Technical Reports and Working Papers

Angewandte Datentechnik (Software Engineering)
Institute of Electrical Engineering and Information Technology
The University of Paderborn

Automata Tools

Verlässliches Programmieren in C/C++

Daniel Dreibrodt, Florian Hemmelgarn,
Fabian Ickerott, Sebastian Kowelek,
Yacine Smaoui, Konstantin Steinmiller

Betreuer: Mutlu Beyazıt

Technical Report 2012/03 (Version 1.0, Aug. 2012)

D-33095 Paderborn, Pohlweg 47-49 Tel.: +49-5251-60-3447 Fax: 60-3246
http://adt.upb.de email: belli@adt.upb.de

Inhaltsverzeichnis

1 Einleitung

Das Projekt *Automata Tools* befasst sich mit der Erstellung einer C++-Bibliothek, die Funktionen im Zusammenhang mit Endlichen Automaten, Regulären Ausdrücken und Regulären Grammatiken bereitstellt.

Konkret wurde gefordert, dass Datenstrukturen für Endliche Automaten, Reguläre Grammatiken und Reguläre Ausdrucke, sowie Minimierungs-, Äquivalenz- und Konvertierungsalgorithmen zu implementieren.

Im folgenden sollen die gewählten Datenstrukturen und implementierten Algorithmen kurz erläutert werden.

2 Zustandsautomat

3 Reguläre Ausdrücke

3.1 Syntax

In der gewählten Implementation stehen die Operatoren Konkatenation ($x.y$), Alternative ($x|y$) und Kleene-Stern ($x*$) zur Verfügung. Im Unterschied zu den meisten gängigen Implementationen (vergl. POSIX[1], Perl[2]) müssen Konkatenationen explizit angegeben werden.

Des weiteren können Ausdrücke durch Klammern zusammengefasst sein und so die Präzedenz der Operatoren festlegen. Wird die Präzedenz nicht durch Klammerung festgelegt, so hat der jeweils am weitesten rechts stehende Operator Präzedenz.

So entspricht $a|b.c*$ dem geklammerten Ausdruck $(a|(b.(c*)))$.

Die Literale bestehen aus beliebig langen Zeichenketten ohne Leerzeichen und Operatoren. Im Gegensatz zu den gängigen Implementation (vergl. POSIX[1], Perl[2]) wird hier die gesamte Zeichenkette als ein Literal aufgefasst und nicht als Konkatenation der einzelnen Zeichen.

Ein besonderes Literal ist die Zeichenfolge $<epsilon>$. Sie entspricht einem leeren Ausdruck.

3.2 Datenstruktur

Reguläre Ausdrücke werden in diesem Projekt als binärer Ausdrucksbaum repräsentiert.

Jeder Knoten enthält dabei entweder einen Operator oder ein Literal. Die Operanden werden in den Nachfolgern des Operator-Knoten gespeichert. Bei einem Kleene-Stern gibt es nur einen Operand, welcher im linken Nachfolger gespeichert wird. Der rechte Nachfolger muss leer bleiben.

Literale können keine Nachfolger haben, sie sind immer Blätter des Ausdrucksbaums.

3.2.1 Implementation

Die Klasse *RegularExpression* repräsentiert einen Regulären Ausdruck. Dies tut sie durch Verweis auf den Wurzelknoten des Ausdrucksbaums.

Die Knoten sind Objekte der Klasse *RETreeNode*.

Ein Knoten hat einen Inhalt, welcher in der Textvariable content gespeichert ist. Diese Variable enthält entweder die textuelle Repräsentation eines Literals oder eines Operanden.

Die Nachfolger eines Knotens sind in den Pointern *p_left* und *p_right* gespeichert.

3.3 Einlesen von Regulären Ausdrücken

Dateiformat

Das gewählte Dateiformat ist sehr simpel. Der reguläre Ausdruck wird einfach in einer einzelnen Zeile in einer Textdatei gespeichert. Die Datei sollte nur ASCII-Zeichen enthalten.

Parser

Das Parsen von Regulären Ausdrücken ist in der Klasse REReaderWriter implementiert. Während des Parsens wird die Eingabezeichenfolge von links nach rechts durchgelaufen und der Ausdrucksbaum von unten nach oben aufgebaut.

Somit ähnelt der Prozess dem eines Shift-Reduce-Parsers[3], der jedoch nicht nach formalen Methoden konstruiert wurde.

3.4 Konvertierungen

3.4.1 Konvertierung zu Endlichem Automaten

Bei der Konvertierung eines Regulären Ausdrucks zu einem endlichen Automaten wird der Ausdrucksbaum in symmetrischer Reihenfolge (in-order) durchlaufen und für jeden Knoten ein Automat erstellt. Die Konvertierung wurde im Rahmen des Projektes selbstständig entworfen. Das Ergebnis der Konvertierung ist ein nichtdeterministischer endlicher Automat.

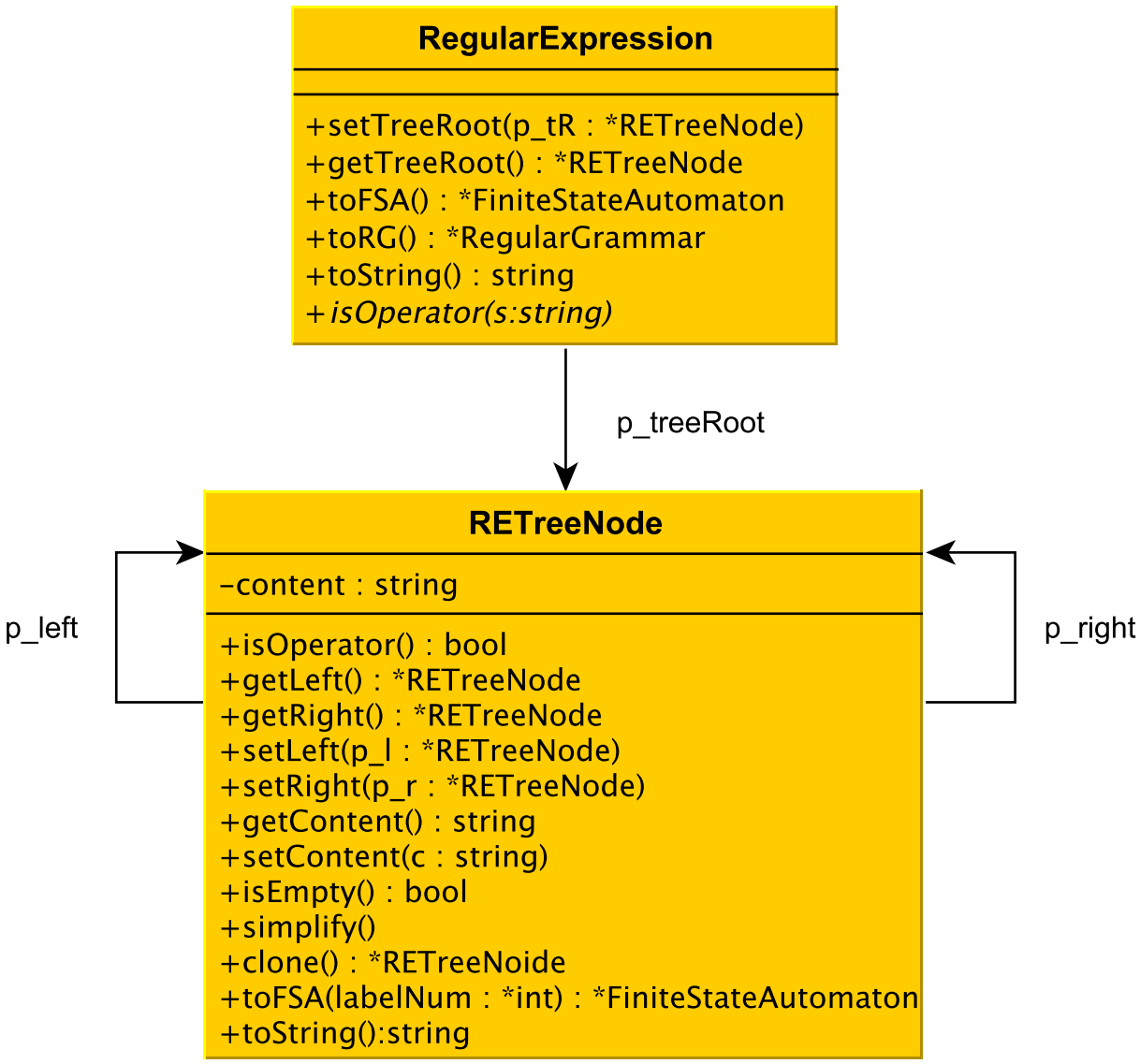

Abbildung 3.1: UML Diagramm zu *RegularExpression* und *RETreeNode*

3.4.1.1 Literal

Enthält ein Knoten ein Literal, so wird ein Automat einem Start- und einem Endzustand erzeugt. Der Startzustand erhält einen Übergang zum Endzustand mit dem Literal als Eingabe.

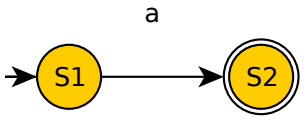

Abbildung 3.2: FSA für a

3.4.1.2 Konkatenation

Bei einer Konkatenation wird der Automat des rechten Knotens an den des linken angehängt. Dazu erhalten alle Endzustände des linken Automaten einen Übergang mit leerer Eingabe zu dem Startzustand des rechten Automaten. Anschließend sind alle Endzustände des linken Automaten nicht mehr Endzustände und der Starzustand des rechten Automaten ist kein Startzustand mehr.

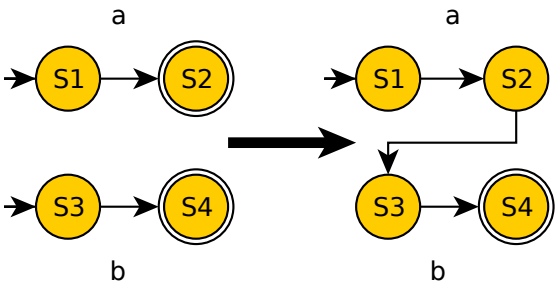

Abbildung 3.3: FSA für $a.b$

3.4.1.3 Alternative

Bei einem Knoten, der eine Alternative darstellt wird ein neuer Automaterzeugt, dessen Startzustand leere Übergänge zu den Startzuständen der Automaten des linken und rechten Nachfolgers hat.

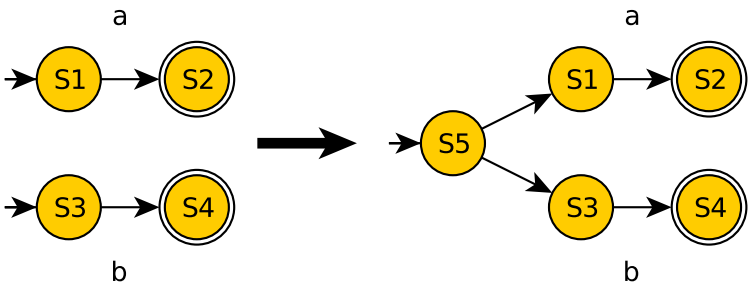

Abbildung 3.4: FSA für $a|b$

3.4.1.4 Kleene-Stern

Enthält ein Knoten einen Kleene-Stern, so wird nur mit dem Automaten des linken Nachfolgers gearbeitet. Alle Endzustände des Automaten erhalten leere Übergänge zum Startzustand. Der Startzustand wird auch zu einem Endzustand.

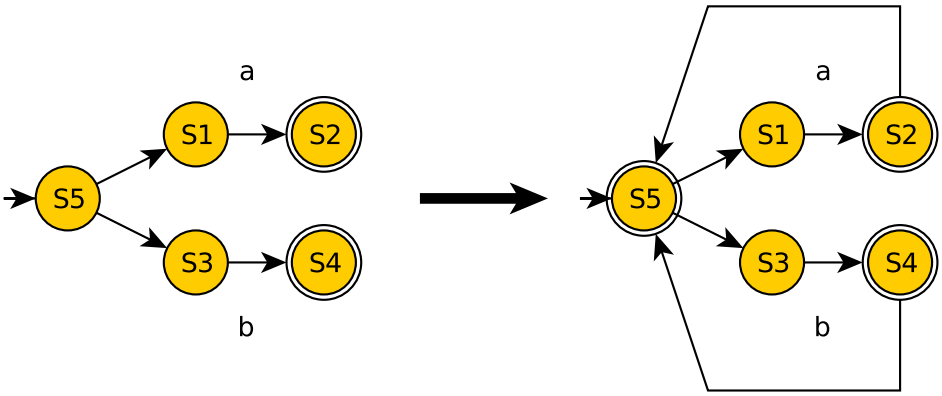

Abbildung 3.5: FSA für $(a|b)*$

3.4.1.5 Entfernung der unnötigen leeren Übergänge

In vielen Fällen sind die bei der Konvertierung entstandenen Zustände mit leeren Übergängen unnötig.

Die implementierten Minimierungs-Algorithmen (Moore und Table-Filling) sind allerdings nicht in der Lage diese unnötigen Zustände und Übergänge zu entfernen. Daher wurde ein eigener Algorithmus entworfen der möglichst alle Knoten welche mit leeren Übergängen verbunden sind zu vereinen, solange dies die akzeptierte Sprache nicht verändert.

In diesem Algorithmus werden alle Zustände des Automaten durchlaufen und für jeden Zustand seine ausgehenden Kanten.

Wird eine leere Kante auf sich selbst gefunden, so wird diese einfach gelöscht.

Werden leere Transitionen zu anderen Zuständen gefunden so können der Ausgangszustand und alle Zielzustände unter Umständen zu einem Zustand vereint werden. Dies kann nur unter der Bedingung geschehen, dass alle Zielzustände keine weiteren Eingangszustände haben, da sonst beim Vereinen der Zustände die Sprache verändert würde.

Wird eine nicht-leere, ausgehende Kante gefunden so kann der Zustand nicht einfach mit anderen Zuständen zu denen er leere Übergänge hat vereint werden, da auch dann die akzeptierte Sprache verändert werden würde.

Dieser Durchlauf wird so-oft durchgeführt, bis keine Vereinigungen mehr gemacht werden konnten.

Somit werden die meisten unnötigen Zustände und leeren Übergänge entfernt.

4 Reguläre Grammatik

4.1 Definition

4.1.1 Definition einer Grammatik

Eine Grammatik G =(T,N,P,S) besteht aus:

T	einer Menge von Terminalsymbolen (kurz Terminalen)
N	einer Menge von Nichtterminalsymbolen (kurz Nichtterminale)
	T und N sind disjunkte Mengen
$S \in N$	einem Startsymbol aus der Menge der Nichtterminale
$P \subseteq N \times V^*$	Menge der Produktionen; (A,x) $\in$ P, A $\in$ N und x $\in V^*$;
	statt (A,x) schreibt man $A \rightarrow x$
$V=T \cup N$	heißt Vokabular, seine Elemente heißen Symbole

4.1.2 Definition einer rechtslinearen Grammatik

Eine Grammatik ist regulär wenn sie entweder eine rechtslineare Grammatik oder eine linkslineare Grammatik.

Eine Grammatik G=G(T,N,P,S) ist einer rechtslineare Grammatik, wenn sie folgenden Anforderungen genügt:

$X \rightarrow a\ Y$
$X \rightarrow a$
$X \rightarrow \varepsilon$
mit X,Y $\in$ N und a $\in$ T

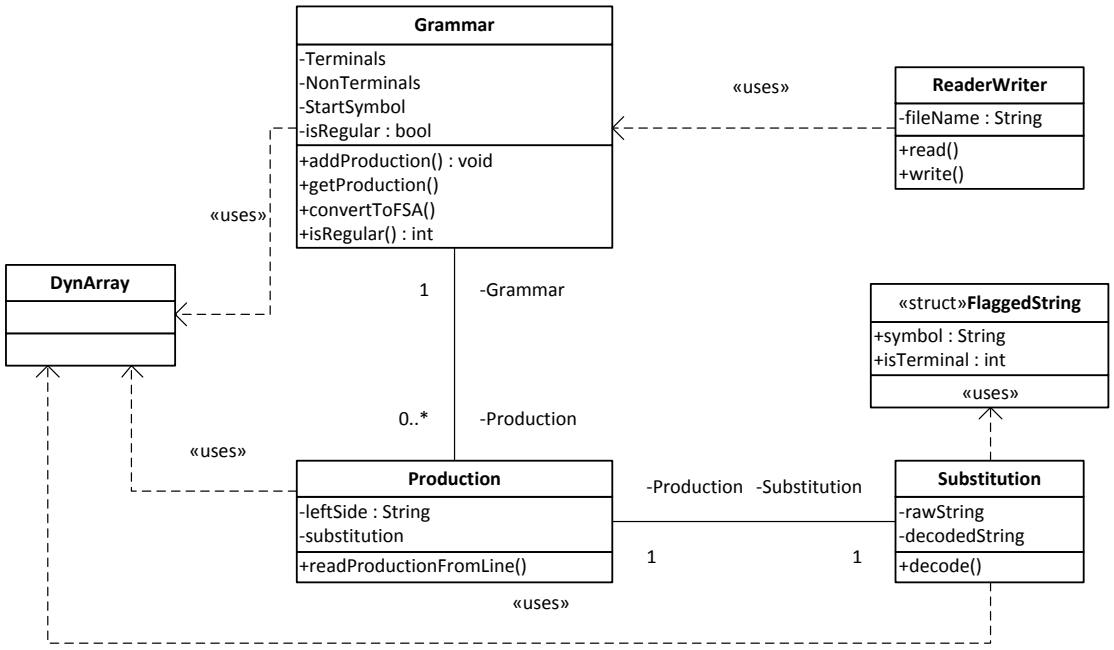

Abbildung 4.1: Reguläre Grammatik UML Diagramm

4.2 Datenstruktur

4.2.1 Implementation

Die Klasse *Grammar* besteht aus zwei *dynArrays Terminals* und *NonTerminals*, in diesen werden alle Terminal- und Nichtterminalsymbole als String gespeichert, einem String *Startsymbol* für das Startsymbol der Grammatik, einer Integer Variablen isRegular, die angibt ob die Grammatik regulär ist und einem weitern *dynArray Productions* für die Produktionen. Die Template-Klasse *dynArray* ermöglicht eine unbekannte Anzahl von Variablen in einem Feld zu speichern, auf die Inhalte des Feldes wird mit dem [ ]-Operator zugegriffen.

Produktionen bestehen aus einem Nichtterminal auf der linken Seite und eine Substitution auf der rechten Seite des Pfeils. Die Klasse *Production* setzt sich aus einer Stringvariablen für die linke Seite und einer Klasse Substitution für die rechte Seite zusammen.

Die Klasse *Substitution* enthält eine Zeichenkette *rawString*. In dieser wird eine unbearbeitet Substitution abgespeichert, also zum Beispiel direkt nach dem Einlesen, bevor diese dann weiter verarbeitet wird. In einem *dynArray* vom Typ *flaggedString* wird dann die vollständig verarbeitete Substitution ,unter dem Namen *decodedSubstitution*,

gespeichert. Der Struct *flaggedString* setzt sich aus einem Integer, der angibt ob das Symbol ein Terminal ist oder nicht und einem String, der das Symbol speichert, zusammen. Die Funktion *decode* wandelt nun den *rawString*, eine Folge von Terminalen und Nichtterminalen, in ein flaggedString Array um, zur Weiterverarbeitung.

4.3 Einlesen von regulären Grammatiken

Es kann mittels der Klasse *RGReaderWriter* eine Grammatik aus einer Datei eingelesen oder in eine Datei geschrieben werden. Hinter einem Tag folgen die zugehörigen Symbole. Das Einlesen erfolg zeichenweise.

```
[Start]
<expression>
[Terminals]
a
b
c
d
[NonTerminals]
<expression>
[Productions]
<expression> —> a
<expression> —> b <expression>
<expression> —> a b <expression>
<expression> —> c d a b <expression>
<expression> —> d <expression>
```

Listing 4.1: Beispiel zu einem Textdatei der einen Grammatik darsttellt.

4.4 Konvertierungen von regulären Grammatiken

Konvertierungen von eine regulären Grammatik zu einem endlichen Zustandsautomat oder einem regulären Ausdruck, sind möglich nur wenn diese Grammatik regulär ist. Im Rahmen dieser Projektarbeit, eine rechtslinieare Grammatik ist gewählt. (Siehe definition einer rechtslinearen Grammatik).

4.4.1 Konvertierung zu einem endlichen Zustandsautomaten

Eine Transition vom Zustand A zum Zustand B mit a als Eingabe wird in einer regulären Grammatik als " A $\rightarrow$ a B " gespeichert. Um Zustand B als Endzustand zu

setzen gibt es mehrere Möglichkeiten, entweder durch z.B B → a und B → a B, also in eine Produktion nur ein Terminal auf der rechten Seite und in eine andere Produktion eine Schleife, oder durch ε als rechte Seite. Ist ein Zustand in einer Transition sowohl Anfang als auch Ende wird dies durch A → a A realisiert: eine Schleife.

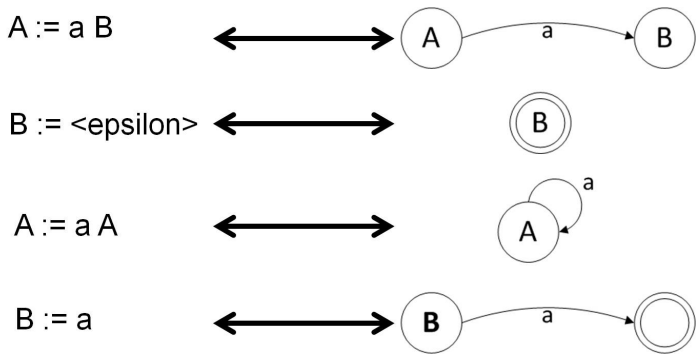

Abbildung 4.2: Zusammenhang zwischen RG und FSA

Für die Umwandlung werden die Produktionen der Reihe nach durchgegangen. Es gibt 4 mögliche Produktionstypen:

1. A → ε
2. A → a
3. A → a A
4. A → b A , wobei b eine Folge von Terminalen ist.

Es wird zuerst überprüft ob die linke Seite im Automaten schon als Zustand vorhanden ist, falls nicht wird diese dem FSA hinzugefügt. Sollte diese das Startsymbol sein wird der Zustand als Startzustand gespeichert. Ein Hilfspointer wird auf diesen Zustand gesetzt.

Ist die rechte Seite das leere Symbol (Fall 1) wird der Zustand als final markiert und die nächste Produktion umgewandelt, falls nicht wird jedes Symbol der Substitution durchgegangen.

Ist das aktuelle Symbol ein Terminal und das letzte Symbol der Substitution(Fall 2) wird ein Zustand Endstate erzeugt und eine Transition vom Hilfpointer zum Endstate mit dem Terminal als Kante dem Automaten hinzugefügt. Der Endstate wird als final markiert.

Ist das aktuelle Symbol ein Terminal, nicht das letzte Symbol und wird von einem Nichtterminal gefolgt (Fall 3), wird das Nichtterminal, falls noch nicht vorhanden, dem Automaten als Zustand hinzugefügt und eine Transition vom Hilfspointer zum neuen Zustand mit dem Terminal als Kante erstellt.

Ist das aktuelle Symbol nicht das letzte Symbol, ein Terminal und wird von einem weiteren Terminal(Fall4)gefolgt, wird zunächst ein Hilfzustand erzeugt und eine Transition vom Hilfspointer zum Hilfszustand mit dem Terminal als Kante hinzugefügt. Der Hilfspointer wird auf den Hilfszustand gesetzt. Dieser Vorgang wird solange wiederholt bis das aktuelle Symbol ein Terminal und das folgende Symbol ein Nichtterminal ist. Dann wird, falls noch nicht vorhanden, ein Zustand des Nichtterminals erzeugt und eine Transition vom Hilfspointer zum neuen Zustand mit dem Terminal als Kante erstellt. Im Anschluss wird die nächste Produktion verarbeitet.

4.4.2 Konvertierung zu einem regulären Ausdrück

Eine direkte Konvertierung von einer reguläre Grammatik zu einer regulären Ausdrück ist im Rahmen dieses Projektes nicht implementiert. Diese Konvertierung wird aber durch die konvertierung von endlichen Zustandsautomaten zu eine regulären Ausdruck realisiert.

Eine direkte Umwandlung ist aber möglich. Die Idee ist im Folgenden dargestellt:

4.4.2.1 Die Grammatik in einer Kompaktform schreiben

Alle Produktionen die dasselbe Linkeseite haben werden zusammengefasst. z.B:

$$\begin{array}{l} X \to a\ Y \\ X \to b\ Z \\ X \to a\ \varepsilon \end{array} \quad \Longrightarrow \quad X \to a\ Y \mid b\ Z \mid \varepsilon$$

4.4.2.2 Die Produktionen als gleichungen interpretieren

Dafür werden die $\to$ zu "=" und die "|" zu "+" umgewandelt. z.B:

$$X \to a\ Y \mid b\ Z \mid \varepsilon \quad \Longrightarrow \quad X = a\ Y + b\ Z + \varepsilon$$

4.4.2.3 Das Gleichungssystem lösen

Gesucht ist eine Gleichung wo die einzige Variable ist das Startsymbol.

Die Distributivität und Assoziation sind hier zwei verwendbare Eigenschaften der gewönliche "+" Operation und um Produktionen der Form A = c A + B können in dieser Form umgewandelt: A = c* B .

4.5 Minimierung eines endlichen Zustandsautomaten mittels den Table-Filling-Algorithmus

4.5.1 Der Algorithmus

Dieses Algorithmus ist abgeleitet vom Äquvalenzsatz von Myhill und Nerode.

Eine Matrix wo in die Zeilen und Spalten alle Zustände von den Automat stehen.

Die Grundidee ist, dass in dieser Matrix alle Paare von Zuständen markiert werden die nicht äquivalent sind.

Formal:
Seien P und Q zwei Zustände, ω ein Satz (eine folge von Terminale), und X = δ(P,ω) und Y = δ(P,ω) die nachfolgende Zustände von P und Q mit eingabe ω .

P und Q sind zwei nicht-äquivalente Zustände wenn nur ein Zustand vom Zustandspaar (X,Y) ist ein akzeptierender Zustand und der andere ist ein nichtakzeptierender Zustand.

Falls für den Paar (P,Q), es kann nicht entschieden werden ob die zwei Zustände äquivalent oder nicht, diese zustände werden zu die Abhängigkeiten (dependencies) von die Nachfolgende Zustände (X,Y) hinzugefügt. (P,Q) werden dann später die Eigenschaft von (X,Y) haben.

```
1. Initialisiere alle Paare in der Matrix als nicht-markiert und ohne
   dependencies
2. Markiere alle Paare von akzeptierenden Zuständen und nicht-
   aktzeptierenden Zuständen als nicht-äquivalent
3. Für alle nicht markierte Paare (P,Q) und jedes Eingabesymbol a:
   Sei X = delta(P,a), Y = delta(Q,a)
     Falls (X,Y) nicht markiert ist,
       füge (P,Q) zu den dependencies von (X,Y) hinzu
     Sonst
       markiere (P,Q) und markiere alle dependencies von (P,Q)
```

Listing 4.2: Table-Filling-Algorithmus : Pseudocode

4.5.2 Implementierung

Um dieser Algorithmus zu impelementieren, eine neue Klasse *TableFillingMinimizer* wurde erstellt. Diese Klasse bietet die Methoden nötig um einen endlichen Zustandsautomat zu minimieren.

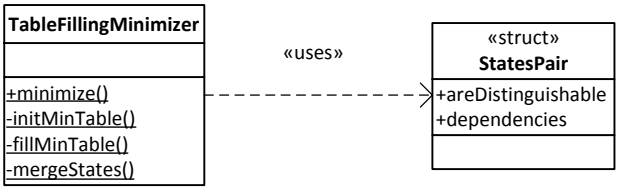

Abbildung 4.3: UML Diagramm zu *TableFillingMinimizer*

Die verwendete Matrix in diesem Algorithmus ist eine Matrix von *StatesPair*, wobei *StatesPair* ist eine Struktur wo gespeichert wird ob ein paar nicht-äquivalent ist oder nicht, und die Liste von Abhängigkeiten dieses Paar.

5 Fazit

Literaturverzeichnis

[1] The IEEE and The Open Group, "IEEE Std 1003.1, 2004 Edition - Regular Expressions." http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html. Accessed: 05/08/2012.

[2] The Perl 5 Porters, "Perl Programming Documentation - Regular Expressions." http://perldoc.perl.org/perlre.html. Accessed: 05/08/2012.

[3] Wikipedia contributors, "Wikipedia - Shift-reduce parser." http://en.wikipedia.org/wiki/Shift-reduce_parser. Accessed: 05/08/2012.

Abbildungsverzeichnis