Compilerbau
Übungsblatt 5

Yangzi Zhang, Michael Gottschalk, Felix J. Oppermann

22. Juni 2006

Aufgabe 14

a)

Folgende antlr-Scannerdefinition erkennt die benötigten Zahlen, Operatoren und Bezeichner. Whitespace wird mit dem Kommando `$setType(Token.SKIP);` automatisch von antlr herausgefiltert und muss im Parser nicht mehr behandelt werden. Mit `NEWLINE` soll eine Eingabe abgeschlossen werden. Nummern werden als Ganzzahlen ohne Kommastelle definiert. Identifier werden, wie in vielen Programmiersprachen üblich, so definiert, dass sie mit einem Buchstaben oder einem Unterstrich beginnen müssen und im Weiteren auch Zahlen beinhalten dürfen.

```
class L extends Lexer;

NUMBER : ('0'..'9')+;
IDENTIFIER : ('a' .. 'z' | 'A' .. 'Z' | '_')
                ('0'..'9'|'a' .. 'z' | 'A' .. 'Z' | '_')*;
MULT : '*';
MINUS : '-';
PLUS : '+';
DIV : '/';
LPAREN : '(';
RPAREN : ')';

WHITESPACE:
    ( ' ' | '\t')
    { $setType(Token.SKIP); }
;

NEWLINE : '\n' | "\r\n";
```

b)

Folgende Grammatik erkennt gültige Eingaben. Wir haben uns in diesem Aufgabenteil schon an die Syntax von antlr angelehnt, um die Erweiterung durch Aktionen im nächsten Schritt zu verdeutlichen. Auch haben wir schon Linksrekursionen entfernt, da diese mit einem LL-Parser nicht aufgelöst werden können.

startRule → expression NEWLINE
expression → literal operator expression? |
LPAREN expression RPAREN (operator expression)?
literal → number | identifier
number → (PLUS | MINUS)? NUMBER
identifier → IDENTIFIER
operator → MULT | DIV | MINUS | PLUS

c)

Folgende antlr-Spezifikation erzeugt einen Parser, der arithmetische Ausdrücke in Postfixausdrücke umwandelt. Jede Regel gibt hier als String die von ihr erzeugte Ausgabe zurück. Das Problem mit den unären und binären Operatoren wurde einfach dadurch gelöst, dass ein eventueller unärer Operator als Bestandteil der Zahl angesehen wird und in der Postfixdarstellung dementsprechend direkt vor der Zahl ausgegeben wird. Von einem binären Operator ist er dadurch unterscheidbar, dass nach einem binären Operator immer ein Leerzeichen folgt. Die Operatorpräzedenz („Punkt-vor Strichrechnung“) wurde dadurch realisiert, dass hochpriorisierte Operatoren immer direkt hinter das auf den Operator folgende Literal geschrieben werden, niedrig priorisierte Operatoren aber immer so weit rechts wie möglich platziert werden. Um dies zu realisieren, wird der Regel für `expression` als Parameter übergeben, welcher eventuelle Operator vor dem aktuellen Ausdruck

stand. Zur Entscheidung, welcher Operator hoch und welcher niedrig priorisiert ist, wurde die Hilfsfunktion `isHighPriority()` definiert.

```
class P extends Parser;

{
    private static boolean isHighPriority(String op) {
        return op.equals("* ") || op.equals("/ ");
    }
}

startRule returns [String res=""] : (res = expression[""]) NEWLINE;

expression [String op_in] returns [String res=""]
            {String e1="", e2 = "", o="", id="";} :

            (id = literal) ((o = operator) (e1 = expression[o]))?
                {
                 res += id;
                 if (isHighPriority(op_in))
                    res += op_in;
                 res += e1;
                 if (!isHighPriority(o))
                    res += o;
                }
            |

             LPAREN (e1=expression[""]) RPAREN
                            ((o=operator) (e2=expression[o]))?
                {res = e1;
                 if (isHighPriority(op_in))
                    res += op_in;
                 res += e2;
                 if (!isHighPriority(o))
                    res += o; }
            ;

literal returns [String res=""] : (res = number) | (res = identifier);

number returns [String res=""] : (PLUS {res="+";} | MINUS {res="-";} )?
                                n: NUMBER {res += n.getText() + " "; };

identifier returns [String res=""]: id : IDENTIFIER {res=id.getText()+ " ";};

operator returns [String res=""]: MULT {res = "* ";}  |
                                  DIV {res = "/ ";}   |
                                  MINUS {res = "- ";} |
                                 (PLUS {res = "+ ";});
```

d)

Folgende Klasse erzeugt aus Scanner und Parser ein lauffähiges Programm. Dazu wird zunächst ein Scanner erzeugt, dem als Eingabe die Standardeingabe übergeben wird. Dem Parser wird das Scanner-Objekt übergeben und es wird der Rückgabewert der Startregel auf der Standardausgabe ausgegeben.

```
import java.io.*;
```

```
class Main {
    public static void main(String[] args) {
        try {
            L lexer = new L(new DataInputStream(System.in));
            P parser = new P(lexer);
            System.out.println("Output: "+ parser.startRule());
        } catch(Exception e) {
            System.err.println("exception: "+e);
        }
    }
}
```

Kompilieren kann man die Klasse mit folgenden Kommandos (Lexer- und Parser-Definition befinden sich in der Datei t.g):

```
java antlr.Tool t.g
javac *.java
```

Dazu muss sich die Datei antlr.jar im Classpath befinden.

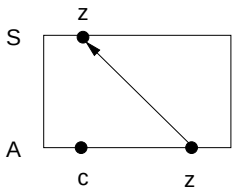
Testbeispiele:

Eingabe	**Ausgabe**
`23+a*3/(4-b)`	`23 a 3 * 4 b - / +`
`1-5+10*(a5-20)`	`1 5 10 a5 20 - * + -`
`1+4*6+5`	`1 4 6 * 5 + +`
`-5 + 40 * -20 + (a-b) * 10`	`-5 40 -20 * a b - 10 * + +`
`(1+ _var10)- (goodVar * 10)*(20-30)/10`	`1 _var10 + goodVar 10 * 20 30 - * 10 / -`
`((1-10)+5`	*Fehler*
`5g + 435`	*Fehler*

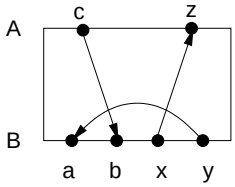
Aufgabe 15

a) Abhängigkeitsgraphen

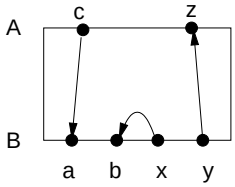
S → A



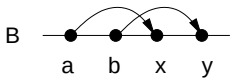
A → aB



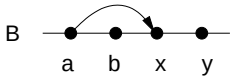
A → bB



B → c



B → d

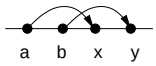


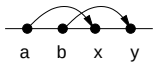
b) IO-Graphen

1. Schritt: IO[A] sei der is-Graph von A ohne Kanten für alle A ∈ NT.

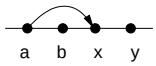
2. Schritt: Fixpunktiteration über alle Produktionen p.

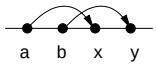
p: B → c

IO[B] = IO[B] ergänzt um die Kanten aus is (D(p)[]) (= 

) = 

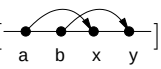
p: B → d

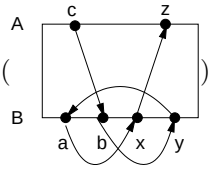
IO[B] = IO[B] ergänzt um die Kanten aus is (D(p)[]) (= 

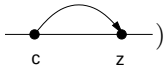
) = 

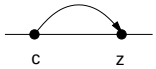
p: A → aB

IO[A] = IO[A] ergänzt um die Kanten aus

is (D(p)[

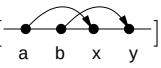
]) (= is (

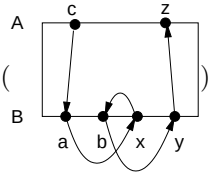
) = 

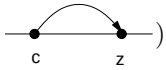
) = 

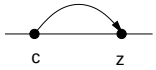
p: A → bB

IO[A] = IO[A] ergänzt um die Kanten aus

is (D(p)[

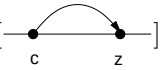
]) (= is (

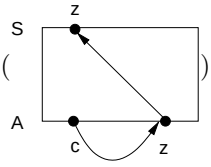
) = 

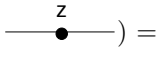
) = 

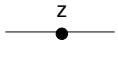
p: S → A

IO[S] = IO[S] ergänzt um die Kanten aus

is (D(p)[

]) (= is (

) = 

) = 

Es sind keine weiteren Iterationen nötig, da sich keine Veränderungen mehr ergeben würden.

Die Grammatik ist absolut nichtzirkulär, da alle D(p)-Graphen, die im obigen Algorithmus erzeugt wurden, keine Zyklen aufweisen.

Aufgabe 16

Quelltext

Listing 1: Main.java

```
/*
 * Main
 *
 * Version 1.0.0
 *
 * 2006-06-20
 *
 * (C) Yangzi Zhang, Michael Gottschalk, Felix J. Oppermann 2006
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.StreamTokenizer;

/**
 * Diese Klasse stellt einen Wrapper zur direkten Interaktion mit der
 * Speicherverwaltung bereit. Der Nutzer kann direkt Komandos an die
 * Speicherverwaltung senden und die Ausgaben beobachten.
 *
 * @author Yangzi Zhang
 * @author Michael Gottschalk
 * @author Felix J. Oppermann
 */
public class Main {

	static MemoryAllocation memory;

	/**
	 * Die Methode gibt den &uuml;bergeben Text an den Benutzer aus und liest
	 * eine numerische Eingabe vom Benutzer. Bei fehlerhaften Eingaben wird der
	 * Nutzer automatisch zur erneuten Eingabe aufgefordert.
	 *
	 * @param text
	 *            Der anzuzeigende Text
	 * @return Die vom Benutzer eigegeben Ganzzahl
	 */
	protected static int readNumber(String text) {
		int number = 0;

		InputStreamReader inputStreamReader = new InputStreamReader(System.in);
		BufferedReader reader = new BufferedReader(inputStreamReader);

		boolean done = false;
		while (!done) {
			System.out.print(text);

			String input = null;
			try {
				input = reader.readLine();
			} catch (IOException e) {
				/*
				 * Sollte bei Tastertureingaben nicht auftreten
				 */
			}

			try {
				number = Integer.decode(input);
				done = true;
			} catch (NumberFormatException e) {
				/*
				 * Nichts zu tun. Der Benutzer wird erneut gefragt.
				 */
			}
		}
		return number;
	}

	/**
	 * Die Methode implementiert einen einfachen Komandointerpreter. Der
	 * Benutzer kann durch fest vorgegeben einfache Komandos (malloc, free,
	 * help, exit) mit dem System interagieren. Die eingebenen Befehle werden
	 * rudiment&auml;r auf Fehler &uuml;berpr&uuml;ft und dann direkt an die
	 * Speicherverwaltung weitergegeben.
	 *
	 * Die Methode setzt voraus, dass das Klassenmenber memory durch eine
```

```
     * Instanz der Speicherverwaltung belegt ist.
     */
    protected static void interpretCommands() {
            boolean done = false;

            InputStreamReader inputStreamReader = new InputStreamReader(System.in);

            while (!done) {
                    System.out.print("> ");

                    StreamTokenizer tokenizer = new StreamTokenizer(inputStreamReader);

                    int token;

                    try {
                            token = tokenizer.nextToken();
                            if (token != StreamTokenizer.TT_WORD) {
                                    System.out.println("Invalid argument");
                                    continue;
                            }
                            if (tokenizer.sval.equals("malloc")) {
                                    if (tokenizer.nextToken() != StreamTokenizer.TT_NUMBER) {
                                            System.out.println("Invalid argument");
                                            continue;
                                    }
                                    int size = (int) tokenizer.nval;
                                    try {
                                            int address = Main.memory.malloc(size);
                                            System.out.println("address: " + address);
                                    } catch (OutOfMemoryException e) {
                                            System.out.println("Out of Memory");
                                    } catch (InvalidBlockSizeException e) {
                                            System.out.println("Invalid size");
                                    }
                            } else if (tokenizer.sval.equals("free")) {
                                    if (tokenizer.nextToken() != StreamTokenizer.TT_NUMBER) {
                                            System.out.println("Invalid argument");
                                            continue;
                                    }
                                    int address = (int) tokenizer.nval;
                                    try {
                                            Main.memory.free(address);
                                            System.out.println("Freed memory");
                                    } catch (InvalidAddressException e) {
                                            System.out.println("Could not free memory");
                                    }
                            } else if (tokenizer.sval.equals("help")) {
                                    System.out.println("Commands:");
                                    System.out.println("malloc SIZE: Allocates memory of size "
                                                    + "SIZE. The address of the allocated "
                                                    + "memory is returned");
                                    System.out.println("free ADDRESS: Frees a memory block "
                                                    + "starting at the address ADDRESS");
                                    System.out.println("exit: Terminates the programme");
                                    System.out.println("help: This information");
                            } else if (tokenizer.sval.equals("exit")) {
                                    done = true;
                            } else {
                                    System.out.println("Invalid command");
                            }

                    } catch (IOException ioe) {
                            /*
                             * Sollte bei Tastertureingaben nicht auftreten
                             */
                    }
            }
    }

    /**
     * Haupprogram. Es werden zun&auml;chst die Parameter der Speicherverwaltung
     * abgefragt. Anschlie&szlig;end kann mit dem System &uuml;ber einen
     * einfachen Komandointerpreter interagiert werden.
     *
     */
    public static void main(String[] args) {

            int n = 0;
            int min = 0;
            int max = 0;

            n = readNumber("Memory size (2^n): ");
            min = readNumber("Minimal block size (2^min): ");
            max = readNumber("Maximal block size (2^max): ");
```

```
                Main.memory = new MemoryAllocation(n, min, max);
                interpretCommands();
        }
}
```

Listing 2: MemoryAllocation.java

```
/*
 * MemoryAllocation
 *
 * Version 1.0.0
 *
 * 2006-06-20
 *
 * (C) Yangzi Zhang, Michael Gottschalk, Felix J. Oppermann 2006
 */

import java.util.Hashtable;

/**
 * Die Klasse MemoryAllocation stellt Methoden zur Speicherverwaltung zur
 * Verf&uuml;gung. Durch die Klasse wird die Belegung von Speicherbl&ouml;cken
 * zur Laufzeit &uuml;berwacht. Speicher kann &uuml;ber die Methode malloc
 * angefordert werden. Die angeforderte Speicherplatzgr&ouml;&szlig;e wird
 * automatisch auf einen passende Blockgr&ouml;&szlig;e umgerechnet. Dies ist
 * f&uuml;r den Anwender transparent. War die Anfrage erfolgreich, so ist
 * garantiert, dass ab der zur&uuml;ckgegeben Anfangsadresse mindestens der
 * angeforderte Speicher zur Verf&uuml;gung steht. Speicher kann mittels der
 * Methode free unter angebe der Startadresse wieder freigegeben werden. Intern
 * Verwendet die Klasse das Buddy-Verfahren zur Speicherverwaltung um eine
 * geringe Fragmentierung bei eine guten reaktionszeit zu erreichen. Die
 * zentralen Datenstrukturen wurden als Felder realisiert um eine Speicherplatz
 * sparende und schnelle Implementierung zu erreichen.
 *
 * @author Yangzi Zhang
 * @author Michael Gottschalk
 * @author Felix J. Oppermann
 */
public class MemoryAllocation {

        /**
         * Gr&ouml;&szlig;e des verwalteten Speichers
         */
        int memorySize;

        /**
         * Minimale Blockgr&ouml;&szlig;e
         */
        int min;

        /**
         * Maximale Blockgr&ouml;&szlig;e
         */
        int max;

        /**
         * Feld zur Verwaltung der Bl&ouml;cke. Das Feld enth&auml;lt die
         * Buddy-Listen. Das Feld ist dabei stufenf&ouml;rmig Aufgebaut. Implizit
         * wird auch die im Buddy-Verfahren verwendete Baumstruktur zwischen den
         * Bl&ouml;cken abgebildet. Die Links dieser Struktur kann leicht duch eine
         * Brechnung der Adressen im Feld rekonstruiert werden.
         */
        Block[][] blockLists;

        /**
         * Hashtabelle zur Verwaltung der bereits belegten Speicherbereiche. Diese
         * Tabelle dient dem leichen Auffinden eines belegten Blocks bei der
         * Freigabe.
         */
        Hashtable<Integer, Block> usedList;

        /**
         * Durch den Standardkonstruktor wird die Klasse zur Verwaltung eins 2^16
         * Byte gro&szlig;er Speichers initialisiert. Die minimale
         * Blockg&ouml;&szlig;e betr&auml;gt 2^8. Die maximale Blockg&ouml;&szlig;e
         * betr&auml;gt 2^14.
         */
        public MemoryAllocation() {
```

```
        this.memorySize = 16;
        this.min = 8;
        this.max = 14;

        initBlockLists();

        usedList = new Hashtable<Integer, Block>();
}

/**
 * Initialisierung der Speicherverwaltung. Die Speicherverwaltung wird mit
 * den &uuml;bergebenen initialisiert und die ben&ouml;tigten
 * Datenstrukturen aufgebaut. Die &uuml;bergeben Gr&ouml;&szlig;en werden
 * auf Konsitenz gepr&uuml;ft. Bei inkompatiblen Angaben wird eine
 * IllegalArgumentException geworfen.
 *
 * @param memorySize
 * @param min
 * @param max
 */
public MemoryAllocation(int memorySize, int min, int max) {
        /*
         * Überprüfen der Parameter
         */
        if (max < min) {
                throw new IllegalArgumentException();
        }
        if (memorySize < max) {
                throw new IllegalArgumentException();
        }

        /*
         * Initialisieren der Parameter der Speicherverwaltung
         */
        this.memorySize = memorySize;
        this.min = min;
        this.max = max;

        /*
         * Anlegen der Struktur zu Blockverwaltung
         */
        initBlockLists();

        /*
         * Anlegen der Hashtabelle zur Verwaltung der belegten Speicherbereiche
         */
        usedList = new Hashtable<Integer, Block>();
}

/**
 * Durch die Methode wird die Speicherstruktur zur Verwaltung der
 * Bl&ouml;cke aufgabaut und initialisiert. Diese Methode wird im
 * Konstruktor verwendet. Die Eigenschaften memorySize, min und max
 * m&uuml;ssen vor dem Aufruf gesetzt sein. Die Methode sollte nicht
 * mehrfach aufgerufen werden.
 */
protected void initBlockLists() {
        /*
         * Berechnung der Zahl der Blöcke auf der obersten Ebene. Ist die
         * maximale Blockgröße kleiner als der zur verfügbare Speicher, so
         * stehen auch für die maximale Blockgrößer mehrere Blöcke zur
         * Verfügung.
         */
        int minBlockNumber = (int) Math.pow(2, this.memorySize - this.max);

        /*
         * Anlegen des Felds zum Verwalten der Blocklisten
         */
        int numberOfBlockLists = this.max-this.min+1;
        blockLists = new Block[numberOfBlockLists][];

        /*
         * Anlegen und initialisieren der Blocklisten für die verschiedene
         * Blockgrößen
         */
        for (int i = 0; i < blockLists.length; i++) {
                /*
                 * Die Zahl der Blöcke auf der aktuellen Stufe hängt auch von der
                 * Zahl der Blöcke auf der obersten Ebene ab.
                 */
                int lengthOfBlockList = (int) Math.pow(2, this.max - this.min - i)
            * minBlockNumber;
                blockLists[i] = new Block[lengthOfBlockList];
```

```
                int blockSize = (int) Math.pow(2, i + this.min);
                for (int j = 0; j < blockLists[i].length; j++) {
                        blockLists[i][j] = new Block(i, j, blockSize);
                }
        }

}

/**
 * Mit malloc kann Speicher einer vorgegeben Gr&ouml;&szlig;e anfordert
 * werden. Kann die Anforderung erf&uuml;llt werden, so wird die
 * Startadresse des Speicherblocks zur&uuml;ck gegeben. Eine
 * Speicheranforderung kann in den folgenden F&auml;llen fehlschlagen:
 * <ul>
 * <li>Der angeforderte Speicher ist gr&ouml;&szlig;er als die maximale
 * Blockgr&ouml;&szlig;e</li>
 * <li>Der noch Verf&uuml;gbare Speicher reicht nicht mehr aus. Der
 * Speicher kann auch durch zu starke Fragmentierung nicht verf&uuml;gbar
 * sein, obwohl insgesammt ausreichend Speicher verf&uuml;gbar w&auml;re.</li>
 * </ul>
 *
 * @param size
 *            Gr&ouml;&szlig;e des angeforderten Speichers
 * @return Anfangsadresse des zugeteilten Speicherbereichs
 * @throws OutOfMemoryException
 *             Wird geworfen, wenn der Verf&uuml;gbare Speicher nicht
 *             ausreicht um die Anfrage zu erf&uuml;llen.
 * @throws InvalidBlockSizeException
 *             Wird geworfen, wenn der angeforderte Speicher
 *             gr&ouml;&szlig;er als die maximale Blockgr&ouml;&szlig;e ist.
 */
public int malloc(int size)
    throws OutOfMemoryException, InvalidBlockSizeException {
        Block[] blockList = getBlockList(size);

        if (blockList == null) {
                /*
                 * Es konnte keine passende Blockgröße gefunden werden
                 */
                throw new InvalidBlockSizeException();
        }

        /*
         * In der Liste der Blöcke der passenden Größe wird der erste freie
         * Block gesucht und zurückgegeben. Der Block wird als belegt markiert
         * und der Block in die Belegtliste eingetragen.
         */
        for (Block block : blockList) {
                if (block.isFree()) {
                        flagBlock(block);
                        usedList.put(block.getStartAddress(), block);
                        return block.getStartAddress();
                }
        }

        /*
         * Konnte kein passender Block gefunden werden, so steht nicht mehr
         * genug Speicher zur Verfügung.
         */
        throw new OutOfMemoryException();
}

/**
 * Durch free kann belegter Speicher wieder freigegeben werden. Der Speicher
 * wird in der Speicherverwaltung wieder als verf&uuml;gbar markiert und aus
 * der Liste belegter Speicherbl&ouml;cke ausgetragen.
 *
 * @param address
 *            Die Anfangsadresse des freizugebenden Speicherbereichs.
 * @throws InvalidAddressException
 *             Wird geworfen, wenn die &uuml;bergeben Adresse nicht
 *             freigegeben werden kann.
 */
public void free(int address) throws InvalidAddressException {
        /*
         * Ermitteln des Blocks zur übergebenen Adresse
         */
        Block block = usedList.get(address);

        if (block == null) {
                throw new InvalidAddressException();
        }
```

```
            /*
             * Falls ein Block zur Adresse gefunden werden konnte, so wird wieder
             * als frei markiert.
             */

        unflagBlock(block);
            usedList.remove(address);
    }

    /**
     * Durch die Methode wird der &uuml;bergebene Block als belegt markiert. Es
     * werden auch rekursiv alle Eltern und alle Kindknoten markiert.
     *
     * @param block
     *            Der als belegt zu markierende Block
     */
    protected void flagBlock(Block block) {
            block.setFree(false);
            flagParent(block);
            flagChildren(block);
    }

    /**
     * Durch die Methode wird der &uuml;bergebene Block als nicht mehrbelegt
     * markiert. Es werden auch rekursiv alle Eltern und alle Kindknoten als
     * nicht belegt markiert. Bei den Eltern wird ber&uuml;cksichtigt, dass eine
     * freigabe nur erfolgen darf, wenn auch der Buddy des Blocks auf der
     * entsprechenden Ebene nicht belegt ist.
     *
     * @param block
     *            Der als frei zu markierende Block
     */
    protected void unflagBlock(Block block) {
            block.setFree(true);
            unflagParent(block);
            unflagChildren(block);
    }

    /**
     * Durch die Methode werden rekursiv alle Nachfahren des &uuml;bergebene
     * Block als belegt markiert. Es werden alle Kinder bis zur untersten Ebene
     * markiert..
     *
     * @param block
     *            Der Block, dessen Nachfahren als belegt markiert werden
     *            sollen.
     */
    protected void flagChildren(Block block) {
            Block leftChild = getLeftChild(block);
            if (leftChild != null) {
                    leftChild.setFree(false);
                    flagChildren(leftChild);
            }

            Block rightChild = getRightChild(block);
            if (rightChild != null) {
                    rightChild.setFree(false);
                    flagChildren(rightChild);
            }
    }

    /**
     * Durch die Methode werden rekursiv alle Nachfahren des &uuml;bergebene
     * Block als nicht mehr belegt markiert. Es werden alle Kinder bis zur
     * untersten Ebene markiert.
     *
     * @param block
     *            Der Block, dessen Nachfahren als nicht mehr belegt markiert
     *            werden sollen.
     */
    protected void unflagChildren(Block block) {
            Block leftChild = getLeftChild(block);
            if (leftChild != null) {
                    leftChild.setFree(true);
                    unflagChildren(leftChild);
            }

            Block rightChild = getRightChild(block);
            if (rightChild != null) {
                    rightChild.setFree(true);
                    unflagChildren(rightChild);
            }
    }
```

```
/**
 * Durch die Methode werden rekursiv alle Vorfahren des &uuml;bergebene
 * Block als belegt markiert. Es werden alle Eltern bis zur obersten Ebene
 * markiert.
 *
 * @param block
 *            Der Block, dessen Vorfahren als belegt markiert werden sollen.
 */
protected void flagParent(Block block) {
        Block parent = getParent(block);

        if (parent != null) {
                        parent.setFree(false);
                        flagParent(parent);
        }
}

/**
 * Durch die Methode werden rekursiv alle Vorfahren des &uuml;bergebene
 * Block als nicht mehr belegt markiert. Es werden alle Eltern bis zur
 * obersten Ebene markiert.
 *
 * @param block
 *            Der Block, dessen Vorfahren als nicht mehr belegt markiert
 *            werden sollen.
 */
protected void unflagParent(Block block) {
        Block buddy = getBuddy(block);
        Block parent = getParent(block);

        if (parent != null && buddy.isFree()) {
                parent.setFree(false);
                unflagParent(parent);
        }
}


/**
 * Die Methode gibt zu einen &uuml;bergeben Block den entsprechenden Buddy
 * zur&uuml;ck. Buddys sind zwei Blocks mit einem gemeinsamen Vorfahren. Bei
 * einer geraden Startadresse ist dies der Block mit der n&auml;chst
 * gr&ouml;&szlig;eren Startadresse auf der gleichen Ebenen, bei einer
 * ungeraden Adresse der Block mit der n&auml;chst kleiner Startadresse auf
 * der entsprechenden Ebene. F&uuml;r die Blocks der obersten Ebene kann
 * kein Buddy bestimmt werden, da dise keinen Vorfahren haben.
 *
 * @param block
 *            Der Block zu dem der Buddy gesucht wird
 * @return Der gesuchte Buddy- Bei Fehlern wird null zur&uuml;ck gegeben.
 */

protected Block getBuddy(Block block) {
        if (block.getBlockList() == this.max - this.min) {
                return null;
        }else if (block.getPosition() % 2 == 0) {
                return blockLists[block.getBlockList()][block.getPosition() + 1];
        } else {
                return blockLists[block.getBlockList()][block.getPosition() - 1];
        }
}

/**
 * Gibt den Vorfahren eines Blocks zur&uuml;ck. Der Vorfahre
 * repr&auml;sentiert einern &uuml;berlappenden Speicherbereich auf der
 * n&uuml;chst h&ouml;heren Blockgr&ouml;&szlig;e
 *
 * @param block
 *            Der Block zu dem der Vorfahre gesucht wird
 * @return Der gesuchte Block oder null falls diese nicht existiert
 */
protected Block getParent(Block block) {
        try {
                return blockLists[block.getBlockList() + 1][block.getPosition() / 2];
        } catch (ArrayIndexOutOfBoundsException e) {
                return null;
        }
}

/**
 * Gibt das linke Kind eines Blocks zur&uuml;ck. Ein Block besitzt immer
 * zwei Kinder (ausgenommen jene der untersten Ebene). Es wird das Kind mit
 * der geringeren Startadresse geliefert. Falls kein Kind existiert wird
 * null zur&uuml;ckgegeben.
 *
```

```
         * @param block
         *            Der Block dessen linkes Kind gesucht wird.
         * @return Der gesuchte Block oder null falls diese nicht existiert.
         */
        protected Block getLeftChild(Block block) {
                try {
                        return blockLists[block.getBlockList() - 1][block.getPosition() * 2];
                } catch (ArrayIndexOutOfBoundsException e) {
                        return null;
                }
        }

        /**
         * Gibt das rechte Kind eines Blocks zur&uuml;ck. Ein Block besitzt immer
         * zwei Kinder (ausgenommen jene der untersten Ebene). Es wird das Kind mit
         * der h&ouml;heren Startadresse geliefert. Falls kein Kind existiert wird
         * null zur&uuml;ckgegeben.
         *
         * @param block
         *            Der Block dessen linkes Kind gesucht wird.
         * @return Der gesuchte Block oder null falls diese nicht existiert.
         */
        protected Block getRightChild(Block block) {
                try {
                        return blockLists[block.getBlockList() - 1][(block.getPosition() * 2) + 1];
                } catch (ArrayIndexOutOfBoundsException e) {
                        return null;
                }
        }

        /**
         * Durch die Methode wird die zu einer &uuml;bergebenen
         * Speichergr&ouml;&szlig;e passende Blockliste gesucht. Falls eine passende
         * Liste gefunden wurde wird das entsprechende Feld zur&uuml;ck gegeben.
         * Kann keine passende Liste gefunden werden so wird null zur&uuml;ck
         * gegeben.
         *
         * @param size
         *            Die angeforderte Speichergr&ouml;&szlig;e
         * @return Die zur angeforderte Speichergr&ouml;&szlig;e passende Blockliste
         */
        protected Block[] getBlockList(int size) {
                for (int i = this.min; i <= this.max; i++) {
                        if (size <= Math.pow(2,i)) {
                        return blockLists[i - this.min];
                        }
                }
                return null;
        }

}
```

Listing 3: Block.java

```
/*
 * Block
 *
 * Version 1.0.0
 *
 * 2006-06-20
 *
 * (C) Yangzi Zhang, Michael Gottschalk, Felix J. Oppermann 2006
 */

/**
 * Die Klasse repr&auml;sentiert einen Block. Die Klase Block enth&alt die
 * Metadaten zu einem Speicherblock.
 *
 * @author Yangzi Zhang
 * @author Michael Gottschalk
 * @author Felix J. Oppermann
 */
public class Block {
        /**
         * Die Nummer der Blockliste zu der dieser Block geh&ouml;rt. Der Wert wird
         * bei der Berechung von Nachbarn, Eltern und Kindern verwendet.
         */
        int blockList;

        /**
         * Die Nummer des Blocks in seiner Blockliste an. Der Wert wird bei der
         * Berechung von Nachbarn, Eltern und Kindern verwendet.
         */
```

```
int position;

/**
 * Der Wert gibt die Gr&ouml;&szlig;e des Verwalteten Speicherblocks an.
 */
int size;

/**
 * Gibt an ob der Block zu Zeit belegt ist. Im gegensatz zu den anderen
 * Eigenschaften ist diese Eigenschaft des Blocks dynamisch.
 */
boolean free;

/**
 * Legt ein neues Blockelement an. Der Block ist zun&auml;st als frei
 * markiert.
 *
 * @param blockList
 *            Die Nummer der Blockliste zu der der Block geh&ouml;rt.
 * @param position
 *            Die Position des Blocks in seiner Blockliste.
 * @param size
 *            Die Gr&ouml;&szlig;e des Verwalteten Speicherblocks.
 */
public Block(int blockList, int position, int size) {
        this.position = position;
        this.blockList = blockList;
        this.size = size;
        this.free = true;
}

/**
 * Gibt an ob der Block belegt ist.
 *
 * @return true falls der Block frei ist und false falls er belegt ist.
 */
public boolean isFree() {
        return this.free;
}

/**
 * Markieren des Blocks als frei oder belegt.
 *
 * @param free
 *            Der neue Verf&oul;gbarkeitszustand des Blocks.
 */
public void setFree(boolean free) {
        this.free = free;
}

/**
 * Gibt die Positionsnummer des Blocks in seiner Blockliste zur&uuml;ck.
 *
 * @return Die Positionsnummer der Blocks in seiner Blockliste
 */
public int getPosition() {
        return this.position;
}

/**
 * Gibt die Nummer des Blockliste in der sich der Block befindet
 * zur&uuml;ck.
 *
 * @return Die Nummer der Blockliste des Blocks.
 */
public int getBlockList() {
        return this.blockList;
}

/**
 * Gibt die Gr&ouml;&szlig;e des verwalteten Speicherblocks zur&uuml;ck..
 *
 * @return Die Gr&ouml;&szlig;e des verwalteten Speicherblocks.
 */
public int getSize() {
        return this.size;
}

/**
 * Gibt die Startaddresse des verwalteten Speicherblocks zur&uuml;ck. Die
 * Adresse wird aus der Position in der Blockliste und der
 * Blockgr&ouml;szlig;e auf der entsprechende Ebene berechnet.
 *
 * @return Die Gr&ouml;&szlig;e des verwalteten Speicherblocks.
```

```
         */
        public int getStartAddress() {
                return this.position * this.size;
        }
}
```

Listing 4: InvalidAddressException.java

```
/*
 * InvalidAddressException
 *
 * Version 1.0.0
 *
 * 2006-06-20
 *
 * (C) Yangzi Zhang, Michael Gottschalk, Felix J. Oppermann 2006
 */

/**
 * Diese Exception wird geworfen, wenn eine Speicheranfreigabe eine
 * ung&uuml;ltige Adresse verwendet. Die Adresse kann entweder au&szlig;erhalb
 * des Verwalteten Speichers liegen oder ist nicht die Startadresse eines
 * belegten Blocks.
 *
 * @author Yangzi Zhang
 * @author Michael Gottschalk
 * @author Felix J. Oppermann
 */
public class InvalidAddressException extends Exception {

}
```

Listing 5: InvalidBlockSizeException.java

```
/*
 * InvalidBlockSizeException
 *
 * Version 1.0.0
 *
 * 2006-06-20
 *
 * (C) Yangzi Zhang, Michael Gottschalk, Felix J. Oppermann 2006
 */

/**
 * Diese Exception wird geworfen, wenn eine Speicheranfrage nicht erf&uuml;llt
 * werden kann, da die Gr&ouml;&szlig;e mit den zur Verf&uuml;gung stehenden
 * Blockgr&ouml;&szlig;en nicht erf&uuml;llbar ist. Es kann dennoch freier
 * Speicherplatz verf&uuml;gbar sein.
 *
 * @author Yangzi Zhang
 * @author Michael Gottschalk
 * @author Felix J. Oppermann
 *
 */
public class InvalidBlockSizeException extends Exception {

}
```

Listing 6: OutOfMemoryException.java

```
/*
 * OutOfMemoryException
 *
 * Version 1.0.0
 *
 * 2006-06-20
 *
 * (C) Yangzi Zhang, Michael Gottschalk, Felix J. Oppermann 2006
 */

/**
 * Diese Exception wird geworfen, wenn eine Speicheranfrage nicht erf&uuml;llt
 * werden kann, da nicht mehr gen&uuml;gent Speicher verf&uuml;gbar ist. Es kann
 * noch freier Speicherplatz geringerer Gr&ouml;&szlig;e verf&uuml;gbar sein.
 *
 * @author Yangzi Zhang
 * @author Michael Gottschalk
 * @author Felix J. Oppermann
 *
 */
public class OutOfMemoryException extends Exception {
```

}

Programmablauf

```
Memory size (2^n): 16
Minimal block size (2^min): 8
Maximal block size (2^max): 14
> malloc 500
address: 0
> malloc 2000
address: 2048
> malloc 1000
address: 1024
> malloc 500
address: 512
> free 0
Freed memory
> malloc 4000
address: 4096
> free 2048
Freed memory
> free 1024
Freed memory
> free 512
Freed memory
> malloc 1500
address: 2048
> malloc 500
address: 0
> exit
```