Kurs 1613 "Einführung in die imperative Programmierung"

Musterlösung zur Klausur am 04.03.2006

Aufgabe 1

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 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 }
```

Anmerkung: In Übereinstimmung mit Muss-Regel 12 übergeben wir das Feld als Referenzparameter, um so Speicherplatz und (Kopier-)Zeit einzusparen. Wir bezeichnen den Parameter aber trotz Referenzübergabe als `inFeld`, da es sich um einen reinen Eingabeparameter handelt.

b) 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, ohne Verwendung eines Stapels oder einer anderen Hilfsdatenstruktur. Dabei müssten wir jedes Feldelement genau einmal betrachten. Beim rekursiven Verfahren müssen nach wie vor alle Feldelemente 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...

Aufgabe 2

a) Zum Einfügen am Listenende wird ein neues Element erzeugt und zwischen Ende und Anfang der Liste eingefügt. Außerdem muss der Listenende-Zeiger natürlich auf das neu eingefügte Element gesetzt werden, da es das neue Listenende bildet. Da dieser Zeiger sich ändert, ist er als **io**-Parameter zu übergeben, während der einzufügende Wert ein reiner **in**-Parameter ist.

```
procedure appendNotEmpty(inZahl:integer;
                         var ioRefEnde:tRefElement);
{fügt inZahl am Ende der nicht-leeren Liste ein.}
```

```
var neu:tRefElement;

begin
  new(neu);
  neu^.info := inZahl;
  { Element einfügen: }
  neu^.next := ioRefEnde^.next;
  ioRefEnde^.next := neu;
  { Neues Listenende markieren }
  ioRefEnde := neu;
end;
```

b) In dieser Teilaufgabe sind sämtliche Sonderfälle zu beachten, nämlich die leere Liste und der Fall, dass das gesuchte Element nicht vorkommt, also nicht gefunden wird. In letzterem Fall wird die Liste bis zum Ende durchlaufen. Die Besonderheit liegt hier nun darin, dass das Listenende nicht an einem NIL-Zeiger erkannt werden kann, sondern dass vielmehr zu überprüfen ist, ob der Laufzeiger auf dem von inRefEnde referenzierten letzten Element angekommen ist.

```
function find(inSuchwert:integer;
              inRefEnde:tRefElement):tRefElement;
{Sucht ein Element mit Info inSuchwert in der in inRefEnde
übergebenen Liste. Rückgabe von nil, falls nicht gefunden,
sonst Rückgabe einer Referenz auf das gefundene Element.}

var lauf:tRefElement;

begin
  {Sonderfall 1: Leere Liste => Ergebnis: NIL}
  if inRefEnde = nil then
    find:=nil
  else
  begin
    {Suchen, Abbruch bei Fundstelle oder am Listenende}
    lauf:=inRefEnde^.next; {erstes Element}
    while (lauf<>inRefEnde) and (lauf^.info<>inSuchwert) do
      lauf:=lauf^.next;

    if lauf^.info<>inSuchwert then
      {Sonderfall 2: Nicht gefunden => Ergebnis: NIL}
      find:=nil
    else
      {Normalfall: Gefunden => Ergebnis: Fundstelle}
      find:=lauf;
```

```
  end; { if }
end; { find }
```

Aufgabe 3

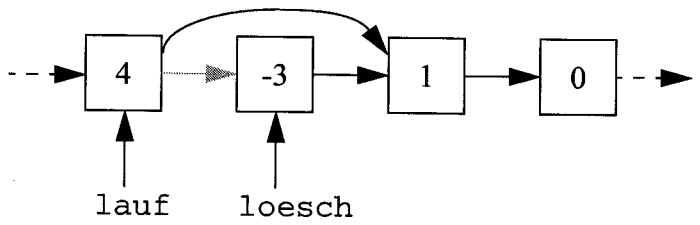
Eine einfache Lösung, welche die vereinfachenden Annahmen (Liste enthält mindestens ein nicht negatives Element) ausnutzt, sieht wie folgt aus:

In einer ersten Schleife wird so lange das erste Element gelöscht, bis die Liste nicht mehr mit einem negativen Element beginnt.
Anschließend wird in einer zweiten Schleife die Liste mit einem Zeiger namens `lauf` durchlaufen. Immer wenn `lauf` dabei auf den Vorgänger eines negativen Elementes zeigt, wird sein Nachfolger aus der Liste gelöscht.
In beiden Fällen wird ein Zeiger namens `loesch` benutzt, um das zu löschende Element nach seinem Entfernen aus der Liste nach wie vor ansprechen zu können, so dass es dann mittels `dispose(loesch)` auch aus dem Speicher gelöscht werden kann.

Die folgenden Grafiken veranschaulichen das Löschen eines Listenelements *innerhalb* einer Liste.

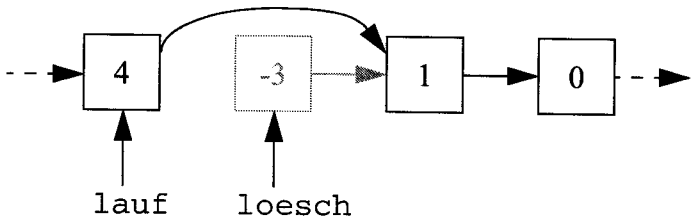
Entfernen eines Elementes aus der Liste:

```
loesch := lauf^.next;
lauf^.next := loesch^.next;
```



Nun kann das Element gelöscht werden.

```
dispose(loesch):
```




```
procedure LoescheNegative(var ioRefAnfang: tRefElement);
{ loescht alle negativen Zahlen aus ioRefAnfang,
  Liste darf nicht leer sein und muss mindestens eine
```

```
  nicht-negative Zahl enthalten! }
var
  loesch,
  lauf: tRefElement;

begin
  { Loeschen am Anfang }
  while ioRefAnfang^.info < 0 do
  begin
    loesch := ioRefAnfang;
    ioRefAnfang := ioRefAnfang^.next;
    dispose(loesch);
  end; { while }
  { Loeschen aller restlichen negativen Zahlen,
    dazu die Liste bis zum letzten Element durchlaufen}
  lauf := ioRefAnfang;
  while lauf^.next <> nil do
  begin
    if lauf^.next^.info < 0 then
    begin
      loesch := lauf^.next;
      lauf^.next := loesch^.next;
      dispose(loesch);
    end
    else
      lauf := lauf^.next;
    { Wichtig: lauf nur im else-Fall weitersetzen,
      im then-Fall wurde bereits der Nachfolger von lauf
      entfernt und als nächstes ist der neue Nachfolger
      von lauf zu untersuchen. }
  end { while }
end; { LoescheNegative }
```

Aufgabe 4

a) Die Implementierung des Suchalgorithmus für binäre Suchbäume aus dem Kurstext kann fast identisch übernommen werden: Es entfällt der Rückgabezeiger, statt dessen wird am Ende eine die Ausgabeanweisung eingefügt:

```
procedure Baumpfad (
            inWurzel : tRefBinBaum;
            inSuch : integer);
{ sucht rekursiv im Suchbaum, auf dessen Wurzel inWurzel
  zeigt, den Wert inSuch und gibt die Werte der besuchten
  Knoten in umgekehrter Reihenfolge auf dem Bildschirm
  aus }

begin
  if inWurzel <> nil then
  begin
    if inSuch < inWurzel^.info then
      Baumpfad (inWurzel^.links, inSuch)
    else
      if inSuch > inWurzel^.info then
        Baumpfad (inWurzel^.rechts, inSuch);
      { else gefunden, Ausgabe erfolgt unten }
    writeln (inWurzel^.info)
  end { if }
end; {Baumpfad}
```

b) Für eine einfache Suche in einem solchen Baum wäre die Rekursion nicht sinnvoll, da — wie beim Durchlaufen einer Liste — nur ein einziger Pfad beschritten wird, was sich iterativ in einer einfachen Schleife realisieren lässt, und zwar ohne einen Stapel oder ähnliche Hilfsmittel.

In diesem Fall liegt das Problem jedoch darin, dass die Werte auf diesem Pfad rückwärts ausgegeben werden sollen. Eine iterative Lösung müsste dazu in einem Vorwärts-Pfaddurchlauf die Werte auf dem Suchpfad explizit auf einem Stapel speichern, um sie anschließend in umgekehrter Reihenfolge ausgeben zu können. Die rekursive Lösung nimmt uns diese Arbeit mit Hilfe des implizt verwalteten Stapels ab. *Deshalb ist die rekursive Lösung hier sinnvoll.*

Aufgabe 5

Wir werten zunächst das Prädikat A) zu true aus, indem wir ein Testdatum wählen, dessen erstes Element 0 ist (die weiteren Werte im Array sind beliebig). Mit dem zweiten Testdatum werten wir A) gezielt zu false aus. Die Prädikate A), B) und F) sind damit schon vollständig überdeckt, als nächstes überdecken wir C), indem wir in einer anderen als der ersten Array-Komponente eine Null eintragen. Danach sind nur noch E) und I) unüberdeckt. Um I) gezielt zu false auszuwerten, wählen wir nun als Testdatum ein unsortiertes Feld, das keine Null enthält.
Mit den vier angegebenen Testdaten erreichen wir die minimale Mehrfach-Bedingungsüberdeckung. Zwei dieser Testdaten decken bereits die Existenz von Fehlern auf (die Funktion enthält übrigens mehr als einen Fehler).

Prädikate		inFeld																			
		0	1	2	3	1	2	3	4	1	0	2	3	2	1	3	4				
A)	`inFeld[1]=0`	T				F				F				F							
B)	`i<FELDMAX`	—				T,F				T,F				T							
C)	`inFeld[i]=0`	—				F				T,F				F							
D)	`inFeld[i-1] <= inFeld[i]`	—				T				F,T				F							
E)	C) **or** D)	—				T				T				F							
F)	B) **and** E) *(while-Bedingung)*	—				T,F				T,F				F							
G)	`inFeld[i]=0`	—				F				T,F				—							
H)	`isNull`	—				F				T				F							
I)	`inFeld[i] > inFeld[i-1]`	—				T				—				F							
erwartete Ausgabe		null				sortiert				null				unsort.							
tatsächliche Ausgabe		null				unsort.				null				sortiert							

Hinweise zur ausgefüllten Tabelle:
Die doppelten Linien kennzeichnen, dass ein Prädikat mit den links davon stehenden Testdaten bereits vollständig überdeckt wurde. Rechts davon haben wir der Vollständigkeit halber die Auswertung der bereits überdeckten Prädikate unter den noch folgenden Testdaten zusätzlich (grau) mit aufgeführt, diese Angaben sind aber nicht notwendig.