Klausur Programmierung WS 2013/2014

Vorname: _____

Nachname: _____

Matrikelnummer: _____

Studiengang (bitte **genau** einen markieren):

- ○ Informatik Bachelor
- ○ Informatik Lehramt (Bachelor)
- ○ Sonstiges: _____
- ○ Mathematik Bachelor
- ○ Informatik Lehramt (Staatsexamen)

	Anzahl Punkte	**Erreichte Punkte**
Aufgabe 1	**15**	
Aufgabe 2	**18**	
Aufgabe 3	**27**	
Aufgabe 4	**20**	
Aufgabe 5	**20**	
Aufgabe 6	**20**	
Summe	**120**	

Allgemeine Hinweise:

- **Auf alle Blätter** (inklusive zusätzliche Blätter) müssen Sie **Ihren Vornamen, Ihren Nachnamen und Ihre Matrikelnummer** schreiben.
- Geben Sie Ihre Antworten in lesbarer und verständlicher Form an.
- Schreiben Sie mit **dokumentenechten** Stiften, nicht mit roten oder grünen Stiften und nicht mit Bleistiften.
- Bitte beantworten Sie die Aufgaben auf den Aufgabenblättern (Antwortbereiche sind durch Kästen markiert).
- Geben Sie für jede Aufgabe **maximal eine** Lösung an. Streichen Sie alles andere durch. Andernfalls werden alle Lösungen der Aufgabe mit **0 Punkten** bewertet.
- Werden **Täuschungsversuche** beobachtet, so wird die Klausur mit **0 Punkten** bewertet und der Täuschungsversuch gemeldet, was eine Exmatrikulation und strafrechtliche Verfolgung nach sich ziehen kann.
- Geben Sie am Ende der Klausur **alle Blätter zusammen mit den Aufgabenblättern ab**.
- Halten Sie bitte Ihren Studierendenausweis und einen Lichtbildausweis zur Kontrolle bereit (die BlueCard gilt innerhalb der RWTH als beides).
- Die Bearbeitungszeit für diese Klausur beträgt zwei Stunden.

Aufgabe 1 **(10 + 5 = 15 Punkte)**

```
int func(int a[ ], int key, int l, int r) throws SearchException {
    if (r < l) {
      throw new SearchException();
    }

    int m = (l + r)/2;
    // Ausgabe
    if (a[m] > key) {
      return func(a, key, l, m − 1);
    } else if (a[m] < key) {
      return func(a, key, m + 1, r);
    } else {
      return m;
    }
}
```

a) Die Methode *func* wird mit den Argumenten $a = [1, 4, 7, 8, 9, 11, 13, 16, 109], key = 8, l = 0, r = 8$ aufgerufen. Bestimmen Sie für jeden rekursiven Aufruf die Werte der Variablen l, m und r an der mit „Ausgabe“ markierten Stelle im Quellcode und vervollständigen Sie die folgende Tabelle.

l	m	r

b) Die obige Funktion durchsucht ein (aufsteigend) sortiertes Array nach einem Wert und liefert den Index dieses Wertes zurück, wenn sie mit $l = 0$ und $r = a.length - 1$ aufgerufen wird und der Wert *key* im Array enthalten ist. Was passiert bei einem Aufruf mit $l = 0$ und $r = a.length - 1$, wenn der Wert *key* **nicht** im Array ist?

Aufgabe 2 **(18 Punkte)**

Die folgende Java-Methode soll für ein übergebenes Array a der Länge n von ganzen Zahlen die Summe der Produkte aller Elementpaare aus a berechnen, d. h.:

$$\sum_{0 \le i < j < n} a[i] \cdot a[j]$$

Wir erwarten vom Array a, daß es eine **Menge** von Zahlen darstellt. Daher soll die Methode eine *NoSetException* werfen, falls mindestens ein Element des Arrays mehr als einmal vorkommt.

Beispiel: *setPairProd*$([4, 1, 3, 5])$ ist $4 \cdot 1 + 4 \cdot 3 + 4 \cdot 5 + 1 \cdot 3 + 1 \cdot 5 + 3 \cdot 5 = 59$. Der Aufruf *setPairProd*$([0, 1, 1, 2])$ hingegen wirft eine *NoSetException*.

Gehen Sie davon aus, daß die Klasse *NoSetException* schon definiert ist.

```
int setPairProd(int a[ ]) throws NoSetException {




}
```

Aufgabe 3 **(16 + 11 = 27 Punkte)**

Im Folgenden sind einige Java-Klassen und -Interfaces gegeben. Darunter finden Sie auch die Klasse *TreeSet*, die eine Menge als geordneten Binärbaum implementiert. Das Ordnungskriterium ist durch eine Instanz des Interfaces *Comparator* festgelegt.

```
interface Visitor<T> {
  void visit(Element<T> e);
}

class Element<T> {
  T val;
  Element<T> left = null;
  Element<T> right = null;

  Element(T val) {
    this.val = val;
  }

  void insert(T v, Comparator<T> comp) {// TODO }

  void accept(Visitor<T> vis) {// TODO }

  T getVal() {
    return this.val;
  }
}

/* Implementierung einer Menge als geordneter Binärbaum. */
public class TreeSet<T> {
  Comparator<T> comp; // Legt die Sortierung fest.
  Element<T> root = null; // Wurzel des Baumes.

  public TreeSet(Comparator<T> comp) {
    this.comp = comp;
  }

  /* Fügt ein Element in die Menge ein. */
  public void insert(T v) {
    if (this.root == null) this.root = new Element<T>(v);
    else this.root.insert(v, this.comp);
  }

  /* Ermöglicht einem Visitor das Besuchen aller Knoten des Baumes. */
  public void accept(Visitor<T> vis) {
    if (this.root != null) this.root.accept(vis);
  }
}
```

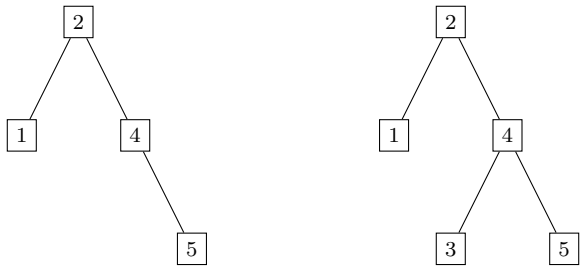

Abbildung 1: Baum vor (links) und nach (rechts) Einfügen des Elements 3

a) Implementieren Sie die Methode *Element.insert*. Diese soll eine Instanz der Klasse *Element* in einen Baum einfügen, ohne dabei die Sortierung des Baumes zu verletzen. Abbildung 1 veranschaulicht beispielhaft, wie sich das Einfügen eines neuen Elements auf einen Baum auswirkt.

b) Implementieren Sie die Methode *Element.accept*. Diese soll dafür sorgen, daß jeder Knoten des Baumes von dem als Argument übergebenem *Visitor* genau einmal besucht wird.

   Erstellen Sie eine Klasse *CountingVisitor*, die das Interface $Visitor\langle Integer\rangle$ implementiert. Diese Klasse soll die Möglichkeit bieten, zu ermitteln, wieviele der Elemente, die in einem $TreeSet\langle Integer\rangle$ gespeichert sind, positiv sind (d. h. $val > 0$). Ihre Implementierung soll auch eine Methode *getRes* zur Verfügung stellen, die das Ergebnis der Berechnung zurückliefert.

Hinweis zur Java-API:
Das Interface $Comparator\langle T\rangle$ deklariert die Methode **int** *compare*(T *o1*, T *o2*). Diese Methode liefert eine negative Zahl zurück, falls *o1* gemäß der durch die *Comparator*-Instanz definierte Ordnung kleiner ist als *o2*. Falls *o1* größer ist als *o2*, liefert *compare* eine positive Zahl zurück. Falls *o1* und *o2* gleich groß sind, wird 0 zurückgegeben.

```
void insert(T v, Comparator<T> comp) {

}
```

```
void accept(Visitor<T> vis) {

}
class CountingVisitor implements Visitor<Integer> {

}
```

Aufgabe 4 **(15 + 5 = 20 Punkte)**

Gegeben sei folgendes Java-Programm P, das zu einer Eingabe $\texttt{n} \geq 0$ den Wert $\texttt{n}^2$ berechnet.

$\langle\ \texttt{n} \geq 0\ \rangle$ (Vorbedingung)
  $i = 0;$
  $res = 0;$
  while $(i < n)$ {
    $i = i + 1;$
    $res = res + 2 * i - 1;$
  }
$\langle\ \texttt{res} = \texttt{n}^2\ \rangle$ (Nachbedingung)

a) Vervollständigen Sie die Verifikation des Algorithmus P auf der nachfolgenden Seite im Hoare-Kalkül, indem Sie die unterstrichenen Teile ergänzen. Hierbei dürfen zwei Zusicherungen nur dann direkt untereinander stehen, wenn die untere aus der oberen folgt. Hinter einer Programmanweisung darf nur eine Zusicherung stehen, wenn dies aus einer Regel des Hoare-Kalküls folgt.

   Hinweise:

   - Es gilt $i^2 + 2 * (i + 1) - 1 = (i + 1)^2$ für alle $i \in \mathbb{Z}$.
   - Sie dürfen beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In unserem Lösungsvorschlag wird allerdings genau die angegebene Anzahl an Zusicherungen benutzt.
   - Bedenken Sie, daß die Regeln des Kalküls syntaktisch sind, weshalb Sie semantische Änderungen (beispielsweise von $x + 1 = y + 1$ zu $x = y$) nur unter Zuhilfenahme der Konsequenzregeln vornehmen dürfen. Beachten Sie, daß Sie bei der Anwendung der Zuweisungsregel eventuell Klammern setzen müssen. Zum Beispiel:
     $\langle (i + 1)^2 = k \rangle$
     $i = i + 1;$
     $\langle i^2 = k \rangle$

Matrikelnummer: **Name:**

$\langle n \geq 0 \rangle$

$\langle$ _____ $\rangle$

```
i = 0;
```

$\langle$ _____ $\rangle$

```
res = 0;
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
while (i < n){
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
   i = i + 1;
```

$\langle$ _____ $\rangle$

```
   res = res + 2 * i - 1;
```

$\langle$ _____ $\rangle$

```
}
```

$\langle$ _____ $\rangle$

$\langle res = n^2 \rangle$

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben und unter Verwendung des Hoare-Kalküls die Terminierung unter der Voraussetzung $\texttt{n} \geq 0$ bewiesen werden. Begründen Sie, warum es sich bei der von Ihnen angegebenen Variante tatsächlich um eine gültige Variante handelt.

Aufgabe 5 **(6 + 6 + 8 = 20 Punkte)**

a) Schreiben Sie eine Funktion *subtractAll*::*Int*→[*Int*]→*Int*, welche als Argument eine Zahl a und eine Liste aus Zahlen bekommt. Das Ergebnis von *subtractAll* ist die Zahl, die man erhält, wenn man von a alle Elemente der Liste subtrahiert. Zum Beispiel ergibt *substractAll* 7 $[1, 2, 3]$ die Zahl 1.

b) Gegeben ist folgendes Haskell-Programm:

mystery = *foldl* (\$x\ y \to y : x$) [ ]

Was ist das Ergebnis von *mystery* $[1, 2, 3, 4, 5, 6]$?

c) Ein Vielwegbaum oder *multi-way tree* ist eine Baumstruktur, in welcher jeder Knoten eine beliebige Anzahl von Kindknoten haben kann. In Haskell können wir einen solchen Baum wie folgt definieren:

data *MultTree a* = *MultNode a* [*MultTree a*]

Schreiben Sie eine Funktion *mapMult*::(*a*→*b*)→*MultTree a*→*MultTree b*, welche einen Vielwegbaum mit Knoten vom Typ *a* auf einen Vielwegbaum gleicher Struktur mit Knoten vom Typ *b* abbildet; entsprechend der im ersten Parameter übergebenen Funktion. Die Eingabefunktion wird also auf alle Knotenwerte eines Knotens im Baum angewendet. Zum Beispiel ist das Ergebnis des Aufrufes

mapMult (\$x \rightarrow x * 2$) (*MultNode* 2[*MultNode* 3[ ], *MultNode* 5[*MultNode* 1[ ]]])

der Baum *MultNode* 4[*MultNode* 6[ ], *MultNode* 10[*MultNode* 2[ ]]]

Denken Sie daran, daß Sie alle Funktionen der Haskell-API, die in der Vorlesung behandelt wurden, verwenden dürfen.

Aufgabe 6 **(4 + 6 + 10 = 20 Punkte)**

a) Geben Sie zu den folgenden Termpaaren jeweils einen allgemeinsten Unifikator an oder begründen Sie, warum sie nicht unifizierbar sind. Hierbei werden Variablen durch Großbuchstaben dargestellt und Funktionssymbole durch Kleinbuchstaben.

i) $f(g(X), Y, Y), f(Z, Z, g(Z))$

ii) $f(X, Z, b), f(Y, a, Y)$

iii) $f(X, b, X), f(a, Y, Y)$

iv) $f(h(X, Y), Z), f(Z, h(b, a))$

b) Gegeben sei das folgende Prolog-Programm.

```
riddle([],[]).
riddle([[]|XS],YS) :- riddle(XS,YS).
riddle([[X|XS]|YS],[X|ZS]) :- riddle([XS|YS],ZS).
```

Geben Sie alle Antwortsubstitutionen zur Anfrage

```
?- riddle([[1],[3,2]],A).
```

in der Reihenfolge an, in der Prolog sie findet. Sie brauchen **nicht** die einzelnen Auswertungsschritte anzugeben (auch wenn Sie das als Nebenrechnung natürlich dürfen - ein schnellerer Weg zur Lösung besteht allerdings darin, ein intuitives Verständnis für das Programm zu entwickeln, sodaß man das Ergebnis direkt angeben kann).

c) Gegeben sei ein Prädikat `max/3` in Prolog, sodaß `max(X,Y,Z)` genau dann wahr ist, wenn `Z` das Maximum von `X` und `Y` ist. Nutzen Sie dieses Prädikat, um ein weiteres Prädikat `add/3` in Prolog zu implementieren, sodaß `add(X,XS,YS)` unter der Annahme, daß `XS` eine aufsteigend sortierte Liste ist, genau dann wahr ist, wenn `YS` aus `XS` entsteht, indem `X` sortiert eingefügt wird. Sollte `XS` keine aufsteigend sortierte Liste sein, darf sich Ihr Prädikat beliebig verhalten. Es gilt also z. B. `add(2,[1,3],[1,2,3])` und `add(1,[1,2],[1,1,2])`, aber nicht `add(2,[1,3],[2,1,3])` oder `add(1,[1,2],[1,2])`. Benutzen Sie **keine** vordefinierten Prädikate außer `max/3`.

Anhang *Hinweise zur Haskell-API:*

- `concat :: [[a]] -> [a]`:
  Konkateniert eine Liste von Listen, etwa wird $[[1,2,3],[4,5],[6,7,8]]$ zu $[1,2,3,4,5,6,7,8]$.

- `foldl :: (a -> b -> a) -> a -> [b] -> a`:
  Der Aufruf `foldl f a [b1,b2,...,bn]` entspricht dem Aufruf `f (... (f (f (f a b1) b2) b3 )... ) bn`, d. h. `f` wird sukzessive auf die Element der Liste aufgerufen, wobei das Ergebnis des vorherigen Aufrufs als erster Parameter übergeben wird.

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
  Die Funktion `foldr` arbeitet wie `foldl`, arbeitet die Liste aber von rechts nach links statt von links nach rechts ab.

- `map :: (a -> b) -> [a] -> [b]`:
  Der Aufruf `map f [a1,a2,...]` ergibt die Liste `[f a1,f a2,...]`.

- `filter :: (a -> Bool) -> [a] -> [a]`:
  Der Aufruf `filter f [a1,a2,...]` liefert eine Liste derjenigen Elemente `ai`, für welche `f ai` wahr ist.

- `zip :: [a] -> [b] -> [(a,b)]`
  Die Funktion zip bildet aus zwei Listen eine Liste von Paaren. Wenn eine Liste länger ist als die andere, werden überschüssige Elemente ignoriert.
  Beispiel: `zip [1,2,3] [2,3,4,5] = [(1,2),(2,3),(3,4)]`.

- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
  zipWidth verallgemeinert zip in der Hinsicht, dass keine Paare erzeugt werden sondern die zwei Elemente aus den jeweiligen Listen durch eine gegebene Funktion abgebildet werden. So erzeugt z.B. `zipWidth (+)` eine Liste von Summen und `zipWidth (\x y -> (x,y))` entspricht schlicht zip.

- `init :: [a] -> [a]`
  Die Funktion `init` liefert eine Liste ohne ihr letztes Element zurück.

- `last :: [a] -> a`
  Die Funktion `last` liefert das letzte Element einer Liste zurück.