Kurs 1613 „Einführung in die imperative Programmierung“

Musterlösung zur Nachklausur am 06.03.2010

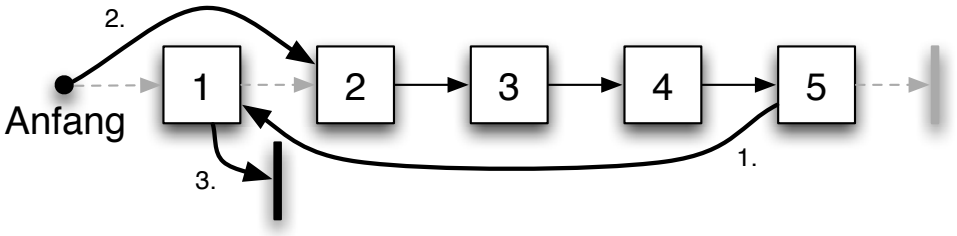
Aufgabe 1

a) Die erste Variante (Verkettungsänderung) ist vorzuziehen. Sie hat zwei wesentliche Vorteile:

- Zunächst ist eine solche Implementierung universell einsetzbar, unabhängig davon, wieviele Informationen welcher Typen auch immer in den Listenelementen gespeichert werden, d.h. die Info-Komponente kann jederzeit durch beliebige andere Datenkomponenten ersetzt oder ergänzt werden, ohne dass diese Prozedur dazu geändert werden muss.
- Vor allem aber sind für die Verkettungsänderung – wie wir in der Lösung zu b) demonstrieren – nur drei Zeiger zu modifizieren, egal wie lang die Liste ist! Denn es genügt, das erste Listenelement ans Listenende zu verschieben, darüber hinaus fällt keine weitere Arbeit zum Aufrücken der restlichen Listenelemente an. Bei der zweiten Variante dagegen müsste der Wert jedes einzelnen Listenelements in ein anderes Listenelement kopiert werden, genauer: Man merkt sich den Wert des ersten Listenelements in einer Variablen, kopiert dann den Wert des zweiten ins erste Element, den Werts des dritten ins zweite Element u.s.w., und abschließend wird der in der Variablen gemerkte ehemals erste Wert ins letzte Listenelement geschrieben. Das wäre unverhältnismäßig viel Arbeit, die mit der Länge der Liste linear zunimmt.

Anmerkung: Zur Antwort auf die Frage genügt das Nennen eines Vorteils.

b) Die folgende Abbildung zeigt, dass nur drei Zeiger manipuliert werden müssen (fett dargestellt), um das erste Element ans Ende der Liste zu verschieben, wodurch automatisch alle anderen Elemente innerhalb der Liste um eine Position nach vorne „aufrücken“:



Die kleinen Nummern an den geänderten Zeigern in obiger Abbildung geben die Reihenfolge für die Änderungen an. So darf z.B. der Anfangszeiger erst auf das zweite Element gerückt werden, wenn ein anderer Zeiger (hier der Next-Zeiger des letzten Elements, es kann natürlich auch ein Hilfszeiger verwendet werden) auf das erste Element zeigt, da letzteres andernfalls verloren ginge.

Entsprechend darf das zweite Element nicht vom ersten abgetrennt werden, bevor es nicht über einen anderen Zeiger (hier den Anfangszeiger, alternativ einen Hilfszeiger) erreichbar ist.

```
procedure RotiereListe(var ioRefAnfang: tRefElement);
{Verschiebt das letzte Listenelement an den Listenbeginn und
rückt so alle weiteren Listenelemente um eine Position nach hin-
ten.}

var ende: tRefElement;

begin
  {Rotation nur bei min. zweielementiger Liste, sonst ist
   nichts zu tun.}
  if ioRefAnfang <> nil then
    if ioRefAnfang^.next <> nil then
    begin
      {Suche zunächst letztes Element}
      ende := ioRefAnfang;
      while ende^.next <> nil do
        ende := ende^.next;

      {Nun Umverkettung vornehmen:}
      {1. Erstes Element ans Ende}
      ende^.next := ioRefAnfang;
      {2. Neuen Listenanfang markieren}
      ioRefAnfang := ioRefAnfang^.next;
      {3. Neues Listenende markieren}
      ende := ende^.next;
      ende^.next:=nil
    end
end;
```

Aufgabe 2

Sowohl zum Einfügen Listenanfang als auch zum Einfügen am Listenende muss das neue Listenelement zwischen dem letzten und dem ersten Listenelement eingefügt werden. Der einzige Unterschied besteht darin, dass beim Einfügen am Listenanfang anschließend der Anfangszeiger auf das neu eingefügte Element zeigen muss, beim Einfügen am Listenende jedoch auf dem bisherigen Listenanfang stehen bleibt.

Wir schlagen folgende Lösung vor (den Sonderfall zum Einfügen in eine leere Liste haben wir der Vollständigkeit halber – kleiner gedruckt – mit aufgenommen, er musste in der Aufgabe nicht berücksichtigt werden):

```
procedure einfuegen ( inInfo: integer;
                      inEinfuegenAmAnfang: Boolean;
                  var ioRefAnfang: tRefElement);
{Fügt ein neues Element mit Info-Wert inInfo in eine zyklische
Liste ein, auf deren erstes Element ioRefAnfang zeigt.
Ist inEinfuegenAmAnfang = true, wird am Listenanfang, andern-
falls am Listenende eingefügt.}

var ende,
    neuesElem: tRefElement;

begin
  {Neues Listenelement erzeugen und Wert inInfo eintragen}
  new(neuesElem);
  neuesElem^.info := inInfo;
  {Sonderfall: Leere Liste}
  if ioRefAnfang = nil then
  begin
    neuesElem^.next := neuesElem;
    ioRefAnfang := neuesElem
  end
  else
  begin
    {suche letztes Listenelement}
    ende := ioRefAnfang;
    while ende^.next <> ioRefAnfang do
      ende := ende^.next;
    {Einfügen zwischen letztem und erstem Element}
    neuesElem^.next := ioRefAnfang;
    ende^.next := neuesElem;
    {Falls am Anfang eingefügt werden soll, Anfang neu setzen}
    if inEinfuegenAmAnfang then
      ioRefAnfang := neuesElem;
  end
end;
```

Aufgabe 3

a) Im leeren Baum kommt der Suchwert nicht vor (da der leere Baum keine Knoten und damit auch keine Werte enthält). Ist der Baum nicht leer und steht der Suchwert im Wurzelknoten, so kommt der Suchwert offensichtlich im Baum vor. Ist der Baum nicht leer und enthält seine Wurzel nicht den Suchwert, so kommt der Wert genau dann im Baum vor, wenn er in einem der beiden Teilbäume vorkommt. Dabei kann er aufgrund der Sortierung des Baums nur in höchstens einem Teilbaum überhaupt vorkommen, weshalb nicht

beide Teilbäume durchsucht werden müssen, sondern nur derjenige, der den Suchwert überhaupt enthalten könnte.

Obige Fallunterscheidungen können schematisch in folgenden Code übertragen werden:

```
function kommtVor(inRefWurzel: tRefKnoten;
                  inSuchwert: integer): boolean;
{ Rückgabe ist true genau dann wenn der Suchbaum mit Wurzel
  inWurzel einen Knoten mit info=inSuchwert enthält. }
begin
  if inRefWurzel = nil then
    kommtVor := false
  else
    if inRefWurzel^.info = inSuchwert then
      kommtVor := true
    else
      if inSuchwert < inRefWurzel^.info then
        kommtVor:=kommtVor(inRefWurzel^.links, inSuchwert)
      else
        kommtVor:=kommtVor(inRefWurzel^.rechts, inSuchwert)
end;
```

b) Wie oben festgestellt, muss beim Suchen im Suchbaum nicht der gesamte Baum durchlaufen werden, sondern lediglich ein Pfad durch den Baum (indem immer nur in einen der beiden Teilbäume abgestiegen wird). Es wird also immer nur Referenzen gefolgt und niemals zu Vorgängerknoten zurückgesprungen, und somit besteht kein Bedarf, besuchte Knoten auf einem Stapel zu speichern. Es gibt eine relativ einfache iterative Lösung, die diesen Pfad durchläuft, analog zu einem Listendurchlauf. Die rekursive Lösung dagegen baut unnötig einen Stapel auf. Aus diesen Gründen ist die iterative Version zu bevorzugen, die Rekursion ist nicht sinnvoll.

Aufgabe 4

a)
```
function pruefeTeilfeld(inSudoku: tSudokuFeld;
                        inStartZeile, inStartSpalte,
                        inEndZeile, inEndSpalte: tZiffer):boolean;
  {Prueft, ob in dem eingegebenen Sudokufeld (inSudoku) inner-
  halb des durch die restlichen vier Parameter definierten Aus-
  schnittes jede der Ziffern von 1 bis 9 mindestens einmal
  vorkommt.}

  var markerFeld: tZifferMarkerFeld;
      i, j: tZiffer;

  begin
    initZifferMarkerFeld(markerFeld);
    {Durchlaufe gegebenen Ausschnitt zeilenweise}
    for i := inStartZeile to inEndZeile do
      for j := inStartSpalte to inEndSpalte do
        {markiere im Marker-Feld die Ziffer,
         die in der Matrix an Position (i,j) steht.}
        markerFeld[inSudoku[i,j]] := true;
    pruefeTeilfeld := alleZiffernMarkiert(markerFeld)
  end;
```

b) Es gibt zwei (verwandte) „Performance-Probleme“:

1. Die For-Schleifen durchlaufen jede Zeile bzw. jede Spalte oder jeden Block immer vollständig, selbst wenn schon vorzeitig (z.B. schon nach Betrachten der ersten Zeile der Matrix) feststeht, dass die Matrix keine Lösung sein kann.
   Das Problem lässt sich durch Ersetzen jeder For-Schleife durch jeweils eine While-Schleife lösen, die im Falle `gueltig = false` vorzeitig abbricht.
2. Es werden stets alle drei Schleifen zum Durchsuchen der Zeilen, Spalten bzw. Blöcke nacheinander ausgeführt. Eine Überprüfung der Spalten und Blöcke ist aber z.B. nur nötig, wenn nicht bereits beim Betrachten der Zeilen festgestellt wurde, dass die Matrix keine Lösung ist.
   Dieses Problem wird durch die oben vorgeschlagene Einführung von While-Schleifen gleich mit gelöst, da damit insbesondere auch nachfolgende While-Schleifen gar nicht mehr ausgeführt werden, wenn in einer vorhergehenden bereits `gueltig` den Wert `false` angenommen hat.

   (So einfach der Lösungsvorschlag aber klingt, ist eine Programmierung mit While-Schleifen doch aufwändiger, da die Laufvariablen nun separat initialisiert und inkrementiert werden müssen und Bereichsüberschreitungen beim letzten Schleifendurchlauf zu vermeiden sind. Im Folgenden geben wir der Vollständigkeit halber eine Realisierung dieser Funktion mit While-Schleifen an, eingebettet in ein vollständiges Pascal-Programm, mit dem die Lösung getestet werden kann.)

```
program sudoku;

type   tZiffer = 1..9;
       tZifferPlus1 = 1..10;
       tZifferMarkerFeld = array [tZiffer] of boolean;
       tSudokuFeld = array [tZiffer, tZiffer] of tZiffer;

var s:tSudokuFeld;
     i,j: tZiffer;

procedure initZifferMarkerFeld(var ioFeld: tZifferMarkerFeld);
{Setzt ein tZifferMarkerFeld zurueck, indem jeder Ziffer der Wert false zu-
geordnet wird.}
var i: tZiffer;
begin
  for i := 1 to 9 do
    ioFeld[i] := false
end;

function alleZiffernMarkiert(inFeld: tZifferMarkerFeld): boolean;
{Gibt genau dann true zurueck, wenn das Array an jedem Index den Wert true
enthaelt.}
var  i:tZiffer;
     erg:boolean;
begin
  erg := true;
  for i := 1 to 9 do
    if not inFeld[i] then
      erg := false;
  alleZiffernMarkiert := erg
end;

function pruefeTeilfeld(inSudoku: tSudokuFeld;
                        inStartZeile, inStartSpalte,
                        inEndZeile, inEndSpalte: tZiffer):boolean;
{Prueft, ob in dem eingegebenen Sudokufeld (inSudoku) innerhalb des durch die
restlichen vier Parameter definierten Ausschnittes jede der Ziffern von 1 bis
9 mindestens einmal vorkommt.}
var  markerFeld: tZifferMarkerFeld;
     i, j: tZiffer;
begin
  initZifferMarkerFeld(markerFeld);
  {Durchlaufe gegebenen Ausschnitt zeilenweise}
  for i := inStartZeile to inEndZeile do
    for j := inStartSpalte to inEndSpalte do
      {markiere im Marker-Feld die Ziffer,
       die in der Matrix an Position (i,j) steht.}
      markerFeld[inSudoku[i,j]] := true;
  pruefeTeilfeld := alleZiffernMarkiert(markerFeld)
end;

function pruefeSudoku(inSudoku: tSudokuFeld): boolean;
{Gibt genau dann true zurueck, wenn die Werte im Feld inSudoku eine gueltige
```

```
Loesung fuer ein Sudoku-Raetsel darstellen, wenn also gilt, dass in jeder Zei-
le, in jeder Spalte und in jedem Block jede Ziffer von 1 bis 9 genau einmal
vorkommt.}
var  i, j: tZifferPlus1;
     gueltig: boolean;
begin
  gueltig:=true;
  i := 1;
  while (i<=9) and gueltig do {Zeilen pruefen}
  begin
    if not pruefeTeilfeld(inSudoku, i, 1, i, 9) then
      gueltig:=false;
    i := i + 1
  end;
  j := 1;
  while (j<=9) and gueltig do {Spalten pruefen}
  begin
    if not pruefeTeilfeld(inSudoku, 1, j, 9, j) then
      gueltig:=false;
    j := j + 1
  end;
  i := 1;
  while (i<=3) and gueltig do
  begin
    j := 1;
    while (j<=3) and gueltig do {Bloecke pruefen}
    begin
      if not pruefeTeilfeld(inSudoku, i*3-2, j*3-2, i*3, j*3) then
        gueltig:=false;
      j := j + 1
    end;
    i := i + 1
  end;
  pruefeSudoku := gueltig
end;

begin
  for i:=1 to 9 do
  begin
    writeln('9 Eingaben fuer ',i,'-te Zeile');
    for j:=1 to 9 do
    begin
      write('[',i,',',j,']: ');
      readln(s[i,j])
    end
  end;
  if pruefeSudoku(s) then
    writeln('Korrekt, die eingegebenen Werte bilden eine Loesung.')
  else
    writeln('Das ist keine Loesung!')
end.
```

Aufgabe 5

a) Die Menge der möglichen Eingaben (eine Eingabe ist ein Paar aus einem in Parameter `inFeld` zu übergebenden Feld und einer im Parameter `inZahl` zu übergebenden Integer-Zahl) wird nach der Betrachtung in der Spezifikation in folgende drei Klassen zerlegt:

E1: Alle Eingabepaare (`inFeld`, `inZahl`) mit (1. Sonderfall)
`inZahl` < 1 oder `inZahl` > `MAXWERT`

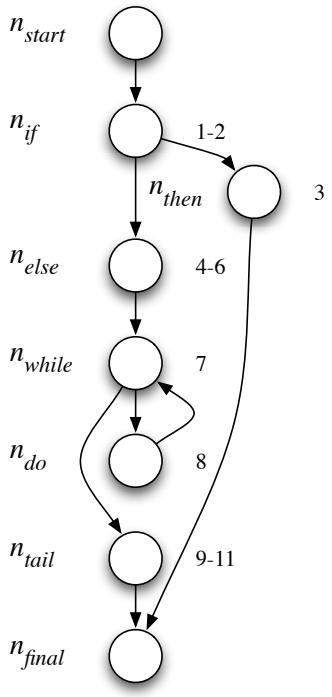
E2: Alle Eingabepaare (`inFeld`, `inZahl`) mit (2. Sonderfall)
`inZahl` ≥ 0 und `inZahl` ≤ `MAXWERT` und `inZahl` kommt nicht in `inFeld` vor.

E3: Alle Eingabepaare (`inFeld`, `inZahl`) mit (Normalfall)
`inZahl` ≥ 0 und `inZahl` ≤ `MAXWERT` und `inZahl` kommt in `inFeld` vor.

Zu Beachten ist, dass diese drei Klassen disjunkt sind und ihre Vereinigung die Menge aller möglichen Eingaben ergibt.

b) 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.

c) Der Vollständigkeit halber zeigen wir in nebenstehender Abbildung den kompakten Kontrollflussgraphen (in der Aufgabe nicht gefordert).



Die drei Testdaten bewirken eine vollständige Zweigüberdeckung: Es gibt zwei Verzweigungen im Kontrollflussgraphen, einmal bei der If- und einmal bei der While-Anweisung.

- Beide von der If-Anweisung ausgehenden Zweige werden überdeckt: Der Then-Zweig durch das Testdatum D_1, der Else-Zweig durch sowohl D_2 als auch D_3.
- Auch beide von der While-Anweisung ausgehenden Zweige werden überdeckt: D_2 und D_3 führen jeweils zu mindestens einem Durchlauf der While-Schleife, und die Schleife terminiert auch. Somit werden sowohl der Schleifeneintritts-Zweig (n_{while}, n_{do}) als auch der Schleifenabbruchs-Zweig (n_{while}, n_{tail}) passiert.

d) Die drei Testdaten erreichen *keine* vollständige 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.) Es wird somit übrigens nicht einmal eine vollständige *einfache* Bedingungsüberdeckung erreicht.

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.