Kurs 1613 "Einführung in die imperative Programmierung"

Musterlösung zur Klausur am 01.03.2008

Aufgabe 1

a) Die Funktion `indexVon` gibt den Index des ersten Vorkommens von `inWert` im Feld `inFeld` zurück. Falls der Wert im Feld gar nicht vorkommt, wird die Zahl 0 ausgegeben.

b) Die While-Schleife soll abbrechen, sobald das erste Vorkommen von `inWert` gefunden wurde — oder wenn das gesamte Feld durchlaufen wurde. Zu beachten ist der gültige Wertebereich der Indexvariablen `i`: Diese darf nicht größer als `MAXINDEX` werden und somit nach einem Lesezugriff auf das letzte Arrayelement nicht nochmals inkrementiert werden. Da sie jedoch den Wert 0 annehmen darf, wählen wir eine Lösung, die `i` nicht nach, sondern vor jedem Arrayzugriff inkrementiert:

```
function indexVon_Neu(var inFeld:tFeld;
                          inWert:integer):tIndex0;
{Funktionsbeschreibung: Siehe Teilaufgabe a)}
var
  ergebnis:tIndex0;
  i:tIndex0;
begin
  i := 0;
  ergebnis := 0;
  { Durchlaufe inFeld von Index 1 bis höchstens MAXINDEX,
    vorzeitiger Abbruch, sobald das Ergebnis gesetzt wurde,
    also der Wert gefunden wurde. }
  while (ergebnis=0) and (i<MAXINDEX) do
  begin
    i := i + 1;
    if inFeld[i] = inWert then
      ergebnis:=i;
  end;
  indexVon_Neu := ergebnis
end;
```

Aufgabe 2

Zu beachten ist bei dieser Aufgabe, dass nicht die Verkettung der übergebenen Liste geändert, sondern eine neue Liste aufgebaut werden soll. Dazu sind zunächst neue Elemente (mit der Prozedur new) zu erstellen, in die die Info-Werte kopiert werden. Die Spiegelung wird dadurch erreicht, dass jede erzeugte Elementkopie nicht am Ende der gerade im Aufbau befindlichen Liste angehängt, sondern ihr als neuer Anfang vorangestellt wird.

```
function gespiegelteKopieVon(inRefOriginal: tRefElement):
                                                tRefElement;
{Erzeugt eine gespiegelte Kopie der durch inRefOriginal referen-
zierten Liste. Die Original-Liste bleibt unverändert.}

var laufQuell,
    anfangKopie,
    neuesElement: tRefElement;
begin
  laufQuell := inRefOriginal;
  anfangKopie := nil;
  {Originalliste durchlaufen und jedes Element vorne in neue
   Liste einfügen.}
  while laufQuell <> nil do
  begin
    {Kopie des betrachteten Quellelementes erzeugen...}
    new(neuesElement);
    neuesElement^.info := laufQuell^.info;
    {... und vorne an neue Liste anfügen}
    neuesElement^.next := anfangKopie;
    anfangKopie := neuesElement;
    {Nächstes Quellelement betrachten}
    laufQuell := laufQuell^.next;
  end;
  gespiegelteKopieVon := anfangKopie;
end;
```

Aufgabe 3

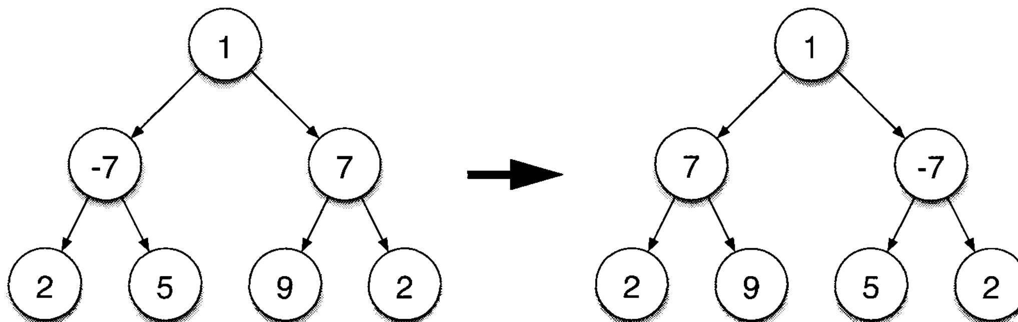
Laut Voraussetzung kommt das Element mit der Null genau einmal vor (d.h. insbesondere auch, dass die Liste nicht leer ist) und bildet nicht den Listenanfang, d.h. es hat sicher ein Vorgängerelement und der Anfangszeiger muss nicht verändert werden. Somit verbleibt nur ein einziger Sonderfall, nämlich, dass die Null bereits das letzte Element der Liste ist. In diesem Fall muss nichts geschehen. Dieser Sonderfall muss aber nicht notwendigerweise abgefangen werden, die folgende Lösung funktioniert unverändert auch mit der Null am Ende der Liste (wobei diese entfernt und wieder angehängt wird):

```
procedure Tausche(inRefAnfang: tRefElement);

  var
  vor,
  ende,
  nullElement : tRefElement;

begin
  {Vorgänger von gesuchtem Element finden}
  vor := inRefAnfang;
  while vor^.next^.info <> 0 do
    vor := vor^.next;

  {zu verschiebendes Element merken und aushängen}
  nullElement := vor^.next;
  vor^.next := nullElement^.next;

  {letztes Element finden}
  ende := vor;
  while ende^.next <> nil do
    ende := ende^.next;

  {gemerktes Element hinten anhängen}
  ende^.next := nullElement;
  nullElement^.next := nil
end; {Tausche}
```

Aufgabe 4

a) Nein, die Typen stellen lediglich sicher, dass jedes Element vom Typ `tBinBaum` bis zu zwei Nachfolger referenziert, aber nicht, dass insgesamt eine hierarchische Baumstruktur daraus gebildet wird. Bedingungen wie z.B., dass jeder Knoten im Baum höchtens einen Voränger haben darf (und mit Ausnahme der Wurzel haben muss) oder dass keine Zyklen entstehen dürfen, werden durch den Typ nicht ausgedrückt.
Es wäre z.B. möglich, aus Elementen des Typs `tBinBaum` eine doppelt verkettete Liste aufzubauen, indem `rechts` auf das Nachfolgerelement und `links` auf das Vorgängerlement zeigt.

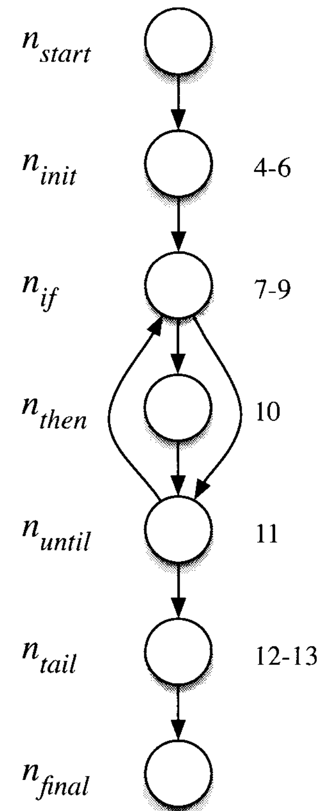
b) Der abgebildete Baum wird wie folgt transformiert:



Die Prozedur spiegelt den ihr übergebenen Baum, indem Sie die Nachfolger eines jeden Knotens vertauscht.
(Hinweis: Obwohl zweimal in den linken Teilbaum eingetreten wird, wird der gesamte Baum durchlaufen: Erst wird der linke Teilbaum gespiegelt, dann werden rechter und linker Teilbaum der Wurzel vertauscht. Anschließend wird mit dem linken Teilbaum der noch unbearbeitete, ehemals rechte Teilbaum gespiegelt. Es handelt sich also um einen In-order-Durchlauf.)

c) Zum Spiegeln muss der gesamte Baum durchlaufen werden. Da man zu bereits besuchten Knoten zurückkehren muss, um deren zweiten, noch nicht besuchten Teilbaum ebenfalls zu durchlaufen, wird zum Baumdurchlauf ein Stapel benötigt. Die rekursive Lösung ist daher sinnvoll, da sie den implizit vorhandenen Laufzeitstapel dafür ausnutzt, während bei einer iterativen Lösung ein Hilfsstapel explizit implementiert werden müsste.

Aufgabe 5



a) Nebenstehende Abbildung zeigt den kompakten Kontrollflussgraphen.
Zu beachten ist, dass laut Definition alle immer zusammen ausgeführten Programmzeilen zu einem einzigen Knoten zusammengefasst werden, also insbesondere auch Zeilen 8 und 9 innerhalb desselben Knotens stehen. Zeilen, die nur Schlüsselworte (wie **`begin`**, **`end`**, oder **`repeat`**) enthalten, werden einem beliebigen Nachbarknoten zugeordnet. Zeile 7 könnte also wahlweise auch Knoten n_{init} zugeordnet werden.
Weiterhin ist zu beachten, dass nur die Zeilen des Anweisungsteils (also Zeilen 4 bis 13) auf die Knoten abgebildet werden und dass den Knoten n_{start} und n_{final} definitionsgemäß keine Zeilen zuzuordnen sind.

b) <u>Keinmaliger Schleifendurchlauf:</u>
Ist bei einer Repeat-Schleife nicht möglich.
<u>Einmaliger Schleifendurchlauf:</u>
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.)
<u>Zweimaliger Schleifendurchlauf:</u>
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.)

c) Ja, denn der einmalige Schleifendurchlauf muss — für den vorzeitigen Abbruch — den Then-Zweig (n_{if}, n_{then}) und anschließend (n_{then}, n_{until}) durchlaufen, während der n-malige Durchlauf (nicht nur für n=2, sondern auch für größere n) zunächst — zum Vermeiden des Abbruchs nach dem ersten Durchlauf — den Else-Zweig (n_{if}, n_{until}) und daraufhin den Rücksprung (n_{until}, n_{if}) passieren muss. Alle weiteren Zweige werden bei jedem beliebigen Testdatum überdeckt, womit insgesamt eine vollständige Zweigüberdeckung erreicht wird.