Kurs 1613 „Einführung in die imperative Programmierung“

Musterlösung zur Klausur am 06.02.2010

Aufgabe 1

```
procedure NachVorn( inWert: integer;
                    var ioRefAnfang: tRefElement);
{Sucht das erste vorkommende Element mit inWert in der
info-Komponente und verschiebt dieses Element an den Anfang
der Liste.
Vorbedingung: Ein solches Element muss in der Liste vorkommen!}

  var lauf,
      elem: tRefElement;

begin
  {Sonderfallprüfung: Falls das erste Element den Wert
  inWert hat, so ist nichts zu tun.}
  if (ioRefAnfang^.info <> inWert) then
  begin
    {Den Vorgaenger des zu verschiebenden Elementes suchen:}
    lauf := ioRefAnfang;
    while lauf^.next^.info <> inWert do
      lauf := lauf^.next;
    elem := lauf^.next;
    {elem zeigt nun auf das zu verschiebende Element,
     lauf auf dessen Vorgaenger. Jetzt verschieben:}
    lauf^.next := elem^.next;
    elem^.next := ioRefAnfang;
    ioRefAnfang := elem;
  end; {if}
end; {NachVorn}
```

Laut Aufgabenstellung durfte vorausgesetzt werden, dass der Suchwert in der Liste vorkommt, was insbesondere bedeutet, dass die Liste nicht leer sein kann, und dass, falls das Element nicht am Anfang steht, die Liste mindestens zwei Elemente hat. Die While-Bedingung oben ist unter dieser Voraussetzung problemlos, es kann nicht zu einer Dereferenzierung eines **`nil`**-Zeigers kommen.

Aufgabe 2

```
function RoemZahlWert(inRoemZahl: tRefRoemZahlStelle): integer;
{Berechnet den Wert der römischen Zahl, auf deren letzte/rechte
Stelle der Zeiger inRoemZahl verweist.
Vorbedingung: inRoemZahl <> nil, d.h. die römische Zahl ist min-
destens einstellig. }

var lauf: tRefRoemZahlStelle;
    summe, wert, vorigerWert: integer;

begin
  lauf := inRoemZahl;
  vorigerWert := 0;
  summe := 0;
  {Betrachte nun der Reihe nach jede Ziffer...}
  while lauf <> nil do
  begin
    wert := RoemZiffWert(lauf^.ziffer);
    {Subtraktion, wenn wert der betrachteten Ziffer kleiner
     als Wert der vorher betrachteten Ziffer, sonst Addition.
     (Bei Betrachtung der ersten (rechten) Ziffer ist
     vorigerWert=0, wert>0 und damit wert>vorigerWert.) }
    if wert < vorigerWert then
      summe := summe - wert
    else
      summe := summe + wert;
    { Vorbereitungen für nächsten Durchlauf }
    vorigerWert := wert;
    lauf := lauf^.links
  end;
  RoemZahlWert := summe
end;
```

Aufgabe 3

Die Werte der ersten beiden Fibonacci-Zahlen f(0) und f(1) sind vorgegebene Konstanten und direkt auszugeben. Für $n>1$ ist die n-te Fibonacci-Zahl aus den beiden vorhergehenden zu berechnen. Wir starten also (für `inZahl` ≥ 2) eine Schleife, in der wir die Folge der Fibonacci-Zahlen bis zur gesuchten `inZahl`-ten Zahl berechen, wobei wir uns jeweils die letzte und vorletzte Fibonacci-Zahl merken müssen, um daraus die jeweils nächste berechnen zu können:

```
function fibonacci(inZahl: tNatZahl): tNatZahl;
{ Berechnet iterativ die inZahl-te Fibonacci-Zahl }

  var letzte, vorletzte, aktuelle, lauf: tNatZahl;

begin
  if inZahl < 2 then
    fibonacci := inZahl
  else
  begin
    { Initialisierung der Vorgänger (f(0) und f(1))
      für die Berechnung von f(2) }
    vorletzte := 0;
    letzte := 1;
    { Berechnung von f(2) bis f(inZahl) }
    for lauf := 2 to inZahl do
    begin
      aktuelle := vorletzte + letzte;
      vorletzte := letzte;
      letzte := aktuelle;
    end;
    fibonacci := aktuelle
  end
end;
```

Aufgabe 4

a) Die Suche kann und soll abbrechen, sobald ein Gegenbeispiel gefunden wurde. Nehmen wir an, bei einem rekursiven Aufruf für irgendeinen (Teil-)baum sei die Wurzel dieses Teilbaums ein Gegenbeispiel (habe also ein „größeres Kind“). Es erscheint sinnvoll, diese Eigenschaft der Wurzel sofort zu erkennen und das Verfahren abzubrechen, ohne zuvor die Teilbäume rekursiv zu durchlaufen. D.h. die Wurzel ist vor den rekursiven Aufrufen zu untersuchen, und genau das passiert bei der Hauptreihenfolge.

   Bei Verwendung der symmetrischen Reihenfolge würde dagegen die Wurzel des (Teil)baumes erst dann untersucht, wenn bereits ihr gesamter linker Teilbaum erfolglos nach einem Gegenbeispiel durchsucht wurde. Effektiv würde der Baum als Erstes bis zum linken Blatt durchlaufen, bevor überhaupt der erste Vergleich stattfindet, ob der Knoten (also zunächst das linke Blatt) ein Gegenbeispiel ist. Die Rekursionstiefe ist somit potentiell höher als bei der Hauptreihenfolge.

   Aus diesen Gründen ziehen wir die Hauptreihenfolge vor.

b) (Gegebener Beginn kleiner gedruckt)

```
function istMaxHeap(inRefWurzel: tRefKnoten): Boolean;
{ Gibt true zurück, genau dann wenn der Baum ein Max-Heap ist. }
var linksGroesser, rechtsGroesser: Boolean;
begin
  if inRefWurzel = nil then
    {Der leere Baum ist ein (leerer) Max-Heap, da er keinen Knoten
     mit größerem Kind enthält.}
    istMaxHeap := true
  else
  begin
    {Untersuchung des Wurzelknotens und seiner Kinder}
    if inRefWurzel^.links = nil then
      linksGroesser := false
    else
      linksGroesser := inRefWurzel^.links^.info > inRefWurzel^.info;
    if inRefWurzel^.rechts = nil then
      rechtsGroesser := false
    else
      rechtsGroesser := inRefWurzel^.rechts^.info > inRefWurzel^.info;

    {Hauptreihenfolge: Erst Wurzelknoten überprüfen}
    if linksGroesser or rechtsGroesser then
      {Wurzel ist Gegenbeispiel}
      istMaxHeap := false
    else {kein Kind ist groesser}
    begin
      {Teilbäume rekursiv prüfen}
      istMaxHeap :=   istMaxHeap(inRefWurzel^.links) and
                      istMaxHeap(inRefWurzel^.rechts);
    end
  end
end; {istMaxHeap}
```

Aufgabe 5

a) Eingabeäquivalenzklassen sind eine Zerlegung der Menge der möglichen Eingaben, hier: $\mathbb{N} \times \mathbb{R}$. Wir suchen hier also vier disjunkte Teilmengen von $\mathbb{N} \times \mathbb{R}$, deren Vereinigung wieder $\mathbb{N} \times \mathbb{R}$ ergibt. Oder anders gesagt: Jede mögliche Eingabe muss in genau einer Äquivalenzklasse enthalten sein.

Anhand der beschriebenen Zerlegungs-Kriterien ergeben sich die folgenden Klassen:

$E_1 = \{(j, m) \mid m \geq 0, j < 3\}$ (Beschäftigungszeit < 3 Jahre, nicht-negativer Lohn)

$E_2 = \{(j, m) \mid m \geq 0, 3 \leq j < 6\}$ (Beschäftigungszeit ≥ 3, < 6 Jahre, nicht-negativer Lohn)

$E_3 = \{(j, m) \mid m \geq 0, j \geq 6\}$ (Beschäftigunszeit ≥ 6 Jahre, nicht-negativer Lohn)

$E_4 = \{(j, m) \mid m < 0\}$ (Beschäftigungszeit beliebig, negativer Lohn)

Jede Klasse sei Teilmenge von $\mathbb{N} \times \mathbb{R}$, d.h. für alle Klassen gelte zusätzlich $j \in \mathbb{N}$ und $m \in \mathbb{R}$.
Eine Mengennotation wie oben ist nicht unbedingt nötig, so lange die Klassen präzise beschrieben sind.

b) Die Elemente eines Testfalls sind nicht lediglich Eingabedaten, sondern Testdaten, die zusätzlich die erwartete Ausgabe spezifizieren. Um aus den Eingabeäquivalenzklassen Testfälle zu bilden, ist also die Spezifikation des erwarteten Ergebnisses zu jeder Eingabe zu ergänzen.
Die vier Testfälle lauten: (Zu nennen war nur einer davon)
$T_1 = \{(j, m, m*0.5) \mid m \geq 0, j < 3\}$
$T_2 = \{(j, m, m*0.75) \mid m \geq 0, 3 \leq j < 6\}$
$T_3 = \{(j, m, m) \mid m \geq 0, j \geq 6\}$
$T_4 = \{(j, m, \text{FEHLERCODE}) \mid m < 0\}$
Jeder Testfall sei Teilmenge von $\mathbb{N} \times \mathbb{R} \times \mathbb{R}'$, d.h. für jeden Testfall gelte $j \in \mathbb{N}$ und $m \in \mathbb{R}$,
dabei sei $\mathbb{R}' := \mathbb{R}^+ \cup \{0.0, \text{FEHLERCODE}\}$ die Menge der laut Spezifikation zulässigen Ausgaben.

c) Nein. Bei der Bildung von Ausgabeäquivalenzklassen wird die Menge der gültigen Ausgaben ($\mathbb{R}'$) in disjunkte Teilmengen zerlegt, aus denen sich dann die Testfälle ableiten lassen, indem, etwas salopp gesprochen, den Ausgaben die sie erzeugenden Eingaben zugeordnet werden.
Eine solche Zerlegung der Ausgabemenge $\mathbb{R}'$ ist anhand der „Weihnachtsgeld-Stufen“ jedoch nicht möglich, da derselbe berechnete Weihnachtsgeld-Betrag mit Eingaben aus mehreren dieser Testfälle hergeleitet werden kann. Beispielsweise berechnet sich ein Weihnachtsgeld i.H.v. 600 € sowohl bei einem Monatslohn von 600 € und einer Beschäftigungsdauer von mindestens 6 Jahren (Stufe 100%) als auch bei einem Monatslohn von 800 € und einer vierjährigen Beschäftigungsdauer (Stufe 75%) sowie bei einem Monatslohn von 1200 € und einer Beschäftigungsdauer von 2 Jahren (Stufe 50%).
Man kann also keiner Ausgabe (außer FEHLERCODE) eindeutig einen dieser vier Testfälle zuordnen und dementsprechend auch keine Menge von Ausgaben bilden, die genau für die Testdaten eines dieser Testfälle (außer T_4) erwartet werden, d.h. diese Testfälle können nicht durch Ausgabeäquivalenzklassen bestimmt werden.
Oder anders herum betrachtet: Bildet man zu jedem der vier Testfälle die Menge der von ihm produzierten Ausgaben, so erhält man keine disjunkten Mengen, d.h. auch keine Zerlegung der Wertemenge zulässiger Ausgaben.

Aufgabe 6

a) Keinmaliger Schleifendurchlauf:
Ist bei einer Repeat-Schleife nicht möglich.
Einmaliger Schleifendurchlauf:
Beispiel-Testdatum: ([0, 1, 2, 3], true)
(Ein einmaliger Durchlauf erfolgt für jedes Array a mit a[1]=0. Erwartete Rückgabe ist demnach jeweils true.)
Zweimaliger Schleifendurchlauf:
Beispiel-Testdatum: ([1, 0, 1, 0], true)
(Ein zweimaliger Durchlauf erfolgt für jedes Array a mit a[1]<>0 und a[2]=0. Erwartete Rückgabe demnach wiederum true.)
Viermaliger Schleifendurchlauf:
Beispiel-Testdatum: ([1, 2, 3, 4], false) oder auch ([1, 2, 3, 0], true)
(Ein viermaliger Durchlauf erfolgt für jedes Array, dessen erste drei Elemente nicht Null sind, d.h. das entweder gar keine Null enthält, oder genau an letzter Stelle eine Null enthält.)

b) Nein.
Ein einfaches Gegenbeispiel ist ein Boundary-Interior-Test, bei dem kein viermaliger Schleifendurchlauf getestet wird. In diesem Fall wird die atomare Bedingung `i=FELDMAX` niemals wahr.
Damit ist die Frage bereits hinreichend beantwortet, der Vollständigkeit halber geben wir im Folgenden jedoch noch eine genauere Betrachtung der Überdeckung der drei vorkommenden einfachen (atomaren) Bedingungen an:

1. Die If-Bedingung `inFeld[i]=0` wird tatsächlich zwangsläufig überdeckt, denn für einen einmaligen Schleifendurchlauf muss diese Bedingung sofort im ersten Durchlauf zu true ausgewertet werden, andernfalls wird die Schleife wiederholt. Für den zweimaligen Schleifendurchlauf wiederum muss diese Bedingung im ersten Durchlauf entsprechend zu false ausgewertet werden.
2. `gefunden` in der While-Bedingung wird beim Interior-Test mit n=2 bzw. n=3 auch zwangsläufig überdeckt: Für einen zweimaligen Durchlauf muss nach dem ersten Durchlauf noch `gefunden`=false gelten (sonst würde die Schleife bereits abbrechen), nach dem zweiten bzw. dritten Durchlauf dagegen `gefunden`=false (sonst würde die Schleife einen weiteren Durchlauf absolvieren).
   Beim Interior-Test mit n=4 Durchläufen jedoch ist die Überdeckung von `gefunden` nicht sichergestellt, denn nach dem vierten Durchlauf terminiert die Schleife auch, falls `gefunden` immer noch den Wert false hat.
3. `i=FELDMAX` in der While-Bedingung wird lediglich beim Interior-Test mit n=4 überdeckt, bei Interior-Tests mit n=2 oder n=3 dagegen wird diese Bedingung stets zu false ausgewertet (s.o.).