Informatik 2

Prüfung im Sommersemester 2004

Name, Vorname: .....

Matrikelnummer: .....

Hochschule Reutlingen - Reutlingen University

Fachbereich:	Informatik
Bachelor-Studiengang/Semester:	Medien- und Kommunikationsinformatik 2
Prüfer:	Prof. Dr. Karlheinz Hug
Prüfungstermin (Datum, Uhrzeit) und Ort:	Mi 7. 7. 2004, 10^{30}-12^{30} Uhr, Aula
Prüfungsdauer:	120 Minuten
Zugelassene Hilfsmittel:	Alle (Vorlesungsmitschrift, Literatur,...)
Anzahl der Aufgabenseiten:	10

Aufgabe	**Punkte**		**Aufgabe**	**Punkte**	
	möglich	**erreicht**		**möglich**	**erreicht**
1	10		6	8	
2	20		7	18	
3	33		8	12	
4	19		9	16	
5	16				
Summe möglich:		**152**	**Summe erreicht:**		

- Zum Bestehen der Prüfung sind 60, für die Note „Eins“ 120 Punkte erforderlich.
- Bearbeiten Sie die Aufgaben möglichst auf dazu freigelassenen Stellen im Text, falls nichts Anderes empfohlen ist!
- Teilaufgaben sind meist unabhängig voneinander lösbar.
- Geben Sie alle Aufgaben- und Lösungsblätter ab!

✌ **Viel Erfolg!**

Aufgabe 1: Programmablaufverfolgung bei Rekursion (Punkte: 10 |.....)

Verfolgen Sie die Ausführung der folgenden rekursiven Funktion bei den unten gegebenen Werten von row; tragen Sie die Werte der aktuellen Parameter, der lokalen Variablen und die von den Funktionsaufrufen gelieferten Ergebniswerte in die Tabelle ein!

```
VAR row : ARRAY 16 OF INTEGER;

PROCEDURE HasInRange (x, le, ri : INTEGER) : BOOLEAN;

   VAR m : INTEGER;

BEGIN
   ASSERT ((0 <= le) & (ri < LEN (row)), BEC.precondition);
   IF le > ri THEN
      RETURN FALSE
   ELSE
      m := (le + ri) DIV 2;
      IF x < row [m] THEN
         RETURN HasInRange (x, le, m - 1)
      ELSIF x > row [m] THEN
         RETURN HasInRange (x, m + 1, ri)
      ELSE
         RETURN TRUE
      END
   END
END HasInRange;
```

row															
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
2	5	11	14	23	36	47	49	50	51	55	61	67	73	89	99

Aufruffolge	**Werte der aktuellen Parameter**			**Wert von**	**Ergebniswert**
HasInRange	x	le	ri	m	result
1. Aufruf	23	0	15		
2. Aufruf					
3. Aufruf					
4. Aufruf					

Beschreiben Sie kurz, was die Funktion liefert!

Aufgabe 2: Mit Keller oder mit Rekursion (Punkte: 20 |.....)

Implementieren Sie die Prozedur

```
PROCEDURE Reverse (VAR str : ARRAY OF CHAR);
```

die die eingegebene Zeichenkette str umkehrt, z.B. 'Reverse' in 'esreveR' wandelt, mit

(1) einem lokalen Objekt der bekannten **Kellerklasse** mit der Schnittstelle:

```
StackDesc = RECORD
   (VAR s : StackDesc) Init, NEW;
   (IN s : StackDesc) IsEmpty () : BOOLEAN, NEW;
   (IN s : StackDesc) Item () : CHAR, NEW;
   (VAR s : StackDesc) Put (x : CHAR), NEW;
   (VAR s : StackDesc) Remove, NEW;
END;
```

(2) einer **rekursiven Prozedur**:

```
PROCEDURE Rec (VAR str : ARRAY OF CHAR; le, ri : INTEGER);
```

Aufgabe 3: Parser mit rekursivem Abstieg (Punkte: 33 |.....)

Zur Sprache, deren Syntax durch die EBNF-Regeln

```
Expr    = [ "+" | "-" ] Term { ( "+" | "-" ) Term }.
Term    = Factor { ( " * " | "/" ) Factor }.
Factor  = Ident | "(" Expr ")".
Ident   = letter { letter | digit }.
```

gegeben ist, ist ein Parser mit rekursivem Abstieg zu entwerfen. Benennen Sie die benötigten **Prozeduren**, stellen Sie ihre **Aufrufbeziehungen** in einem Diagramm dar und skizzieren Sie die **Algorithmen** der Prozeduren (bitte auf Extrablatt)!

Aufgabe 4: Zeiger (Punkte: 19 |.....)

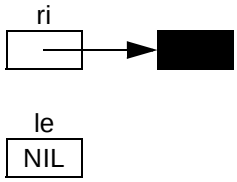
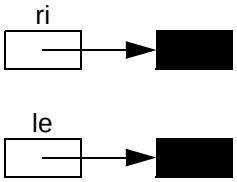
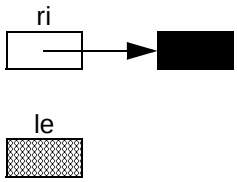
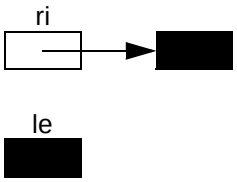
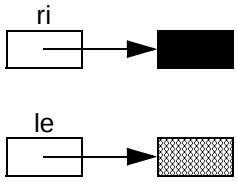
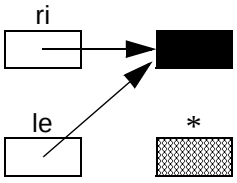
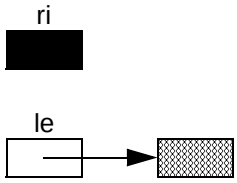
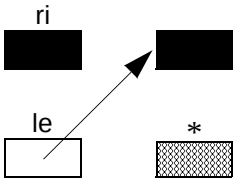
Geben Sie unter der Voraussetzung der Vereinbarungen

```
TYPE  A     = POINTER TO ADesc;
      ADesc = RECORD i : INTEGER; END;

VAR   sD, tD : ADesc;   s, t : A;
```

zu jeder der folgenden Zuweisungen an, unter welcher Bedingung sie korrekt oder warum sie falsch ist!

Zuweisung	korrekt, falls...	falsch, weil...
sD^ := tD		
s := tD		
s := t^		
s^ := tD		
sD.i := tD		
sD.i := t		
sD.i := tD.i		
sD.i := t.i		

Geben Sie zu jedem der folgenden Diagramme den Typ der Variablen le, ri (ADesc oder A) an und Anweisungen, die den Vorzustand in den Nachzustand überführen!

Typ von		**Vorzustand**	**Anweisungen**	**Nachzustand**
le	ri			
				
				
				
				

Was passiert mit den mit * markierten Objekten?

Aufgabe 5: Vererbung, Polymorphie (Punkte: 16 |.....)

```
TYPE  A = POINTER TO ABSTRACT RECORD END;
      B = POINTER TO RECORD (A) END;
      C = POINTER TO ABSTRACT RECORD (A) END;
      D = POINTER TO RECORD (C) END;

PROCEDURE (a : A) DoA, NEW, ABSTRACT;

PROCEDURE (b : B) DoA; BEGIN Out.String ('DoA of B') END DoA;

PROCEDURE (b : B) DoB; BEGIN Out.String ('DoB of B') END DoB;

PROCEDURE (c : C) DoA, EXTENSIBLE; BEGIN Out.String ('DoA of C') END DoA;

PROCEDURE (c : C) DoC, NEW, ABSTRACT;

PROCEDURE (d : D) DoA; BEGIN Out.String ('DoA of D') END DoA;

PROCEDURE (d : D) DoC; BEGIN Out.String ('DoC of D') END DoC;
```

Zeichnen Sie zu den Vereinbarungen ein **Klassendiagramm** und geben Sie an, welche Klassen die **Rollen** *Konzept-*, *Schnittstellen-* und *Implementationsklasse* spielen!

Die Vereinbarungen `a : A; b : B; c : C; d : D` vorausgesetzt, geben Sie zu jeder der folgenden Anweisungsfolgen an: Solange die Anweisungen korrekt sind, ihren Effekt (erzeugte Objekte, Ausgabetext); bei fehlerhaften Anweisungen den Grund des Fehlers.

Anweisungsfolge	**Effekt im Speicher, Ausgabe im Log, Fehlerursache**
`NEW (a);` `a.DoA`	

`NEW (b);`
`a := b;`
`a.DoA`	

`NEW (b);`
`a := b;`
`a.DoB`	

`NEW (d);`
`d.DoA;`
`d.DoC`	

`NEW (d);`
`c := d;`
`c.DoA;`
`c.DoC`	

Aufgabe 6: Dynamische Objektstruktur lineare Liste (Punkte: 8 |.....)

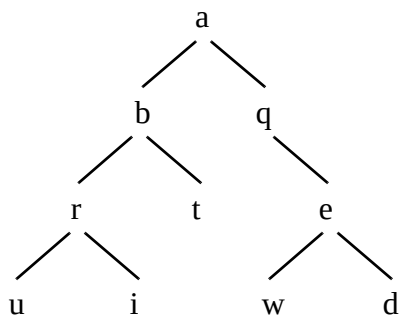
Die Behälterklasse List ist als **einfach verkettete Liste** implementiert mit den Vereinbarungen:

```
TYPE  Element* =  INTEGER;
      List*    =  POINTER TO RECORD first : Node END;

      Node     =  POINTER TO RECORD
                     item : Element;
                     next : Node;
                  END;
```

Implementieren Sie die Abfrage, ob eine Liste ein gegebenes Element enthält!

```
PROCEDURE (list : List) Has* (x : Element) : BOOLEAN, NEW;


BEGIN




END Has;
```

Aufgabe 7: Dynamische Datenstruktur Binärbaum (Punkte: 18 |.....)



Geben Sie zu dem **Binärbaum** die Reihenfolgen an, in der die Traversierungen die Elemente besuchen!

Preorder:.....

Inorder:.....

Postorder:.....

Aus einem leeren Baum entsteht durch Einfügen der Elemente

d y n a m i s c h

dieser **geordnete Binärbaum**:

Im Folgenden gelten die Vereinbarungen:

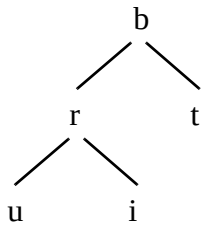
```
TYPE  Element  =  CHAR;

      Node     =  POINTER TO RECORD
                     item        : Element;
                     left, right : Node;
                  END;
```

Die Prozedur

```
PROCEDURE Count (tree : Node) : INTEGER;
BEGIN
   IF tree = NIL THEN
      RETURN 0
   ELSE
      RETURN Count (tree.left) + 1 + Count (tree.right)
   END
END Count;
```

wird mit dem Baum rechts als aktuellem Parameter aufgerufen. Geben Sie den **Aufrufbaum** an, wobei Sie als Parameter tree.item statt tree hinschreiben, falls tree # NIL ist.

Count (b)



Die **Höhe** eines Baums ist die größte Anzahl von Knoten der Wege von der Wurzel zu den Blättern; der leere Baum hat die Höhe 0. Implementieren Sie die Höhenfunktion!

```
PROCEDURE Height (tree : Node) : INTEGER;
BEGIN



END Height;
```

Aufgabe 8: Entwurfsmuster Schablonenmethode (Punkte: 12 |.....)

Die bekannte Mengenklasse ContainersSetsOfString.SetDesc ist um einen **All**- und einen **Existenzquantor** zu erweitern. Der Individuenbereich ist die Menge. Das Prädikat bestimmt der Kunde mit der Implementation der bekannten Schablonenmethode:

```
TYPE ActionDesc* = ABSTRACT RECORD END;

PROCEDURE (IN a : ActionDesc) Valid* (item : Element) : BOOLEAN, NEW, ABSTRACT;
```

Vorausgesetzt sind die bekannten Vereinbarungen:

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

      SetDesc*   =  EXTENSIBLE RECORD root : Node; END;
```

Implementieren Sie die **Baumfunktionen**!

```
PROCEDURE AreAllValid (tree : Node; VAR action : ActionDesc) : BOOLEAN;
BEGIN




END AreAllValid;

PROCEDURE (IN set : SetDesc) AreAllValid* (VAR action : ActionDesc) : BOOLEAN, NEW;
   (*! Does action.Valid (item) hold for each element item of set? !*)
BEGIN
   RETURN AreAllValid (set.root, action);
END AreAllValid;


PROCEDURE ExistsSomeValid (tree : Node; VAR action : ActionDesc) : BOOLEAN;
BEGIN




END ExistsSomeValid;

PROCEDURE (IN set : SetDesc)
   ExistsSomeValid* (VAR action : ActionDesc) : BOOLEAN, NEW;
   (*! Does action.Valid (item) hold for at least one element item of set? !*)
BEGIN
   RETURN ExistsSomeValid (set.root, action);
END ExistsSomeValid;
```

Aufgabe 9: Multimengen und Wörter zählen (Punkte: 16 |.....)

Eine **Multimenge** (*bag*) unterscheidet sich von einer Menge dadurch, dass jedes Element mehrfach vorkommen kann (in einer Menge höchstens einmal). Es ist nicht nur abfragbar, ob eine Multimenge ein Element enthält, sondern auch wie oft.

Entwerfen Sie ein Modul ContainersBagsOfComparable, das eine Klasse Bag für Multimengen bereitstellt und möglichst viel Bekanntes wiederverwendet!

Zeichnen Sie ein **Klassendiagramm**, das Bag und seinen Elementtyp zu bekannten Klassen in Beziehung setzt!

Welche Ideen, Konzepte und Programmteile kann ContainersBagsOfComparable von ContainersSetsOfComparable **übernehmen**?

Welche Teile von ContainersSetsOfComparable sind für ContainersBagsOfComparable **anzupassen**?

Nachdem Sie die Probleme *Zeichen sammeln*, *Zeichen zählen*, *Wörter sammeln* gelöst haben, stellt sich das Problem ***Wörter zählen***. Entwerfen Sie ein Modul StudWordCounter, das Kommandos zum Lesen von Texten und Analysieren der Häufigkeiten von Wörtern bereitstellt und möglichst viel Bekanntes wiederverwendet!