Prof. Dr. J. Giesl

S. Dollase, M. Hark, D. Korzeniewski

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **2 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 oben links!**
- 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 Mittwoch, den 31.10.2018 um 08:00** 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 Mittwoch, den 31.10.2018 um 08:00 an Ihre Tutorin/Ihren Tutor.
  Stellen Sie sicher, dass Ihr Programm von `javac` **akzeptiert** wird, ansonsten werden keine Punkte vergeben.
- Benutzen Sie in ihrem Code keine Umlaute, auch nicht in Strings und Kommentaren. Diese führen oft zu Problemen, da diese Zeichen auf verschiedenen Betriebssystemen unterschiedlich behandelt werden. Dadurch kann es dazu führen, dass Ihr Programm bei Ihrer Tutorin/Ihrem Tutor bei Verwendung von Umlauten nicht mehr von `javac` akzeptiert wird.
- Halten Sie sich beim Lösen von Programmieraufgaben an die auf der Website zur Vorlesung verfügbaren Codekonventionen. Verstöße, die zu unleserlichem Code führen, können zu Punktabzug führen.
- Für einige Programmieraufgaben benötigen Sie die Java Klasse `SimpleIO`. Diese können Sie auf der Website herunterladen.
- Einige Hausaufgaben müssen im Spiel Codescape gelöst werden:
  https://codescape.medien.rwth-aachen.de/progra/
  Diese Aufgaben werden getrennt von den anderen Hausaufgaben gewertet.

Tutoraufgabe 1 (Typcasting):

Bestimmen Sie den Typ und das Ergebnis der folgenden Java-Ausdrücke. **Begründen** Sie Ihre Antwort und **geben Sie dabei für alle auftretenden Typkonvertierungen den resultierenden Typ und den resultierenden Wert an. Geben Sie darüber hinaus an, ob es sich um explizite oder implizite Konvertierungen handelt.** Sollte der Ausdruck nicht typkorrekt sein, begründen Sie, worin der Fehler besteht.
Dabei seien die Variablen `x`, `y` und `z` wie folgt deklariert: `var x = 1; var y = 2; var z = 3;`

a) `false && true`

b) `10 / 3`

c) `10 / 3.`

d) `x == y ? x > y : y < z`

e) `(byte) (127 + 1)`

f) `'x' + y + z`

g) `x + y + "z"`

h) `1 || 0`

i) `5 + (2 > 3? 7.2 : 4)`

Aufgabe 2 (Typcasting): (1.5 + 1 + 1 + 2 + 1 + 1.5 + 1 = 9 Punkte)

Bestimmen Sie den Typ und das Ergebnis der folgenden Java-Ausdrücke. **Begründen** Sie Ihre Antwort und **geben Sie dabei für alle auftretenden Typkonvertierungen den resultierenden Typ und den resultierenden Wert an. Geben Sie darüber hinaus an, ob es sich um explizite oder implizite Konvertierungen handelt.** Sollte der Ausdruck nicht typkorrekt sein, begründen Sie, worin der Fehler besteht.
Dabei seien die Variablen `x`, `y` und `z` wie folgt deklariert: `var x = 5.; var y = false; var z = 3;`

a) `(float) (2/(long) 3)`

b) `x == y && x != y`

c) `x == z && x != z`

d) `'a'/98 == 3.F/(int)4.6`

e) `(x < z) + 2.2F`

f) `result`, wobei `result` folgende Variablendeklaration hat: `int result = x < (byte) 'a'? 'a': 'b'`

g) `u`, wobei `u` folgende Variablendeklaration hat: `var u = true? 1 : 2.0;`

Tutoraufgabe 3 (Programmierung):

Schreiben Sie ein einfaches Java-Programm, welches den Benutzer auffordert, eine positive ganze Zahl (d. h. größer als 0) einzugeben. Danach soll das Programm eine Zahl einlesen. Diese Eingabeaufforderung mit anschließendem Einlesen soll solange wiederholt werden, bis der Benutzer wirklich eine positive Zahl eingibt. Wenn die Benutzereingabe keine Zahl ist, darf sich das Programm beliebig verhalten. Anschließend soll der Benutzer aufgefordert werden, ein Wort einzugeben. Das Wort soll eingelesen und schließlich so oft hintereinander geschrieben ausgeben werden, wie durch die eingegebene positive Zahl festgelegt wurde.
Ein Ablauf des Programms könnte z.B. so aussehen:

```
Bitte geben Sie eine positive Zahl ein
0
Bitte geben Sie eine positive Zahl ein
3
Bitte geben Sie ein Wort ein
Programmierung
ProgrammierungProgrammierungProgrammierung
```

Hinweise:

- Verwenden Sie die Klasse `SimpleIO` zum Einlesen und Ausgeben von Werten.

Aufgabe 4 (Programmierung): (9 Punkte)

Implementieren Sie einen einfachen 8–Bit-Zweierkomplement–Konverter in Java. Für die Ein-/Ausgabe soll die bereitgestellte Klasse `SimpleIO` genutzt werden. Die Benutzung anderer vordefinierten Methoden als der hier explizit genannten ist **nicht** gestattet.
Um einen String `str1` in einem Fenster mit dem Titel[1] `str2` auszugeben, nutzen Sie `SimpleIO.output(str1, str2)`. Um einen Wert vom Typ `type` einzulesen, nutzen Sie `SimpleIO.getType(str)`, wobei `str` der Text ist, der dem Benutzer im Eingabefenster angezeigt wird. Um einen Wert vom Typ `int` einzulesen, benutzen Sie also z.B. `SimpleIO.getInt("Bitte eine ganze Zahl eingeben")`.

Das Programm soll den Benutzer zunächst auswählen lassen, ob von Dezimaldarstellung in Binärdarstellung oder von Binärdarstellung in Dezimaldarstellung konvertiert werden soll. Zur Dezimaldarstellung benutzen wir den primitiven Datentyp `int`, zur Binärdarstellung den vordefinierten Typ `String`. Bedenken Sie, dass mit dem 8–Bit–Zweierkomplement die ganzen Zahlen $\{-128 \dots 127\}$ dargestellt werden können und alle Binärdarstellungen genau 8 Einträge haben. Zur Auswahl sollen folgende Schlüsselwörter verwendet werden:

Decimal Bei dieser Auswahl soll der Benutzer eine ganze Zahl als Dezimalzahl eingeben. Falls diese nicht im 8–Bit–Zweierkomplement dargestellt werden kann, soll die Eingabe wiederholt werden, bis eine darstellbare Zahl eingegeben wird. Anschließend soll die Eingabe mit dem Verfahren aus der Vorlesung ins Zweierkomplement konvertiert und ausgegeben werden.

Binary Bei dieser Auswahl soll das Programm den Benutzer nach einer ganzen Zahl im 8–Bit–Zweierkomplement fragen, die als `String` eingelesen wird. Das Einlesen soll wiederholt werden bis ein `String` der Länge 8 eingegeben wird. Enthält der String Zeichen, die weder `0` noch `1` sind, soll eine Fehlermeldeung ausgegeben und das Programm beendet werden. Eine korrekte Eingabe soll mit dem Verfahren aus der Vorlesung in eine Dezimalzahl umgewandelt und ausgegeben werden.

Gibt der Nutzer weder `Decimal` noch `Binary` ein, so soll er erneut gefragt werden.

Ein Beispiellauf des Programms könnte also so aussehen:

```
Bitte waehlen Sie aus:
Decimal: Dezimalzahl ins Zweierkomplement konvertieren.
Binary: 8-Bit-Zweierkomplement als Dezimalzahl darstellen.
Programmierung
Bitte waehlen Sie aus:
Decimal: Dezimalzahl ins Zweierkomplement konvertieren.
Binary: 8-Bit-Zweierkomplement als Dezimalzahl darstellen.
Decimal
Bitte geben Sie eine ganze Zahl zwischen -128 und 127 ein.
-13
Ihre Dezimalzahl im Zweierkomplement dargestellt ist 11110011
```

bzw.

```
Bitte waehlen Sie aus:
Decimal: Dezimalzahl ins Zweierkomplement konvertieren.
Binary: 8-Bit-Zweierkomplement als Dezimalzahl darstellen.
Binary
Bitte geben Sie eine ganze Zahl im 8-Bit-Zweierkomplement ein.
11110011
Ihr 8-Bit-Zweierkomplement entspricht der Dezimalzahl -13
```

Hinweise:

- Um zwei `Strings` `str1` und `str2` auf Gleichheit zu testen, verwenden Sie `str1.equals(str2)` (und nicht `str1 == str2`).
- Für die Berechnung der Länge eines `String str` rufen Sie `str.length()` auf.

[1] Sie dürfen in dieser Aufgabe immer z.B. "Zweierkomplement" als Titel verwenden.

- Um den `i`-ten `char` eines `String str` zu bestimmen, rufen Sie `str.charAt(i)` auf. Der vorderste Eintrag eines `String` hat den Index `0`. Der letzte Buchstabe hat den Index `str.length()-1`.
- Die Benutzung von impliziter Typinferenz durch `var x = ...;` ist nicht gestattet, da diese zu unleserlichem Programmcode führt.
- Legen Sie die bereitgestellte Datei `SimpleIO.java` einfach im gleichen Verzeichnis wie ihre Lösung ab. Dann findet `javac` diese automatisch und kompiliert sie.
- Ihr Konverter muss die Verfahren aus der Vorlesung verwenden, um die Eingabe zu konvertieren. Ansonsten werden keine Punkte vergeben.

Tutoraufgabe 5 (Verifikation):

Gegeben sei folgendes Java-Programm über den Integer-Variablen `x`, `y`, `z` und `r`:

$\langle 0 \leq \texttt{x} \wedge 0 < \texttt{y} \rangle$ (Vorbedingung)

```
   z = 0;
   r = x;
   while (r >= y) {
      r = r - y;
      z = z + 1;
   }
```

$\langle \texttt{z} = \texttt{x} \text{ div } \texttt{y} \rangle$ (Nachbedingung)

Vervollständigen Sie die folgende Verifikation der partiellen Korrektheit 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.

Hinweise:

- div steht für die Integer-Division, d.h. 5 div 3 ergibt z.B. 1.
- 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.
- Es empfiehlt sich oft, bei der Erstellung der Zusicherungen in der Schleife von unten (d. h. von der Nachbedingung aus) vorzugehen.

$\langle 0 \leq \texttt{x} \wedge 0 < \texttt{y} \rangle$

$\langle$ _____ $\rangle$

```
z = 0;
```

$\langle$ _____ $\rangle$

```
r = x;
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
while (r >= y) {
```

⟨_____⟩

⟨_____⟩

```
    r = r - y;
```

⟨_____⟩

```
    z = z + 1;
```

⟨_____⟩

```
}
```

⟨_____⟩

⟨z = x div y⟩

Aufgabe 6 (Verifikation): (7 Punkte)

Gegeben sei folgendes Java-Programm über den Integer-Variablen `x`, `res` und `c`:

⟨x ≥ 0⟩ (Vorbedingung)

```
    res = -x;
    c = x;
    while (c > 0) {
       res = res + 2 * c;
       c = c - 1;
    }
```

⟨res = x · x⟩ (Nachbedingung)

Vervollständigen Sie die folgende Verifikation der partiellen Korrektheit 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.

Hinweise:

- Sie können die Gauß'sche Summenformel benutzen:
  Für eine natürliche Zahl x gilt $0 + 1 + \cdots + x = \sum_{k=0}^{x} k = \frac{x \cdot (x+1)}{2}$. Somit ist also $x^2 = 2 \cdot \sum_{k=0}^{x} k - x$.
- Die leere Summe (bei der der Startindex größer als der Zielindex ist) wird als 0 interpretiert. Beispielsweise gilt also $\sum_{k=x+1}^{x} k = 0$.
- 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.
- Es empfiehlt sich oft, bei der Erstellung der Zusicherungen in der Schleife von unten (d. h. von der Nachbedingung aus) vorzugehen.

$\langle 0 \leq \texttt{x} \rangle$

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

```
res = -x;
```

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

```
c = x;
```

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

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

```
while (c > 0) {
```

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

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

```
    res = res + 2 * c;
```

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

```
    c = c - 1;
```

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

```
}
```

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

$\langle \texttt{res} = \texttt{x} \cdot \texttt{x} \rangle$

Tutoraufgabe 7 (Verifikation):

Gegeben sei folgendes Java-Programm, das zu zwei Eingaben $\texttt{x} = \texttt{a}$ und $\texttt{y} = \texttt{b}$ den Wert $|\texttt{a} - \texttt{b}|$ berechnet.

$\langle\ \texttt{x} = \texttt{a} \wedge \texttt{y} = \texttt{b}\ \rangle$ (Vorbedingung)

```
res = 0;
while (x != y) {
    if (x > y) {
        x = x - 1;
    } else {
        y = y - 1;
    }
    res = res + 1;
}
```

$\langle\ \texttt{res} = |\texttt{a} - \texttt{b}|\ \rangle$ (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.

Hinweise:

- Für alle $x, y \in \mathbb{Z}$ gilt:
$$x > y \implies |(x-1) - y| = |x - y| - 1$$
und
$$x \neq y \wedge \neg(x > y) \implies |x - (y-1)| = |x - y| - 1.$$
Achtung: Falls lediglich $x \geq y$ bzw. $x \leq y$ gilt, gelten beide Gleichungen **nicht** im Allgemeinen.
- 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. Klammern dürfen und müssen Sie jedoch eventuell bei der Anwendung der Zuweisungsregel setzen.
- Es empfiehlt sich oft, bei der Erstellung der Zusicherungen in der Schleife von unten (d. h. von der Nachbedingung aus) vorzugehen.

$\langle \mathtt{x} = \mathtt{a} \wedge \mathtt{y} = \mathtt{b} \rangle$

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

```
res = 0;
```

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

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

```
while (x != y) {
```

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

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

```
    if (x > y) {
```

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

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

```
        x = x - 1;
```

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

```
    } else {
```

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

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

```
        y = y - 1;
```

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

```
    }
```

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

```
    res = res + 1;
```

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

```
}
```

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

$\langle \mathtt{res} = |\mathtt{a} - \mathtt{b}| \rangle$

b) Beweisen Sie die Terminierung des Algorithmus im Hoare-Kalkül.

Aufgabe 8 (Verifikation): (9 + 3 = 12 Punkte)

Gegeben sei folgendes Java-Programm über den Integer-Variablen `a`, `b`, `x`, `res` und `y`:

$\langle \texttt{b} \geq 0 \rangle$ (Vorbedingung)

```
x = b;
res = a;
y = 1;
while (x > 0) {
   if (x % 2 == 0) {
      y = 2 * y;
      x = x / 2;
   } else {
      res= res + y;
      y = 2 * y;
      x = (x - 1) / 2;
   }
}
```

$\langle \texttt{res} = \texttt{a} + \texttt{b} \rangle$ (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.

Hinweise:

- Die Programmanweisung `x / 2` berechnet das abgerundete Ergebnis der Ganzzahldivision, d.h. $\lfloor \frac{x}{2} \rfloor$.
- 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.
- Es empfiehlt sich oft, bei der Erstellung der Zusicherungen in der Schleife von unten (d. h. von der Nachbedingung aus) vorzugehen.

$\langle \texttt{b} \geq 0 \rangle$

$\langle$ _____ $\rangle$

```
x = b;
```

$\langle$ _____ $\rangle$

```
res = a;
```

$\langle$ _____ $\rangle$

```
y = 1;
```

$\langle$ _____ $\rangle$

```
                                    ⟨_____⟩
while (x > 0) {
                                    ⟨_____⟩
  if (x % 2 == 0) {
                                    ⟨_____⟩
                                    ⟨_____⟩
    y = 2 * y
                                    ⟨_____⟩
    x = x / 2
                                    ⟨_____⟩
  } else {
                                    ⟨_____⟩
                                    ⟨_____⟩
    res = res + y
                                    ⟨_____⟩
    y = 2 * y
                                    ⟨_____⟩
    x = (x - 1) / 2;
                                    ⟨_____⟩
  }
                                    ⟨_____⟩
}
                                    ⟨_____⟩
                                    ⟨res = a + b⟩
```

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und mit Hilfe des Hoare-Kalküls die Terminierung bewiesen werden.

Aufgabe 9 (Deck 1): (Codescape)

Lösen Sie die Räume von Deck 1 des Spiels Codescape.
Ihre Lösung für Räume dieses Codescape Decks wird nur dann für die Zulassung gezählt, wenn Sie die Lösung bis Mittwoch, den 31.10.2018 um 08:00 abschicken.