
Aufgaben zur Klausur **C** und **Objektorientierte Programmierung** im SS 2009 (WI h103, II h105, MI h353)

Zeit: 150 Minuten

erlaubte Hilfsmittel: keine

Bitte tragen Sie Ihre Antworten und fertigen Lösungen ausschließlich an den gekennzeichneten Stellen in das Aufgabenblatt ein. Ist ihre Lösung wesentlich umfangreicher, so überprüfen Sie bitte nochmals Ihren Lösungsweg.

Sollten Unklarheiten oder Mehrdeutigkeiten bei der Aufgabenstellung auftreten, so notieren Sie bitte, wie Sie die Aufgabe interpretiert haben.

Verwenden Sie in den zu entwickelnden Java Programmteilen keine impliziten Konversionen zwischen einfachen Datentypen und kein Autoboxing und Autounboxing.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 18 Seiten.

Aufgabe 1:

Gegeben sei das folgende C-Programm zur Verarbeitung von Mengen als Bitstrings.

```
#include <stdio.h>

typedef unsigned char Set;
#define SetMax 8

void print(Set s) {
    char * digits = "0123456789abcdef";
    printf("%c%c\n",
           (char)(digits[(s>>4) & 0x0f]),
           (char)(digits[s & 0x0f]));
}

#define single(i) ( (Set)(1 << (i)) )
#define first(n) (single(n) - 1)
#define interval(n,m) (first(m+1) ^ first(n))

int main(void) {
    Set s1;

    print( (Set)-1 );
    print( single(2) );
    print( single(SetMax / 2 + 1) );

    print( interval(2,5) );
    print( interval(5,2) );
    print( interval(1,SetMax) );

    print( 5 | 10 );
    print( 5 || 10 );
    print( first(SetMax / 2 - 1) );

    s1 = interval(2,4) & ~interval(3,5); print(s1);
    s1 = interval(2,4) ^ interval(3,5); print(s1);

    s1 = 4 + 16 + 64;
    s1 ^= s1 & (~s1 + 1); print(s1);

    return 0;
}
```

Die Mengen sind in diesem Beispiel 8 Bits lang, können also die Elemente 0, 1, . . . , 7 enthalten. *print* gibt eine Menge als 2-stellige Hexadezimalzahl aus. Die Menge, die die Elemente 1,2 und 3 enthält, würde als *0e* ausgegeben werden.

Welche 12 Ausgabezeilen erzeugt dieses Programm?

1)

2)

3)

4)

5)

6)

7)

8)

9)

10)

11)

12)

Aufgabe 2:

Seien f_1, f_2, g_1, g_2 Funktionen vom Typ $\mathbb{N} \rightarrow \mathbb{R}$.

Weiter gelte $f_i(n) \in O(g_i(n))$ und $c_i \in \mathbb{R}$ für $i \in \{1, 2\}$.

1. Aus welcher Komplexitätsklasse ist $f(n) = f_1(n) + c_1 * n^2$

.....

2. Aus welcher Komplexitätsklasse ist $f(n) = (f_1(n))^2 + f_1(n)$

.....

3. Aus welcher Komplexitätsklasse ist $f(n) = f_1(n) * (f_2(n) + c_2 * f_1(n))$

.....

4. Aus welcher Komplexitätsklasse ist $f(n) = (c_1 * f_1(n))^2$

.....



Aufgabe 3:

Gegeben seien die C Makro-, Typ-, Variablen- und Funktionsdeklarationen und -definitionen:

```
typedef double Element;
```

```
typedef Element *Row;
```

```
typedef Row *Rows;
```

```
struct Descr
```

```
{
```

```
    Rows rows;
```

```
    Element *elems;
```

```
    int width;
```

```
    int height;
```

```
};
```

```
#define trunc(x) ((int)(x))
```

```
#define at(a,i,j) ((a)->rows[i][j])
```

```
typedef struct Descr * Matrix;
```

```
typedef Matrix (* MatrixConstr)(int w, int h);
```

```
typedef Matrix (* UnaryMatrixOp)(Matrix m);
```

```
extern Matrix newMatrix (int w, int h);
```

```
extern Matrix zeroMatrix (int w, int h);
```

```
extern Matrix unitMatrix (int w, int h);
```

```
extern Matrix addMatrix (Matrix m1, Matrix m2);
```

```
extern Matrix transposeMatrix (Matrix m);
```

```
Matrix m1;
```

```
MatrixConstr c1;
```

Bestimmen Sie für die folgenden Ausdrücke den Typ gemäß ANSI-C. Sollten fehlerhafte oder logisch nicht sinnvolle Ausdrücke vorkommen, so kennzeichnen Sie diese mit dem Wort FEHLER. Nutzen Sie für die Typbestimmung die in dem Programmstück deklarierten Typnamen.

- m1->elems
- *((*m1).elems)
- m1->rows[1][2]
- m1->rows[m1->width]
- m1->rows[1,2]
- at(m1,0,0)
- at(m1,trunc(at(m1,1,1)),+1)
- zeroMatrix
- c1(1,1)
- at(c1(1,1),1,1)
- c1 = unitMatrix
- c1 = transposeMatrix
- (*addMatrix(m1,m1)).rows + 1

Aufgabe 4:

Gegeben sind die folgenden Datentypdefinitionen und Prototypen

```
#include <string.h>
```

```
#include <assert.h>
```

```
typedef char * Element;
```

```
typedef struct Node * List;
```

```
struct Node {
```

```
    List next;
```

```
    Element e;
```

```
};
```

```
extern int invList(List l);
```

```
extern List cons(Element e, List l);
```

```
extern List insert(Element e, List l);
```

Entwickeln die Testfunktion *invList*, mit der überprüft werden kann, ob die in einer Liste enthaltenen Elemente in aufsteigender Reihenfolge gespeichert sind. Es soll dabei ausgeschlossen werden, dass Elemente doppelt in der Liste enthalten sind.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Entwickeln Sie die Funktion *insert*, die ein Element in eine sortierte Liste einfügt. Die Funktion soll rekursiv arbeiten. Sie soll so arbeiten, dass die Invariante erhalten bleibt, d.h. wenn vor dem Einfügen in eine Liste *l* die Invariante gilt, so auch nach dem Einfügen.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Die Funktion *cons*, die ein Element vorne an eine Liste anhängt, kann bei der Implementierung verwendet werden.

Aufgabe 5:

Bitte lesen Sie diese Aufgabe vollständig durch, bevor Sie beginnen, die einzelnen Lösungsteile zu entwickeln.

In dieser Aufgabe soll ein einfacher Baukasten für die Verarbeitung von Folgen von Textzeilen entwickelt werden. Diese Zeilenfolgen werden durch eine Schnittstelle *LineStream* beschrieben.

```
interface LineStream {  
    boolean hasMoreLines();  
    String nextLine();  
}
```

Das Prädikat *hasMoreLines* zeigt an, ob noch Zeilen in dem Zeilenstrom enthalten sind, mit *nextLine* kann man dann die nächste Zeile aus dem Zeilenstrom lesen.

Um aus einem *Reader*-Object einen Datenstrom zu konstruieren, kann die Klasse *ReaderLineStream* verwendet werden.

```
import java.io.Reader;
```

```
public class ReaderLineStream  
    implements LineStream {  
  
    private Reader r;  
    private int lookAhead;  
  
    public ReaderLineStream(Reader r1) {  
        r = r1;  
        advance();  
    }  
  
    private boolean eof() {  
        return lookAhead == -1;  
    }  
  
    private void advance() {  
        try {  
            lookAhead = r.read();  
        }  
        catch (Exception e) {  
            lookAhead = -1;  
        }  
        if ( eof() ) {  
            try {  
                r.close();  
            }  
            catch (Exception e) { }  
        }  
    }  
}
```

```

    }
}

public boolean hasMoreLines() {
    return ! eof();
}

public String nextLine() {
    StringBuffer l = new StringBuffer(" ");

    while ( ! eof() ) {
        if ( lookAhead == '\n' ) {
            advance();
            return new String(l);
        }

        if ( lookAhead == '\r' ) {
            advance();
            if ( lookAhead == '\n' )
                advance();
            return new String(l);
        }

        l.append((char)lookAhead);
        advance();
    }

    return new String(l);
}
}
}

```

In Java wird bei Methodenaufrufen immer das dynamische Binden verwendet. Dieses ist von der Laufzeit her ineffizienter als statisches Binden. In der Klasse *ReaderLineStream* werden sehr häufig die Hilfsmethoden *advance* und *eof* aufgerufen. Ist es sinnvoll, die Anweisungen für diese Methoden an die Aufrufstellen einzukopieren und diese Hilfsroutinen zu löschen, um Laufzeit für dynamisches Binden zu vermeiden?

ja nein

Begründung:

.....

Analog zu einem *Reader*-Objekt kann man aus einem *String*-Objekt einen Zeilenstrom konstruieren. Dieses wird durch die Klasse *StringLineStream* beschrieben. Vervollständigen Sie die Klasse *StringLineStream* so, dass sie eine analoge Semantik besitzt wie die *Reader*-Klasse.

Hinweis: In der *String*-Klasse sind die Methoden *length* und *charAt* definiert.

```
public class StringLineStream
  implements LineStream {

  private String s;
  private int lookAhead;

  .....

  public StringLineStream(String s1) {

    .....

    .....
    advance();
  }

  private boolean eof() {

    .....

    .....
  }

  private void advance() {

    .....

    .....

    .....

    .....

    .....

    .....

    .....
  }
}
```

```

public boolean hasMoreLines() {
    return ! eof();
}

public String nextLine() {
    StringBuffer l = new StringBuffer(" ");

    while ( ! eof() ) {
        if ( lookAhead == '\n' ) {
            advance();
            return new String(l);
        }

        if ( lookAhead == '\r' ) {
            advance();
            if ( lookAhead == '\n' )
                advance();
            return new String(l);
        }

        l.append((char)lookAhead);
        advance();
    }

    return new String(l);
}
}

```

Ist diese Implementierung der beiden Klassen *ReaderLineStream* und *StringLineStream* aus Software-technischer Sicht eine gute Lösung?

ja nein

Begründung:

.....

.....

.....

Weiter sei folgende Klasse gegeben:

```
public class CatLines
  implements LineStream {
  protected LineStream s;
  protected String lookAhead;

  public CatLines(LineStream s1) { init(s1); }
  protected CatLines() { }

  protected void init(LineStream s1) {
    s = s1;
    advance();
  }
  protected boolean isRedundantLine() {
    return false;
  }
  protected final void advance() {
    if ( s.hasMoreLines() ) {
      lookAhead = s.nextLine();
      if ( isRedundantLine() )
        advance();
    }
    else
      lookAhead = null;
  }
  public boolean hasMoreLines() {
    return lookAhead != null;
  }
  public String nextLine() {
    String res = lookAhead;
    advance();
    return res;
  }
}
```

Ist es aus Software-technischer Sicht sinnvoll, diese Klasse zu realisieren?

ja nein

Begründung:

.....
.....

Macht es aus Software-technischer Sicht einen Unterschied, ob die Methode *advance* in der Klasse *CatLines* als *private* oder *protected final* deklariert wird?

ja nein

Begründung:

.....
.....

Entwickeln Sie die einfachste Klasse *FastCatLines*, die die gleiche Funktionalität implementiert wie *CatLines*.

.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....

Entwickeln Sie eine Klasse *RemoveEmptyLines*, die aus einem Zeilenstrom alle Leerzeilen (Zeilen die keine Zeichen enthalten) löscht.

```
public class RemoveEmptyLines
{
    .....
    public RemoveEmptyLines(LineStream s1) {
        .....
        .....
    }
    .....
    .....
    .....
    .....
    .....
    .....
}
```


RemoveEmptyLines ist ein ganz spezieller Filter zum Selektieren von Zeilen. Eine Verallgemeinerung wird dann möglich, wenn man beliebige Prädikate auf die Zeilen eines Zeilenstroms anwenden kann. Hierzu sei folgende Schnittstelle gegeben:

```
interface StringPredicate {  
    boolean isValid(String s);  
}
```

Vervollständigen Sie die folgende Klasse *GrepLines*, mit der über ein Prädikat die Zeilen ausgewählt werden können, die dieses Prädikat erfüllen. Dieses ist eine Verallgemeinerung der Klasse *RemoveEmptyLines*. Die Klasse wird also eine ähnliche Struktur besitzen.

```
public class GrepLines  
  
    .....  
    {  
    private StringPredicate p;  
  
    public GrepLines(StringPredicate p1, LineStream s1) {  
  
        .....  
  
        .....  
  
        .....  
  
        .....  
  
        .....  
  
        .....  
  
        .....  
  
        .....  
  
        .....  
  
        .....  
  
        .....  
  
        .....  
  
    }  
  
    .....  
  
    .....  
  
    .....  
  
    .....  
  
    .....  
  
    .....  
  
    }  
  
    }
```

Zwei (oder mehrere) Zeilenströme können zu einem neuen Strom kombiniert werden. Die Klasse *CombineLineStreams* legt hierfür das gemeinsame Muster fest:

```

abstract public class CombineLineStreams
  implements LineStream {
  protected LineStream s1;
  protected LineStream s2;

  public CombineLineStreams(LineStream l1, LineStream l2) {
    s1 = l1;
    s2 = l2;
  }
}

```

Vervollständigen Sie die Klasse *ConcatLineStreams*. Diese soll ermöglichen, dass zwei Zeilenströme zu einem konkateniert werden, ähnlich dem *cat*-Kommando unter UNIX.

```

public class ConcatLineStreams extends CombineLineStreams {
  public ConcatLineStreams(LineStream l1, LineStream l2) {
    super(l1,l2);
  }
  public boolean hasMoreLines() {
    .....
    .....
    .....
    .....
    .....
    .....
    .....
  }
  public String nextLine() {
    .....
    .....
    .....
  }
}

```