Kurs 1613 "Einführung in die imperative Programmierung"

Klausur am 04.03.2000

Aufgabe 1 (6+5+3+4= 18 **Punkte)**

In der Mathematik gibt es die sogenannten Armstrong-Zahlen, die wie folgt definiert werden:

Eine Armstrong-Zahl ist eine Zahl, bei der die Summe der einzelnen Ziffern, die jeweils mit der Anzahl der Ziffern der Zahl potenziert sind, gleich der Zahl ist.

z.B.: $153 = 1^3 + 5^3 + 3^3$ (3 Ziffern je hoch 3)

a) Schreiben Sie eine Funktion countZiff, welche die Anzahl der Ziffern einer positiven Zahl zurückliefert (in unserem Beispiel die Zahl 3). Benutzen Sie den im Programmrahmen ArmstrongZahlen (siehe Seite 3) vorgegebenen Funktionskopf.

b) Schreiben Sie eine Funktion potenziere. Der Sonderfall des Undefinierten Wertes 0° muss nicht berücksichtigt werden. Die Funktion potenziere soll die n-te Potenz einer Zahl berechnen. Benutzen und ergänzen Sie den vorgegebenen Funktionskopf (siehe Seite 3).

c) Die Funktion isArmstrong (siehe unten) gibt true zurück wenn der übergebene Parameter eine Armstrong-Zahl ist, sonst false. In dieser Funktion werden die Funktionen countZiff und potenziere benutzt.

Bestimmen Sie, an welcher der Stellen (1) - (4) in der Funktion isArmstrong die Funktion potenziere sinnvoll angewandt wird und geben Sie die komplette Programmzeile an.

d) Schreiben Sie ein Programm, das zwei positive ganze Zahlen x und y (x > 0, y < MAXINT, x < y) einliest und dann alle Armstrong-Zahlen mit x ^ a < y ausgibt. Hinweis: Auch wenn Sie die Teilaufgaben a) - c) nicht gelöst haben, können Sie die Funktionen voraussetzen. Nutzen Sie dazu den vorgegebenen Programmrahmen.

```
function isArmstrong (inZahl : tNatZahl): boolean;
{prüft ob inZahl eine Armstrong-Zahl ist}
  var
  i,
  j,
  ggfArmstrong,
  AnzDerStellen
  ZwischErg,
  EndErg,
  Letzteziffer : tNatZahl;
begin
    ggfArmstrong := inZahl;
    AnzDerStellen := countZiff(inZahl);
    EndErg := 0;
    for i := l to AnzDerStellen do
    begin
      (1)
      Letzteziffer := ggfAnnstrong mod 10;
      (2)
```

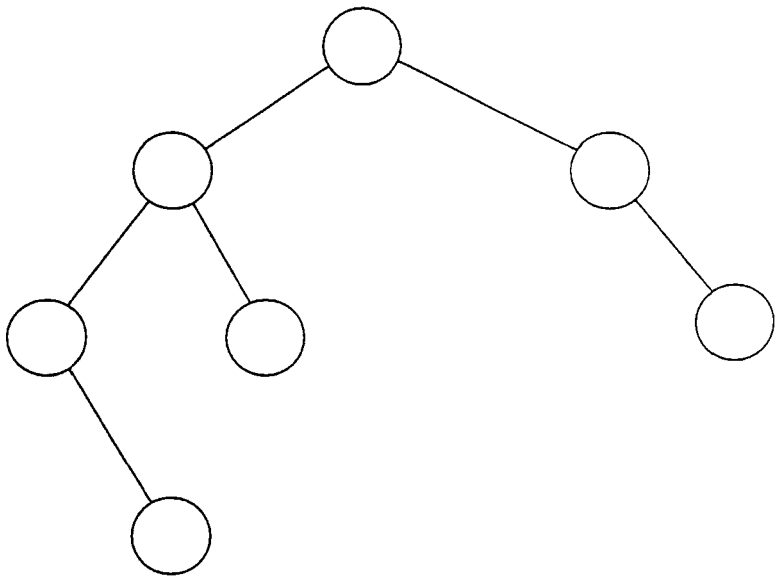
```
        ggfArmstrong := ggfAnnstrong div 10;
        (3)
        EndErg := EndErg + ZwischErg
        (4)
    end;

    if EndErg = inZahl then
    isArmstrong := true eise

    isArmstrong := false;
end;
program ArmstrongZahlen (input, Output) ;
  type
  tNatZahl = 0..maxint;
  var
  function countZiff (inZahl : tNatZahl): tNatZahl;
  {gibt die Stellenanzahl der inZahl}
  function potenziere ( ??? ): tNatZahl;
  {berechnet die n-te Potenz einer Zahl}
  function isArmstrong (inZahl : tNatZahl): boolean;
  {prüft ob inZahl eine Armstrong-Zahl ist}
```

Aufgabe 2 (10 + 4 = 14 Punkte)

Gegeben sei ein (nicht sortierter) binärer Baum und ein Zeiger (inRefWurzel) auf die Wurzel des Baumes. Die Funktion countLeaf soll die Anzahl der Blätter in dem Baum bestimmen. Wird der Funktion countLeaf ein Zeiger auf die Wurzel des Baumes der folgenden Abbildung übergeben, so liefert sie als Ergebnis den Wert 3. Im Fall eines leeren Baumes wird der Wert 0 zurückgegeben.



a) Ihre Aufgabe ist es, die Funktion countLeaf zu implementieren. Gehen Sie dabei von den unten gegebenen Typdefinitionen und der Funktionsdeklaration aus. Wählen Sie eine rekursive Lösung.

b) Zeichnen Sie für die Menge (36, 11, 18, 3, 37, 25, 8, 21) einen degenerierten und einen vollständigen Suchbaum.

```
type
   tNatZahl = 0..maxint ;
   tRefBinBaum = ^tBinBaum;
        tBinBaum = record
        info : tNatZahl;
        links,
        rechts : tRefBinBaum
        end;
   function countLeaf(??): ??;
   { ?? }
```

Aufgabe 3 (9 + 9 = 18 Punkte)

a) Schreiben Sie eine PASCAL-Prozedur Zahlenliste, welche eine nicht-leere Folge von in-teger-Zahlen mit **readln** einliest und in einer linearen Liste abspeichert. Die Reihenfolge der Zahlen in der Liste soll der Reihenfolge der Eingabe entsprechen, d.h. es muss jeweils am Ende der Liste angefügt werden. Die Eingabe einer Null, die nicht in der Liste abgespeichert wird, schließt die Eingabe ab. Benutzen Sie folgende Typdefinitionen und den Prozedurkopf von Zahlenliste:

```
type tRefListe = ^tListe;
tListe = record
            info : integer;
            next : tRefListe end;
procedure Zahlenliste (var outListe : tRefListe) ;
{ liest integer-Zahlen von der Tastatur ein und speichert
sie in der Reihenfolge der Eingabe in der linearen Liste
outListe; Null beendet die Eingabe }
```

b) Schreiben Sie nun ein Programm Zahleninversion, das unter Benutzung der Prozedur Zahlenliste eine Liste mit Zahlen aufbaut und dann die Zahlen der Liste von hinten nach vorne ausgibt. Verwenden Sie für die Ausgabe die **unit** Stapel, deren Interface unten angegeben ist, wie folgt: Zuerst sind die in der Liste gespeicherten Zahlen auf dem Stapel abzulegen, dann werden die Zahlen vom Stapel geholt und ausgegeben. (Haben Sie Teil a) nicht bearbeitet, können Sie die Prozedur Zahlenliste als entsprechend vereinbart annehmen.)

```
unit Stapel;
{ ein Stapel zur Verwaltung von integer-Zahlen }
interface
   function isEmpty : boolean;
   { liefert true, wenn der Stapel leer ist, sonst false }
   procedure top (var outElem : integer) ;
   { liefert das oberste Stapelelement }
   procedure push (inElem : integer);
   { stapelt ein neues Element }
   procedure pop ;
```

{ entfernt das oberste Stapelelement }

implementation

... { hier nicht relevant }

end. { Unit Stapel }

Zusammenfassung der Muß-Regeln

1. Selbstdefinierte Konstantenbezeichner bestehen nur aus Großbuchstaben. Bezeichner von Standardkonstanten wie z.B. maxint sind also ausgenommen
2. Typbezeichnem wird ein t vorangestellt. Bezeichner von Zeigertypen beginnen mit tRef. Bezeichner formaler Parameter beginnen mit in, i o oder out.
3. Jede Anweisung beginnt in einer neuen Zeile; **begin** und end stehen jeweils in einer eigenen Zeile
4. Anweisungsfolgen werden zwischen **begin** und **end** um eine konstante Anzahl von 2-4 Stellen eingerückt, **begin** und **end** stehen linksbündig unter der zugehörigen Kontrollanweisung, sie werden nicht weiter eingerückt.
5. Anweisungsteile von Kontrollanweisungen werden genauso eingerückt.
6. Im Programmkopf wird die Aufgabe beschrieben, die das Programm löst. -
7. Jeder Funktions- und Prozedurkopf enthält eine knappe Aufgabenbeschreibung als Kommentar. Ggf. werden zusätzlich die Parameter kommentiert.
8. Die Parameter werden sortiert nach der Übergabeart: Eingangs-, Änderungs- und Ausgangsparameter.
9. Die Übergabeart jedes Parameters wird durch Voranstellen von in, io oder out vor den Parameternamen gekennzeichnet.
10. Das Layout von Funktionen und Prozeduren entspricht dem von Programmen.
11. Jede von einer Funktion oder Prozedur benutzte bzw. manipulierte Variable wird als Parameter übergeben. Es werden keine globalen Variablen manipuliert. Einzige Ausnahme sind Modul-lokale Variablen, die in den Parameterlisten der exportierten Prozeduren und Funktionen des Moduls nicht auftauchen, selbst wenn sie von diesen geändert werden.
12. Jeder nicht von der Prozedur veränderte Parameter wird als Wertparameter übergeben. Lediglich Felder können auch anstatt als Wertparameter als Referenzparameter übergeben werden, um den Speicherplatz für die Kopie und den Kopiervorgang zu sparen. Der Feldbezeichner beginnt aber stets mit dem Präfix in, wenn das Feld nicht verändert wird.
13. Funktionsprozeduren werden wie Funktionen im mathematischen Sinne benutzt, d.h. sie besitzen nur Wertparameter. Wie bei Prozeduren ist eine Ausnahme nur bei Feldern erlaubt, um zusätzlichen Speicherplatz und Kopieraufwand zu vermeiden.
14. Wertparameter werden nicht als lokale Variable mißbraucht.
15. Die Schlüsselworte unit, interface und implementation werden ebenso wie begin und end des Initialisierungsteils linksbündig positioniert. Nach dem Schlüsselwort unit folgt ein Kommentar, der die Aufgabe beschreibt, welche die Unit löst.
16. Für die Schnittstelle gelten dieselben Programmierstilregeln wie für ein Programm. Dies betrifft Layout und Kommentare. Nach dem Schlüsselwort interface folgt im Normalfall kein Kommentar.
17. Für den Implementationsteil gelten dieselben Programmierstilregeln wie für ein Programm. Nach dem Schlüsselwort **implementation** folgt nur dann ein Kommentar, wenn die Realisierung einer Erläuterung bedarf (z.B. wegen komplizierter Datenstrukturen und/oder Algorithmen).
18. In Programmen oder Moduln, die andere Moduln importieren („benutzen"), wird das Schlüsselwort **uses** auf dieselbe Position eingerückt wie die Schlüsselworte const, type, var usw.
19. Die Laufvariable wird innerhalb einer for-Anweisung nicht manipuliert.
20. Die Grundsätze der strukturierten Programmierung sind strikt zu befolgen.