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

2.1 Definition

Ein endlicher Zustandsautomat kann im Allgemeinen als 5-Tupel dargestellt werden. Seine Form, $A = (Q, \Sigma, \delta, q_0, F)$, beinhaltet die Elemente:

Q endliche Menge aller Zustände

Σ Eingabealphabet

$\delta : Qx\Sigma \to Q$ Übergangsfunktion

$q_0 \in Q$ Startzustand

$F \subseteq Q$ Endzustände

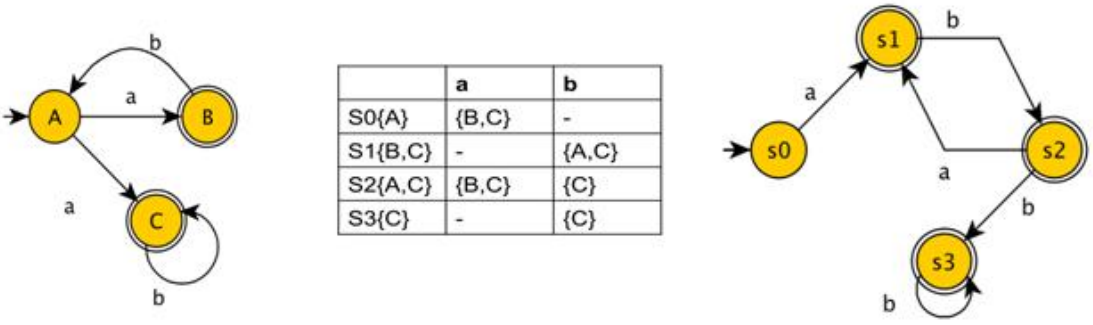
Hierbei ist die Übergangsfunktion eine Funktion, mit der man von einem Zustand der Menge Q über eine definierte Eingabe der Menge Σ zu einem weiteren Zustand der Menge Q gelangt. Der Startzustand dient als Einstiegspunkt des Automaten. Jeder Automat besitzt weiterhin mindestens einen Endzustand, auch akzeptierender Zustand genannt. Sollte durch passende Eingabe ein solcher Zustand erreicht werden, so kann die Abarbeitung des Automaten enden.

2.2 Deterministische und nichtdeterministische Zustandsautomaten

Es gibt grundsätzlich zwei Arten der endlichen Zustandsautomaten. Ist es möglich, das Ziel einer Übergangsfunktion vollständig vorherzusagen, d.h. kann durch Eingabe eines Symbols nur ein einziger Folgezustand erreicht werden, so spricht man von einem deterministischen Automaten. Im Gegensatz zu einem deterministischen Automaten können bei einem nichtdeterministischen Automaten also bei Eingabe eines Symbols mehrere Zustände erreicht werden, daher ist das Verhalten dieses Automaten nicht vorhersagbar. Da der Umgang mit nichtdeterministischen Automaten äußerst kritisch ist und jeder nicht deterministische Automat in einen äquivalenten deterministischen Automaten umgewandelt werden kann, wird im Folgenden der Algoritmus

zur Umwandlung eines nichtdeterministischen Automaten in einen deterministischen Automaten vorgestellt.

Zur Umwandlung der Automaten wurde der Algorithmus der Potenzmengenkonstruktion verwendet. Hierbei wird eine Tabelle erstellt, in der ersichtlich ist, über welche Eingaben welche Zustände erreicht werden können. Begonnen wird dabei im Startzustand des Automaten. Für jede neue Kombination aus Zielzuständen wird nun ein neuer Zustand erzeugt und für diesen die Zielzustände überprüft, bis sich keine neuen Kombinationen ergeben. Der neue Automat kann nun einfach aus der Tabelle abgelesen werden, wie in folgender Abbildung ersichtlich.



	a	b
S0{A}	{B,C}	-
S1{B,C}	-	{A,C}
S2{A,C}	{B,C}	{C}
S3{C}	-	{C}

Abbildung 2.1: Beispiel zur Potenzmengenkonstruktion

2.3 Datenstruktur

Endliche Zustandsautomaten (FSA) werden in diesem Projekt als Listen von States (Zuständen) und Transitionen (Übergangsfunktionen) aufgebaut. Eine gesonderte Reihenfolge der Elemente einer Liste ist hierbei nicht zu beachten, da die Struktur des Automaten durch die einzelnen Transitionen abgebildet wird.

2.4 Implementation

Die Klasse FiniteStateAutomaton besitzt zwei Listen bzw. Vektoren, die TransitionList und die StateList, durch welche die Abbildung eines kompletten Automaten möglich ist.

Die TransitionList setzt sich aus Elementen der Klasse Transition zusammen. Diese Elemente wiederum bestehen aus zwei Pointern, welche auf Elemente der Klasse State

zeigen, sowie dem für den Übergang zuständigen Eingabesymbol, welches als Text in der String-Variablen szEdge gespeichert ist.

Die Elemente der Klasse State, welche in der StateList gespeichert werden, enthalten eine String-Variable szName und zwei Boolean-Variablen, welche angeben, ob der State entweder ein Startzustand (bStartState) oder ein Endzustand (bFinalState) ist. Natürlich kann ein State gleichzeitig Start- als auch Endzustand sein.

2.5 Einlesen von endlichen Zustandsautomaten

Das Einlesen eines endlichen Zustandsautomaten aus einer Textdatei ist als Funktion der Klasse FiniteStateAutomaton realisiert, selbiges gilt für das Schreiben in eine Textdatei. Beim Aufruf der Funktionen *write* und *read* ist jeweils der Name der Textdatei, aus der gelesen oder in die geschrieben werden soll, in Stringform zu übergeben.

Das Einlesen und Schreiben erfolgt Zeilenweise und ist so strukturiert, dass die Objekte durch ihre Entsprechenden Tags gekapselt sind. Beispielhaft ist diese Darstellung im Folgenden für die States dargestellt:

<States>
State1
State2
</States>

Diese Art der Darstellung ermöglicht einen inhomogenen Aufbau der Textdatei, so dass es mehrere Bereiche geben kann, an denen States aufgeführt sind und sie trotzdem auch alle als solche erkannt werden.

Die Textdateien sind so aufgebaut, dass zuerst alle States, dann der StartState und zum Schluss alle Transitionen ausgegeben werden.

2.6 Konvertierung

Die Konvertierungen zwischen regulärem Ausdruck und endlichem Zustandsautomat, sowie regulärer Grammatik und endlichem Zustandsautomaten, sind direkt möglich. Bei der Konvertierung zwischen regulärem Ausdruck und regulärer Grammatik muss man einen Umweg über den endlichen Zustandsautomaten in Kauf nehmen.

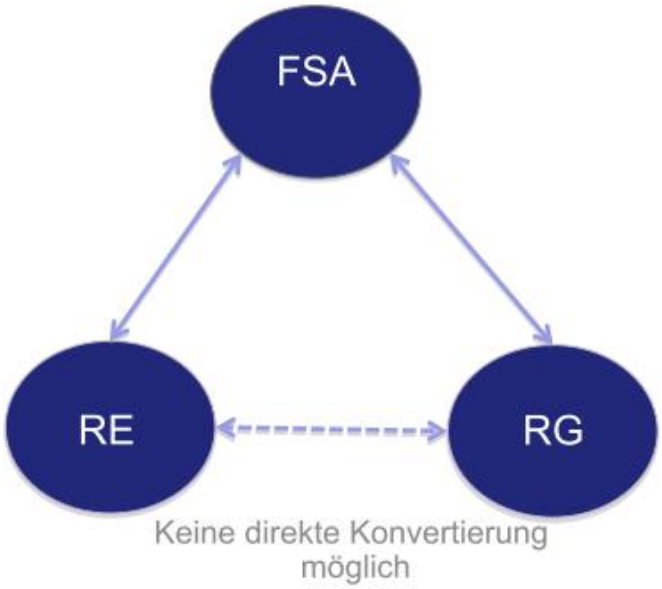


Abbildung 2.2: Konvertierungen

2.6.1 Konvertierung zu Regulärem Ausdruck

Es gibt zahlreiche Methoden um Endliche Automaten zu Regulären Ausdrücken zu konvertieren, welche unterschiedliche Ergebnisse liefern. Die Konvertierung von Endlichem Automaten zu Regulärem Ausdruck ist nicht eindeutig.

In diesem Projekt wurde die Algebraische Methode nach Brzozowski implementiert.

2.6.1.1 Algebraische Methode

Bei der algebraischen Methode wird der Automat als Gleichungssystem aus Regulären Ausdrücken betrachtet.

Für alle ausgehenden Übergänge eines Zustands Q_i wird eine Gleichung aufgestellt. Sei a das Eingabesymbol des Übergangs und Q_j der Zielzustand der Transition. Dann lautet der Reguläre Ausdruck für diese Transition $a.Q_j$.

Für jeden Zustand Q_i werden dann die Regulären Ausdrücke der ausgehenden Transitionen mit einer Oder-Verknüpfung zusammengefasst, z.B. $Q_0 = a.Q_1|b.Q_2|c.Q_3$.

Es ergibt sich ein Gleichungssystem aus Regulären Ausdrücken für jeden Zustand.

Der Reguläre Ausdruck, welcher die Sprache des Endlichen Automaten beschreibt, ergibt sich durch Lösung des Gleichungssystem nach dem Startzustand.

2.6.1.2 Lösungsreihenfolge nach Brzozowski

Je nach Lösungsreihenfolge des Gleichungssystem ergibt sich ein unterschiedlicher Regulärer Ausdruck.

Die algebraische Methode nach Brzozowski beschreibt eine solche Lösungsreihenfolge. Der folgende Pseudocode beschreibt die Lösung des Gleichungssystems nach Brzozwski.

Listing 2.1: Pseudocode für die algebraische Methode nach Brzozowski.[1]

```
Sei m die Anzahl der Zustände.
Sei Q der Vektor der Größe m der alle Zustände enthält.
Sei Q[0] der Startzustand.
Sei Σ das Eingabealphabet.
Sei B ein Vektor der Größe m. In ihm werden die Regulären Ausdrücke
  für jeden Zustand gespeichert
Sei A eine Matrix der Dimension m x m. In ihr werden alle Transitionen
  gespeichert.
Sei e der Reguläre Ausdruck, welcher die Sprache des Automaten beschreibt.

Für i = 0 bis m
  Wenn Q[i] Endzustand ist
    B[i] := ϵ
  Sonst
    B[i] := NULL

  Für j = 0 bis m
    Für jedes Eingabesymbol a aus Σ
      Wenn ein Übergang von Q[0] nach Q[1] mit a existiert
        a[i,j] := a[i,j] | a

Für n = m–1 bis 0
  B[n] = A[n,n]* . B[n]
  Für j = 1 bis n
    A[n,j] := A[n,n]* . A[n,j];
  Für i = 1 bis n
    B[i] := B[i] | A[i,n] . B[n]
    Für j = 1 bis n
      A[i,j] := A[i,j] | A[i,n] . A[n,j]

e = B[0]
```

2.7 Minimierung nach dem Moore-Algorithmus

Bei der Minimalisierung nach dem Moore Algorithmus werden zuerst alle States in accepting und rejecting States unterteilt. Accepting sind all diejenigen, die ein Endzustand sind und rejecting alle anderen. Von nun an werden diese beiden Bereiche getrennt behandelt.

Das Vorgehen ist nun wie folgt. Man stelle eine Tabelle auf, die auf der Vertikalen alle möglichen Eingaben enthält. Dann trage man auf der Horizontalen zwei Gruppen auf. Die eine Gruppe enthält die Namen aller rejecting States (im folgenden Beispiel G_0 der ersten Zeile), die andere enthält die Namen aller accepting States (im folgenden Beispiel G_1 der ersten Zeile). Nun fülle man die Elemente der Tabelle, indem man zu jeder Eingabe und jedem Statename angibt, welche Gruppe das Resultat der Transition aus gegebenem State und Eingabe wäre, ähnlich der Vorgehensweise in der Potenzmengenkonstruktion.

Sind alle Elemente ausgefüllt, so beginnt man neue Gruppen zu erstellen. Dafür wird wie folgt vorgegangen: Man vergleicht die Elemente, die unter einem State aufgelistet sind, mit den Elementen, die unter den Nachbarstates der gleichen Ausgangsgruppe aufgelistet sind. Haben alle States einer Ausgangsgruppe für die jeweilige Eingabe die gleiche Folgegruppe, so ist die Ausgangsgruppe minimal. Gibt es jedoch einen State, der für die gleiche Eingabe eine andere Folgegruppe hat, so wird eine neue Gruppe erstellt und der State in diese Gruppe verschoben.

Dieser Ablauf wiederholt sich so lange, bis alle Gruppen minimal sind. Die minimalen Gruppen entsprechen abschließend dann den States des minimalisierten endlichen Automaten. Das folgende Beispiel beschreibt diesen Vorgang nochmal beispielhaft:

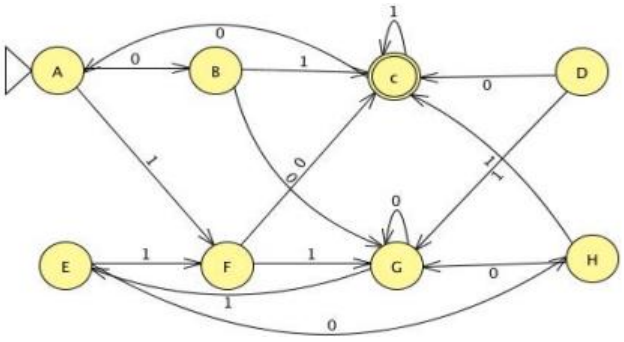


Abbildung 2.3: Nicht minimierter Automat

	Rejecting states							Accepting states
	G_0{A	B	D	E	F	G	H}	G_1{C}
0	G_0	G_0	G_0	G_0	G_1	G_0	G_0	G_0
1	G_0	G_1	G_0	G_0	G_0	G_0	G_1	G_1

	G_0{A	D	E	G}	G_1{B	H}	G_2{F}	G_3{C}
0	G_1	G_0	G_1	G_0	G_0	G_0	G_3	G_0
1	G_2	G_0	G_2	G_0	G_3	G_3	G_0	G_3

Abbildung 2.4: Tabelle zum Minimierungsvorgang

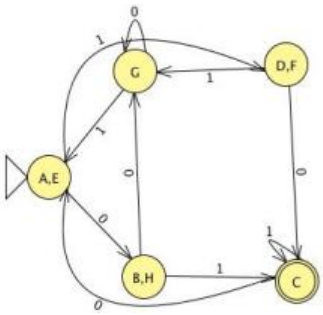


Abbildung 2.5: Resultierender minimierter Automat

2.8 Minimierung mittels des Table-Filling-Algorithmus

2.8.1 Der Algorithmus

Der Table-Filling Algorithmus[2, 3, 4] ist abgeleitet vom Äquivalenzsatz von Myhill und Nerode.

Es wird eine Matrix benötigt, in der in den Zeilen und Spalten alle Zustände des Automaten vorhanden sind.

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 ein Zustand vom Zustandspaar (X,Y) ein akzeptierender Zustand und der Andere ein nichtakzeptierender Zustand ist.

Falls für den Paar (P,Q) nicht entschieden werden kann ob die zwei Zustände äquivalent sind oder nicht, werden die Zustände zu die Abhängigkeiten (dependencies) der Nachfolgenden Zustände (X,Y) hinzugefügt. (P,Q) werden 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-akzeptierenden Zuständen als nicht-äquvalent.
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,
                füge (P,Q) zu den dependencies von (X,Y) hinzu,
        Sonst markiere (P,Q), und markiere alle dependencies von (P,Q).
```

Listing 2.2: Table-Filling-Algorithmus : Pseudocode

2.8.2 Implementierung

[5, 6] Um diesen Algorithmus zu impelementieren, wurde eine neue Klasse *TableFilling-Minimizer* erstellt. Diese Klasse bietet die benötigten Methoden um einen endlichen Zustandsautomat zu minimieren.

Die verwendete Matrix in diesem Algorithmus ist eine Matrix von *StatesPair*, wobei jedes *StatesPair* eine Struktur ist in der gespeichert wird ob ein Paar nicht-äquivalent ist, und eine Liste mit den Abhängigkeiten dieses Paares.

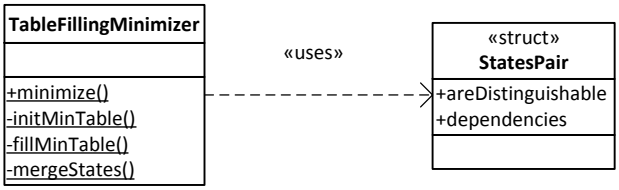


Abbildung 2.6: UML Diagramm zu *TableFillingMinimizer*

2.9 Äquivalenzprüfung

Eine der Anforderungen an das Projekt war es, eine Methode zum Überprüfen der Äquivalenz zwischen verschiedenen Darstellungen einer Regulären Sprache zu finden.

Generell kann jede Darstellungsform in einen Endlichen Automaten überführt werden. Somit war es nur notwendig eine Methode zu implementieren, welche die Äquivalenz zweier endlicher Automaten überprüft.

Hierzu werden die beiden zu vergleichenden Automaten zunächst minimiert.

Laut Definition müssen minimale Automaten, welche die gleiche Sprache darstellen, die gleiche Anzahl an Zuständen haben.

Falls dies gegeben wird, dann sind die beiden endlichen Automaten äquivalent, wenn die beiden Startzustände äquivalent sind.

Äquivalenz von Zuständen ist dabei wie folgt definiert:

Seien A und B Zustände.

Ist A Endzustand und B nicht (oder umgekehrt), so sind A und B nicht äquivalent.

Ansonsten sind A und B genau dann äquivalent, wenn für jedes Eingabesymbol e aus dem Eingabealphabet Σ, mit $\delta(A, e, P)$ und $\delta(B, e, Q)$, gilt, dass P und Q äquivalent sind.

In der gewählten Implementation wird diese Definition rekursiv von den Starzuständen aus angewandt und so die Äquivalenz der Automaten festgestellt.

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[7], Perl[8]) 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[7], Perl[8]) 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.

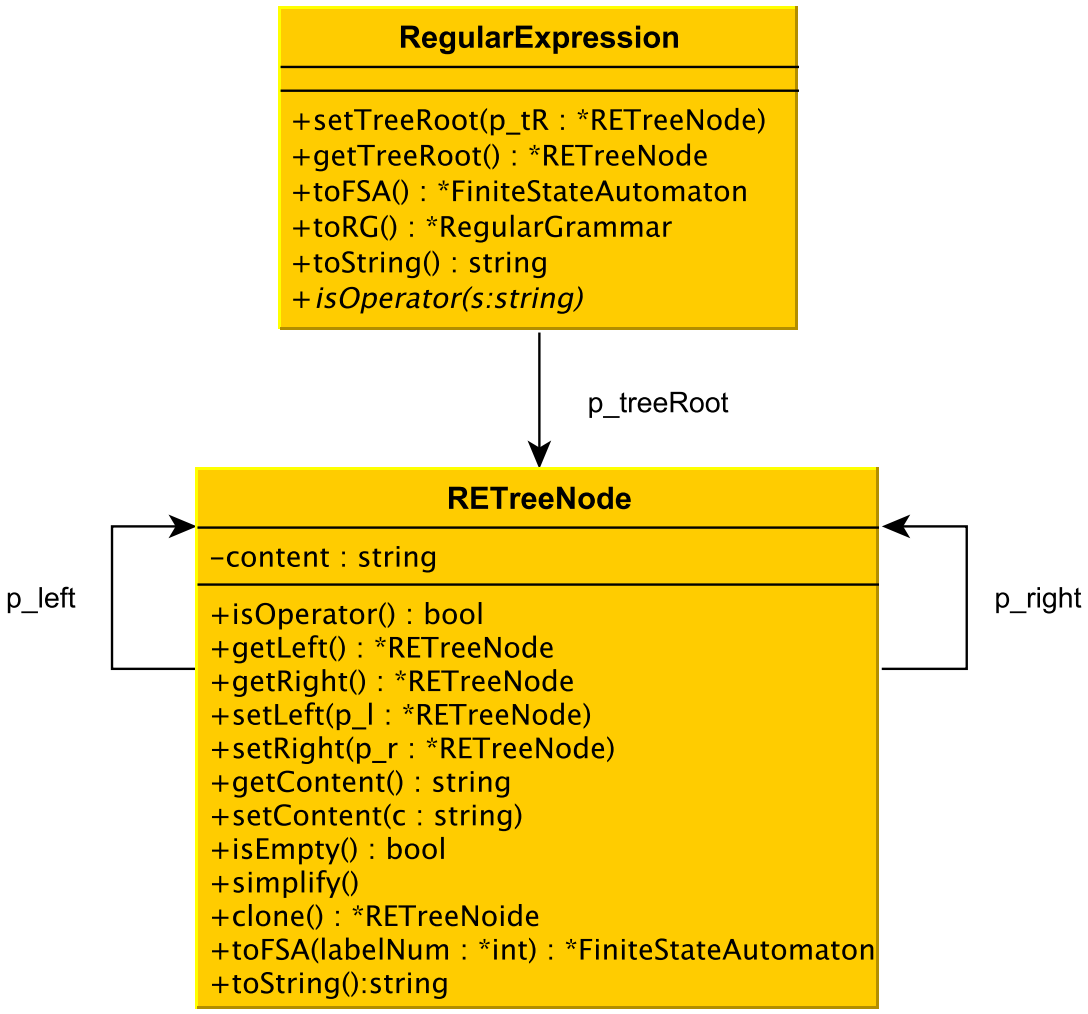


Abbildung 3.1: UML Diagramm zu *RegularExpression* und *RETreeNode*

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[9], 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.

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.

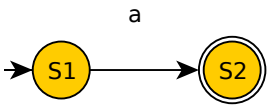


Abbildung 3.2: FSA für a

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.

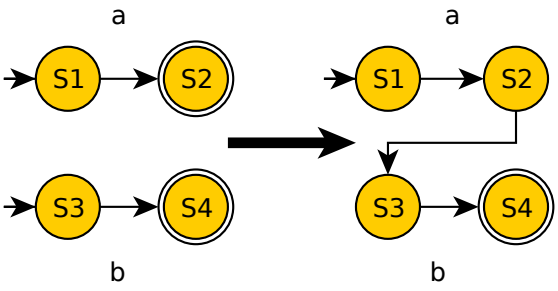


Abbildung 3.3: FSA für $a.b$

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.

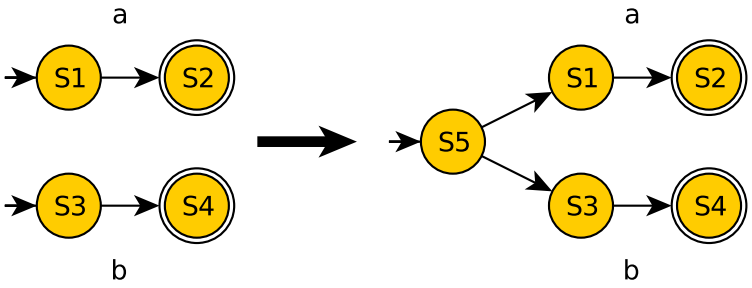


Abbildung 3.4: FSA für $a|b$

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.

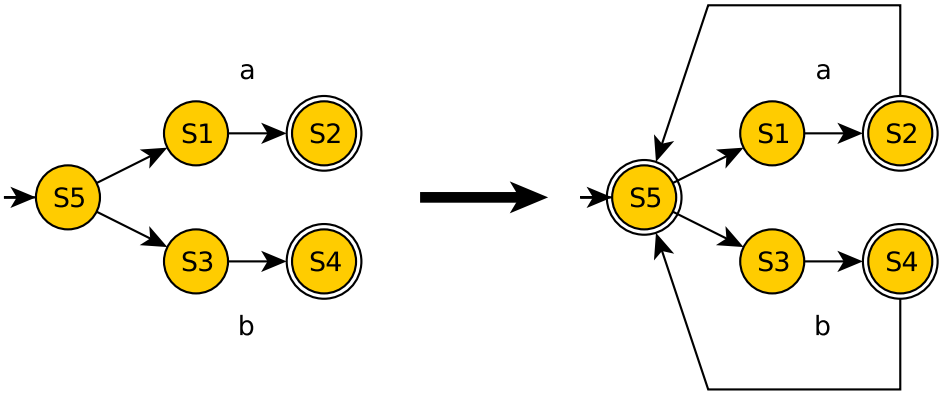


Abbildung 3.5: FSA für $(a|b)*$

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 sooft 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 ist.

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 $a \in T$

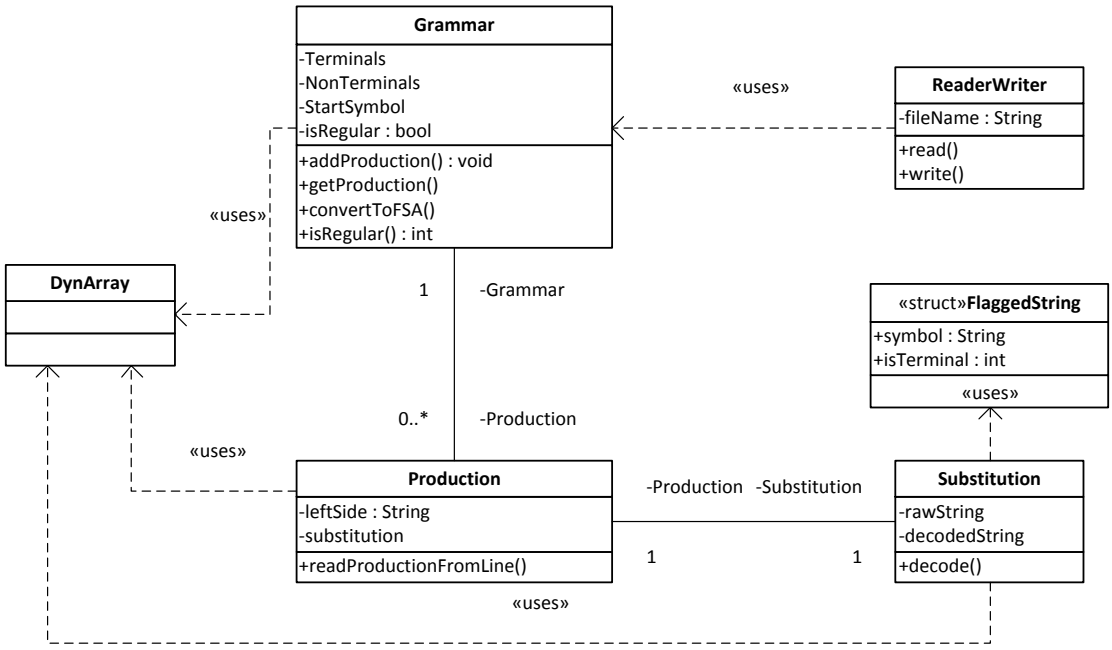


Abbildung 4.1: Reguläre Grammatik UML Diagramm

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. Des weiteren besteht ein *Grammar*-Objekt aus 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 einer Substitution auf der rechten Seite des Pfeils. Die Klasse *Production* setzt sich aus einer Stringvariablen für die linke Seite und einem Objekt der Klasse *Substitution* für die rechte Seite zusammen.

Die Klasse *Substitution* enthält eine Zeichenkette *rawString*. In dieser wird eine unbearbeitete 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.

Die 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 einen *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 darstellt.

4.4 Konvertierungen von regulären Grammatiken

Konvertierungen von einer regulären Grammatik zu einem endlichen Zustandsautomaten oder einem regulären Ausdruck sind nur möglich, wenn die vorliegende Grammatik regulär ist. Im Rahmen dieser Projektarbeit wurde eine rechtslinieare Grammatik 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.

1.Fall:

B $\rightarrow$ a $\quad \Longrightarrow \quad$ B ist ein Endzustand
B $\rightarrow$ a B

Dieser Fall ist aber nicht genau so implementiert: Wird eine Produktion gefunden, wo die rechte Seite nur ein Terminal ist, dann wird eine neuer Endzustand erzeugt (leztes Beispiel in Abbildung 4.2). Diese Entscheidung wurde getroffen, damit nur eine Produktion pro Bearbeitung betrachtet werden muss. Die zusätzlich erzeugten Zustände werden später entfernt durch die Konvertierung des Automatens zu einem deterministischen, endlichen Zustandsautomaten.

2.Fall:

B $\rightarrow \varepsilon \quad \Longrightarrow \quad$ B ist ein Endzustand

Ist ein Zustand in einer Transition sowohl Anfang als auch Ende wird dies durch A $\rightarrow$ a A realisiert: eine Schleife.

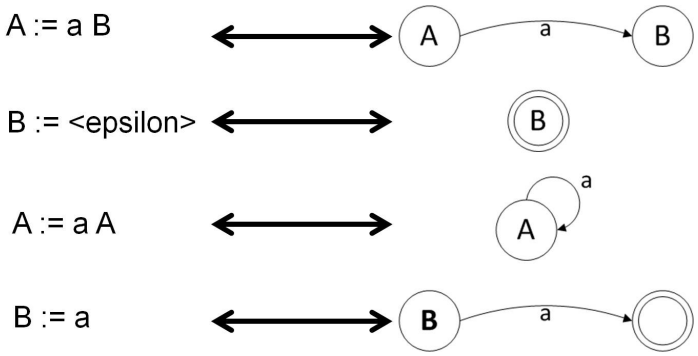


Abbildung 4.2: Zusammenhang zwischen RG und FSA

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

1. A $\rightarrow \varepsilon$
2. A $\rightarrow$ a
3. A $\rightarrow$ a A
4. A $\rightarrow$ 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 einer Grammatik zu einem endlichen Zustandsautomaten und schließlich zu einem regulären Ausdruck realisiert.

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

Die Grammatik in einer Kompaktform schreiben

Alle Produktionen mit gleicher linker Seite werden zusammengefasst. z.B:

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

Die Produktionen als gleichungen interpretieren

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

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

Das Gleichungssystem lösen

Gesucht ist eine Gleichung die nur das Startsymbol als Variable enthält.

Die Distributivität und Assoziativität sind hier zwei verwendbare Eigenschaften der gewöhlichen "+" Operation. Zusätzlich, können Produktionen der Form A = c A + B in die Form A = c* B umgewandelt werden.

5 Fazit

Es wurde eine vollständig funktionsfähige C++-Bibliothek zur Bereistellung von Funktionen, welche zur Arbeit mit regulären Sprache dient, erstellt. Alle damit einhergehenden Forderungen wurden erfüllt. Zusätzlich wurden statt einem zwei verschiedene Minimierungsalgorithmen implementiert.

Literaturverzeichnis

[1] jmad, "How to convert finite automata to regular expressions - Brzozowski algebraic method." http://cs.stackexchange.com/a/2392/1997. Accessed: 05/08/2012.

[2] S. Chawla, "Lecture 6: State minimization in finite automata." http://pages.cs.wisc.edu/~shuchi/courses/520-S08/handouts/Lec6.pdf. Accessed: 05/07/2012.

[3] S. Chawla, "Lecture 7: State minimization in finite automata." http://pages.cs.wisc.edu/~shuchi/courses/520-S08/handouts/Lec7.pdf. Accessed: 05/07/2012.

[4] P. A. Schulz, "Table-filling algorithmus." http://wwwmath.uni-muenster.de/logik/Personen/Schulz/BETHWIKI/index.php/Table-Filling_Algorithmus. Accessed: 05/07/2012.

[5] J. Southern, "Equivalence and minimization of deterministic finite automata." http://adammikeal.org/courses/compute/presentations/DFA_minimization.ppt. Accessed: 06/08/2012.

[6] D. Forster, "Introduction to the theory of computation." http://www.cse.yorku.ca/course_archive/2001-02/S/2001B/lectures/gh.pdf. Accessed: 06/08/2012.

[7] 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.

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

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

Abbildungsverzeichnis

Listings