Kurs 1613 "Einführung in die imperative Programmierung"

Musterlösung zur Nachklausur am 31.03.2007

Aufgabe 1

Analysiert man das Verfahren anhand des angegebenen Beispiels, ist schnell zu erkennen, dass das erste Element von `inFeld2` nach `outFeld` an Index 2 kopiert wird, das zweite den Index 4 bekommt, das dritte den Index 6 u.s.w.. Der „Ziel-Index" für ein aus `Feld2` zu kopierendes Element ist also genau doppelt so groß wie dessen „Quell-Index", d.h. der Wert von `inFeld2[i]` ist nach `outFeld[2*i]` zu kopieren. Elemente aus `Feld1` dagegen stehen jeweils eine Position davor, d.h. der Wert von `inFeld1[i]` ist nach `outFeld[2*i-1]` zu kopieren — wobei `i` jeweils ein gültiger Index vom Typ `tIndex` sein muss.

Hat man dies erkannt, lässt sich folgende sehr kurze Lösung entwickeln:

```
procedure merge(inFeld1, inFeld2: tIntFeld;
                var outFeld: tIntFeldDoppelt);
{Fügt die Felder inFeld1 und inFeld2 zu outFeld zusammen, indem
immer abwechselnd ein Element aus inFeld1 und eines aus inFeld2
nach outFeld übernommen wird.}
var i:integer;
begin
  for i:=1 to MAX do
  begin
    outFeld[2*i-1] := inFeld1[i];
    outFeld[2*i] := inFeld2[i];
  end;
end;
```

Alternativ besteht natürlich auch die Möglichkeit, für den „Ziel-Index" eine zweite Zählvariable einzuführen.

Aufgabe 2

Die folgende Funktion stellt eine mögliche Realisierung dar. Die boolsche Variable `plus` gibt an, ob der Listenwert addiert oder subtrahiert werden muss.

```
function AltSumme(inRefAnfang: tRefElement): integer;
{ Berechnet die alternierende Summe der Listenelemente (Summe
der Elemente an ungerader Position abzüglich Summe der Elemente
an gerader Position). }

var
  summe: integer;
  lauf : tRefElement;
  plus: boolean;
```

```
begin
  summe := 0;
  lauf := inRefAnfang;
  plus := true; {Beginnen mit Addition}
  while lauf <> nil do
  begin
    if plus then
      summe := summe + lauf^.info
    else
      summe := summe - lauf^.info;
    lauf := lauf^.next;
    plus := not plus {Umschalten zwischen Addition/Subtr.}
  end
  AltSumme := summe;
end;
```

Aufgabe 3

a)

```
function vorMaximum(inRefAnfang: tRefElement): tRefElement;
{Sucht das Maximum in der übergebenen Liste und gibt einen Zeiger
auf dessen Vorgängerelement zurück.
Vorbedingung: Die Liste muss mindestens zwei Elemente besitzen.}
var lauf, vorMax:tRefElement;
begin
  {Wir betrachten zunächst das zweite Listenelement als Maximum,
   also das erste Element als dessen Vorgänger.}
  vorMax := inRefAnfang;

  {Nun werden alle weiteren Elemente überprüft und der vorMax-
   Zeiger aktualisiert, falls ein größeres Element gefunden wird}
  lauf := inRefAnfang^.next;
  while lauf <> inRefAnfang do
  begin
    if lauf^.next^.info > vorMax^.next^.info then
      vorMax := lauf;
    lauf := lauf^.next;
  end;
  {vorMax zeigt nun auf den Vorgänger des Maximums.}
  vorMaximum := vorMax;
end;
```

---

b)

```
procedure loescheMaximum(var ioRefAnfang: tRefElement);
{Löscht das Listenelement mit dem größten Wert aus der Liste.
Vorbedingung: Die Liste muss mindestens zwei Elemente besitzen.}
var vorMax, loesch: tRefElement;
begin
  vorMax := vorMaximum(ioRefAnfang);
  {Sonderfall: Falls das erste Listenelement gelöscht werden
   soll, wird das bisher zweite zum neuen Listenanfang:}
  if vorMax^.next = ioRefAnfang then
    ioRefAnfang := ioRefAnfang^.next;

  {Jetzt Maximum löschen}
  loesch := vorMax^.next;
  vorMax^.next := loesch^.next;
  dispose(loesch);
end;
```

Aufgabe 4

a) Ein leerer Baum enthält keinen Knoten mit irgendeinem Suchwert. Ein nicht-leerer Baum mit dem Suchwert in der Wurzel enthält den Suchwert. Für nicht-leere Bäume, die den Suchwert nicht in der Wurzel enthalten, ist derjenige Teilbaum zu bestimmen, der den Wert enthalten könnte (aufgrund der Sortierung des Suchbaumes kann dies nur einer der beiden Teilbäume sein). Der Suchwert kommt im Baum vor, falls er in dem ausgewählten Teilbaum vorkommt.

```
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) Da zum Suchen in einem Suchbaum lediglich ein Pfad durchlaufen wird, also immer nur Referenzen gefolgt und niemals zu Vorgängerknoten zurückgesprungen wird, 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 5

a) Bei der Ausführung des Tests mit dem Testdatum ([0, 2, 3, 4], true) ∈ T3 werden zwar alle Anweisungen überdeckt, jedoch wird die Funktion tatsächlich den Wert false ausgeben, der vom laut Testdatum erwarteten Ergebnis true abweicht. Es wird also ein Fehler in der Funktion aufgedeckt. Der Anweisungsüberdeckungsgrad von 100% sagt offensichtlich nichts über die Fehlerfreiheit aus.
Allgemeiner kann man festhalten, dass dei der Testausführung nicht nur auf die jeweiligen Übderdeckungsmaße, sondern natürlich auch auf das tatsächliche Ergebnis im Vergleich zum in den Testdaten angegebenen (und aus der Spezifikation abgeleiteten) erwarteten Ergebnis zu achten ist und klar erkennbare Programmierfehler nicht einfach ignoriert werden. Die Überdeckungsmaße bieten sich vor allem als Testendekriterien an.

b) Es gibt nur zwei Verzweigungen im Kontrollfluss, die zu überwachen sind:
(1) Die If-Anweisung innerhalb der Schleife: Hier gibt es einen Then- und einen (leeren) Else-Zweig, die beide zu überdecken sind, wozu also mindestens zwei Schleifendurchläufe oder zwei Testdaten benötigt werden.
(2) Die Abbruchbedingung der Repeat-Schleife: Hier kann die Schleife wahlweise verlassen oder wiederholt werden. Um beide Zweige abzudecken, muss die Schleife also mindestens zweimal durchlaufen werden (und darf nicht endlos sein).

Nur T1 führt *nicht* zu einer vollständigen Zweigüberdeckung:

Bei Eingabe des Arrays [2, 0, 4, 5] wird die Repeat-Schleife ein einziges Mal durchlaufen. Weder der Zweig zur Schleifenwiederholung (siehe (2)) noch der Else-Zweig (siehe (1)) werden überdeckt.

Bei T2 und T3 werden sämtliche Zweige überdeckt: Alle Testdaten aus T2 und T3 führen zu mindestens einem zweiten Schleifendurchlauf. Die beiden alternativen Zweige nach der If-Anweisung werden beim einzigen Testdatum aus T2 beide durchlaufen, bei T3 sorgt jedes Testdatum für die Überdeckung eines der beiden Zweige.

c) Es gibt drei einfache (atomare) Bedingungen, die zu überdecken sind:
(1) `inFeld[i] = 0` (If-Bedingung)
(2) `gefunden` (Teil der Until-Bedingung)
(3) `i = FELDMAX` (Teil der Until-Bedingung)

T1 und T2 überdecken *nicht* sämtliche dieser drei Bedingungen, lediglich T3 führt zu einer vollständigen Bedingungsüberdeckung:

Bei Eingabe des einzigen Testdatums aus T1 wird bereits Bedingung (1) nur ein einziges Mal ausgewertet, und zwar zu true, also niemals zu false.

Bei Eingabe des einzigen Testdatums aus T2 wird Bedingung (1) zwar überdeckt, und auch Bedingung (2) ist nach dem ersten Schleifendurchlauf false, nach dem zweiten true, wird also überdeckt. Bedingung (3) wird dagegen niemals true, da die Schleife immer schon vor Erreichen des Feldendes abbricht.

In T3 sorgt das erste Testdatum dafür, dass Bedingung (1) zu true ausgewertet wird und die Schleife vorzeitig aufgrund von Bedingung (2) abbricht, während mit dem zweiten Testdatum die Bedingung (2) immer zu false ausgewertet wird und die Schleife das ganze Feld durchläuft und dann aufgrund von Bedingung (3) abbricht. Insgesamt werden alle drei Bedingungen überdeckt.

d) Nein: Das Überdecken der atomaren Bedingungen stellt nicht sicher, dass jedes aus diesen zusammengesetzte Prädikat, das über eine Verzweigung im Kontrollfluss entscheidet, auch überdeckt wird.
Betrachtet man z.B. die folgende If-Anweisung
`if (A or B) then ...`
wobei A und B jeweils eine atomare Bedingung darstellen, so ist es möglich, in einem Testlauf A zu true und B zu false auszuwerten und in einem zweiten Testlauf A zu false und B zu true, so dass die Bedingungen A und B vollständig überdeckt wurden, jedoch die If-Bedingung immer erfüllt ist, der else-Zweig also unüberdeckt bleibt.