Informatik II (D-ITET)
Frühjahrssemester 2017
Prof. Friedemann Mattern
Hossein Shafagh (shafagh@inf.ethz.ch)

ETH Zürich
Institut für Pervasive Computing
Gruppe für Verteilte Systeme
http://vs.inf.ethz.ch

Übungsserie Nr. 3

Ausgabe: 15. März 2017
Abgabe: 21. März 2017

Hinweise

Für diese Serie benötigen Sie das Archiv
https://www.vs.inf.ethz.ch/edu/I2/downloads/u3.zip.

1. Aufgabe: (5 Punkte) Programmverifikation

ETH Codeboard link: https://codeboard.ethz.ch/ifee2u3t1

In der Vorlesung wurde der Algoritmus der altägyptischen Multiplikation vorgestellt und die Korrektheit einer konkreten Implementierung mithilfe einer Schleifeninvariante gezeigt. Unten ist eine Implementierung eines klassischen Multiplikationsalgorithmus angegeben, bei dem das Endresultat durch Akkumulation in einer Schleife berechnet wird.

```
1  static int f(int i, int j) {
2     int u = i;
3     int z = 0;
4     while (u > 0) {
5        z = z + j;
6        u = u - 1;
7     }
8     return z;
9  }
```

(1a) *(2 Punkte)* Geben Sie eine Schleifeninvariante für die oben gegebene Implementierung an, d.h. eine Bedingung, die vor und nach der Schleife (Zeilen 3 und 8) und am Anfang und am Ende eines jeden Schleifendurchlaufs gilt (Zeilen 4 und 7). *Hinweis: Betrachten Sie hierzu* z*,* $u \times j$ *und* $i \times j$*.*

(1b) *(3 Punkte)* Beweisen Sie mittels der Schleifeninvariante die Korrektheit der Implementierung.

(1c) *(0 Punkte)* Nehmen Sie an, der Schleifenkörper `z = z + j; u = u - 1;` wird ersetzt durch folgende Dummy-Statements: `z = z; u = u;`. Gilt dann noch die Schleifeninvariante? Wenn ja, ist Ihr Beweis aus der vorigen Teilaufgabe dann immer noch gültig?

2. Aufgabe: (5 Punkte) String und StringBuffer

ETH Codeboard link: `https://codeboard.ethz.ch/ifee2u3t2`

Strings gibt es in Java in zwei Varianten: den unveränderlichen (immutable) `String` und den veränderlichen (mutable) `StringBuffer`. Unveränderliche Objekte können nach ihrer Erzeugung nicht mehr modifiziert werden. Dem Compiler und der Laufzeitumgebung sind dann Optimierungen möglich, z.B. dass sich mehrere Objekte den selben Speicher teilen können. Falls jedoch eine Modifikation nötig ist, muss stattdessen eine Kopie angelegt werden, was Laufzeit und Speicher kostet. Es hängt also vom Anwendungsfall ab, welcher Typ effizienter ist.

In dieser Aufgabe implementieren Sie eine Variante der Caesar-Verschlüsselung[1]. Hierbei handelt es sich um eine der einfachsten Verschlüsselungen: Beim Verschlüsseln eines Texts (*encryption*) wird jeder Buchstabe des Klartexts (*plaintext*) durch den Buchstaben ersetzt, der im Alphabet 3 Zeichen weiter hinten steht. Bei der Entschlüsselung (*decryption*) wird entsprechend im Schlüsseltext (*ciphertext*) jedes Zeichen um 3 reduziert.

Wir berufen uns hier auf die ASCII-Tabelle und ignorieren der Einfachheit halber Überläufe am Ende des Alphabets. Das Verschlüsseln von *Xaver5* führt somit zum Schlüsseltext *[dyhu8*.

(2a) *(3 Punkte)* Implementieren Sie die Methode `decrypt`, welche einen gegebenen Schlüsseltext wieder in Klartext umwandelt. Verwenden Sie (ohne die Methodensignatur zu ändern) `StringBuffer` anstelle von `String`, um die Implementierung effizient zu gestalten. Testen Sie Ihre Implementierung mit Hilfe der Unittests.

Hinweis: Sie können sich an der Implementierung von `encrypt` orientieren, die allerdings Strings verwendet.

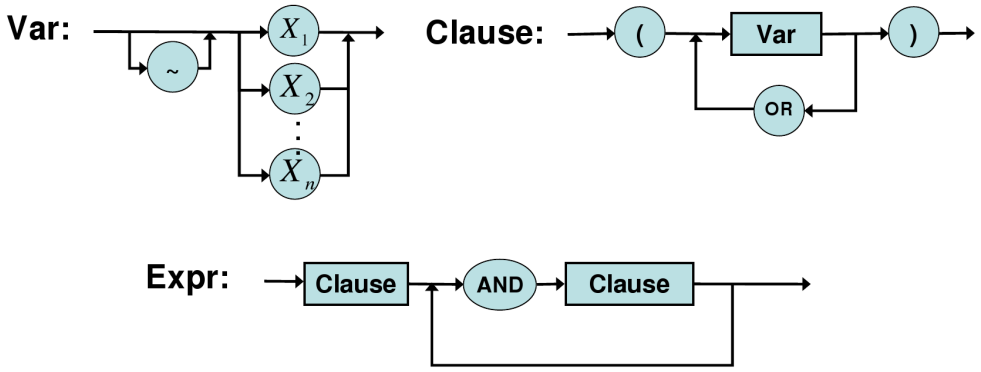
(2b) *(2 Punkte)* Das Framework enthält ausserdem eine Main-Methode in der Datei `Main.java`. Schauen Sie sich die Main-Methode genau an und beschreiben Sie ihr Verhalten. Führen Sie die Main-Methode aus – wie lautet die Ausgabe? Erklären Sie die unterschiedlichen Laufzeiten bei der Verschlüsselung und der Entschlüsselung.

[1] `http://en.wikipedia.org/wiki/Caesar_cipher`

3. Aufgabe: (3 Punkte) Syntaxdiagramm

ETH Codeboard link: `https://codeboard.ethz.ch/ifee2u3t3`

Betrachten Sie folgendes Syntaxdiagramm:



(3a) *(2 Punkte)* Geben Sie an, welche der folgenden Ausdrücke nach dem Diagramm *Clause* (*Klausel*) korrekt erzeugt werden können.

	erzeugbar	**nicht erzeugbar**		**erzeugbar**	**nicht erzeugbar**
X_2	○	○	$\sim (X_1 \text{ OR } \sim X_2)$	○	○
$(\sim X_1)$	○	○	$(X_2) \text{ OR } (\sim X_1 \text{ OR } X_2)$	○	○

(3b) *(1 Punkt)* Geben Sie an, welche der folgenden Ausdrücke nach dem Diagramm *Expr* (*Ausdruck*) korrekt erzeugt werden können.

	erzeugbar	**nicht erzeugbar**
$(X_1 \text{ OR } X_2) \text{ AND } (\sim X_1)$	○	○
$(X_1) \text{ AND } (\sim X_1 \text{ OR } \sim X_2) \text{ AND } (X_2)$	○	○

4. Aufgabe: (7 Punkte) Syntaxchecker

ETH Codeboard link: `https://codeboard.ethz.ch/ifee2u3t4`

(4a) *(1 Punkt)* Ergänzen Sie das Syntaxdiagramm auf Folie 455 im Skript, sodass leere Bäume und leere Teilbäume generiert werden können. Ein leerer Baum bzw. Teilbaum soll dabei durch ein '-' repräsentiert werden. Achten Sie darauf, dass durch Ihre Modifikation keine ungültigen Bäume generiert werden können.

(4b) *(6 Punkte)* Die Klasse *u3a4.KD* soll ein Syntaxchecker für Wurzelbäume in Klammerdarstellung werden. Vervollständigen Sie die Implementierung.

Hinweis: Schreiben Sie für *Baum*, *Nachfolger* und *Knoten* je eine eigene Funktion, die mit Hilfe der jeweils anderen Funktionen und analog zum jeweiligen Syntaxdiagramm den String sequentiell überprüft und die Anzahl der überprüften Zeichen zurück gibt.