Praktische Informatik II
FSS 2012 Programmierklausur

Prof. Dr. Heiner Stuckenschmidt

20.04.2012

Name, Vorname: _____

Matrikelnummer: _____

CVS-Username: *automatisch generierter Benutzername*

CVS-Password: *automatisch generiertes Passwort*

Unterschrift: _____

Viel Erfolg bei der Bearbeitung der Aufgaben!

Aufgabe 1: Logische Operationen (de.unima.ki.pi2.ptest.aufgabe1)

In der Klasse `LogicBWImpl` finden Sie die leeren Methodenrümpfe der Methoden `computeAnd`, `computeOr`, und `computeNot`. Die Methode `computeAnd` soll zwei boolsche Feldvariablen elementweise verunden und das Resultat in einer boolschen Feldvariable zurückgeben. Entsprechend soll `computeOr` die elementweise Veroderung der Eingabevariablen und `computeNot` die elementweise Negation einer Eingabevariable zurückgeben. Hier ein Beispiel, in dem die drei Methoden verwendet werden:

```
boolean[] x = new boolean[]{true,  false, true}
boolean[] y = new boolean[]{false, false, true}
LogicBW logic = new LogicBW();
boolean[] z1 = logic.computeAnd(x,y)
boolean[] z2 = logic.computeOr(x,y)
boolean[] z3 = logic.computeNot(x)
```

Führt man dieses Codefragment aus, so sollen die Variablen `z1`, `z2`, und `z3` folgende Werte haben:

- `z1 == {false, false, true}`
- `z2 == {true, false, true}`
- `z3 == {false, true, false}`

Weiterhin soll `null` zurückgegeben werden, wenn die Eingabefeldvariablen nicht die gleiche Anzahl an Elementen haben. Ebenso soll `null` zurückgegeben werden, wenn das erste oder zweite Eingabeargument kein Element hat (Länge = 0).

Ihre Aufgabe besteht darin, die Methoden `computeAnd`, `computeOr`, und `computeNot` zu vervollständigen, so dass die oben beschriebene Funktionalität implementiert wird!

Aufgabe 2: Buchstaben zählen (de.unima.ki.pi2.ptest.aufgabe2)

Die Klasse `CharCounterImpl` soll zunächst einen Eingabetext parsen (Methode `parse`) und zählen, wie oft ein Buchstabe in diesem Text vorkommt.[1] Nachdem ein Text geparst wurde, soll mittels eines Aufrufs von `getNumOfChars` abgefragt werden, wie oft ein bestimmter Buchstabe aufgetreten ist.

```
CharCounter counter = new CharCounterImpl();
counter.parse("alpha und beta");
int n1 = this.counter.getNumOfChars('a');
int n2 = this.counter.getNumOfChars(' ');
int n3 = this.counter.getNumOfChars('z');
```

Wird dieses Codefragement ausgeführt, so gilt

- `n1 == 3`
- `n2 == 2`
- `n3 == 0`.

Wird die Funktion `parse` mehrfach aufgerufen, so sollen die Rückgabewerte von `getNumOfChars` sich stets auf den jeweils letzten Aufruf beziehen. Wurde `parse` noch nicht aufgerufen, so soll `getNumOfChars` für eine beliebige Eingabe 0 zurückgeben.

Ihre Aufgabe besteht darin, die Klasse `CharCounterImpl` zu vervollständigen, so dass die oben beschriebene Funktionalität implementiert wird!

Zusätzlicher Hinweis Die Klasse `Integer` verhält sich zu dem einfachen Datentyp `int` so wie die Klasse `Character` sich zu dem einfachen Datentyp `char` verhält. Weiterhin finden Sie in der `main` Methode der Klasse `CharCounterImpl` ein Codefragment, das für eine einfache Lösung der Aufgabe nützlich sein kann.

[1] Eine solche Funktionalität wird zum Beispiel als Vorbereitung für die Huffman Codierung benötigt.

Aufgabe 3: Addition mit Übertrag (de.unima.ki.pi2.ptest.aufgabe3)

In dieser Aufgabe geht es um die Addition ganzer positiver Binärzahlen. Eine Binärzahl ist definiert als Folge von Ziffern $b_n, b_{n-1}, ..., b_1, b_0$ mit $b_i \in \{0,1\}$. Der dargestellte Zahlenwert ist $\sum_{i=0}^{n} b_i \cdot 2^i$.

In der Klasse `AdderImpl` wird die Ziffernfolge durch ein Array aus boolschen Variablen dargestellt. Die Binärzahl 0100, die der Dezimalzahl 4 entspricht, kann also wie folgt dargestellt werden:

```
boolean[] num = new boolean[]{false, false, true, false}
```

Die Variable `num[i]` entspricht der Ziffer b_i.

Die Klasse `AdderImpl` soll mittels der Methode `add` zwei ganze positive Zahlen der obigen Darstellung entsprechend addieren. Die Methode erhält als Eingabe zwei Zahlen (als boolsche Feldvariablen in obigem Format) und gibt das Resultat der Addition zurück (ebenfalls als boolsche Feldvariable). Zudem wird hierbei das Carry-Bit auf `true` gesetzt, wenn es zu einem Überlauf gekommen ist, ansonsten wird das Carry-Bit auf `false` gesetzt. Hierzu ein Beispiel:

```
Adder adder = new AdderImpl();
boolean[] a = new boolean[]{true, true,  true};
boolean[] b = new boolean[]{true, false, false};
boolean[] result = adder.add(a, b);
boolean carry = adder.getCarryBit();
```

Nach Ausführung dieses Codes hat `result` den Wert {`false, false, false`} und `carry` hat den Wert `true`. Haben die Eingabeargumente der Methode `add` nicht die gleiche Länge, so soll die Methode `null` zurückgeben. Der Wert des Carry-Bits ist in diesem Fall beliebig. Weiterhin soll die Länge des Ergebnisses, das zurückgegeben wird, der Länge der Eingabeargumente entsprechen, was somit die Wortgröße bestimmt in der gerechnet wird. Kommt es, was diese Wortgröße betrifft zu einem Überlauf, so soll das Carry-Bit gesetzt werden.

Erweitern Sie die Methode `add`, so dass die beschriebene Funktionalität implementiert wird!

Zusätzlicher Hinweis In der Klasse finden Sie eine Methode `getNumOfBitsSet`. Sie dürfen (müssen aber nicht) diese Hilfsmethode in ihrem Code verwenden.

Aufgabe 4: Hamming Code (de.unima.ki.pi2.ptest.aufgabe4)

Im folgenden beschäftigen uns mit dem Hamming-Code. Insbesondere betrachten wir die aus Vorlesung und Übung bekannte Variante 4/7 bei der 4 Datenbits in 7 Bits codiert werden indem 3 Paritätsbits hinzugefügt werden. Im folgenden wollen wir annehmen, dass die Datenwörter durch boolsche Feldvariablen der Länge 4 und codierte Bitfolgen durch boolsche Feldvariablen der Länge 7 implementiert werden.

In der Klasse `HammingCodeImpl` sind zwei Methoden definiert. Die Methode `encode` ist bereits vollständig definiert. Sie bildet einen zu übertragenden Datenblock auf eine codierte Nachricht ab. Die Methode `decode` soll eine codierte Nachricht decodieren, d.h. auf den ursprünglichen Datenblock abbilden. Dabei soll entsprechend des 4/7 Hamming Codes auch eine Korrektur vorgenommen werden, falls Übertragungsfehler erkennbar sind. Der Rückgabewert der Methode `encode` ist eine boolsche Feldvariable der Länge 7; der Rückgabewert der Methode `decode` ist eine boolsche Feldvariable der Länge 4. Dementsprechend ist der Eingabewert der Methode `encode` eine boolsche Feldvariable der Länge 4; der Eingabewert der Methode `decode` ist eine boolsche Feldvariable der Länge 7. Im folgenden kann davon ausgegegangen werden, dass diese Konvention befolgt wird (d.h. diesbezügliche Fehler müssen nicht abgefangen werden).

Erweitern Sie die Methode `decode`, so dass die beschriebene Funktionalität implementiert wird!

Zusätzlicher Hinweis Betrachten Sie die Methode `decode`, falls Sie sich nicht mehr sicher sind wie der Hamming-Code arbeitet. Dies sollte es Ihnen ermöglichen die notwendigen Zusammenhänge leicht zu rekonstruieren.