Inhaltsverzeichnis

1 Aufgabenbeschreibung

Diese Aufgabe ist für maximal 5 Personen gedacht. Im Falle einer Dreiergruppe müssen nur der Top-Down-Parser und die beiden Visitoren 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
  - er erzeugt im positiven Fall einen AST, d.h. (abstrakten) Syntaxbaum des regulären Ausdrucks
  - er wirft im negativen Fall eine Exception
- Einen ***Visitor*** implementieren
  - er durchwandert in Tiefensuche (L → R → W) den AST
  - 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
- Einen ***zweiten Visitor*** implementieren
  - 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 → R → 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 ∈ ***lastpos***(linker Sohnknoten) und führt folgende Operationen aus:
      - ***followpos***(Knoten an Position i) = ***followpos***(Knoten an Position i) ∪ ***firstpos***(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 ∈ ***lastpos***(akt. innerem Knoten) und führt folgende Operationen aus:
      - ***followpos***(Knoten an Position i) = ***followpos***(Knoten an Position i) ∪

firstpos(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

- Eine Klasse implementieren, welche den DEA (***Pseudocode auf Seite 216*** des Drachenbuches) realisiert
  - Sie ermittelt aus derjenigen Tabelle, die der 2. Visitor angelegt hat, das Alphabet Σ des Ziel-DEAs
  - Sie liefert als Ergebnis eine Tabelle (***Übergangsmatrix***!)
    - die Tabelle hat 1 + |Σ| Spalten
    - erste Spalte repräsentiert einen Ausgangszustand
    - jede weitere Spalte repräsentiert ein mögliches Eingabesymbol, d.h. ab der zweiten Spalte steht ***in jeder Zelle ein Folgezustand***, abhängig vom Ausgangszustand der aktuellen Zeile und dem Eingabesymbol der aktuellen Spalte
  - Sie markiert den Startzustand der Übergangsmatrix.
    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

- Eine Klasse implementieren, die einen ***generischen Lexer*** realisiert
  - Der Konstruktor erhält als Parameter, die oben erwähnte Übergangsmatrix, die den gesuchten DEA darstellt
  - Eine Methode **`match`**, die überprüft, ob ein eingegebener String vom übergebenen 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

2 Schnittstellenbeschreibung

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

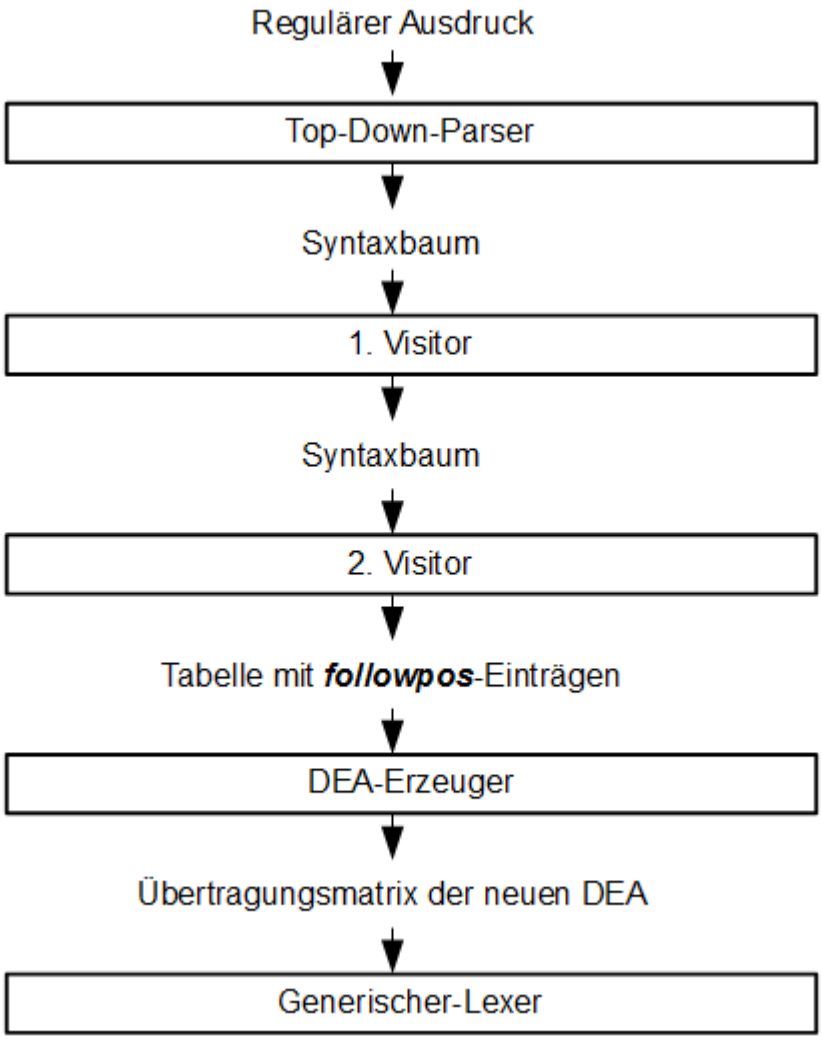

Abb. 1 Schaubild mit Schnittstellen

Gemäß Abb. 1 ergeben sich folgende Schnittstellen:

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

2.1 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 vistor)
   {
      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 vistor)
   {
      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 vistor)
   {
      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!

2.2 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;
   }
}
```

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!

2.3 Die Übergangsmatrix des neuen DEA

Zunächst führen wir eine package-private Struktur ein, die einen Zustand beschreibt und die bereits durch ihren Member **`index`** eindeutig identifiziert werden kann.

```
class DFAState implements Comparable<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)
   {
     if(this == obj)                  return true;
     if(obj == null)                  return false;
     if(getClass() != obj.getClass()) return false;

     DFAState other = (DFAState)obj;
     return (other.index == this.index);
```

```
   }

   @Override
   public int hashCode()
   {
     final int prime = 31;
     int result = 1;

     result = prime * result + this.index;
     return result;
   }

   @Override
   int compareTo(DFAState other)
   {
      return (this.index – other.index);
   }
}
```

Beachten Sie!
Die Klasse **`DFAState`** kann in Abhängigkeit ihres Members **`index`** sortiert werden! Wenn wir also einen allgemeinen Zähler mitführen, ihn bei jeder neuen Instanz von **`DFAState`** um eins inkrementieren und ihn als Wert für den Index der neuen **`DFAState`**-Instanz verwenden, muss die **`DFAState`**-Instanz mit dem kleinsten Index der ***Startzustand*** sein!

Eine Konsequenz davon ist, dass wir auch hier eine TreeMap verwenden, um die Übergangsmatrix der DEA darzustellen.

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

2.4 Implementierungsvorgabe bei den Visitoren

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

```
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!");
    }
}
```

3 Bewertung

Was wird bewertet	Zu welchem Anteil
die korrekte Umsetzung	max. 55 Punkte
die Kompilierfähigkeit	max. 5 Punkte
Einhalten der Schnittstellenvorgaben	max. 15 Punkte
Einhalten der Programmiererkonventionen	max. 25 Punkte

4 Abgabe

Was	eine ZIP-Datei (Quelldatei + Testprotokoll)
Wohin	farid.derradji@web.de
Eingabefrist	15.02.2019 (verspätet zugeschickte ZIP-Dateien werden nicht berücksichtigt!) Jedes Team erhält nach Einreichen der ZIP-Datei via Email eine Empfangsbestätigung.

5 Hinweise

5.1 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_1r_2 (Konkatenation)
    2. $r_1 \mid r_2$ (Alternative)
    3. r_i* (Kleenesche Hülle)
    4. r_i+ (Positive Hülle)
    5. r_i? (Option)

    ebenfalls reguläre Ausdrücke.
iii) Der Einfachheit halber seien Whitespaces nicht zulässig

5.2 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'       →    Ɛ
RE'       →    '|' Term RE'

Term      →    FactorTerm
Term      →    Ɛ

Factor    →    Elem HOp
HOp       →    Ɛ
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'       →    Ɛ {RE'.return = RE'.parameter;}

RE'       →    '|'
               {Term.parameter = null;}
               Term
               {
                    root = new BinOpNode('|',
                                         RE'.parameter,
                                         Term.return);
                    RE₁'.parameter = root;
               }
               RE₁'
               {RE'.return = RE₁'.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      →    Ɛ {Term.return = Term.parameter;}

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

HOp       →    Ɛ {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');}
...            ...
```

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

6 Quellen

Das Visitoren-Entwurfsmuster
Compilerbau – Prinzipien, Techniken und Werkzeuge (google book: Kapitel 3.9 ff.)