le Y \ \vee\ Y \le 0 \ \wedge\ i \ge Y) \\
\Leftrightarrow\ & summe = X + i \ \wedge\ (Y > 0 \ \wedge\ i \le Y \ \vee\ Y \le 0 \ \wedge\ i \ge Y) \\
\Leftrightarrow\ & INV
\end{aligned}
$$

Nach Teilaufgabe a) ist

$$
\begin{aligned}
& \{\ P_{if}\ \} \\
& S_{if} \\
& \{\ Q_{if}\ \}
\end{aligned}
$$

eine gültige Programmformel und es gilt $(INV \ \wedge\ i \ne Y) \Leftrightarrow P_{if}$ (siehe Behauptung A). Nach der Konsequenzregel 2 ist dann auch

$$
\begin{aligned}
& \{\ INV \ \wedge\ i \ne Y\ \} \\
& S_{if} \\
& \{\ Q_{if}\ \}
\end{aligned}
$$

eine gültige Programmformel. Es gilt weiterhin $Q_{if} \Leftrightarrow INV$ (siehe Behauptung B), also ist auch

$$
\begin{aligned}
& \{\ INV \ \wedge\ i \ne Y\ \} \\
& S_{if} \\
& \{\ INV\ \}
\end{aligned}
$$

nach der Konsequenzregel 1 eine gültige Programmformel.

Lösung 6 (10 Punkte)

w1) Im Schleifenrumpf und in der Schleifenbedingung kommen die Variablen `summe` und `i` und die Konstante `Y` vor. Wir suchen also eine dreistellige Terminierungsfunktion und wählen

$\tau: \mathbb{Z}^3 \rightarrow \mathbb{Z}$ mitt

$$
\tau(summe,\ i,\ Y) = \begin{cases} Y - i, & \text{falls } Y \ge 0 \\ -Y + i, & \text{falls } Y < 0 \end{cases}
$$

Es ist zu zeigen, daß τ auch die Eigenschaften w2) und w3) erfüllt.

w2) Ist $Y \ge 0$, dann wird bei jedem Durchlauf der **`then`**-Zweig der **`if`**-Anweisung ausgeführt. Darin wird `i` um eins erhöht. Da vor Ausführung der Schleife $i < Y$ galt, wird $Y - i$ echt kleiner. Entsprechend argumentiert man für $Y < 0$.

---

w3) Wir wählen t* = 0. Für alle möglichen Werte von `Y` und `i` folgt aus der Invarianten und der **`while`**-Schleifenbedingung

$(Y \geq 0 \wedge i \leq Y \vee Y < 0 \wedge i \geq Y) \wedge i \neq Y$ stets
$\tau(\text{summe}, i, Y) \geq 0$ (= t*).

Lösung 7 (6 + 3 + 5 + 3 = 17 Punkte)

a) siehe nächste Seite

b)
du(Summe, n_{init}) ={n_{then1}, n_{else2}}
du(Summe,n_{then1})={n_{then1}, n_{else2}}
du(Zahl,n_{repeat}) = {(n_{repeat}, n_{then1}), (n_{repeat}, n_{until}), n_{then1},(n_{until}, n_{repeat}), (n_{until}, n_{if2})}

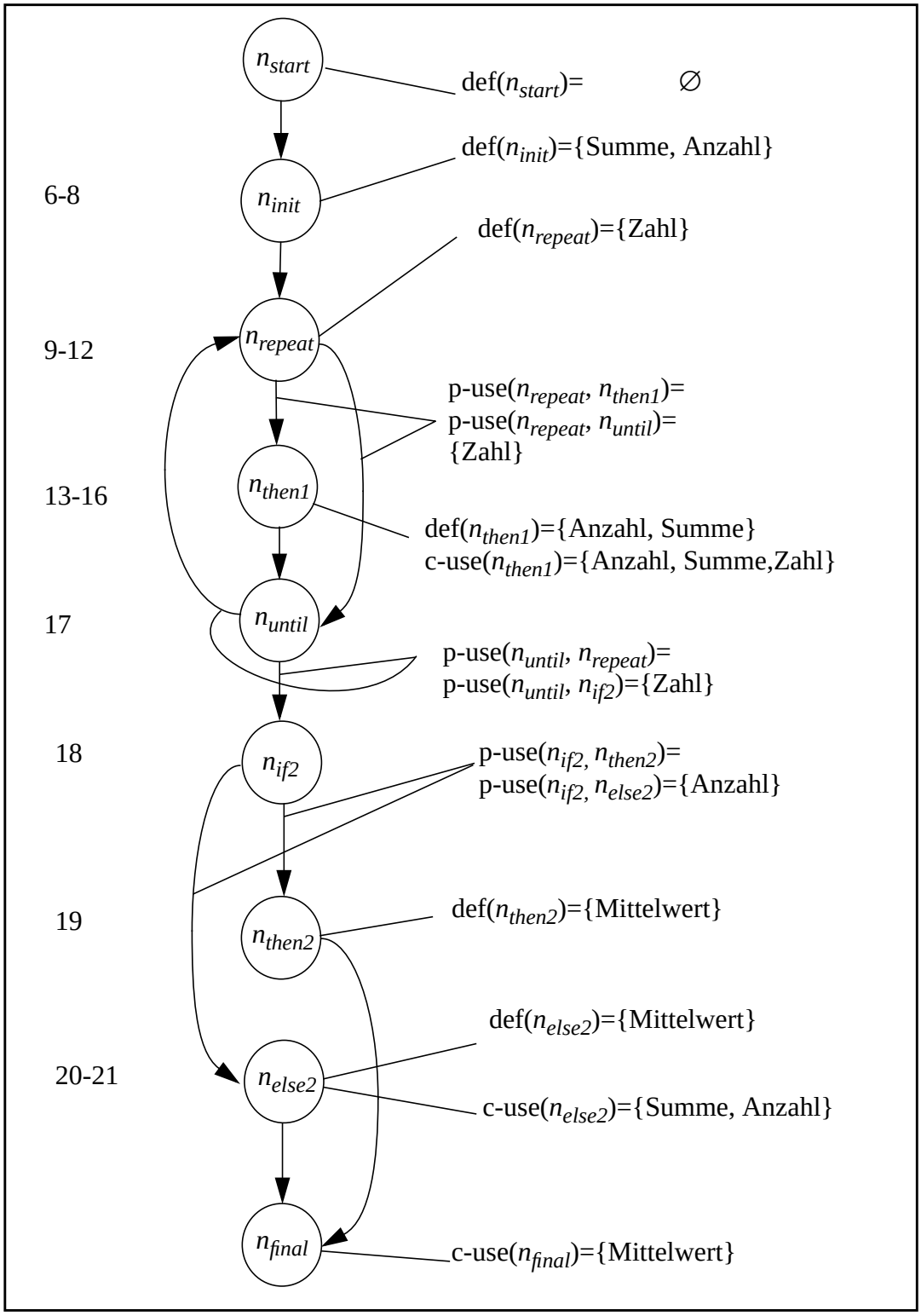
c)
du (Summe, n_{init}):

n_{init} -->	n_{then1}	:{n_{init}, n_{repeat}, n_{then1}}	(1)
	n_{else2}	:{n_{init}, n_{repeat}, n_{until}, n_{if2}, n_{else2}}	(2)

du (Summe, n_{then1}):

n_{then1} -->	n_{then1}	:{n_{then1}, n_{until}, n_{repeat}, n_{then1}}	(3)
	n_{else2}	:{n_{then1}, n_{until}, n_{repeat}, n_{until}, n_{if2}, n_{else2}}	(4)

d) Der Pfad (n_{start}, n_{init}, n_{repeat}, n_{then1}, n_{until}, n_{repeat}, n_{then1}, n_{until}, n_{repeat}, n_{until}, n_{if2}, n_{else2}, n_{final}) deckt die Teilpfade `(1)`, (3) und `(4)`ab. Ein Testdatum ist T((2,4,0), 3).
Der Teilpfad (2) ist auf Grund der Programmlogik nicht ausführbar.


n_{start}
def(n_{start})= ∅
n_{init}
def(n_{init})={Summe, Anzahl}
6-8
n_{repeat}
def(n_{repeat})={Zahl}
9-12
p-use(n_{repeat}, n_{then1})=
p-use(n_{repeat}, n_{until})=
{Zahl}
n_{then1}
13-16
def(n_{then1})={Anzahl, Summe}
c-use(n_{then1})={Anzahl, Summe,Zahl}
n_{until}
17
p-use(n_{until}, n_{repeat})=
p-use(n_{until}, n_{if2})={Zahl}
n_{if2}
18
p-use(n_{if2}, n_{then2})=
p-use(n_{if2}, n_{else2})={Anzahl}
n_{then2}
19
def(n_{then2})={Mittelwert}
n_{else2}
20-21
def(n_{else2})={Mittelwert}
c-use(n_{else2})={Summe, Anzahl}
n_{final}
c-use(n_{final})={Mittelwert}