

Inhaltsverzeichnis

1 Aufgabenbeschreibung.....	2
2 Bewertung.....	3
3 Abgabe.....	4
4 Schnittstellenbeschreibung.....	5
4.1 Der Syntaxbaum.....	6
4.2 Die Tabelle mit followpos-Einträgen.....	7
4.3 Die Übergangsmatrix des neuen DEA.....	9
5 Einschränkung der Sprache der regulären Ausdrücke.....	10
6 Hinweise für das Erstellen des Parsers.....	11
6.1 Beschreibung der Grammatik.....	11
6.2 Die Parsertabelle.....	14
6.3 Codeskelett für die Klasse Parser.....	14
6.4 Richtiger Zeitpunkt zum Einlesen des nächsten Zeichens.....	15
7 Hinweise für das Erstellen der Visatoren.....	19
7.1 Codeskelett des ersten Visitors.....	19
7.2 Funktionalität des ersten Visitors.....	19
7.3 Codeskelett des zweiten Visitors.....	19
7.4 Funktionalität des zweiten Visitors.....	20
7.5 Die Iteratorklasse.....	20
8 Hinweise für das Erstellen des DEA-Erzeugers.....	22
8.1 Codeskelett des DEA-Erzeugers.....	23
8.2 Ermitteln des Eingabealphabets.....	23
8.3 Aufbau der Übergangsmatrix.....	24
8.4 Der Algorithmus.....	24
8.4.1 Initialisierungsphase.....	24
8.4.2 Die Hauptschleife.....	25
8.4.3 Berechnung des Folgezustands.....	25
8.4.4 Gleichheit zweier Zustände.....	26
9 Hinweise für das Erstellen des Lexers.....	27
9.1 Codeskelett.....	27
9.2 Funktionalität.....	27
10 Hinweise für das Erstellen der Standalone-Tests.....	28
10.1 Standalone-Test für den Parser (erfolgreicher Parse-Vorgang).....	28
10.2 Der Vergleich zweier Syntaxbäume.....	28
10.3 Standalone-Test für den Parser (fehlgeschlagener Parse-Vorgang).....	29
10.4 Standalone-Test für den ersten Visitor.....	29
10.5 Der Vergleich zweier Syntaxbäume.....	29
10.6 Standalone-Test für den zweiten Visitor.....	30
10.7 Standalone-Test für den DEA-Erzeuger.....	31
10.8 Standalone-Test für den generischen Lexer.....	31
11 Quellen.....	32

1 Aufgabenbeschreibung

Diese Aufgabe ist für maximal 5 Personen gedacht. Im Falle einer Dreiergruppe müssen nur der Top-Down-Parser und die beiden Visatoren implementiert werden. Im Falle einer Vierergruppe kann der generische Lexer entfallen.

Analog zum Drachenbuch (Seiten 209 bis 216) soll ein Verfahren implementiert werden, das aus einem regulären Ausdruck **direkt** einen deterministischen endlichen Automaten (DEA) erzeugt, der exakt die Worte akzeptiert, welche den regulären Ausdruck matchen.

Folgende Teilschritte muss die jeweilige Projektgruppe realisieren

- Einen **rekursiven Top-Down-Parser** implementieren
 - er überprüft die Korrektheit des Ausdrucks
(→ **Kapitel 2 der Vorlesung** ab Folie 39 und
→ Kapitel 6)
 - er erzeugt im positiven Fall mittels **syntaxgesteuertem Übersetzungsverfahren** einen AST, d.h. (abstrakten) Syntaxbaum des regulären Ausdrucks
(→ **Kapitel 2 der Vorlesung** ab Folie 30 und
→ Kapitel 6)
 - er wirft im negativen Fall eine **RuntimeException**
- Einen **Visitor** implementieren zum Bestimmen von **nullable**, **firstpos** und **lastpos**
(→ **Kapitel 3 der Vorlesung** ab Folie 65 und
→ Kapitel 7)
- Einen **zweiten Visitor** implementieren zum Bestimmen der **followpos**-Tabelle
(→ **Kapitel 3 der Vorlesung** ab Folie 65 und
→ Kapitel 7)
- Eine Klasse implementieren, welche den **DEA-Erzeuger** realisiert
(→ **Pseudocode auf Seite 216** des Drachenbuches oder
→ **Kapitel 3 der Vorlesung** ab Folie 65 und
→ Kapitel 8)
- Eine Klasse implementieren, die einen **generischen Lexer** realisiert
(→ Kapitel 9)

2 Bewertung

Was wird bewertet	Zu welchem Anteil
die korrekte Umsetzung	max. 50 Punkte
die Kompilierfähigkeit	max. 5 Punkte
Einhalten der Schnittstellenvorgaben	max. 20 Punkte
Einhalten der Programmiererkonventionen	max. 25 Punkte

3 Abgabe

Was	eine ZIP-Datei (Quelldateien der Umsetzung und der Standalone-Tests)
Eingabefrist	01.03.2021 (verspätet zugeschickte ZIP-Dateien werden nicht berücksichtigt!)

4 Schnittstellenbeschreibung

Damit alle Beteiligten einer Projektgruppe möglichst unabhängig voneinander (und somit auch gleichzeitig!) arbeiten können, müssen gemeinsame Schnittstellen definiert werden.

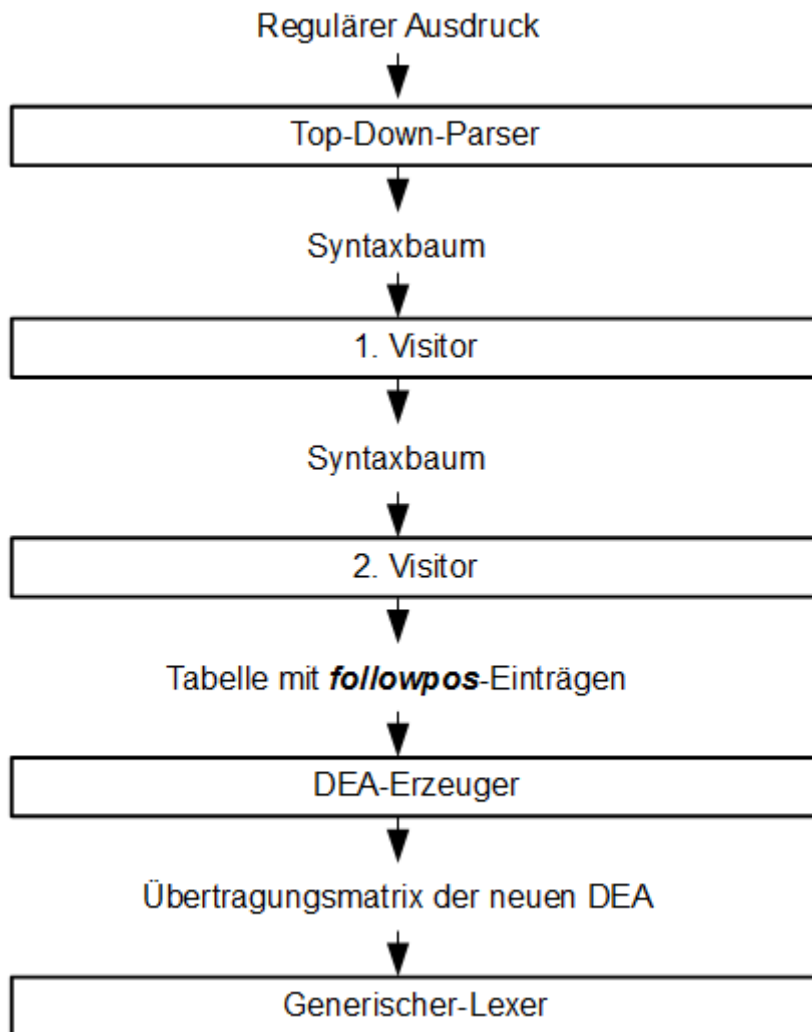


Abb. 1 Schaubild mit Schnittstellen

Gemäß Abb. 1 ergeben sich folgende Schnittstellen:

- der Syntaxbaum
- die Tabelle mit followpos-Einträgen sowie
- die Übergangsmatrix des neuen DEA

4.1 Der Syntaxbaum

Zunächst einmal müssen zwei Interfaces (**Visitor** und **Visitable**) aufgenommen werden, die für die Implementierung der Visitoren wichtig sind (package-private Definition der Interfaces genügt)

```
interface Visitor
{
    public void visit(OperandNode node);
    public void visit(BinOpNode  node);
    public void visit(UnaryOpNode node);
}

interface Visitable
{
    void accept(Visitor visitor);
}
```

Nun folgen die entsprechenden Klassendefinitionen für alle Knoten des Syntaxbaums:

```
public abstract class SyntaxNode
{
    public Boolean nullable;
    public final Set<Integer> firstpos = new HashSet<>();
    public final Set<Integer> lastpos  = new HashSet<>();
}

public class OperandNode extends SyntaxNode implements Visitable
{
    public int    position;
    public String symbol;

    public OperandNode(String symbol)
    {
        position    = -1;    // bedeutet: noch nicht initialisiert
        this.symbol = symbol;
    }

    @Override
    public void accept(Visitor visitor)
    {
        visitor.visit(this);
    }
}

public class BinOpNode extends SyntaxNode implements Visitable
```

```

{
    public String    operator;
    public Visitable left;
    public Visitable right;

    public BinOpNode(String operator, Visitable left, Visitable
right)
    {
        this.operator = operator;
        this.left      = left;
        this.right     = right;
    }

    @Override
    public void accept(Visitor visitor)
    {
        visitor.visit(this);
    }
}

public class UnaryOpNode extends SyntaxNode implements Visitable
{
    public String    operator;
    public Visitable subNode;

    public UnaryOpNode(String operator, Visitable subNode)
    {
        this.operator = operator;
        this.subNode  = subNode;
    }

    @Override
    public void accept(Visitor visitor)
    {
        visitor.visit(this);
    }
}

```

Beachten Sie, dass alle referenziellen Verweise auf andere Knoten des Syntaxbaums als Visitables angesehen werden, damit die Iteration der Visitoren über alle Knoten des Syntaxbaums sehr generisch implementiert werden kann!

4.2 Die Tabelle mit followpos-Einträgen

Zunächst folgt eine package-private Definition eines Zeileneintrages.

```

class FollowposTableEntry
{

```

```

public final int      position;
public final String   symbol;
public final Set<Integer> followpos = new HashSet<>();

public FollowposTableEntry(int position, String symbol)
{
    this.position = position;
    this.symbol   = symbol;
}

@Override
public boolean equals(Object obj)
{
    if (this == obj)
    {
        return true;
    }

    if (! (obj instanceof FollowposTableEntry))
    {
        return false;
    }

    FollowposTableEntry other = (FollowposTableEntry) obj;
    return this.position == other.position &&
           this.symbol.equals(other.symbol) &&
           this.followpos.equals(other.followpos);
}

@Override
public int hashCode()
{
    int hashCode = this.position;
    hashCode = 31 * hashCode + this.symbol.hashCode();
    hashCode = 31 * hashCode + this.followpos.hashCode();

    return hashCode;
}
}

```

Gefolgt von der Deklaration einer TreeMap, welche die gesamte Tabelle repräsentiert.

```

import java.util.SortedMap;
import java.util.TreeMap

...

private SortedMap<Integer, FollowposTableEntry> followposTableEntries = new TreeMap<>();

```

Der Schlüssel ist die Position und der Wert ist der gesamte Zeileneintrag.

Beachten Sie!

Eine TreeMap sortiert alle Einträge nach dem Wert ihrer Schlüssel. Außerdem muss der **höchste Positionseintrag das Endmarkersymbol** (hier: #) repräsentieren!

4.3 Die Übergangsmatrix des neuen DEA

Zunächst führen wir eine package-private Struktur ein, die einen Zustand beschreibt.

```
class DFAState
{
    public final int      index;
    public final Boolean  isAcceptingState;
    public final Set<Integer> positionsSet;

    public DFAState(int      index,
                    Boolean  isAcceptingState,
                    Set<Integer> positionsSet)
    {
        this.index      = index;
        this.isAcceptingState = isAcceptingState;
        this.positionsSet = positionsSet;
    }

    @Override
    public boolean equals(Object obj) return
    {
        if(this == obj)          return true;
        if(obj == null)          return false;
        if(getClass() != obj.getClass()) return false;

        DFAState other = (DFAState)obj;
        return equals(this.positionsSet, other.positionsSet);
    }

    @Override
    public int hashCode()
    {
        return this.positionsSet == null ? 0 : this.positionsSet.hashCode();
    }

    private static boolean equals(Object o1, Object o2)
    {
        if (o1 == o2)      return true;
        if (o1 == null)    return false;
        if (o2 == null)    return false;

        return o1.equals(o2);
    }
}
```

Darauf verwenden wir eine HashMap, um die Übergangsmatrix der DEA darzustellen.

```
private Map<DFAState, Map<Char, DFAState>> stateTransitionTable = new HashMap<>();
```

5 Einschränkung der Sprache der regulären Ausdrücke

Folgende Konstruktionsvorschrift soll als Grundlage dienen.

- i) jeder Buchstabe und jede Ziffer von 0 bis 9 ist ein regulärer Ausdruck
- ii) seien zwei reguläre Ausdrücke r_1 und r_2 gegeben, so sind
 1. (r_1) und (r_2) (Klammerung)
 2. r_1r_2 (Konkatenation)
 3. $r_1 \mid r_2$ (Alternative)
 4. r_1^* (Kleenesche Hülle)
 5. r_1^+ (Positive Hülle)
 6. $r_1?$ (Option)ebenfalls reguläre Ausdrücke.
- iii) Der Einfachheit halber seien Whitespaces nicht zulässig

6 Hinweise für das Erstellen des Parsers

6.1 Beschreibung der Grammatik

Gemäß dem Algorithmus auf den Seiten 209 bis 216 des „Drachenbuches“ muss Ihr Parser den Ausdruck **(r)#** auswerten, wobei:

1. **r** ein regulärer Ausdruck (gemäß obiger Einschränkung) ist und
2. **#** nicht als Zeichen in **r** vorkommen darf.

Eine entsprechende LL(1)-Grammatik (Linksrekursionen beseitigt!) sieht wie folgt aus:

Start	→	'#'
Start	→	'(' 'RegExp' ')' '#'
RegExp	→	Term RE'
RE'	→	ε
RE'	→	' ' Term RE'
Term	→	FactorTerm
Term	→	ε
Factor	→	Elem HOp
HOp	→	ε
HOp	→	'*'
HOp	→	'+'
HOp	→	'?'
Elem	→	Alphanum
Elem	→	'(' 'RegExp')'
Alphanum	→	'A'
...		

Im folgenden wird diese Grammtik mit (semantischen) Aktionen (blau gefärbt) angereichert, so dass beim Ableiten eines Zielausdrucks gleichzeitig dessen Syntaxbaum erstellt wird. Zuvor müssen allerdings noch alle Nichtterminalen Symbole um die Attribute **parameter** und **return** erweitert werden:

- **parameter**
Die Referenz auf den Wurzelknoten eines bis dato konstruierten Syntaxbaums. Ziel ist es, für den aktuellen Knoten des Parsebaums einen neuen Syntaxbaum zu erzeugen, der den bis dato ermittelten Syntaxbaum als direkten Unterbaum enthält.
- **return**
Die Referenz auf den Wurzelknoten des neuen Syntaxbaums.

Start	→	'#' {Start.return = new OperandNode('#');}
Start	→	{RegExp.parameter = null;} '(' 'RegExp' ')' '#' { leaf = new OperandNode('#'); root = new BinOpNode('°', RegExp.return, leaf); Start.return = root; }
RegExp	→	{Term.parameter = null;} Term {RE'.parameter = Term.return;} RE' {RegExp.return = RE'.return;}
RE'	→	ε {RE'.return = RE'.parameter;}
RE'	→	' ' {Term.parameter = null;} Term { root = new BinOpNode(' ', RE'.parameter, Term.return); RE ₁ '.parameter = root; } RE ₁ ' {RE'.return = RE ₁ '.return;}

```

Term      →      {Factor.parameter = null;}
                  Factor
                  {
                      if (Term.parameter != null)
                      {
                          root = new BinOpNode('°',
                                                  Term.parameter,
                                                  Factor.return);
                          Term1.parameter = root;
                      }
                      else
                      {
                          Term1.parameter = Factor.return;
                      }
                  }
                  Term1
                  {Term.return = Term1.return;}

Term      →      ε {Term.return = Term.parameter;}

Factor    →      {Elem.parameter = null;}
                  Elem
                  {HOp.parameter = Elem.return;}
                  HOp
                  {Factor.return = HOp.return;}

HOp       →      ε {HOp.return = HOp.parameter}

HOp       →      '*' {HOp.return = new UnaryOpNode('*',
HOp.parameter);}

HOp       →      '+' {HOp.return = new UnaryOpNode('+',
HOp.parameter);}

HOp       →      '?' {HOp.return = new UnaryOpNode('?',
HOp.parameter);}

Elem      →      {Alphanum.parameter = null;}
                  Alphanum
                  {Elem.return = Alphanum.return;}

Elem      →      {RegExp.parameter = null;}
                  '(' RegExp ')'
                  {Elem.return = RegExp.return;}

Alphanum  →      'A' {Alphanum.return = new OperandNode('A');}
...          ...

```

6.2 Die Parsertabelle

Die folgende Parsertabelle zeigt an, bei welcher Kombination aus Nichtterminalem Symbol und Eingabesymbol, welche Regel ausgeführt wird.

	0 ... 9, A ... Z, a ... z	'('	'*'	'+'	'?'	' '	')'	'#'	'\$'
Start		Start \rightarrow '(' RegExp ')' '#'						Start \rightarrow '#'	
RegExp	RegExp \rightarrow Term RE'	RegExp \rightarrow Term RE'							
RE'						RE' \rightarrow ' ' Term RE'	RE' \rightarrow ϵ		
Term	Term \rightarrow FactorTerm	Term \rightarrow FactorTerm				Term \rightarrow ϵ	Term \rightarrow ϵ		
Factor	Factor \rightarrow Elem HOp	Factor \rightarrow Elem HOp							
HOp	HOp \rightarrow ϵ	HOp \rightarrow ϵ	HOp \rightarrow '*'	HOp \rightarrow '+'	HOp \rightarrow '?'	HOp \rightarrow ϵ	HOp \rightarrow ϵ		
Elem	Elem \rightarrow Alphanum	Elem \rightarrow '(' RegExp ')'							
Alphanum	Alphanum \rightarrow ...								

6.3 Codeskelett für die Klasse Parser

Die Klasse sollte folgenden Aufbau haben. Beachten Sie insbesondere, dass das Inkrementieren des Positionindex innerhalb der Methode **match** gekapselt ist.

```
public class Parser
{
    private      int      position;
    private final String eingabe;
    ...

    public Parser(String eingabe)
    {
        this.eingabe = eingabe;
        this.position = 0;
    }
    ...

    // pro Nichtterminal eine Methode !

    // Nichtterminal Start
    //
    // i)  Nur diese Methode ist oeffentlich !!
    // ii) Nur in dieser Methode auf Eingabeende ueberpruefen !!!
    public Visitable start(Visitable parameter)
    {
        ...
    }
    ...

    // Nichtterminal Alphanum
    private Visitable alphanum(Visitable parameter)
    {
```

```

    ...
}

private void match(char symbol)
{
    if ((eingabe == null) || ("".equals(eingabe)))
    {
        throw new RuntimeException("Syntax error !");
    }
    if (position >= eingabe.length())
    {
        throw new RuntimeException("End of input reached !");
    }
    if (eingabe.charAt(position) != symbol)
    {
        throw new RuntimeException("Syntax error !");
    }

    position++;
}

//-----
// 1. wird benoetigt bei der Regel Start -> '(' RegExp ')' '#'
// 2. wird benoetigt bei der Regel Start -> '#'
// 3. wird sonst bei keiner anderen Regel benoetigt
//-----
private void assertEndOfInput()
{
    if (position < eingabe.length())
    {
        throw new RuntimeException(" No end of input
reached !");
    }
}
}

```

6.4 Richtiger Zeitpunkt zum Einlesen des nächsten Zeichens

Schauen Sie sich als erstes den Abschnitt **Syntaxanalyse** in Kapitel 2 der Vorlesung an (Folien 44 und 45). Dadurch erhalten Sie einen ersten intuitiven Eindruck.

Auf unsere Grammatik angewendet ergibt sich:

1. Falls der Rumpf der Produktionsregel keine terminalen Symbole enthält, darf kein neues Zeichen eingelesen werden, d.h. der Positionsindex darf nicht erhöht werden.

Term \rightarrow Factor Term



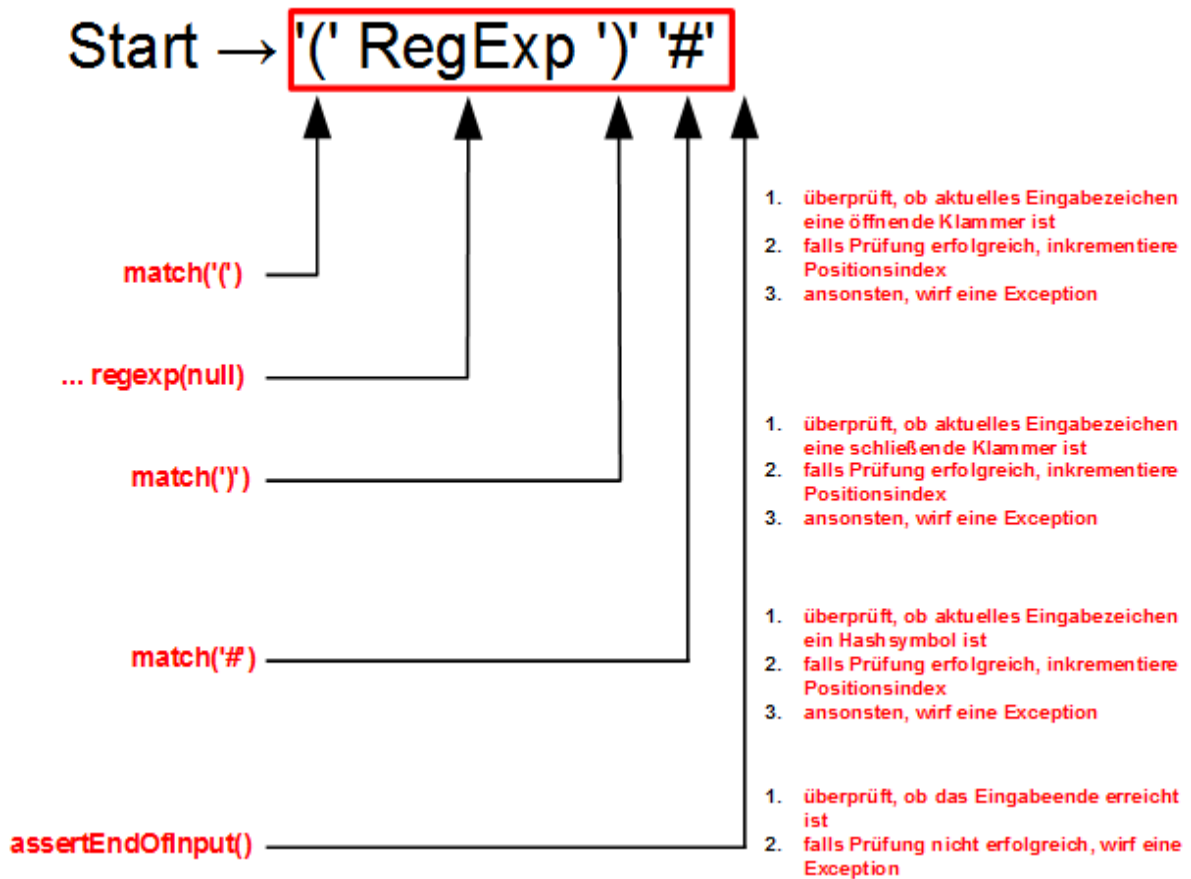
Regel hat kein terminales Symbol auf der rechten Seite => Positionsindex darf hier nie inkrementiert werden!!!

```
public class Parser
{
    private      int    position;
    private final String eingabe;
    ...

    private Visitable term(Visitable parameter)
    {
        if (eingabe.charAt(position) == ... /* 0...9, A...Z, a...z, ( */)
        {
            if (parameter != null)
            {
                return term(new BinOpNode("°", parameter, factor(null)));
            }

            return term(factor(null));
        }
        else if (eingabe.charAt(position) == ... /* | oder ) */)
        {
            ...
        }
        else
        {
            throw new RuntimeException("Syntax error !");
        }
    }
    ...
}
```

2. Falls der Rumpf der Produktionsregel terminale Symbole enthält, muss pro terminales Symbol das jeweils aktuelle Eingabezeichen überprüft und, im positiven Fall, der Positionsindex inkrementiert werden.



```

public class Parser
{
    private int position;
    private final String eingabe;
    ...

    public Visitable start(Visitable parameter)
    {
        if (eingabe.charAt(position) == '(')
        {
            match('(');
            Visitable subTree = regexp(null);
            match(')');
            match('#');
            assertEndOfInput(); /* wird nur innerhalb dieser Methode
benoetigt */

            return new BinOpNode('°', subTree, new OperandNode('#'));
        }
        else if (eingabe.charAt(position) == '#')
        {
            ...
        }
        else
        {
            throw new RuntimeException("Syntax error !");
        }
    }
    ...
}

```

3. Auch falls der Rumpf der Produktionsregel nur aus dem leeren Wort besteht, wird der

Positionsindex nicht inkrementiert

Term $\rightarrow \varepsilon$



Regel hat kein terminales Symbol auf der rechten Seite => Positionsindex darf hier nie inkrementiert werden!!!

```
public class Parser
{
    private      int    position;
    private final String eingabe;
    ...

    private Visitable term(Visitable parameter)
    {
        if (eingabe.charAt(position) == ... /* 0...9, A...Z, a...z, ( */ )
        {
            ...
        }
        else if (eingabe.charAt(position) == ... /* | oder ) */)
        {
            return parameter;
        }
        else
        {
            throw new RuntimeException("Syntax error !");
        }
    }
    ...
}
```

7 Hinweise für das Erstellen der Visitoren

7.1 Codeskelett des ersten Visitors

```
public class SyntaxTreeEvaluator implements Visitor
{
    public void visit(OperandNode node)
    {
        ...
    }

    public void visit(BinOpNode node)
    {
        ...
    }

    public void visit(UnaryOpNode node)
    {
        ...
    }
}
```

7.2 Funktionalität des ersten Visitors

Er durchwandert in Tiefensuche ($L \rightarrow R \rightarrow W$) den Syntaxbaum:

- bei jedem inneren Knoten vermerkt er
 - die Auswertung von **nullable** am aktuellen Knoten
 - die Auswertung von **firstpos** am aktuellen Knoten
 - die Auswertung von **lastpos** am aktuellen Knoten
- bei jedem Blattknoten vermerkt er
 - die **Position**
 - die Auswertung von **nullable** am aktuellen Knoten
 - die Auswertung von **firstpos** am aktuellen Knoten
 - die Auswertung von **lastpos** am aktuellen Knoten

7.3 Codeskelett des zweiten Visitors

```
public class FollowPosTableGenerator implements Visitor
{
    public void visit(OperandNode node)
    {
        ...
    }

    public void visit(BinOpNode node)
    {
        ...
    }
}
```

```

    }

    public void visit(UnaryOpNode node)
    {
        ...
    }
}

```

7.4 Funktionalität des zweiten Visitors

- er legt eine Tabelle an (erste Spalte die Position, zweite Spalte das entsprechende Zeichen, dritte Spalte die Auswertung von followpos des Knotens mit der aktuellen Position)
- er durchwandert in Tiefensuche ($L \rightarrow R \rightarrow W$) den AST
 - **bei jedem Blattknoten** führt er folgende Schritte durch
 - eine neue Zeile in der Tabelle anlegen
 - in der ersten Spalte die Position des Blattknotens vermerken
 - in der zweiten Spalte das Zeichen, das der Blattknoten repräsentiert, vermerken
 - die dritte Spalte mit einer leeren Menge initialisieren (in Java eine Instanz der Klasse `HashSet<Integer>` erzeugen!)
 - **bei jedem inneren Knoten** führt er folgende Schritte durch
 - falls der innere Knoten weder eine Konkatenation, noch eine Kleenesche noch eine Positive Hülle ist, tut er nichts!
 - falls der innere Knoten eine **Konkatenation** darstellt, iteriert er über alle Positionen $i \in \text{lastpos}(\text{linker Sohnknoten})$ und führt folgende Operationen aus:
 - $\text{followpos}(\text{Knoten an Position } i) = \text{followpos}(\text{Knoten an Position } i) \cup \text{firstpos}(\text{rechter Sohnknoten})$
 - aktualisiere den entsprechenden Eintrag in der dritten Spalte der Tabelle
 - falls der innere Knoten eine **Kleenesche Hülle** darstellt, iteriert er über alle Positionen $i \in \text{lastpos}(\text{akt. innerem Knoten})$ und führt folgende Operationen aus:
 - $\text{followpos}(\text{Knoten an Position } i) = \text{followpos}(\text{Knoten an Position } i) \cup \text{firstpos}(\text{akt. innerem Knoten})$
 - aktualisiere den entsprechenden Eintrag in der dritten Spalte der Tabelle
 - falls der innere Knoten eine **Positive Hülle** darstellt, verfährt er exakt so wie im Falle einer Kleeneschen Hülle

7.5 Die Iteratorklasse

Um ein einheitliches Abarbeiten des Syntaxbaumes durch beide Visitoren zu garantieren, soll der folgende Iterator verwendet werden.

```

public class DepthFirstIterator
{
    public static void traverse(Visitable root, Visitor visitor)
    {
        if (root instanceof OperandNode)
        {
            root.accept(visitor);
            return;
        }
    }
}

```

```
    if (root instanceof BinOpNode)
    {
        BinOpNode opNode = (BinOpNode) root;

        DepthFirstIterator.traverse(opNode.left, visitor);
        DepthFirstIterator.traverse(opNode.right, visitor);
        opNode.accept(visitor);
        return;
    }
    if (root instanceof UnaryOpNode)
    {
        UnaryOpNode opNode = (UnaryOpNode) root;

        DepthFirstIterator.traverse(opNode.subNode, visitor);
        opNode.accept(visitor);
        return;
    }

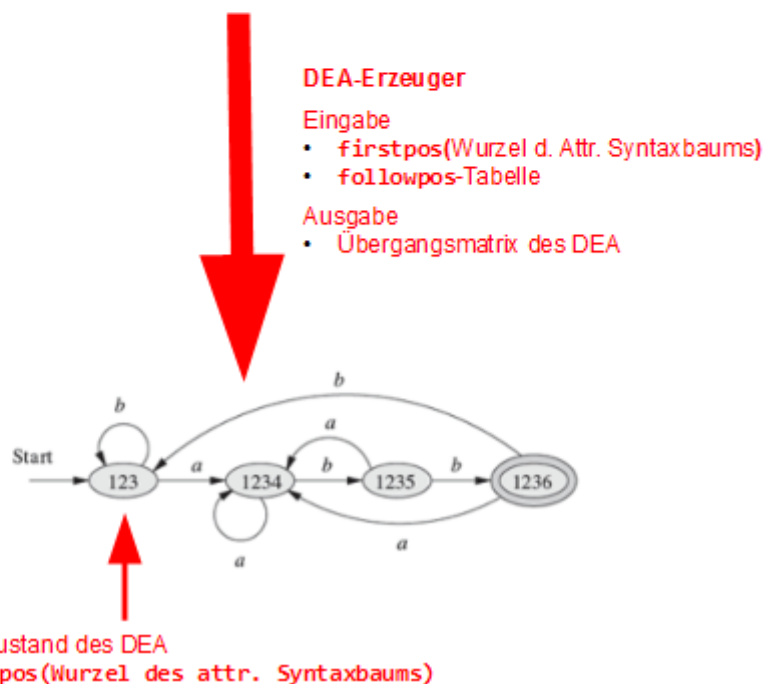
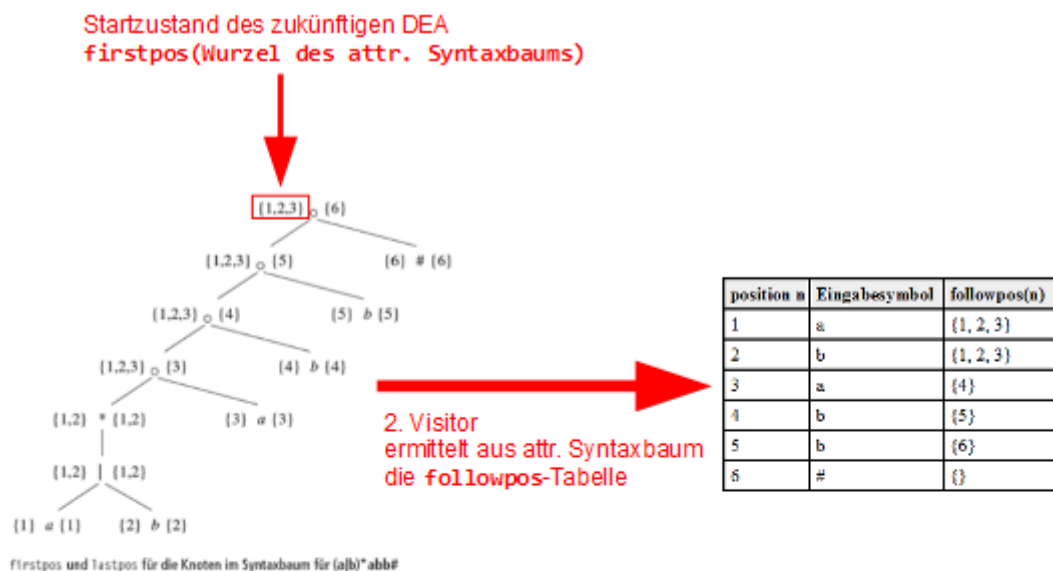
    throw new RuntimeException("Instance root has a bad type!");
}
}
```

8 Hinweise für das Erstellen des DEA-Erzeugers

Mögliche alternative Quellen sind:

- Pseudocode auf Seite 216 des Drachenbuches
- Abschnitt **Direkte Konvertierung zu DEAs** aus Kapitel 3 der Vorlesung (ab Folie 65)

Das folgende Schaubild zeigt den grundsätzlichen Zusammenhang zwischen dem attribuierten Syntaxbaum, der followpos-Tabelle und der Übergangsmatrix des DEA.



8.1 Codeskelett des DEA-Erzeugers

```
public class DFACreator
{
    private final Set<Integer>                positionsForStartState;
    private final SortedMap<Integer, FollowposTableEntry> followposTable;
    private final Map<DFASState, Map<Char, DFASState>> stateTransitionTable;

    /**
     * Man beachte ! Parameter <code>positionsForStartState</code> muss vom Aufrufer
     * mit der firstpos-Menge des Wurzelknotens des Syntaxbaums initialisiert werden !
     */
    public DFACreator(Set<Integer>                positionsForStartState,
                     SortedMap<Integer, FollowposTableEntry> followposTable)
    {
        this.positionsForStartState = positionsForStartState;
        this.followposTable          = followposTable;
        this.stateTransitionTable    = new HashMap<>();
    }


    // befüllt die Uebergangsmatrix
    public void populateStateTransitionTable()
    {
        ...
    }

    public Map<DFASState, Map<Char, DFASState>> getStateTransitionTable()
    {return stateTransitionTable;}

    ...
}
```

8.2 Ermitteln des Eingabealphabets

Der erste Schritt zum Erstellen der Übergangsmatrix ist das Bestimmen des Eingabealphabets. Zu diesem Zwecke müssen Sie lediglich alle Eingabesymbole aus der zweiten Spalte der followpos-Tabelle entnehmen und das Hash-Symbol ('#') sowie alle doppelten Einträge entfernen (siehe folgendes Schaubild).



position	Eingabesymbol	followpos(n)
1	a	{1, 2, 3}
2	b	{1, 2, 3}
3	a	{4}
4	b	{5}
5	b	{6}
6	#	{}

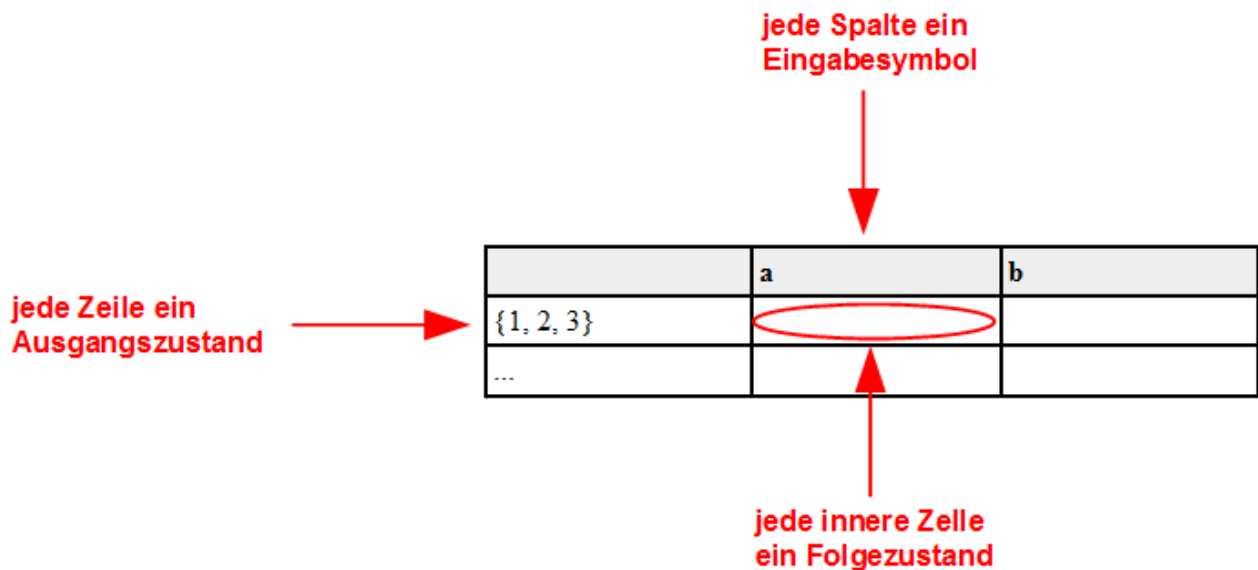
Das Eingabealphabet Σ ermittelt sich aus den Eingabesymbolen der followpos-Tabelle.

Allerdings müssen aus Σ alle doppelten Einträge und das Hashsymbol ('#') entfernt werden.

Hier: $\Sigma = \{a, b\}$

8.3 Aufbau der Übergangsmatrix

Das folgende Abbild zeigt den prinzipiellen Aufbau der Übergangsmatrix.

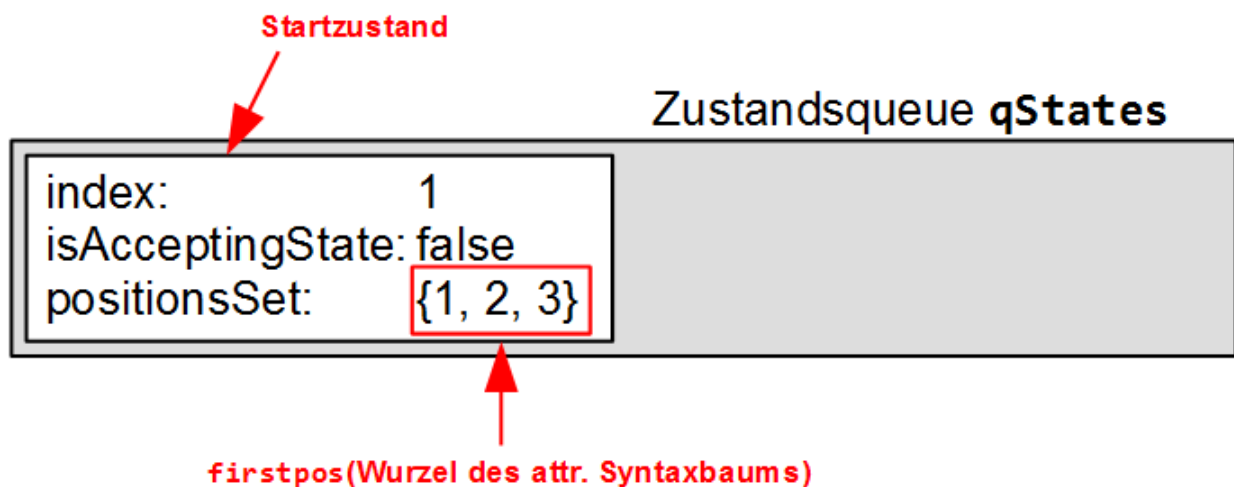


Die Übergangsmatrix hat folgende Eigenschaften

- die erste Spalte repräsentiert einen **Ausgangszustand**
- jede weitere Spalte repräsentiert ein mögliches Eingabesymbol
- die erste Zeile ist der Startzustand (= **firstpos**(Wurzel des Syntaxbaums))
- Sie markiert alle akzeptierenden Zustände.
Ein Zustand ist genau dann ein akzeptierender Zustand, wenn seine zugeordnete Positionsmenge die Position von # enthält

8.4 Der Algorithmus

8.4.1 Initialisierungsphase



firstpos(Wurzel des attr. Syntaxbaums) wird zum Startzustand des neuen DEA und wird in einer Queue namens **qStates** gespeichert (siehe Schaubild).

- Member **index** wird gleich dem aktuellen Wert einer allgemeinen Laufvariablen gesetzt,

- die Laufvariable im Anschluss daran inkrementiert.
- Member **isAcceptingState** wird genau dann auf **true** gesetzt, falls **positionsSet** die Position von Eingabesymbol **#** enthält. In unserem Beispiel also (**#** hat Position 6) **false**.

Außerdem wird eine leere Übergangsmatrix erzeugt

	a	b
--	---	---

8.4.2 Die Hauptschleife

Folgender Pseudocode beschreibt die Hauptschleife.

```
while (qStates ist nicht leer)
{
    1) s = erster Zustand aus qStates;
       entferne s aus qStates;

    2) füge für s neue Zeile in Übergangsmatrix ein;
       s wird zum Ausgangszustand dieser Zeile;

    for (jedes Eingabesymbol a aus  $\Sigma$ )
    {
        3) berechne für Kombination <s, a> den Folgezustand;
           trage Folgezustand in entsprechende innere Zelle der neuen Zeile ein;

        if ((Folgezustand ist kein Ausgangszustand in Übergangsmatrix)
            && (Folgezustand ist nicht in qStates enthalten))
        {
            4) füge Folgezustand am Ende von qStates ein;
        }
    }
}
```

8.4.3 Berechnung des Folgezustands

Das folgende Abbild zeigt die Ausgangssituation, Ausgangszustand **s** und Eingabesymbol **a**.

Ausgangszustand s

index:	...
isAcceptingState:	...
positionsSet:	{1, 2, 3}

+

Eingabesymbol a

Zunächst ermittelt der Algorithmus anhand der *followpos*-Tabelle, welche Positionen aus **s.positionsSet** auf Eingabesymbol **a** abgebildet werden. In unserem Beispiel (siehe folgendes Abbild) sind das die Positionen 1 und 3.

alle Positionen von **s.positionsSet**,
die auf **a** abgebildet werden:

=> {1, 3}

position n	Eingabesymbol	followpos(n)
1	a	{1, 2, 3}
2	b	{1, 2, 3}
3	a	{4}
4	b	{5}
5	b	{6}
6	#	{}

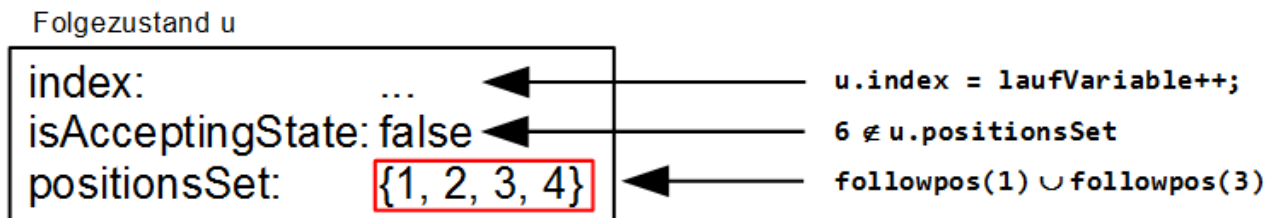
Der Algorithmus liest darauf aus der *followpos*-Tabelle für jede ermittelte Position **i** die Menge **followpos(i)** aus, in unserem Beispiel also **followpos(1)** und **followpos(3)**.

Die Positionenmenge des Folgezustands **u** berechnet sich aus der *Vereinigung über alle ausgelesenen followpos-Mengen*. In unserem Beispiel ergibt sich also:

u.positionsSet = **followpos(1)** \cup **followpos(3)**
= {1, 2, 3} \cup {4}
= {1, 2, 3, 4}

Folgezustand **u** wird genau dann zum *akzeptierenden Zustand* erklärt, wenn **u.positionsSet** die Position von Eingabesymbol **#** enthält (hier: Position 6). In unserem Beispiel ergibt sich also:
u.isAcceptingState = **false**

Schließlich setzt der Algorithmus **u.index** gleich dem Wert einer *allgemeinen Laufvariablen*, die ihrerseits unmittelbar darauf inkrementiert wird. Im Abbild darauf sehen wir den endgültigen Folgezustand.



8.4.4 Gleichheit zweier Zustände

Die Hauptschleife (→ Kapitel 8.4.2) schreibt u.a. vor, dass ein Folgezustand **u** nur dann in **qStates** eingefügt werden soll, wenn **u** weder ein Ausgangszustand der Übergangsmatrix ist noch in **qStates** vorkommt. Dies bedeutet wiederum, dass ständig zwei Zustände miteinander verglichen werden müssen.

Da jeder Zustand des zu konstruierenden DEAs im Wesentlichen eine Menge von Positionsnummern ist, bedeutet die Gleichheit zweier Zustände die *Gleichheit ihrer Positionsmengen* (→ Kapitel 4.3, Methode **DFASState.equals**).

9 Hinweise für das Erstellen des Lexers

9.1 Codeskelett

```
public class Lexer
{
    private Map<DFASState, Map<Char, DFASState>> stateTransitionTable;

    public Lexer(Map<DFASState, Map<Char, DFASState>> stateTransitionTable)
    {
        this.stateTransitionTable = stateTransitionTable;
    }

    public boolean match(String word)
    {
        ...
    }
}
```

9.2 Funktionalität

- Der Konstruktor erhält als Parameter eine Übergangsmatrix, die den erzeugten DEA darstellt
- Eine Methode **match**, die überprüft, ob ein eingegebener String vom erzeugten DEA akzeptiert wird (Beachten Sie! Genau dann matcht der übergebene String den ursprünglichen regulären Ausdruck.)
 - den eingegebenen String in ein **Char**-Array umzuwandeln
 - eine Variable **state** = Startzustand des DEAs setzen
 - über das **Char**-Array iterieren
 - **letter** = aktuelles Zeichen des **Char**-Arrays
 - aus der Übergangsmatrix den Folgezustand in Abhängigkeit von **state** und **letter** ermitteln
 - falls ein Folgezustand existiert, **state** = Folgezustand setzen
 - sonst, Methode **match** mit **false** beenden
 - falls **state** ein akzeptierender Zustand ist, Methode **match** mit **true** beenden
 - sonst, Methode **match** mit **false** beenden

10 Hinweise für das Erstellen der Standalone-Tests

Allgemein bedeutet Standalone-Test *das isolierte Testen einer Komponente* (hier: Parser, 1. Visitor, 2. Visitor, DEA-Erzeuger und generischer Lexer).

10.1 Standalone-Test für den Parser (erfolgreicher Parse-Vorgang)

Formulieren Sie ein oder mehrere Junit-Tests wie folgt:

Eingabe

- Ein String, der einen *gültigen* regulären Ausdruck darstellt
- Ein Syntaxbaum hartverdrahtet erstellt (erwartetes Ergebnis einer Auswertung des regulären Ausdrucks)

Durchführung

- Vom Parser den regulären Ausdruck analysieren und den Syntaxbaum erstellen lassen

Abschließender Test

- Der Test ist genau dann erfolgreich, wenn der hartverdrahtete Syntaxbaum (das erwartete Ergebnis) gleich dem erzeugten Syntaxbaum (das aktuelle Ergebnis) ist (→ Kapitel 10.2)

10.2 Der Vergleich zweier Syntaxbäume

Um zwei Syntaxbäume zu vergleichen, müssen beide Bäume in *Tiefensuche durchwandert* werden. Ihr JUnit-Test sollte also folgende Methode implementieren und beim Vergleich anwenden.

```
private static boolean equals(Visitable v1, Visitable v2)
{
    if (v1 == v2)
        return true;

    if (v1 == null)
        return false;

    if (v2 == null)
        return false;

    if (v1.getClass() != v2.getClass())
        return false;

    if (v1.getClass() == OperandNode.class)
    {
        OperandNode op1 = (OperandNode) v1;
        OperandNode op2 = (OperandNode) v2;
        return op1.position == op2.position && op1.symbol.equals(op2.symbol);
    }
    if (v1.getClass() == UnaryOpNode.class)
    {
        UnaryOpNode op1 = (UnaryOpNode) v1;
        UnaryOpNode op2 = (UnaryOpNode) v2;
        return op1.operator.equals(op2.operator) && equals(op1.subNode,
op2.subNode);
    }
    if (v1.getClass() == BinOpNode.class)
    {
        BinOpNode op1 = (BinOpNode) v1;
```

```

        BinOpNode op2 = (BinOpNode) v2;
        return op1.operator.equals(op2.operator) &&
               equals(op1.left, op2.left)      &&
               equals(op1.right, op2.right);
    }

    throw new IllegalStateException("Ungueltiger Knotentyp");
}

```

10.3 Standalone-Test für den Parser (fehlgeschlagener Parse-Vorgang)

Formulieren Sie ein oder mehrere Junit-Tests wie folgt:

Eingabe

- Ein String, der einen **ungültigen** regulären Ausdruck darstellt

Durchführung

- Vom Parser den regulären Ausdruck analysieren lassen

Abschließender Test

- Der Test ist genau dann erfolgreich, wenn der Parser eine **RuntimeException** wirft.

10.4 Standalone-Test für den ersten Visitor

Formulieren Sie ein oder mehrere JUnit-Tests wie folgt:

Eingabe

- Ein Syntaxbaum hartverdrahtet erstellt
- Eine Kopie des ersten Syntaxbaums mit zusätzlichen hartverdrahteten Werten für die Attribute **nullable**, **firstpos** und **lastpos** (erwartetes Ergebnis)

Durchführung

- Den Visitor den ersten Syntaxbaum durchwandern und dessen Attribute **nullable**, **firstpos** und **lastpos** mit Werten belegen lassen.

Abschließender Test

- Der Test ist genau dann erfolgreich, wenn der erste, vom Visitor durchwanderte, Syntaxbaum (das aktuelle Ergebnis) gleich dem hartverdrahteten zweiten Syntaxbaum (das erwartete Ergebnis) ist (→ Kapitel 10.5).

10.5 Der Vergleich zweier Syntaxbäume

Wieder müssen zwei Syntaxbäume – der Hartverdrahtete (erwartetes Ergebnis) und derjenige Syntaxbaum, der vom ersten Visitor durchwandert wurde – in Tiefensuche durchwandert werden. Ihr Junit-Test sollte folgende Methode implementieren.

```

private boolean equals(Visitable expected, Visitable visited)
{
    if (expected == null && visited == null) return true;
    if (expected == null || visited == null) return false;
    if (expected.getClass() != visited.getClass()) return false;

    if (expected.getClass() == BinOpNode.class)
    {
        BinOpNode op1 = (BinOpNode) expected;

```

```

        BinOpNode op2 = (BinOpNode) visited;

        return op1.nullable.equals(op2.nullable) &&
            op1.firstpos.equals(op2.firstpos) &&
            op1.lastpos.equals(op2.lastpos) &&
            equals(op1.left, op2.left) &&
            equals(op1.right, op2.right);
    }

    if (expected.getClass() == UnaryOpNode.class)
    {
        UnaryOpNode op1 = (UnaryOpNode) expected;
        UnaryOpNode op2 = (UnaryOpNode) visited;

        return op1.nullable.equals(op2.nullable) &&
            op1.firstpos.equals(op2.firstpos) &&
            op1.lastpos.equals(op2.lastpos) &&
            equals(op1.subNode, op2.subNode);
    }

    if (expected.getClass() == OperandNode.class)
    {
        OperandNode op1 = (OperandNode) expected;
        OperandNode op2 = (OperandNode) visited;

        return op1.nullable.equals(op2.nullable) &&
            op1.firstpos.equals(op2.firstpos) &&
            op1.lastpos.equals(op2.lastpos);
    }

    throw new IllegalStateException(
        String.format( "Beide Wurzelknoten sind Instanzen der Klasse
%1$s !"
                        + " Dies ist nicht erlaubt!",
                        expected.getClass().getSimpleName())
    );
}

```

10.6 Standalone-Test für den zweiten Visitor

Formulieren Sie ein oder mehrere JUnit-Tests wie folgt:

Eingabe

- Ein Syntaxbaum mit hartverdrahteten Werten für die Attribute **nullable**, **firstpos** und **lastpos**
- Eine followpos-Tabelle hartverdrahtet erstellt (erwartetes Ergebnis)

Durchführung

- Den Visitor den Syntaxbaum durchwandern und eine neue followpos-Tabelle erstellen lassen.

Abschließender Test

- Der Test ist genau dann erfolgreich, wenn die hartverdrahtete followpos-Tabelle (erwartetes Ergebnis) gleich der Neuen ist (aktuelles Ergebnis).
Beachten Sie! Um einen Test auf Gleichheit durchzuführen, genügt es, die statische Methode **assertEquals** aufzurufen (mit beiden followpos-Tabellen als Parameter)!

10.7 Standalone-Test für den DEA-Erzeuger

Formulieren Sie ein oder mehrere JUnit-Tests wie folgt:

Eingabe

- Eine Positionenmenge, die den Startzustand des DEAs darstellt
- Eine followpos-Tabelle
- Eine DEA-Übergangsmatrix hartverdrahtet (erwartetes Ergebnis des DEA-Erzeugers)

Durchführung

- Den DEA-Erzeuger aus Positionenmenge und followpos-Tabelle eine neue DEA-Übergangsmatrix erstellen lassen.

Abschließender Test

- Der Test ist genau dann erfolgreich, wenn die hartverdrahtete DEA-Übergangsmatrix (erwartetes Ergebnis) gleich der Neuen ist (aktuelles Ergebnis).
Beachten Sie! Um einen Test auf Gleichheit durchzuführen, genügt es, die statische Methode **assertEquals** aufzurufen (mit beiden DEA-Übergangsmatrizen als Parameter)!

10.8 Standalone-Test für den generischen Lexer

Formulieren Sie ein oder mehrere JUnit-Tests wie folgt:

Eingabe

- Eine DEA-Übergangsmatrix
- Ein zu testender String
- Ein boolescher Wert (das erwartete Ergebnis)

Durchführung

- Einen generischen Lexer erzeugen, abhängig von der DEA-Übergangsmatrix
- Den String mit der Methode **match** testen

Abschließender Test

- Der Test ist genau dann erfolgreich, wenn die Auswertung der Methode **match** gleich dem erwarteten booleschen Wert ist.

11 Quellen

Das [Visitoren-Entwurfsmuster](#)

[Compilerbau – Prinzipien, Techniken und Werkzeuge](#) (google book: Kapitel 3.9 ff.)