RWTH Aachen
Lehrgebiet Theoretische Informatik
Frohn–Reidl–Rossmanith–Sánchez–Ströder

WS 2013/14
Probeklausur

Probeklausur zur Vorlesung Programmierung

Aufgabe 1 (10 Punkte)

Vervollständigen Sie folgende Java-Methode, welche von einem Array von Zahlen a das viertkleinste Element berechnet. Für $[4, 3, 8, 2]$ soll beispielsweise das Ergebnis 8 sein und für $[3, 3, 3, 2, 2, 2, 1, 1, 1]$ ist 2 das richtige Ergebnis.

Für unsinnige Eingaben soll Ihre Methode eine *BadInputException* werfen; gehen Sie davon aus, daß die Klasse *BadInputException* schon definiert ist.

```
int viertKleinstesElement(int a[ ]) throws BadInputException {




}
```

Aufgabe 2 **(6+2 Punkte)**

Gegeben sei die folgende Java-Methode *func*.

```
static void func(int[ ] a) {
  for (int i = 0; i < a.length; i++) {
    for (int j = 1; j < a.length − i; j++) {
      if (a[j − 1] > a[j]) {
        int tmp = a[j − 1];
        a[j − 1] = a[j];
        a[j] = tmp;
      }
    }
    // Ausgabe
  }
}
```

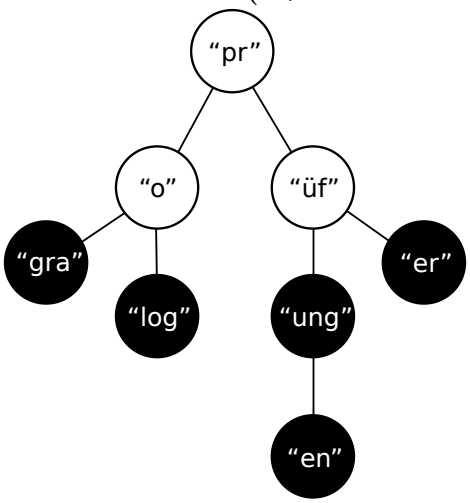
a) Die Methode wird auf dem Array $[6, 5, 4, 3, 1, 7, 2]$ aufgerufen. Bestimmen Sie für jeden Durchlauf der äußeren Schleife die Werte der Variablen i, a an der mit "Ausgabe" markierten Stelle im Quellcode und vervollständigen Sie die unten stehende Tabelle.

i	a
-	$[6, 5, 4, 3, 1, 7, 2]$
0	$[5, 4, 3, 1, 6, 2, 7]$
1	

b) Was müssten Sie an dem obigen Programm ändern, damit das Array a nach dem Aufruf absteigend sortiert ist? Ihre Änderung soll keine weiteren Programmzeilen hinzufügen!

Aufgabe 3 **(1+9=10 Punkte)**

Die auf der rechten Seite abgebildete Datenstruktur ist ein sogenannter *Präfixbaum*. Er wird benutzt um eine Menge von Zeichenketten möglichst platzsparend zu speichern.
Um die in ihm gespeicherten Wörter zu rekonstruieren, beginnt man bei der Wurzel (hier der Knoten mit der Zeichenkette "pr") und bewegt sich zu einem der schwarzen Knoten. Dabei konkateniert man die Zeichenketten der Knoten, die man auf diesem Weg besucht. Jedes im Baum gespeicherte Wort entspricht also einem schwarzen Knoten.



a) Geben Sie die Zeichenketten an, die in dem obigen Präfixbaum gespeichert sind.

b) Ergänzen Sie die unten gegebene Java-Implementierung des Präfixbaums um eine Methode **void** *printAllWords*(), welche alle im Baum gespeicherten Zeichenketten über die Standardausgabe auflistet. Natürlich können Sie zusätzliche Hilfsmethoden implementieren.

```
public class PTNode {
  // Die Zeichenkette dieses Knotens
  private String content;
  // Die Kindknoten. Das Array ist niemals null.
  private PTNode[ ] children;
  // 'true' wenn hier ein Wort aufhört, sonst 'false'
  private boolean isEndOfWord;

  public void printAllWords() {









  }









}
```

Aufgabe 4 **(6+6 Punkte)**

Gegeben seien eine Implementierung der Java-Klasse *ListSet*⟨*T*⟩, das interface *Visitor*⟨*T*⟩ und eine partielle Implementierung der Klasse *Element*⟨*T*⟩. Ein *ListSet*⟨*T*⟩ speichert Element in einer verketteten Liste entsprechend einer Ordnung, die durch ein Objekt des Typs *Comparator*⟨*T*⟩ gegeben ist. Wie der Name verrät, soll dabei jedes Element höchstens einmal in der Liste enthalten sein.

a) Implementieren Sie die Methoden *Element*⟨*T*⟩.*insert* und *Element*⟨*T*⟩.*accept*. Wie sich diese Methoden verhalten müssen, wird deutlich, wenn Sie die beiden Methoden *ListSet*⟨*T*⟩.*insert* und *ListSet*⟨*T*⟩.*accept* betrachten.

```
public interface Visitor⟨T⟩ {
  void visit(Element⟨T⟩ e);
}

public class ListSet⟨T⟩ {
  private Element⟨T⟩ head; // Erstes Element der Liste
  public Comparator⟨T⟩ comp; // Legt fest, wie die Liste sortiert wird

  /* Erzeugt eine leere Menge. */
  public ListSet(Comparator⟨T⟩ comp) {
    this.head = null;
    this.comp = comp;
  }

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

  public void accept(Visitor⟨T⟩ vis) {
    if (this.head != null) this.head.accept(vis);
  }
}

public class Element⟨T⟩ {
  public T val;
  private Element⟨T⟩ prev;
  private Element⟨T⟩ next;

  Element(T val, Element⟨T⟩ prev, Element⟨T⟩ next) {
    this.val = val;
    this.prev = prev;
    this.next = next;
  }
```

```
Element⟨T⟩ insert(T v, Comparator⟨T⟩ comp) {









  }

  void accept(Visitor⟨T⟩ vis) {









  }
}
```

b) Implementieren Sie darüber eine Klasse *ReorderingListSetVisitor*⟨*T*⟩, die das Interface *Visitor*⟨*T*⟩ implementiert und darüber hinaus folgende Methoden zur Verfügung stellt:

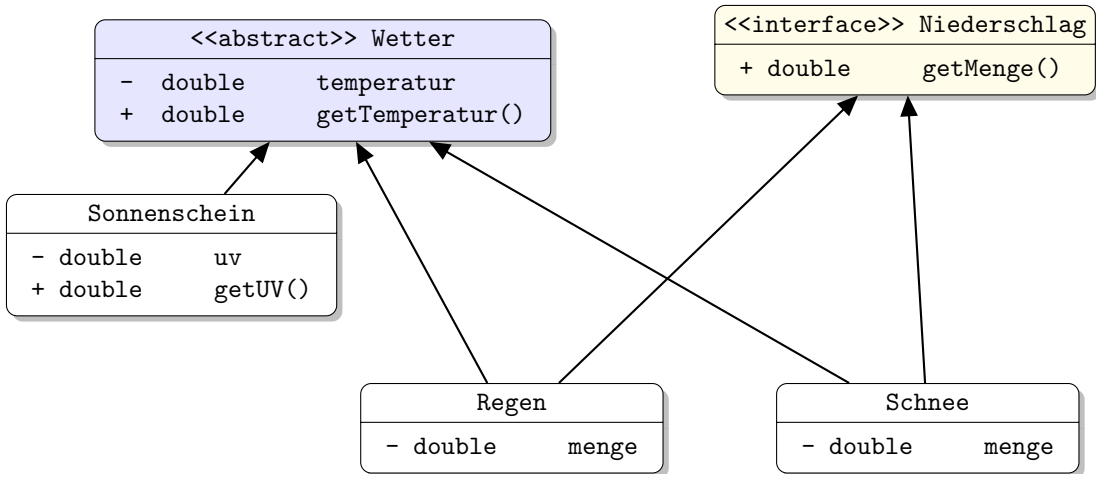
```
/* Der Comparator comp legt das zu verwendende Sortierkriterium fest. */
public ReorderingListSetVisitor(Comparator⟨T⟩ comp);
/* Liefert die neue Menge zurück. */
public ListSet⟨T⟩ getRes();
```

Diese Klasse soll die Möglichkeit bieten, eine neue Menge zu erzeugen, welche die gleichen Elemente wie die besuchte Menge enthält. Zur Sortierung der Elemente kann jedoch ein anderes Sortierkriterium verwendet werden. Bitte schreiben Sie ihre Lösung auf die nächste Seite.

Sie können bei der Lösung dieser Aufgabe davon ausgehen, dass $compareTo(o1, o2) == 0 \iff o1.equals(o2)$ für alle Implementierungen des Interfaces *Comparator* gilt.

Aufgabe 5 **(5 Punkte)**

Betrachten Sie folgendes Klassendiagramm. Die Sichtbarkeiten + und - bedeuten `public` und `private`. Die Methoden in den Klassen sind Getter für die entsprechenden Attribute.



Implementieren Sie die untenstehende Methode in der Klasse `Wetter`, welche die gesamte Niederschlagsmenge zurückgeben soll, die bei den im übergebenen Array enthaltenen Wetterphänomenen vorkommt. Sollten keine Phänomene mit Niederschlag vorhanden sein, soll die Methode `0.0` zurückgeben.

public static double *niederschlag*(*Wetter*[ ] *phaenomene*) {

}

Aufgabe 6 **(12 + 3 = 15 Punkte)**

Gegeben sei folgendes Java-Programm P, das zu einer Eingabe $\texttt{n} \geq 0$ den Wert $\frac{3^{\texttt{n}+1}-1}{2}$ berechnet.

$\langle\ \texttt{n} \geq 0\ \rangle$ (Vorbedingung)
  $i = 1;$
  $k = 3;$
  $res = 1;$
  while $(i <= n)$ {
    $res = res + k;$
    $k = k * 3;$
    $i = i + 1;$
  }
$\langle\ \texttt{res} = \frac{3^{\texttt{n}+1}-1}{2}\ \rangle$ (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 $\frac{3^i-1}{2} + 3^i = \frac{3^{i+1}-1}{2}$ für alle $i \in \mathbb{N}$.
- 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. Klammern dürfen Sie allerdings beliebig gemäß dem Assoziationsgesetz setzen oder weglassen, da diese keine semantischen Änderungen darstellen.

$\langle n \geq 0 \rangle$

$\langle \_\_\_\_\_ \rangle$

```
i = 1;
```

$\langle \_\_\_\_\_ \rangle$

```
k = 3;
```

$\langle \_\_\_\_\_ \rangle$

```
res = 1;
```

$\langle \_\_\_\_\_ \rangle$

$\langle \_\_\_\_\_ \rangle$

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

$\langle \_\_\_\_\_ \rangle$

$\langle \_\_\_\_\_ \rangle$

```
   res = res + k;
```

$\langle \_\_\_\_\_ \rangle$

```
   k = k * 3;
```

$\langle \_\_\_\_\_ \rangle$

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

$\langle \_\_\_\_\_ \rangle$

```
}
```

$\langle \_\_\_\_\_ \rangle$

$\langle \texttt{res} = \frac{3^{n+1}-1}{2} \rangle$

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 7 **(6 Punkte)**

Schreiben Sie eine Methode *findSum*::[*Int*]→*Int*→*Bool* in Haskell, welche prüft, ob die Elemente einer Liste ganzer Zahlen so miteinander addiert und subtrahiert werden können, dass das Ergebnis genau eine gegebene Zahl (der zweite Parameter) ist. Dabei muss jedes Element genau einmal addiert oder subtrahiert werden.

Zum Beispiel evaluiert *findSum* $[1, 0, 4, 5, 1]$ 1 zu 'True', denn $1 + 0 - 4 + 5 - 1 = 1$. Hingegen ist *findSum* $[1, 0, 4, 5, 1]$ 4 'False', denn keine Kombination von Additionen und Subtraktionen dieser Listenelemente ergibt 4.

Aufgabe 8 **(7+7=14 Punkte)**

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

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

Im Folgenden dürfen Sie alle Funktion der Haskell-API, die in der Vorlesung behandelt wurden, verwenden.

a) Entwerfen Sie eine Funktion *zipMult*::*MultTree* $a \rightarrow$ *MultTree* $b \rightarrow$ *MultTree* (a, b), welche zwei Vielwegbäume zusammenführt. Dazu dürfen Sie annehmen, dass die beiden Bäume die exakt gleiche Struktur haben: jeder Knoten des ersten Baumes hat genauso viele Kinder wie sein gegenüber im zweiten Baum; sie unterscheiden sich also nur durch ihren Inhalt. Der ausgegebene Baum soll ebenfalls dieselbe Struktur aufweisen, jedoch die Inhalte beider Bäume in Tupeln vereinen.

   Enthält etwa der erste Baum im Wurzelknoten das Element a_0 und der zweite im Wurzelknoten b_0, so enthält der Ergebnisbaum das Tupel (a_0, b_0) in seiner Wurzel.

b) Schreiben sie eine Funktion *bfs*::*MultTree* $a \rightarrow [a]$ welche die Elemente des Vielwegbaums in eine Liste überführt. Dabei sollen zunächst alle Elemente der Tiefe eins, dann der Tiefe zwei usw. in der Liste in dieser Reihenfolge enthalten sein. Insbesondere ist der Inhalt der Wurzel das erste Element in der Liste.

Aufgabe 9 **(4 + 4 + 12 = 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) $f(a, Y, X), f(X, b, Y)$

ii) $f(X, Y, Z), f(h(c), g(X, X), g(Y, Y))$

iii) $f(X, Y, Z), f(Z, a, h(X))$

iv) $f(X, Y, Z), f(g(Y, Y), g(Z, Z), h(X))$

b) Schreiben Sie ein Prolog-Programm, welches ein Prädikat *permutation* definiert. Für Listen A, B soll $permutation(A, B)$ genau dann wahr sein, wenn die Liste A dieselben Elemente genausooft enthält wie die Liste B. Mit anderen Worten: A ist eine Permutation von B.

c) Gegeben sei das folgende Prolog-Programm.

```
% Klausel 1:
append([],YS,YS).
% Klausel 2:
append(XS,[],XS).
% Klausel 3:
append([X|XS],YS,[X|ZS]) :- append(XS,YS,ZS).
```

Geben Sie alle Antwortsubstitutionen zu den folgenden Anfragen in der Reihenfolge an, in der Prolog sie findet (dieselbe Antwortsubstitution kann mehrfach vorkommen). Sollte die jeweilige Anfrage unendlich viele Antwortsubstitutionen berechnen, geben Sie die ersten vier Antwortsubstitutionen in der Reihenfolge an, in der Prolog sie findet. Falls die Auswertung nicht terminiert (wenn man alle Antwortsubstitutionen berechnen will), geben Sie außerdem an, welche Klauseln in welcher Reihenfolge von Prolog in dieser nicht-terminierenden Auswertung verwendet werden.

i) `append([1],[],A).`

ii) `append(A,[],A).`

iii) `append(A,B,[1,2,3]).`

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.

Hinweise 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 definierten Ordnung kleiner ist als *o2*. Falls *o1* größer ist als *o2*, liefert *compare* eine positive Zahl zurück. Fall *o1* und *o2* gleich groß sind, wird 0 zurückgegeben.