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

Blablabla

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[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 Präzendenz nicht durch Klammerung festgelegt, so hat der jeweils am weitesten rechts stehende Operator Präzendenz.

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, Perl) 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.

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.

1 http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html

2 http://perldoc.perl.org/perlre.html#Regular-Expressions

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.

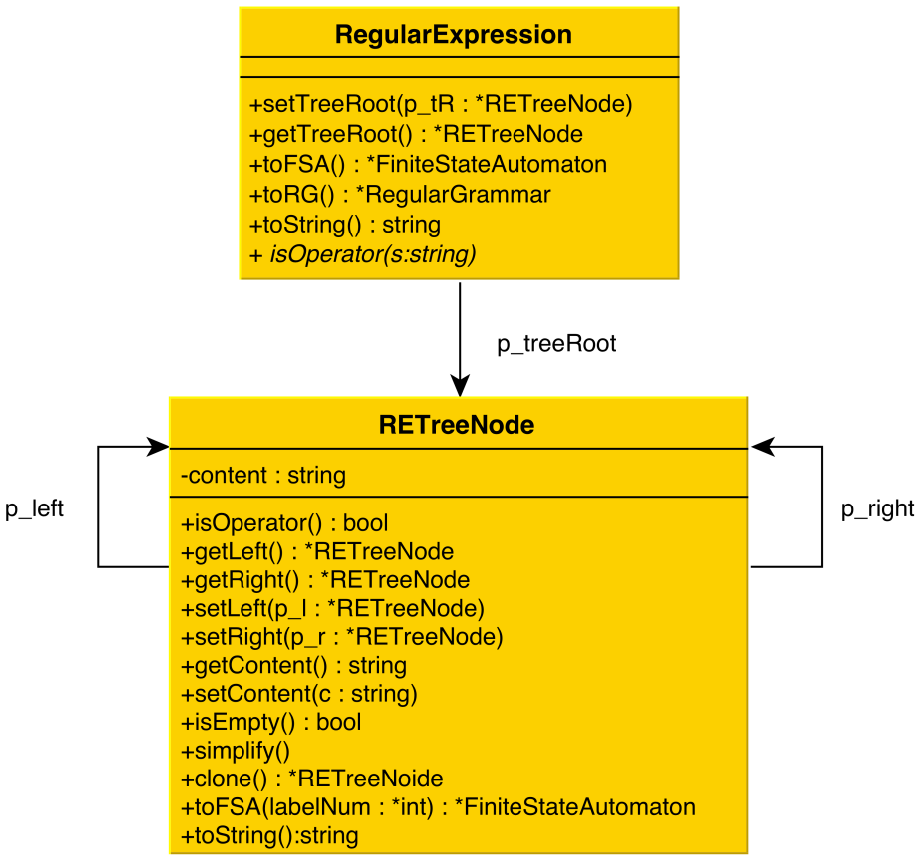

Abbildung 1: UML Diagramm zu RegularExpression und RETreeNode

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

3 http://en.wikipedia.org/wiki/Shift-reduce_parser

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 nicht-deterministischer endlicher Automat.

2.4.1.a 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.

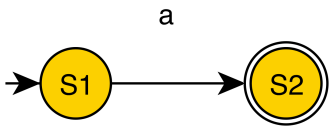

Abbildung 2: FSA für "a"

2.4.1.b 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.

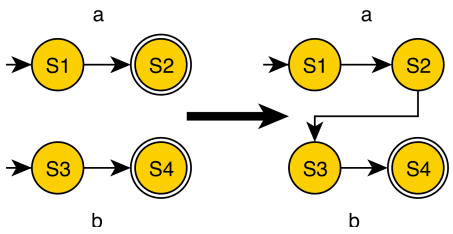

Abbildung 3: FSA für "a.b"

2.4.1.c Alternative

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

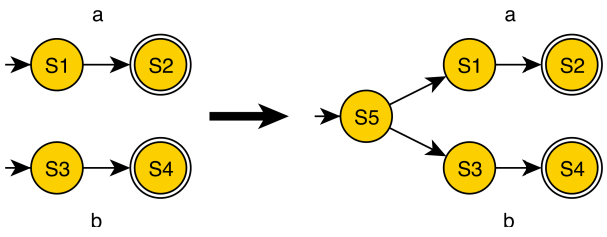

Abbildung 4: FSA für "a|b"

2.4.1.d 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.

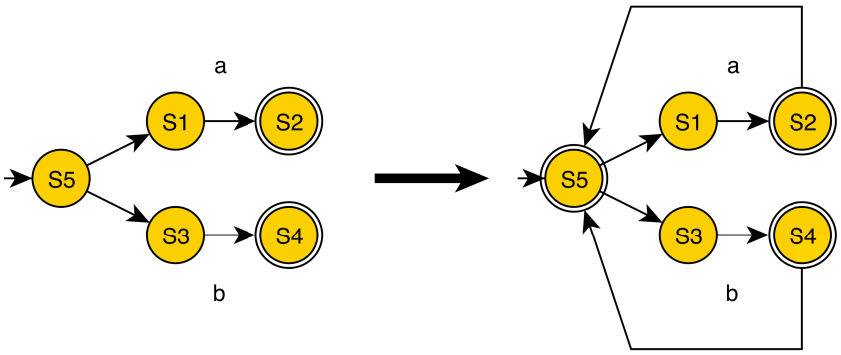

Abbildung 5: FSA für "(a|b)"*

2.4.1.e 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.

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∈N	einem Startsymbol aus der Menge der Nichtterminale
P⊆N×V*	Menge der Produktionen; (A,x)∈P, A∈N und x∈V*; statt (A,x) schreibt man A → x
V=T⋃N	heißt Vokabular, seine Elemente heißen Symbole

3.1.2 Definition einer rechtslinearen Grammatik

Eine rechtslineare Grammatik G=G(T,N,P,S) ist einer 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

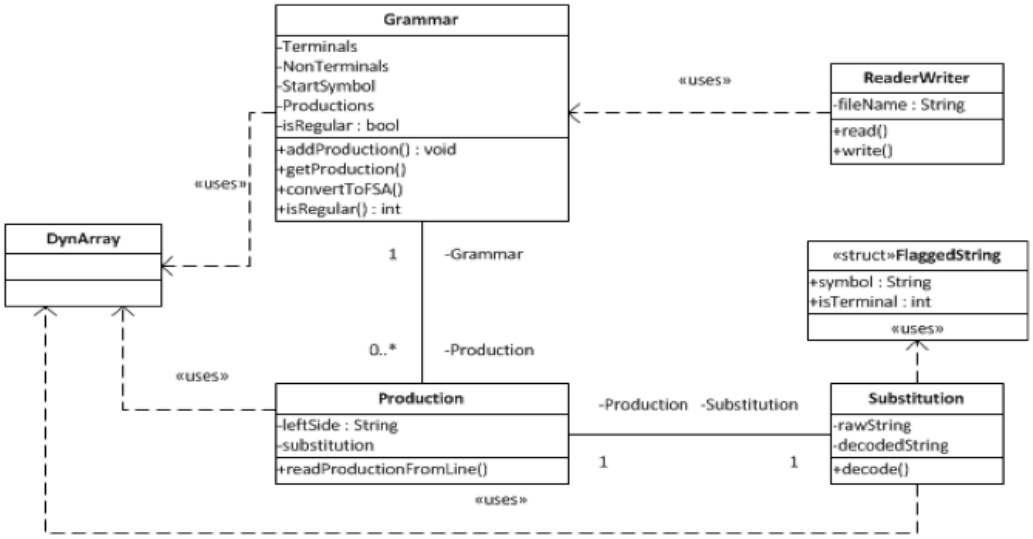

Abbildung 6: Reguläre Grammatik UML Diagramm

Die Klasse 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 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

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.

3.4 Konvertierungen

Eine Transition vom Zustand A zum Zustand B mit a als Bedingung wird in einer regulären Grammatik als A → aB gespeichert. Um Zustand B als Endzustand zu setzen gibt es mehrere Möglichkeiten, entweder durch z.B B → a, also nur ein Terminal auf der rechten Seite, oder durch ε als rechte Seite. Ist ein Zustand in einer Transition sowohl Anfang als auch Ende wird dies durch A → aA realisiert.

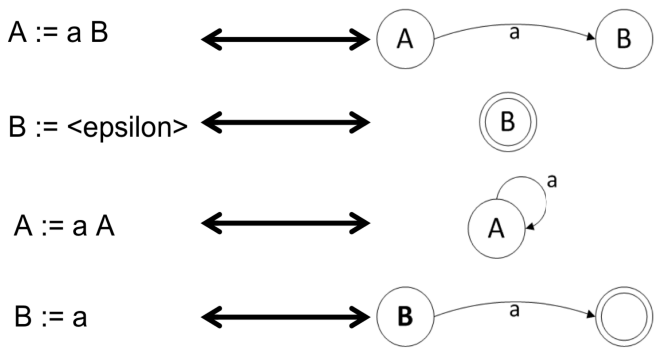

Abbildung 7: Zusammenhang zwischen FSA und RG

3.4.1 Konvertierung zu einem endlichen Zustandsautomaten

Im Rahmen dieser Projektarbeit haben wir uns nur auf die Konvertierung einer regulären Grammatik zu einem endlichen Zustandsautomaten befasst. Der Quellcode kann aber erweitert werden, so dass jede Grammatik verarbeitet werden kann.

Für die Umwandlung werden die Produktionen der Reihe nach durchgegangen.

Es gibt 4 mögliche Produktionstypen:

1. A → ε

2. A → a

3. A → aA

4. A → bA , 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 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 Terminale 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.

3.5 Minimierung eines endlichen Zustandsautomaten

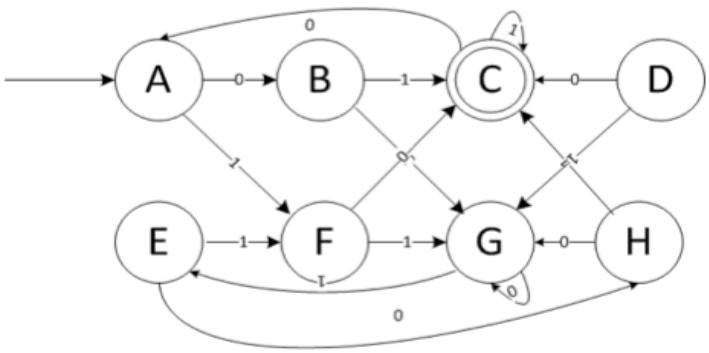

Abbildung 8: Beispielautomat

Zur Minimierung eines endlichen totalen Zustandsautomaten wurde der „Table-fillig-Algorithmus“ verwendet, abgeleitet vom Äquvalenzsatz von Myhill und Nerode.

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

Formal: Zwei Zustände P,Q sind nicht äquivalent, wenn es einen Satz ω gibt, so dass einer von δ(P,ω) oder δ(Q,ω) ein akzeptierender und der andere ein nichtakzeptierender Zustand ist.

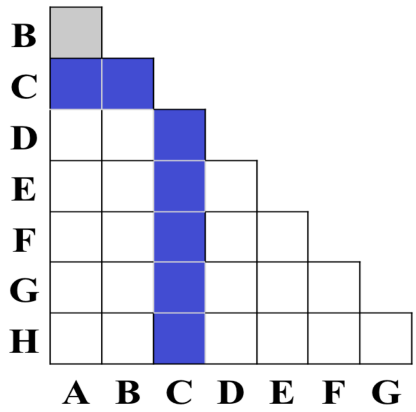

Abbildung 9: Tabelle der nichtäquivalenten Zustände

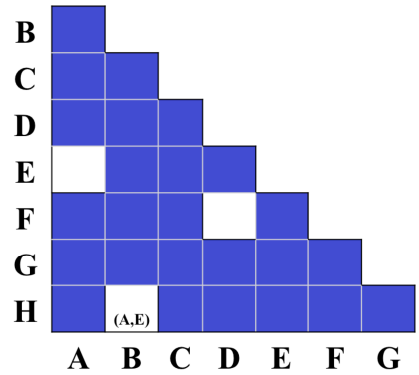

Abbildung 10: endgültige Tabelle der nichtäquivalenten Zustände

Zunächst werden alle Paare markiert, in denen der Zustand C vorkommt, denn C ist der einzig akzeptierende Zustand.

Betrachtet man nun das Paar der Zustände (A, B).

Man erhält $\delta(A,0)=B$,$\delta(B,0)=G$ und $\delta(A,1)=F$, $\delta(B,1)=C$. Und muss also überprüfen ob B und G, bzw. F und C äquivalent sind.Das Paar B und G ist nicht markier. Man kann also noch keine Aussage über Äquivalenz treffen. F und C hingegen sind nicht äquivalent, da das Paar (F,C) bereits markiert wurde. Führt man dieses Vorgehen komplett durch kommt man auf folgende Tabelle(Abbildung 10). Die Zustände (A,E), (B,H) und (D,F) sind äquivalent und können zusammengefasst werden.

4 Fazit