

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

Copyright All rights including translation into other languages is reserved by the authors.

No part of this report may be reproduced or used in any form or by any means - graphically, electronically or mechanically, including photocopying, recording, taping or information and retrieval systems - without written permissions from the authors and or projects.

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

D-33095 Paderborn, Pohlweg 47-49
<http://adt.upb.de>

Tel.: +49-5251-60-3447
email: belli@adt.upb.de

Fax: 60-3246

Inhaltsverzeichnis

1	Einleitung	1
2	Reguläre Ausdrücke	2
2.1	Syntax	2
2.2	Datenstruktur	2
2.2.1	Implementation	3
2.3	Einlesen von Regulären Ausdrücken	3
2.4	Konvertierungen	3
2.4.1	Konvertierung zu Endlichem Automaten	3
3	Reguläre Grammatik	8
3.1	Definition	8
3.1.1	Definition einer Grammatik	8
3.1.2	Definition einer rechtslinearen Grammatik	8
3.2	Datenstruktur	8
3.2.1	Implementation	8
3.3	Einlesen von regulären Grammatiken	10
3.4	Konvertierungen	10
3.4.1	Konvertierung zu einem endlichen Zustandsautomaten	10
3.5	Minimierung eines endlichen Zustandsautomaten	10
4	Zustandsautomat	11
5	Zusammenfassung	12
	Literaturverzeichnis	13
	Abbildungsverzeichnis	14

1 Einleitung

bla bla bla

2 Reguläre Ausdrücke

[?, ?]

2.1 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[Rega], Perl[Regb]) 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[Rega], Perl[Regb]) wird hier die gesamte Zeichenkette als ein Literal aufgefasst und nicht als Konkatenation der einzelnen Zeichen.

Ein besonderes Literal ist die Zeichenfolge $\langle \textit{epsilon} \rangle$. Sie entspricht einem leeren Ausdruck.

2.2 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.

2.2.1 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.

2.3 Einlesen von Regulären Ausdrücken

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[Regc], der jedoch nicht nach formalen Methoden konstruiert wurde.

2.4 Konvertierungen

2.4.1 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.

2.4.1.1 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.

2.4.1.2 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 Startzustand des rechten Automaten ist kein Startzustand mehr.

2.4.1.3 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.

2.4.1.4 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.

2.4.1.5 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.

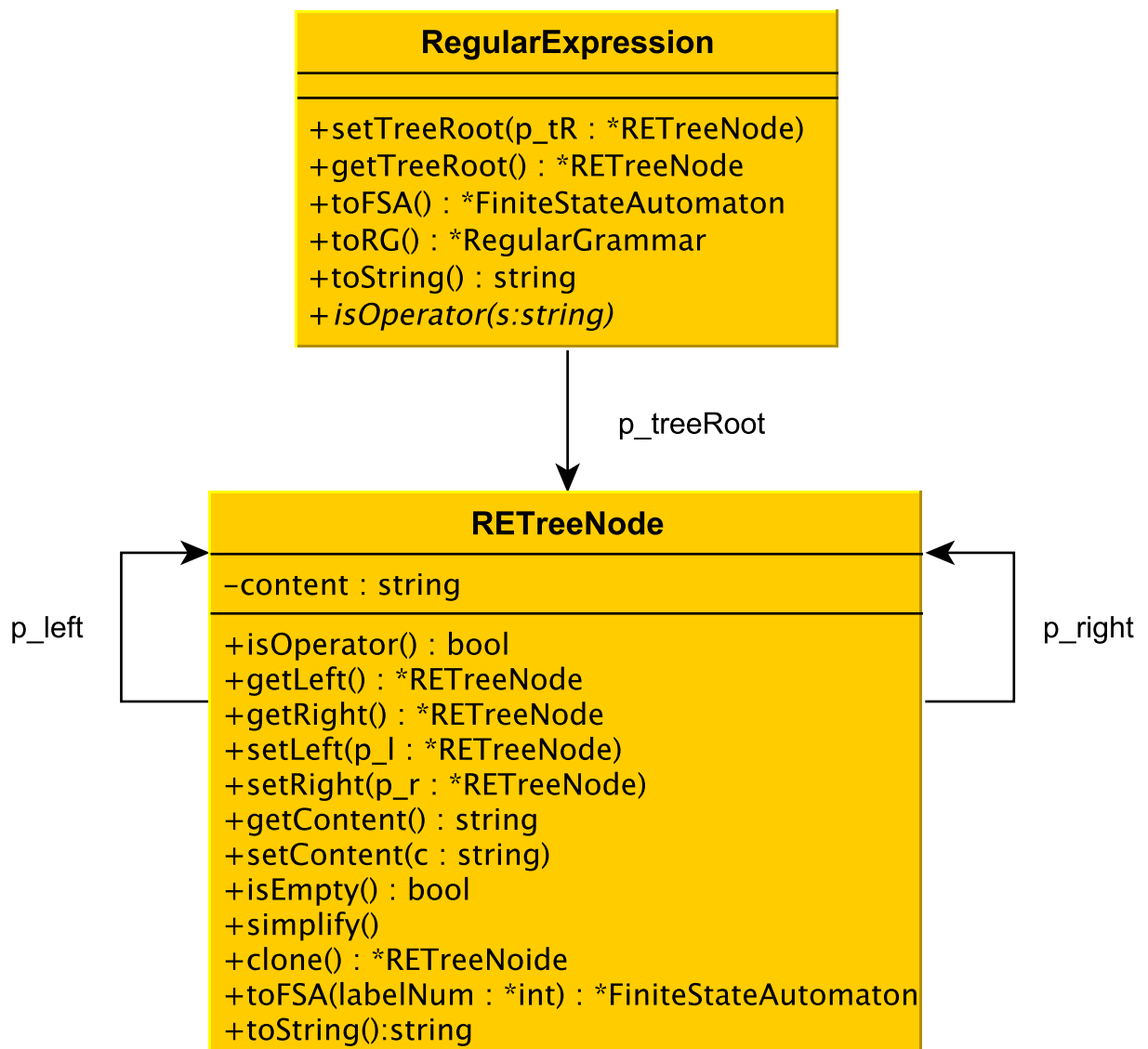


Abbildung 2.1: UML Diagramm zu *RegularExpression* und *RETreeNode*

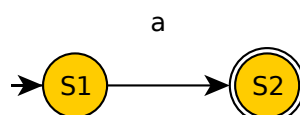


Abbildung 2.2: FSA für *a*

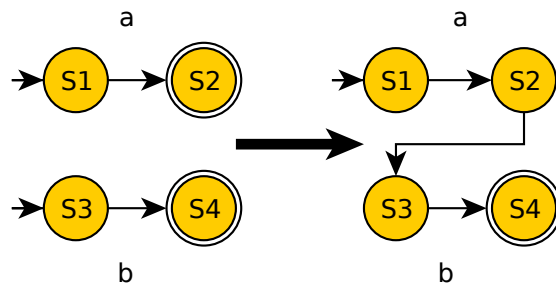


Abbildung 2.3: FSA für $a.b$

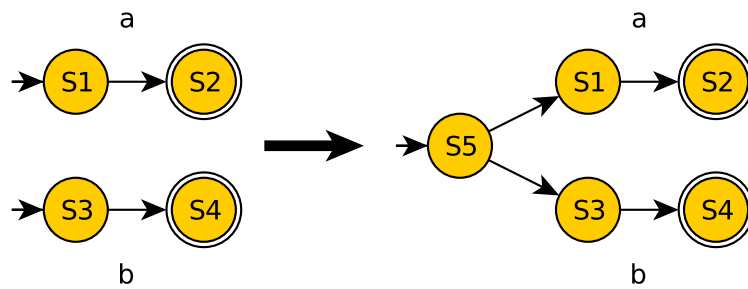


Abbildung 2.4: FSA für $a|b$

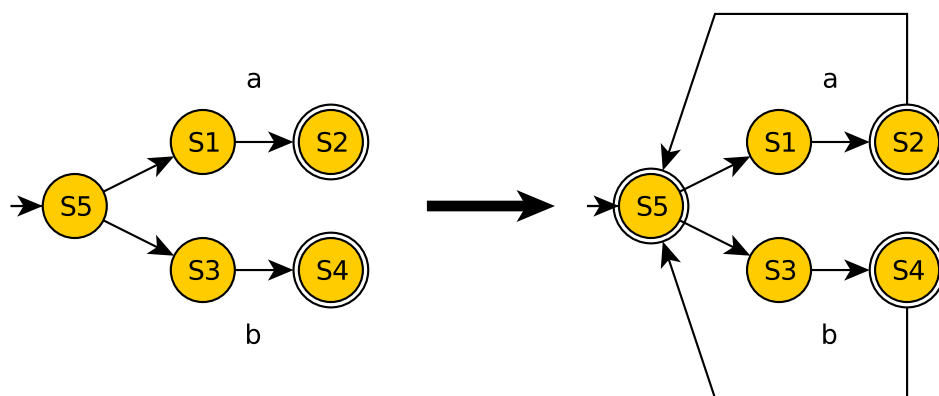


Abbildung 2.5: FSA für $(a|b)^*$

3 Reguläre Grammatik

3.1 Definition

3.1.1 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

3.1.2 Definition einer rechtslinearen Grammatik

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

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

3.2 Datenstruktur

3.2.1 Implementation

Die Klasse *Grammar* besteht aus zwei *dynArrays* *Terminals* und *NonTerminals*, in diesen werden alle Terminal- und Nichtterminalsymbole als String gespeichert, einem

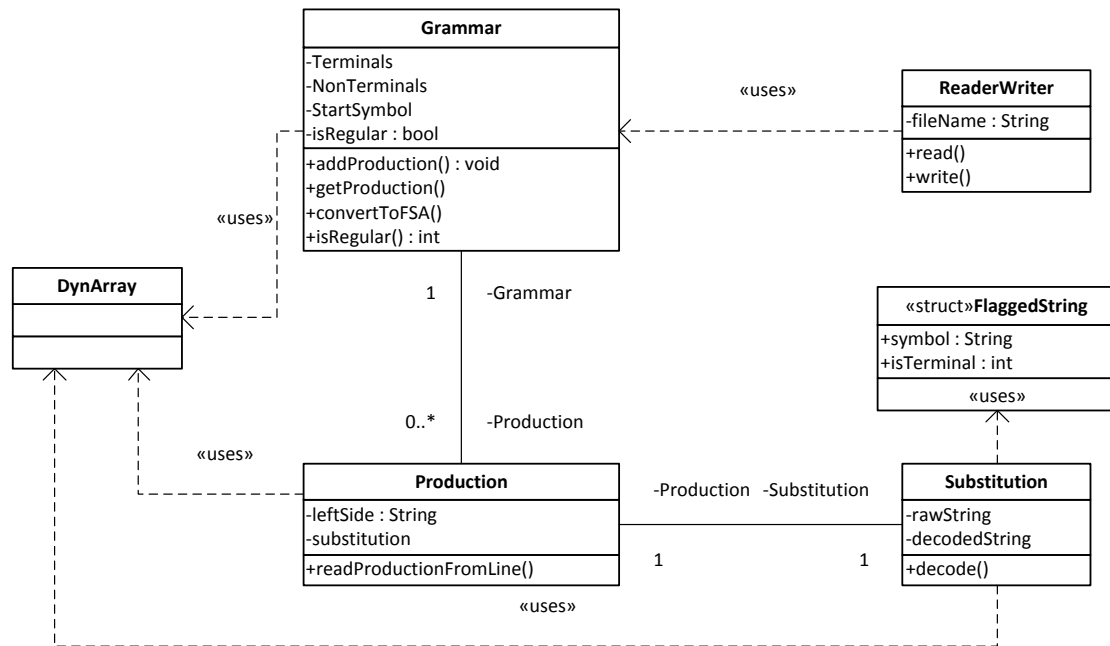


Abbildung 3.1: Reguläre Grammatik UML Diagramm

String *Startsymbol* für das Startsymbol der Grammatik, einer Integer Variablen *isRegular*, die angibt ob die Grammatik regulär ist und einem weitem *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.

3.3 Einlesen von regulären Grammatiken

3.4 Konvertierungen

3.4.1 Konvertierung zu einem endlichen Zustandsautomaten

3.5 Minimierung eines endlichen Zustandsautomaten

4 Zustandsautomat

5 Zusammenfassung

Literaturverzeichnis

- [Rega] Regular expression. http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html. Accessed: 05/08/2012.
- [Regb] Regular expression. <http://perldoc.perl.org/perlre.html>. Accessed: 05/08/2012.
- [Regc] Regular expression. http://en.wikipedia.org/wiki/Shift-reduce_parser. Accessed: 05/08/2012.

Abbildungsverzeichnis

2.1	UML Diagramm zu <i>RegularExpression</i> und <i>RETreeNode</i>	6
2.2	FSA für <i>a</i>	6
2.3	FSA für <i>a.b</i>	7
2.4	FSA für <i>a b</i>	7
2.5	FSA für <i>(a b)*</i>	7
3.1	Reguläre Grammatik UML Diagramm	9