Prof. Dr. M. Müller C. Aschermann, J. Hensel, J. Protze, P. Reble

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **3 Studierenden** aus der **gleichen Kleingruppenübung (Tutorium)** bearbeitet werden. **Namen und Matrikelnummern** der Studierenden sind auf jedes Blatt der Abgabe zu schreiben. **Heften bzw. tackern Sie die Blätter!**
- Die **Nummer der Übungsgruppe** muss **links oben** auf das **erste Blatt** der Abgabe geschrieben werden. Notieren Sie die Gruppennummer gut sichtbar, damit wir besser sortieren können.
- Die Lösungen müssen **bis Montag, den 23.11.2015 um 15:00 Uhr** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Alternativ können Sie die Lösungen auch vor der Abgabefrist direkt bei Ihrer Tutorin/Ihrem Tutor abgeben.
- In einigen Aufgaben müssen Sie in Java programmieren und `.java`-Dateien anlegen. **Drucken** Sie diese aus **und** schicken Sie sie per **E-Mail** vor Montag, dem 23.11.2015 um 15:00 Uhr an Ihre Tutorin/Ihren Tutor.
  Stellen Sie sicher, dass Ihr Programm von `javac` **akzeptiert** wird, ansonsten werden keine Punkte vergeben.

Tutoraufgabe 1 (Verifikation mit Arrays):

Gegeben sei folgendes Java-Programm P:

$\langle$ `a.length` $> 0 \rangle$ (Vorbedingung)

```
res = a[0];
i = 1;
while (i < a.length) {
    if (a[i] > res) {
        res = a[i];
    }
    i = i + 1;
}
```

$\langle$ `res` $= \max\{$ `a[`j`]` $\mid 0 \leq j <$ `a.length` $\} \rangle$ (Nachbedingung)

a) Vervollständigen Sie die folgende Verifikation des Algorithmus 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.

Beachten Sie bei der Anwendung der "Bedingungsregel 1" mit Vorbedingung φ und Nachbedingung ψ, dass $\varphi \wedge \neg B \implies \psi$ gelten muss. D. h. die Nachbedingung der `if`-Anweisung ψ muss aus der Vorbedingung der `if`-Anweisung φ und der negierten Bedingung $\neg B$ selbst folgen. Geben Sie beim Verwenden der Regel einen entsprechenden Beweis an.

Hinweise:

- Sie dürfen beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In der Musterlösung werden allerdings genau die angegebenen 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.

$\langle$`a.length` $> 0\rangle$

```
res = a[0];
```
$\langle$________________________________$\rangle$

$\langle$________________________________$\rangle$
```
i = 1;
```

$\langle$________________________________$\rangle$

$\langle$________________________________$\rangle$
```
while (i < a.length) {
```

$\langle$________________________________$\rangle$

$\langle$________________________________$\rangle$
```
    if (a[i] > res) {
```

$\langle$________________________________$\rangle$

$\langle$________________________________$\rangle$
```
        res = a[i];
```

$\langle$________________________________$\rangle$
```
    }
```

$\langle$________________________________$\rangle$
```
    i = i + 1;
```

$\langle$________________________________$\rangle$
```
}
```

$\langle$________________________________$\rangle$

$\langle \texttt{res} = \max\{\, \texttt{a[}j\texttt{]} \mid 0 \leq j < \texttt{a.length} \,\}\rangle$

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und unter Verwendung des Hoare-Kalküls die Terminierung unter der Voraussetzung `a.length` > 0 bewiesen werden.

Geben Sie auch bei dieser Teilaufgabe einen Beweis für die Aussage $\varphi \wedge \neg B \implies \psi$ bei der Anwendung der "Bedingungsregel 1" an.

Aufgabe 2 (Verifikation mit Arrays): (7+2=9 Punkte)

Gegeben sei folgender Algorithmus P, der als Eingabe ein Array a mit Elementen $a[0], a[1], \ldots, a[n-1]$ entgegennimmt. Dieses Array wird in umgekehrter Reihenfolge in das Array b mit den Elementen $b[0], b[1], \ldots, b[n-$

1] gespeichert.

```
k = 0;
while (k < n) {
    b[n − 1 − k] = a[k];
    k = k + 1;
}
```

a) Vervollständigen Sie die Verifikation des Algorithmus P 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. Beachten Sie also insbesondere, dass Sie pro Beweisschritt jeweils nur eine einzige Regel des Hoare-Kalküls einsetzen dürfen!

Hinweis: Im Fall $n = 3$ bedeutet

$$\bigwedge_{j=0}^{n-1} b[n-1-j] = a[j]$$

die Aussage

$$b[2] = a[0] \wedge b[1] = a[1] \wedge b[0] = a[2].$$

Im Fall $n = 0$ bedeutet

$$\bigwedge_{j=0}^{n-1} b[n-1-j] = a[j]$$

die Aussage “true”.

$\langle n \geq 0 \rangle$

$\langle$ ______________________________ $\rangle$

$k = 0;$

$\langle$ ______________________________ $\rangle$

$\langle$ ______________________________ $\rangle$

`while` $(k < n)$ $\{$

$\langle$ ______________________________ $\rangle$

$\langle$ ______________________________ $\rangle$

$b[n-1-k] = a[k];$

$\langle$ ______________________________ $\rangle$

$\langle$ ______________________________ $\rangle$

$k = k + 1;$

$\langle$ ______________________________ $\rangle$

$\}$

$\langle$ ______________________________ $\rangle$

$\langle \bigwedge_{j=0}^{n-1} (b[n-1-j] = a[j]) \rangle$

b) Beweisen Sie die Terminierung des Algorithmus P. Geben Sie hierzu eine Variante für die `while`-Schleife an. Zeigen Sie, dass es sich tatsächlich um eine Variante handelt, und beweisen Sie damit unter Verwendung des Hoare-Kalküls mit der Voraussetzung $n \geq 0$ die Terminierung.

Tutoraufgabe 3 (Pilze):

In dieser Aufgabe beschäftigen wir uns mit dem berühmten Gaunerpärchen *Bonnie und Clyde*. Wenn die beiden nicht gerade Banken ausrauben, gehen sie gerne im Wald Pilze sammeln (bzw. klauen).
Wir verwenden hier die Klassen `Main`, `Mensch` und `Pilz`, die Sie auf der Homepage herunterladen können.
Jeder dieser (beiden) **Mensch**en hat einen Korb, in den eine feste Anzahl von Pilzen passt. Weiterhin hat jeder Mensch einen Namen. Wie Sie in der Klasse `Mensch` sehen können, gibt es hierfür drei Attribute. Das Attribut `anzahl` gibt hierbei an, wie viele Pilze bereits im Korb enthalten sind.
Zu jedem **Pilz** kennen wir den Namen.

a) Vervollständigen Sie die Klasse `Main` wie folgt:

- Ergänzen Sie an den mit `TODO a.1)` markierten Stellen den Code so, dass die Variablen `steinpilz`, `champignon`, `pfifferling` auf Pilz-Objekte mit passenden Namen verweisen.
- Ergänzen Sie an den mit `TODO a.2)` markierten Stellen den Code so, dass die Variablen `bonnie` und `clyde` auf passende Mensch-Objekte zeigen. Setzen Sie hierfür jeweils den passenden Namen und sorgen Sie dafür, dass in Bonnies Korb maximal 3 Pilze Platz haben. Bei Clyde passen 4 Pilze in den Korb.

b) Gehen Sie in dieser Teilaufgabe davon aus, dass die Attribute bereits alle auf vernünftige Werte gesetzt sind.

- Erweitern Sie die Klasse `Mensch` um eine Methode `hatPlatz()`, die `true` genau dann zurückgibt, wenn im Korb Platz für einen weiteren Pilz ist. Anderenfalls wird `false` zurückgegeben.
- Schreiben Sie für die Klasse `Mensch` eine Methode `ausgabe()`. Diese gibt kein Ergebnis zurück, aber gibt den Namen und eine lesbare Übersicht der von der Person gesammelten Pilze aus. Geben Sie in der ersten Zeile den Namen der Person gefolgt von einem Doppelpunkt („:“) aus. Schreiben Sie pro Pilz im Korb eine weitere Zeile, in der (nur) der Name des jeweiligen Pilzes steht.

  Eine Beispielausgabe von einem Menschen mit Namen „Gustav“ und einem Korb, der einen Pilz mit Namen „Morchel“ und einen Pilz mit Namen „Steinpilz“ enthält:

  ```
  Gustav:
  Morchel
  Steinpilz
  ```

c) In der Klasse *Main* sehen Sie eine Variable `wald`.

Schreiben Sie an die mit `TODO c)` markierte Stelle eine Schleife, die die Pilze im `wald`-Array nach und nach abarbeitet. Hierbei wird **zuerst** überprüft, ob **Bonnie** Platz für einen weiteren Pilz hat. Wenn dies der Fall ist, wird der Pilz in Bonnies Korb hinzugefügt. Benutzen Sie hierfür auch das Attribut `anzahl` und passen Sie dieses entsprechend an.

Nur wenn der Pilz bei Bonnie nicht hinzugefügt werden konnte, probieren wir den Pilz bei **Clyde** hinzuzufügen. Überprüfen Sie also in diesem Fall ebenfalls, ob der Korb voll ist und fügen Sie den Pilz ggf. hinzu.

Wenn ein Pilz weder bei Bonnie noch bei Clyde hinzugefügt werden kann, wird der Pilz keinem Korb hinzugefügt.

Rufen Sie am Ende einer jeder Schleifeniteration die Methode `ausgabe()` zuerst für Bonnie und anschließend für Clyde auf. Geben Sie anschließend eine Zeile aus, in der nur „`---`“ (drei Bindestriche) steht.

Listing 1: Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Pilz steinpilz = // TODO a.1)
4
5      Pilz champignon = // TODO a.1)
6
7      Pilz pfifferling = // TODO a.1)
8
9      Mensch bonnie = // TODO a.2)
10
11     Mensch clyde = // TODO a.2)
12
13     Pilz[] wald = new Pilz[] {steinpilz, champignon, champignon, pfifferling,
14       steinpilz, pfifferling, champignon};
15
16     // TODO c)
17   }
18 }
```

Listing 2: Mensch.java

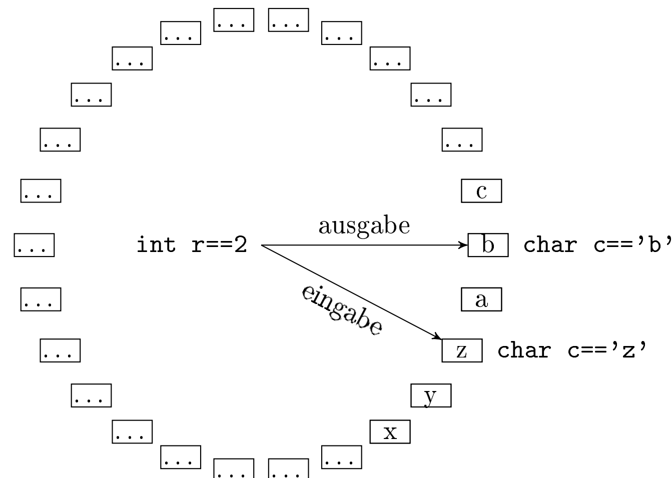
```
1  public class Mensch {
2    String name;
3    Pilz[] korb;
4    int anzahl = 0;
5  }
```

Listing 3: Pilz.java

```
1  public class Pilz {
2    String name;
3  }
```

Aufgabe 4 (Verschlüsselung): (1+2+1+1+2+1* = 7+(1*) Punkte)

In dieser Aufgabe soll ein Algorithmus zur Verschlüsselung von einfachen Text-Nachrichten in Java implementiert werden. Wir beschränken uns dabei auf Wörter, die lediglich aus kleinen Buchstaben (a ... z) des lateinischen Alphabetes ohne Umlaute und Sonderzeichen bestehen.
Als Verschlüsselungs-Algorithmus soll jeder Buchstabe im Alphabet um eine vorgegebene Anzahl von Stellen im Alphabet in aufsteigender Reihenfolge verschoben werden. Kommt es zu einem Überlauf (z.B. 'z' verschoben um 2 Stellen), wird eine Verschiebung am Beginn des Alphabetes fortgesetzt.
Diese Abbildungsvorschrift ist in der folgenden Grafik illustriert:



a) Schreiben Sie eine Klasse `Geheim` mit einem Attribut Nachricht, das ein zwei-dimensionales Array vom Typ `char` ist. Ein Objekt vom Typ `Geheim` soll eine Nachricht der Länge n repräsentieren.

b) Schreiben Sie eine Methode `char encrypt(char c, int r)`, welche für ein Zeichen i eine Rotation um r Stellen vornimmt und das resultierende Zeichen zurückgibt.

c) Erweitern Sie die Klasse `Geheim` um eine Methode `void input()`, welche den Benutzer zur Eingabe von n Wörtern auffordert, diese einliest und das Array im Attribut mit entsprechenden Werten belegt.

Hinweise:

- Um einen `String` als Array aus `char` Werten abzulegen, ist in Java die Methode `toCharArray()` vordefiniert.

Hinweise:

- Unter einigen Entwicklungsumgebungen funktioniert Konsoleninput nicht (z.B. BlueJ). Sie dürfen beliebigen Code benutzen, um die Userwerte zu erfragen. Eine mögliche Klasse für Userabfragen in BlueJ ist auf der Homepage der Vorlesung veröffentlicht (SimpleIO).

d) Schreiben Sie eine ausführbare `main`-Methode, welche den Benutzer auffordert, eine Nachricht einzugeben. Dazu wird er erst einmal gefragt, wieviele Wörter die Nachricht haben soll, und dann wird diese Wort für Wort eingelesen.

Hinweise:

- Um einen `int` Wert einzulesen, können Sie die Methode `Integer.parseInt()` verwenden, die einen `String` als Argument erhält und den entsprechenden `int` Wert zurückliefert.

Dann soll das Programm die Nachricht in verschlüsselter Form ausgeben.

e) Wie lautet die Ausgabe von folgender Nachricht bei einer Verschiebung um 13 Stellen? Was fällt Ihnen bei wiederholter Verschlüsselung der Nachricht auf?

```
thg trznpug rva obahfchaxg
```

Tutoraufgabe 5 (Spielfeld):

In dieser Tutoraufgabe soll die Ausgabe eines Spielfelds als Java-Programm implementiert werden. Hierzu existiert bereits eine Klasse `Feld` mit einem Attribut `fieldsize`, das die Seitenlänge des quadratischen Spielfelds angibt.

Listing 4: Feld.java

```
1  public class Feld {
2
3      // Seitenlaenge des quadratischen Spielfeldes
4      public static int fieldsize = 10;
5
6      // Gibt das Spielfeld aus.
7      public static void ausgabe(
8          boolean[][] spieler1,
9          boolean[][] spieler2)
10     {
11         // TODO
12
13     }
14
15     // Fuehrt das Programm aus.
16     public static void main(String[] args) {
17       Feld.fieldsize = 3;
18       boolean[][] p1 = {{true, true, true},
19                         {false, false, false},
20                         {false,false,false}};
21       boolean[][] p2 = {{true, false, false},
22                         {true,false,false},
23                         {true,false,true}};
24       ausgabe(p1,p2);
25     }
26 }
```

Implementieren Sie die Methode `ausgabe` in der Klasse `Feld`. Diese Methode bekommt zwei 2-dimensionale Arrays von `booleans` als Eingabe, die das Spielfeld darstellen. Ein Feld (x, y) auf dem Spielfeld wird mit "O" beschriftet, falls nur Spieler 1 auf dieses Feld gesetzt hat (d.h., falls `spieler1[x][y] == true`). Falls nur Spieler 2 auf das Feld gesetzt hat, wird ein "X" angezeigt. Haben beide Spieler auf dasselbe Feld gesetzt, so wird "@" ausgegeben. Freie Felder werden mit einem " " (Space) angezeigt. Die Ausführung der `main` Methode der Klasse `Feld` sollte dann folgende Ausgabe erzeugen.

```
 |0|1|2|
-+-+-+-+
A|@|O|O|
-+-+-+-+
B|X| | |
-+-+-+-+
C|X| |X|
-+-+-+-+
```

Aufgabe 6 (Labyrinth): (1+2+2+2+4+3* = 11+(3*) Punkte)

In dieser Aufgabe sollen Sie ein Programm schreiben, das den Weg eines Reisenden durch ein Labyrinth simuliert. Ihrem Programm wird eine Karte zur Verfügung gestellt, der Reisende soll jedoch nur Informationen zu den direkt benachbarten Feldern seiner Position erhalten. In jedem Simulationsschritt soll sich der Reisende mit einer Schrittweite von 1 auf ein freies Feld bewegen können, in horizontaler oder vertikaler Richtung. Diagonale Bewegungen sind nicht erlaubt. Zur Vereinfachung nehmen wir an, dass der Rand einer Karte immer aus blockierten Feldern besteht und der Reisende einen fixen Startpunkt `(1,1)` und ein fixes Ziel `(n-2,n-2)` besitzt. Außerdem betrachten wir nur Labyrinthe mit mindestens einem Pfad von Start zum Ziel. Zum Einlesen einer Karte benutzen Sie bitte die auf der Webseite zur Verfügung gestellte Klasse `Loader`, welche die Methode `static boolean[][] load(String path)` zur Verfügung stellt. Diese Methode lädt die Datei, die sich am

angegebenen Pfad (`path`) befindet, und erzeugt daraus ein 2-dimensionales Array von `booleans`. Eine solche Beispieldatei ist “`test.maze`”.

a) Zuerst erstellen Sie eine Methode `static char avatar(int r)` zur Darstellung der Blick-Richtung des Reisenden auf der Karte. Der Reisende wird abhängig von seiner Blickrichtung (Süd,Ost,Nord,West) mit einem anderen Zeichen (`char`) “`v`”, “`>`”, “`^`” oder “`<`” repräsentiert. Nutzen Sie innerhalb der Methode eine `switch` Anweisung und folgende Kodierung zur Parameter-Übergabe der Richtung (`r`):

Int-Wert	Orientierung	Darstellung
0	Ost	`'>'`
1	Nord	`'^'`
2	West	`'<'`
3	Süd	`'v'`

b) Implementieren Sie eine Methode `static void print(boolean[][] field, int[] pos, char r)`, die folgendes entgegenimmt: ein Karte, ein Array `pos` der Größe 2 mit x- und y-Position des Reisenden, und ein Zeichen `char r` als Repräsentation des Reisenden. Der Rand soll nicht ausgegeben werden. Ansonsten wird jede blockierte Position durch ein “`X`” und jede offene Position, außer an (x,y), durch ein “`O`” repräsentiert.

Die Methode kann mit dem Beispiel in “`test.maze`” getestet werden. Das Ergebnis sollte wie folgt aussehen:

```
v O O O O X O
X X O X O O X
O X O O X X X
X X X O O X O
X X X X O O X
O O O O O O O
X X O X X X O
```

Hier: `pos[0]==1, pos[1]==1, char r=='v'`

c) Implementieren Sie eine Methode `static boolean[] viewport(boolean[][] t, int[] pos)`, die eine Karte sowie eine Position auf der Karte übergeben bekommt. Innerhalb der Methode wird ein neues Array der Größe 4 angelegt und mit den Wahrheitswerten der Nachbarfelder von `pos` belegt. Dabei zählen als Nachbarfelder ausgehend von (x, y) alle Felder mit $\{ (x+x_o, y+y_o) \mid (x_o \in \{0\} \wedge y_o \in \{-1,1\}) \vee (x_o \in \{-1,1\} \wedge y_o \in \{0\})\}$. Schließlich soll eine Referenz auf das Array zurückgegeben werden. Nutzen sie die Zuordnung aus der Tabelle in **a)** zur Kodierung des Indexes.

d) Erstellen Sie eine Methode `static int move(boolean[] field)`, welche für einen gegebenen Kartenausschnitt, äquivalent zur Rückgabe der Methode `viewport`, die Zugrichtung des Reisenden im Labyrinth bestimmt. Dabei soll der Benutzer gefragt werden, in welche Richtung sich der Reisende bewegen soll (entsprechend der oben dargestellten Tabelle).

Prüfen Sie ob die Eingabe korrekt ist (Bewegung nur auf freies Feld erlaubt) und lassen sie ansonsten die Eingabe wiederholen.

Geben Sie den Index des Feldes, auf das sich bewegt wurde, zurück.

e) Schreiben Sie eine `main`-Methode `public static void main(String[] args)`, so dass eine Karte aus einer Datei geladen wird, deren Pfad in `args[0]` angegeben wird. Das Programm sollte eine Fehlermeldung ausgeben, wenn die Länge von `args` nicht eins ist.

Führen sie dann die Simulation durch Aufruf der Methode `move(...)` in einer Schleife aus, wobei die Abbruchbedingung das Erreichen des Zielpunktes ist.

Nach der Entscheidung über die Richtung und vor der Bewegung dreht sich der Reisende immer in Laufrichtung.

Geben Sie die Karte in jedem Simulationsschritt aus.

f)* Sie erhalten 3 Bonuspunkte, wenn Sie die Methode `move(...)` so modifizieren, dass sich der Reisende ohne Benutzereingabe durch ein Labyrinth mit den beschriebenen Eigenschaften bewegt und das Ziel findet. Zum Beispiel kann dazu die sogenannte Rechte-Hand-Regel implementiert werden, wobei der Reisende stets versucht, sich an der Wand zu seiner Rechten entlang zu bewegen. Dies ist eine schwierigere Bonusaufgabe.