Klausur zu Programmieren II

SS 95

15. Juli 1995

Name ________________________________

Vorname ________________________________

Matrikel-Nr. ________________________________

Aufgabe		Punkte	Bewertung
1	a)	3	
	b)	3	
	c)	3	
	d)	3	
2	a)	4	
	b)	2	
	c)	3	
3		10	
4		6	
5		10	
6	a1)	3	
	a2)	3	
	b1)	6	
	b2)	6	
7	a)	8	
	b1)	3	
	b2)	5	
8		8	
Summe		**88**	

☞ Bearbeitungszeit: 180 Min.

Aufgabe 1

a) Was versteht man unter **Information Hiding** (Idee, Vorteil)?

Nur Schnittstelle, das WAS sichtbar machen, Realisierung oder implementierungstechnische Details verbergen

keine versteckten Abhängigkeiten, klare Schnittstellen, höhere Flexibilität und Änderungsfreundlichkeit

b) Beschreiben Sie kurz die Idee der **Abstrakten Datentypen** und zeichnen Sie eine Realisierungsmöglichkeit in Modula-2 auf!

Vom Typ einer Datenstruktur werden nur der Name und die Zugriffsoperationen bekanntgegeben, strukturelle Informationen werden verborgen

Modul-Konzept und hidden types

c) Was versteht man in der UNIX-Welt unter **Filter**?

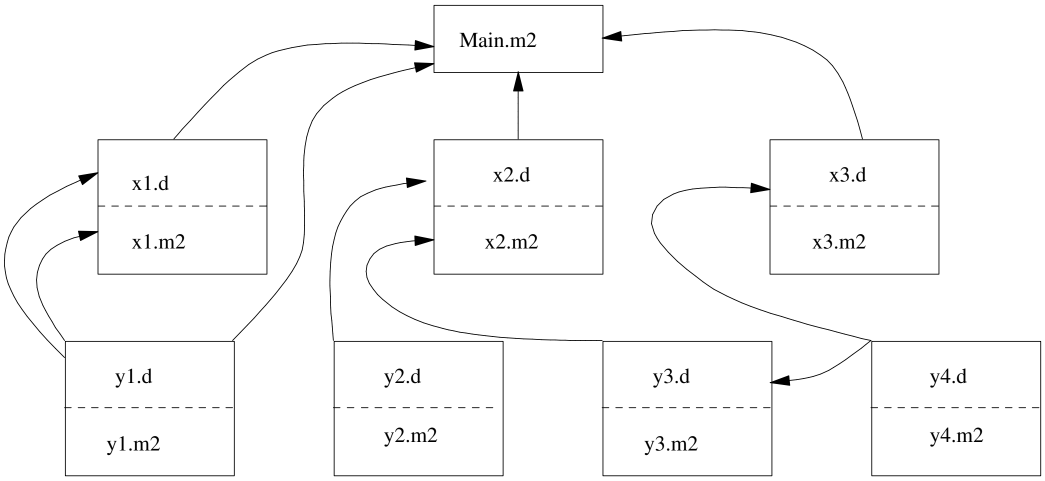
Ein Programm, das von der Standardeingabe liest, Inhalt verarbeitet und Ergebnis an die Standardausgabe weiterleitet - Einsatz in Pipes

d) Nennen Sie wenigstens einen plausiblen Grund für die Trennung der Ausgabe eines Programms unter UNIX in **stdin** und **stderr**!

Zur Realisierung von Filter-Programmen - Fehlermeldungen können vom folgenden Programm nicht sinnvoll verarbeitet werden, Möglichkeit, eigentliches Ergebnis und Fehlermeldungen zu trennen (z.B. in verschiedenen Dateien zu schreiben

Aufgabe 2

Gegeben sei folgende Programmstruktur:



Anm.: Die Pfeile symbolisieren die IMPORT-Beziehung!

a) Geben Sie eine mögliche Übersetzungsreihenfolge an!

y1.d y2.d y4.d y3.d y1.m2 y2.m2 y3.m2 y4.m2
x1.d x1.m2 x2.d x2.m2 x3.d x3.m2
Main.m2

b) Was enthält das sogenannte *makefile* (nur das Prinzipielle, nicht den syntaktischen Inhalt)?

Die durch die Import-Beziehungen definierten Abhängigkeiten, Compiler-Aufrufe (in einer nach topsort definierten Reihenfolge) mit den richtigen Optionen

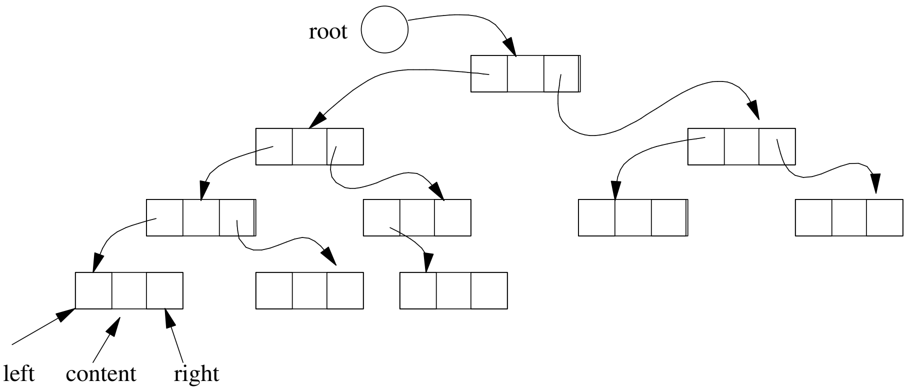
c) Welchen Vorteil bietet die Verwendung von *make*?

make prüft anhand der Zeitstempel, welche Teile seit der letzten Übersetzung verändert wurde und führt all das der Übersetzung zu, was aufgrund dessen neu übersetzt werden muß (incl. der Module, die von geänderten abhängig sind

Aufgabe 3

Ein nach inorder sortierter Binär-Baum sei wie folgt vereinbart:

```
TYPE
    Tree = POINTER TO Node;
    Node = RECORD
            content: CARDINAL;
            left,right: Tree;
           END;
```



Die Höhe eines Knotens ist die Länge des Weges von der Wurzel zu diesem Knoten (die Wurzel hat also die Höhe 0).

Aufgabe: Schreiben Sie eine rekursive Prozedur

```
PROCEDURE levelPrint(root: Tree; level:CARDINAL)
```

die den Inhalt (*content*) aller Knoten mit Höhe = *level* absteigend nach *stdout* ausgibt (sie können davon ausgehen, daß jeder Pointer einen definierten Wert hat)

```
PROCEDURE levelPrint(wurzel:Baum; level: CARDINAL);
(*levelOrder Traversierung, Inhalte der Knoten
  mit Hoehe "level" ausgeben; *)
BEGIN
    IF wurzel = NIL THEN RETURN END;
    IF level = 0 THEN
        WriteCard(wurzel^.inhalt, 6);
    END;
    IF (wurzel^.rechts <> NIL) & (level > 0) THEN
        levelPrint(wurzel^.rechts, level-1)
    END;
    IF (wurzel^.links <> NIL) & (level > 0) THEN
        levelPrint(wurzel^.links,level-1)
    END;
END levelPrint;
```

Aufgabe 4

Für einen Rechner mit Dateiblockgröße 1kByte will jemand eine B-Baum-Verwaltung realisieren. Die abzulegenden Daten entstammen folgender Typ-Vereinbarung:

```
TYPE
    DataRec = RECORD
                key: CARDINAL;
                name: ARRAY[0..19] OF CHAR;
                married: BOOLEAN
          END
```

Bestimmen Sie für diesen "Jemand" den Typ **k** des B-Baumes - sollten Sie dazu Annahmen zugrundelegen, geben Sie diese unbedingt an!

Annahmen:

- eine Cardinal umfaßt 4 Bytes
- eine BOOLEAN umfaßt 2 Bytes
- eine CHAR umfaßt 1 Byte, also das ARRAY 20 Byte
- ein Pointer umfaßt 4 Bytes

also

2*k*(4 + 4 + 2 + 20) + 4 = 1024

2*k*30 = 1020

k = 17

Aufgabe 5

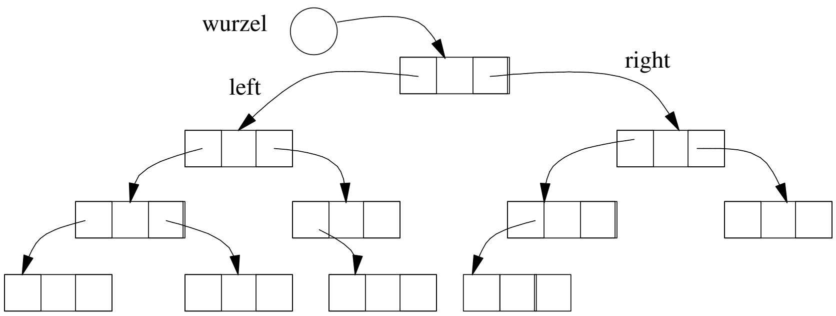
Gegeben sei folgender Auszug aus einem Modul:

```
TYPE
    Tree = POINTER TO NODE;
    Node = RECORD
        content: CARDINAL;
        left, right: Tree
           END;

    Direction = (l,r);
CONST   MaxLevel = 3;

PROCEDURE oha(   VAR root: Tree;
        dir: Direction;
        level: CARDINAL );
BEGIN
    IF root = NIL THEN RETURN END;
    IF (level < MaxLevel)
          & (root^.left <> NIL)
          & (root^.right <> NIL) THEN
       IF dir = l THEN
        oha(root^.right,r,level+1)
       ELSE
        oha(root^.left,l,level+1)
       END
    ELSE
       IF dir = l THEN
        root := root^.right
       ELSE
        root := root^.left
       END
    END
END oha;
```

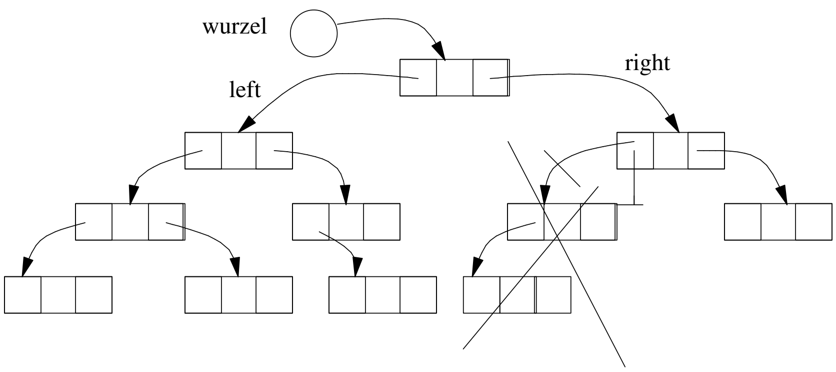
Für folgenden Baum wird die Prozedur mit `oha(wurzel,l,1) aufgerufen:`



(Alle "Pointer-Kästchen", von denen kein Pfeil weggeht, sind mit dem NIL-Pointer gesetzt)

Was bewirkt dieser Aufruf:

- Er verändert die Struktur des Baumes (Bitte in Skizze eintragen WIE! Was ist in der Prozedur dafür verantwortlich?)



Verantwortlich ist die Übergabe von *root* als VAR-Parameter

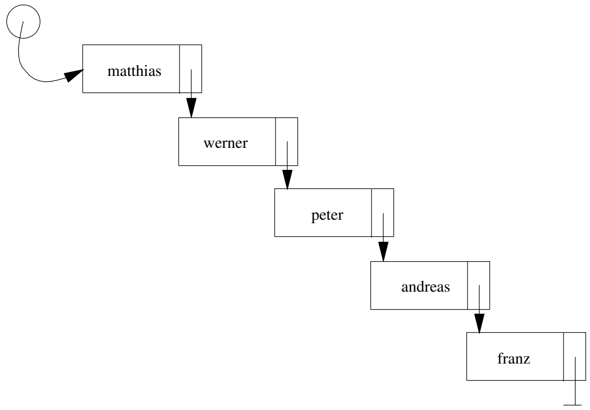
Aufgabe 6

Gegeben sei folgende Typ-Vereinbarung:

```
TYPE Element = ARRAY [0..MAX] OF CHAR
```

a) Geben Sie eine dynamische Datenstruktur an für die Repräsentation einer Menge von Elementen vom Typ `Element`, und zwar

a1) einmal in grafischer Darstellung für die konkrete Menge M = {"matthias", "werner", "peter", "andreas", "franz"}



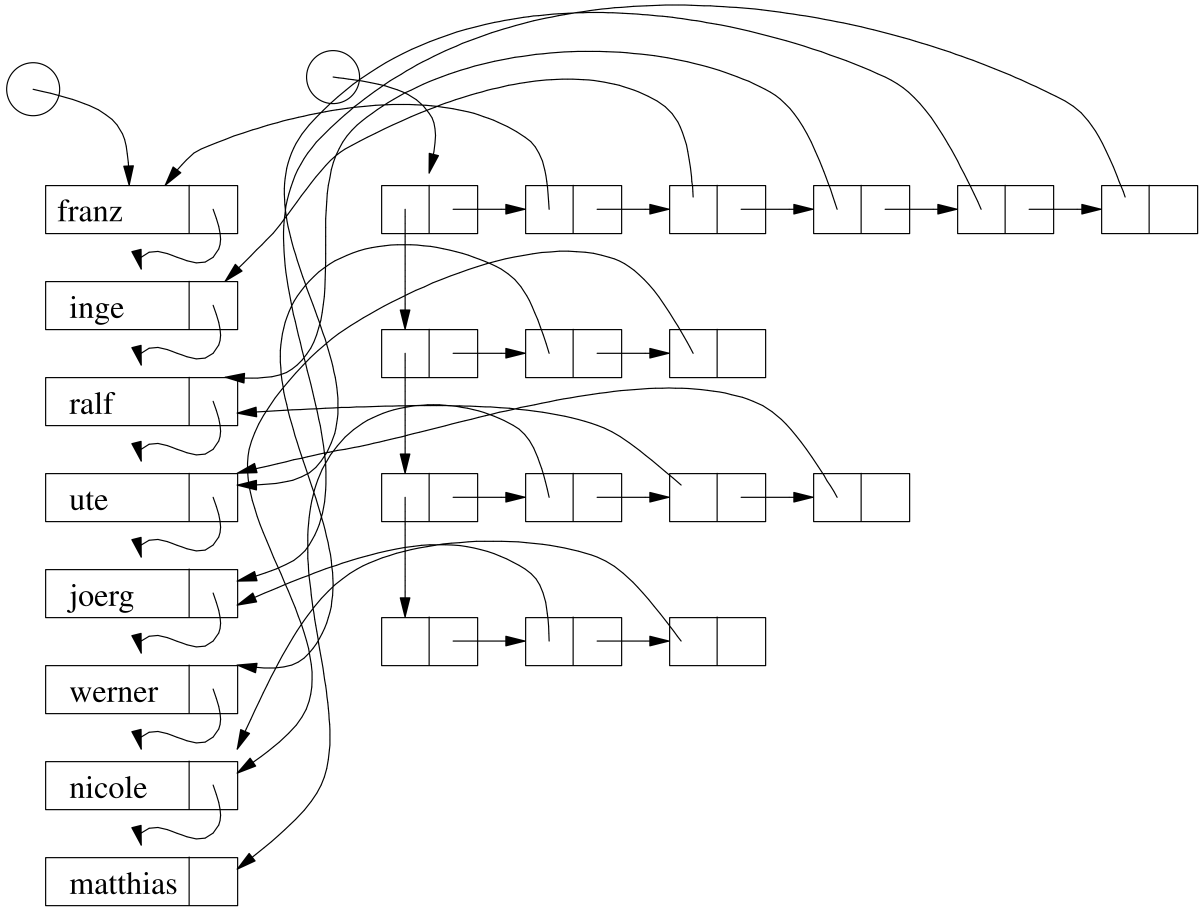
a2) und zum anderen als Typ-Definition(en) in Modula-2

```
TYPE
   Liste = POINTER TO Node;
   Node = RECORD
   entry: Element;
   next: Liste
         END;
```

b) Geben Sie eine dynamische Datenstruktur für die Repräsentation einer Menge von Mengen von Elementen vom Typ `Element`, so daß jedes elementare Element (vom Typ `Element`) nur einmal gespeichert wird, und zwar

b1) einmal in grafischer Darstellung für die konkrete Menge

```
K = { {"franz", "inge", "ralf", "ute", "joerg"},
      {"werner", "nicole"},
      {"matthias", "ralf", "ute"},
      {"joerg", "nicole"}
    }
```



b2) und zum anderen als Typ-Definition(en) in Modula-2

```
TYPE
   Name = POINTER TO NameNode;
   NameNode = RECORD
              name: Element;
              next: Name;
             END;
   SetList = POINTER TO SetNode;
   SetNode = RECORD
              el: Name;
              next: SetList;
            END;
```

```
SetSetList = POINTER TO SetSetNode;
SetSetNode = RECORD
        set: SetList;
        next: SetSetList;
         END;
```

Aufgabe 7

Irgend jemand möchte ein Programm, das die Gewichtsausgeglichenheit eines sortierten Binärbaumes prüfen soll. **Sie** sollen ihm dazu eine Hilfsprozedur liefern, die zu einem Baum, gegeben durch einen Zeiger auf dessen Wurzel, die Zahl der Knoten (incl. Wurzel) liefert. Da er ein Fan der Rekursion ist, verlangt er von Ihnen auch die **Verwendung der Rekursion**.

Folgende Typvereinbarungen legt er zugrunde:

```
TYPE
    Tree = POINTER TO Node;
    Node = RECORD
             left,right: Tree
           END;
```

a) Er will zunächst eine Funktionsprozedur mit folgender Vereinbarung:

```
PROCEDURE count1(r: Tree): CARDINAL;
```

```
PROCEDURE count1(r: Baum): CARDINAL;
BEGIN
    IF r = NIL THEN
        RETURN 0
    ELSE
        RETURN (1+count1(r^.rechts) + count1(r^.links))
    END
END count1;
```

b) Aus welchem Grund auch immer will er von Ihnen das gleiche nocheinmal, allerdings nicht als Funktionsprozedur (sondern als "normale" Prozedur), sie dürfen aber auch keine globalen Variablen verwenden.

b1) Geben Sie den Prozedurkopf für `count2`!

```
PROCEDURE count2(r: Baum; VAR c: CARDINAL);
```

b2) Formulieren Sie die Prozedur aus!

```
PROCEDURE count2(r: Baum; VAR c: CARDINAL);
   VAR cl,cr: CARDINAL;
BEGIN
   IF r=NIL THEN
   c := 0;
   ELSE
   count2(r^.links, cl);
   count2(r^.rechts, cr);
   c := 1+cl + cr;
   END
END count2;
```

Aufgabe 8

Gegeben sei folgendes Programm:

```
MODULE Passage;
    IMPORT InOut;
    IMPORT Storage;
    IMPORT SYSTEM;

    TYPE NodePointer = POINTER TO Node;
        Node = RECORD
                data: CHAR;
                next: NodePointer;
                  END;

    VAR currentPtr: NodePointer;

    PROCEDURE change(tempPtr: NodePointer);
    BEGIN
        tempPtr^.data := "C";
        tempPtr := tempPtr^.next;
        tempPtr^.data := "D";
    END change;
BEGIN
    Storage.ALLOCATE(currentPtr,SYSTEM.TSIZE(Node));
    currentPtr^.data := "A";
    Storage.ALLOCATE(currentPtr^.next,SYSTEM.TSIZE(Node));
    currentPtr^.next^.data := "B";
    InOut.Write(currentPtr^.data);
    InOut.Write(currentPtr^.next^.data);

    change(currentPtr);

    InOut.Write(currentPtr^.data);
    InOut.Write(currentPtr^.next^.data);
END Passage.
```

a) Was gibt dieses Programm aus?

ABCD

b) Was verändert sich in der Ausgabe und im Programmverhalten, wenn die Prozedur *change* wie folgt verändert wird:

```
PROCEDURE change(VAR tempPtr: NodePointer)
```

ABD und dann *segmentation fault (Core dumped)*