Kurs 1613 "Einführung in die imperative Programmierung"

Musterlösung zur Nachklausur am 05.04.2008

Aufgabe 1

a)

```
function ZeileMalSpalte(inMatL: tMatrixLxM; inMatR: tMatrixMxN;
                        inI: tIndexL; inJ:tIndexN): integer;
{Berechnet für 1 <= k <= M jeweils das Produkt des k-ten Elements
der inI-ten Zeile von inMatL und des k-ten Elements der inJ-ten
Spalte von inMatR und summiert diese M Produkte.}

var k: tIndexM;
    summe: integer;

begin
  summe := 0;
  for k := 1 to M do
    summe := summe + inMatL[inI,k] * inMatR[k,inJ];
  ZeileMalSpalte := summe;
end;
```

b)

```
procedure matrProdukt(inMatL: tMatrixLxM; inMatR: tMatrixMxN;
                      var outMat: tMatrixLxN);
{Berechnet das Matrizenprodukt inMatL * inMatR.}

var i:tIndexL;
    j:tIndexN;

begin
  {Für jedes Paar (i,j) den Matrizeneintrag berechnen.}
  for i := 1 to L do
    for j := 1 to N do
      outMat[i,j] := ZeileMalSpalte(inMatL, inMatR, i, j);
end;
```

Kurs 1613 "Einführung in die imperative Programmierung"

Musterlösung zur Nachklausur am 05.04.2008

Aufgabe 2

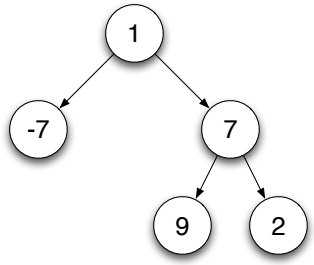
```
procedure Tausche(var ioRefAnfang: tRefElement);
{Sucht das voraussetzungsgemäß vorkommende Element mit Info=0
und macht dieses zum neuen Listenanfang, indem die ab 0 begin-
nende Restliste der vorhergehenden Teilliste vorangestellt
wird.}
var lauf, endeNeu: tRefElement;
begin
  if ioRefAnfang^.info <> 0 then
  { Nur wenn 0 nicht das erste Element ist, muss ueberhaupt
    etwas gemacht werden }
  begin
    { Suchen & Merken des Vorgängers der 0 (das neue Listenende)}
    lauf := ioRefAnfang;
    while lauf^.next^.info <> 0 do
      lauf := lauf^.next;
    endeNeu := lauf;

    { Suchen des (alten) Listenendes }
    while lauf^.next <> nil do
      lauf := lauf^.next;

    { Verkettungsänderung: }
    lauf^.next := ioRefAnfang;
    ioRefAnfang := endeNeu^.next;
    endeNeu^.next := nil
  end
end;
```

Aufgabe 3

```
function DVKopie(inRefQuell: tRefElement): tRefDVElement;
{Erzeugt aus einer einzugebenen einfach verketteten Quellliste
eine neue doppelt verkettete Liste, ohne die Quellliste zu ver-
ändern.}

var laufQuell: tRefElement; { Laufzeiger für eingegebene Liste }
    anfangKopie,    { Anfangszeiger für neu aufzubauende Liste }
    endeKopie,      { Endzeiger für neu aufzubauende Liste }
    neuesElement:tRefDVElement; { Hilfszeiger zum Erzeugen neuer
                                  Elemente (falls benötigt). }

begin
  { Initialisierung (neue Liste noch leer) }
  anfangKopie := nil;
  endeKopie := nil;
  laufQuell := inRefQuell;
  { Quellliste durchlaufen und eine Kopie jedes Elementes ans
    Ende der neuen Liste anfügen. }
  while laufQuell <> nil do
  begin
    { Kopie des betrachteten Quellelementes erzeugen... }
    new(neuesElement);
    neuesElement^.info := laufQuell^.info;
    neuesElement^.next := nil;
    { ... und hinten an neue Liste anfügen }
    if endeKopie = nil then { Sonderfall: neue Liste noch leer }
    begin
      anfangKopie := neuesElement;
      neuesElement^.prev := nil
    end
    else { Normalfall: hinter endeKopie anfügen }
    begin
      endeKopie^.next := neuesElement;
      neuesElement^.prev := endeKopie;
    end;
    { Listenende-Zeiger aktualisieren (in beiden Fällen gleich)}
    endeKopie := neuesElement;

    { Nächstes Quellelement betrachten }
    laufQuell := laufQuell^.next;
  end;
  DVKopie := anfangKopie;
end;
```

Aufgabe 4

a) Die Prozedur `whatDoIDo` liefert die Ausgabe *true*, genau dann wenn jeder innere Knoten genau zwei Nachfolger hat, oder anders ausgedrückt, wenn kein Knoten mit genau einem Nachfolger existiert.
(Für den rechts abgebildeten nicht vollständigen Baum würde sie beispielsweise mit *true* antworten.)



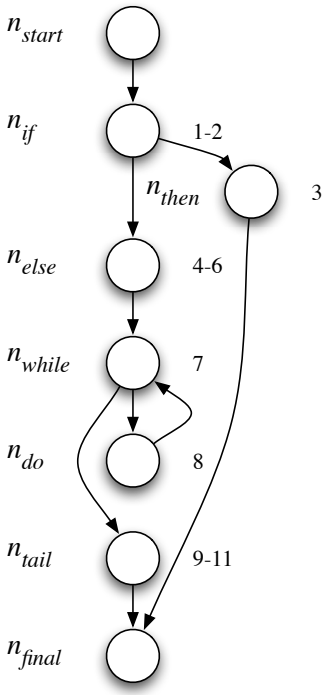
b) Eine kurze Erläuterung vorab:
Zur Feststellung der Vollständigkeit eines Baumes ist im Rekursionsschritt zusätzlich zu überprüfen, ob beide Teilbäume auch gleich hoch sind. Die Prozedur `pruefeVollstaendigkeit` entstand daher, indem der Prozedur `whatDoIDo` der Vergleich, ob beide Teilbäume gleich hoch sind, hinzugefügt wurde.
Nun lässt sich folgende Beobachtung treffen:
Ist die Höhe beider Teilbäume eines Knotens = 0, so ist der Knoten ein Blatt. Ist die Höhe beider Teilbäume > 0, so hat der Knoten genau zwei Nachfolger (nicht-leere Teilbäume). Das heißt: Die Bedingung, dass beide Teilbäume gleiche Höhe haben (Zeilen 16 & 17) impliziert bereits, dass der Knoten entweder ein Blatt ist oder zwei Nachfolger hat.
Somit kann Zeile 15 ersatzlos entfallen. Mit ihr entfallen weiterhin die Zeilen 12 & 13 sowie die Variablendeklarationen in Zeile 4:

```
1 procedure pruefeVollstaendigkeit(inWurzel: tRefKnoten;
2                                  var outErgebnis:boolean);
3 var ergL, ergR: boolean;
4     Blatt, ZweiNachf: boolean;
5 begin
6   if inWurzel = nil then
7     outErgebnis := true
8   else
9   begin
10    pruefeVollstaendigkeit(inWurzel^.links, ergL);
11    pruefeVollstaendigkeit(inWurzel^.rechts, ergR);
12    Blatt := (inWurzel^.links=nil) and (inWurzel^.rechts=nil);
13    ZweiNachf:=(inWurzel^.links<>nil)and(inWurzel^.rechts<>nil);
14    outErgebnis :=  ergL and ergR
15                    and (Blatt or ZweiNachf)
16                    and (hoehe(inWurzel^.links)
17                       = hoehe(inWurzel^.rechts));
18  end
19end;
```

Aufgabe 5

a) Die jeweils dritte Komponente eines solchen Testdatums gibt das laut Spezifkation erwartete Ergebnis an, d.h. die erwartete Ausgabe unter Eingabe der zuvor genannten Werte.

b) Die rechts stehende Abbildung zeigt den kompakten Kontrollflussgraphen. Die Zuordnung der Schlüsselwortzeilen 4 und 5 kann dabei wahlweise zu n_{then} oder n_{else} erfolgen. Den Knoten n_{start} und n_{final} werden definitionsgemäß keine Zeilen zugeordnet.



c) Die drei Testdaten bewirken eine vollständige Zweigüberdeckung: Es gibt zwei Verzweigungen im Kontrollflussgraphen, einmal bei Knoten n_{if} und einmal bei Knoten n_{while}.
- Beide von n_{if} ausgehenden Zweige werden überdeckt: (n_{if}, n_{then}) durch das Testdatum D_1, (n_{if}, n_{else}) durch sowohl D_2 als auch D_3.
- Auch beide von n_{while} ausgehenden Zweige werden überdeckt: D_2 und D_3 führen jeweils zu mindestens einem Durchlauf der While-Schleife, wodurch sowohl der Schleifeneintritts-Zweig (n_{while}, n_{do}) als auch der Schleifenabbruchs-Zweig (n_{while}, n_{tail}) passiert werden.

d) Die drei Testdaten erreichen *keine* minimale Mehrfachbedingungsüberdeckung: Das zweite Teilprädikat der If-Bedingung (`inZahl > MAXWERT`) wird unter allen drei Testdaten zu *false* ausgewertet, niemals zu *true*. Es wird also nicht überdeckt. (Anschaulich: Der Then-Zweig zur Sonderfallbehandlung wird zwar überdeckt, vgl. c), jedoch nur für einen von zwei möglichen Sonderfällen getestet.)

Obige Begründung genügt bereits zur Lösung der Aufgabe, der Vollständigkeit halber demonstrieren wir diese Tatsache aber auch noch einmal anhand der gegebenen Tabelle:

Prädikate	***Testdaten*** (`inFeld`, `inZahl`, *erwartetes Ergebnis)* mit A := [6,2,3,4,1]		
	(A, 0, -1)	(A, 2, 2)	(A, 10, 0)
`inZahl <= 0`	T	F	F
`inZahl > MAXWERT`	F	F	F
`if`-Bedingung	T	F	F
`i < FELDMAX`	—	T	T,F
`inFeld[i] <> inZahl`	—	T,F	F
`while`-Bedingung	—	T,F	T,F

Anhand dieser Tabelle ist zu erkennen, dass alle vorkommenden Bedingungen mit Ausnahme von `inZahl > MAXWERT` überdeckt werden.