1. Einführung

1.1. Motivation

Das Konzept der regulären Ausdrücke[1] in Verbindung mit endlichen Automaten wurde ursprünglich im Hinblick auf Neuronen-Netze und Schaltkreise entwickelt[2]. Mit der Zeit hat es sich aber als nützliches Werkzeug zur Entwicklung von lexikalischen Analysern erwiesen.[3] Eine Vielzahl anderer Anwendungsmöglichkeiten für reguläre Ausdrücke gibt es bei Text-Editoren, Muster-Erkennern sowie Textverarbeitungs- und Such-Programmen.[4]
Mit dem vorliegenden Programm hat man nun eine Anwendung zur Verfügung, mit der man auf einfache Art und Weise zwei gegebene reguläre Ausdrücke auf Äquivalenz überprüfen kann.
Implementiert wurde die Anwendung in der Programmiersprache JAVA, benutzt wurde hierbei der Compiler JDK 1.1.6 unter dem Betriebssystem Solaris 2.6 .
Für die schriftliche Ausarbeitung wurde das Textverarbeitungsprogramm Microsoft Word 97 verwendet. Die Graphiken wurden mit xv 3.1 erstellt und mittels Microsoft Photo Editor 3.0 in das Dokument eingebunden.

1.2. Grundlegende Begriffe

Zum leichteren Verständnis der nachfolgenden Ausführungen werden in diesem Abschnitt zunächst einige grundlegende Begriffe kurz eingeführt. Hierzu werden die

[1] Vgl. Kleene 1956. Die Grundidee der regulären Ausdrücke geht auf einen Entwurf von Kleene zurück. Dieser führte 1956 die Konzeption der regulären Ausdrücke ein.

[2] Vgl. ebenda. Kleene simulierte neuronale Netze mittels endlicher Automaten und bewies die Äquivalenz beider Konzepte.

[3] Vgl. Hopcroft und Ullman 1988.

[4] Vgl. ebenda.

Ausführungen von Blum (1998), Hopcroft und Ullman (1988) und Stetter (1988) verwendet. Eine umfangreiche Einführung in dieses Thema bieten auch Aho, Sethi und Ullman (1988).

- Ein *Symbol* ist eine abstrakte Einheit, die nicht weiter zerteilt werden kann. Die im hier betrachteten Programm verwendeten *Symbole* sind die Buchstaben a,..,z,A,..,Z.
- Der Begriff *Alphabet* bezeichnet eine beliebige endliche nichtleere Menge von *Symbolen.*
- Ein *Wort* über einem *Alphabet* Σ ist eine endliche Folge von *Symbolen* aus diesem *Alphabet*, die ohne Zwischenraum hintereinander geschrieben werden. In der Sprachentheorie werden die Begriffe *Satz* und *String* oft synonym für *Wort* verwendet.
- Der Begriff *Sprache* bezeichnet eine beliebige Menge von *Worten* über einem gegebenen *Alphabet* Σ. Diese Definition ist sehr allgemein. Danach sind abstrakte *Sprachen* wie die leere Menge ϕ oder die Menge, die nur das leere Wort {ε} enthält, ebenso *Sprachen* wie die Menge aller syntaktisch richtigen JAVA-Programme. Auf *Sprachen* sind eine Reihe wichtiger Operationen anwendbar. Hinsichtlich des hier zu besprechenden Programms interessieren besonders die Vereinigung zweier *Sprachen* L und M geschrieben als L&M, die Konkatenation zweier *Sprachen* L und M geschrieben als LM und der Kleene´sche Stern-Operator einer *Sprache* L geschrieben als L*.
- Ein *regulärer Audruck* ist eine Notation, mit der die im obigen Abschnitt aufgeführten Mengen von *Worten* exakt definiert werden können. Ein *regulärer Audruck* über einem gegebenen *Alphabet* Σ und die von ihm beschriebenen Mengen werden wie folgt rekursiv definiert:

  a.) ϕ ist ein *regulärer Ausdruck* und bezeichnet die leere Menge.

  b.) ε ist ein *regulärer Ausdruck* und bezeichnet die Menge, die nur das leere Wort {ε} enthält.

  c.) Für jedes a aus dem *Alphabet* Σ ist a ein *regulärer Ausdruck* und bezeichnet die Menge {a}.

  d.) Wenn r und s *reguläre Ausdrücke* sind, die die *Sprachen* R und S bezeichnen, so sind (r|s), (rs) und (r*) ebenfalls *reguläre Ausdrücke* und bezeichnen die Mengen R&S, RS und R*.

- Die von einem *regulären Ausdruck* bezeichnete *Sprache* heißt *reguläre Menge* bzw. *reguläre Sprache*.
- Ein *endlicher Automat* (EA) ist ein mathematisches Modell eines Systems mit diskreten Ein- und Ausgaben. Dieses System besteht aus einer endlichen Anzahl von Zuständen und einer Menge von Transitionen, die auf einem Eingabe-*Symbol* aus einem *Alphabet* Σ arbeiten und einen Zustand in einen anderen überführen. Jeder Zustand dieses Systems umfaßt also die Informationen, die sich aus den bisherigen Eingabe-*Symbolen* ergeben haben und die benötigt werden, um die Reaktionen des Systems auf noch folgende Eingaben zu bestimmen.
- Ein *deteministischer endlicher Automat* (DEA) liegt vor, wenn für jeden Zustand für ein bestimmtes Eingabe-*Symbol* immer nur ein Übergang zu einem anderen Zustand existiert.

1.3. Grundlegende Vorgehensweise

Man nennt zwei *reguläre Ausdrücke* r und s äquivalent, in Zeichen r = s, wenn sie die gleiche *Sprache* bezeichnen. Für *reguläre Ausdrücke* gelten eine Reihe algebraischer Gesetze, die zur Umwandlung *regulärer Ausdrücke* in äquivalente Formen benutzt werden können. Allerdings ist es sehr schwierig, die Äquivalenz zweier beliebiger *regulärer Ausdrücke* mit Hilfe solcher Äquivalenzregeln nachzuweisen[5].

Eine wesentlich einfachere Methode liefern die sogenannten Ableitungen von *regulären Ausdrücken*. Mit Hilfe bestimmter Ableitungsregeln läßt sich so zu jedem *regulären Ausdruck* genau ein *regulärer Ausdruck* finden, der zu diesem die Ableitung bildet[6].

In dem hier zu besprechenden Programm wurde eine weitere Möglichkeit benutzt, um zwei *reguläre Ausdrücke* auf Äquivalenz zu vergleichen. Hier macht man sich die Tatsache zunutze, daß die von *endlichen Automaten* akzeptierten *Sprachen* genau die *Sprachen* sind, die durch *reguläre Ausdrücke* spezifiziert werden können. Demnach sind auch die *Sprachen* der *endlichen Automaten reguläre Mengen* bzw. *reguläre*

[5] Vgl. Stetter 1988.

[6] Vgl. ebenda.

Sprachen[7]. Somit kann man zu jedem *regulären Ausdruck* einen äquivalenten *deterministischen endlichen Automat* konstruieren, der dessen Sprache akzeptiert[8]. Der Vergleich zweier *deterministischer endlicher Automaten* auf Äquivalenz ist viel einfacher als der zweier *regulärer Ausdrücke.* Zwei *deterministische endliche Automaten* R und S heißen äquivalent, in Zeichen R ≅ S, wenn die von ihnen erzeugten *Wortfunktionen* gleich sind. Die *Wortfunktion* eines *DEA* kennzeichnet die Beziehung zwischen Eingabe und Ausgabe[9].

Kann nun ein *DEA* für einen bestimmten Zustand bei keiner Eingabe in diesen Zustand übergehen, so trägt dieser Zustand nichts zu der vom *DEA* erzeugten *Wortfunktion* bei. Folglich kann man diesen überflüssigen Zustand entfernen, ohne daß sich am Verhalten des *DEA* etwas ändert. Ein *DEA* ohne überflüssige Zustände heißt vereinfachter DEA. Die Konstruktion eines zu einem vorgegebenen *DEA* äquivalenten *DEA*, der weniger Zustände hat, wird als *Reduktion* bezeichnet. Ist eine *Reduktion* eines *DEA* nicht möglich, so heißt dieser *reduziert*[10].

Im Programm wird somit für jeden der beiden zu vergleichenden *regulären Ausdrücke* r und s der zugehörige *deterministische endliche Automat* konstruiert und reduziert. Sind dann die beiden reduzierten DEA´s gleich, so kann man daraus schließen, daß die ihnen zugrunde liegenden *regulären Ausdrücke* r und s äquivalent sind.

2. Benutzerhandbuch

Das vorliegende Programm erlaubt den Vergleich zweier *regulärer Ausdrücke* auf Äquivalenz.

[7] Vgl. Hopcroft und Ullman 1988.

[8] Vgl. ebenda. Hopcroft und Ullman beweisen diese Äquivalenz mittels vollständiger Induktion über die Größe des regulären Ausdrucks, d.h. die Anzahl der in ihm enthaltenen Operatoren.

[9] Vgl. Stetter 1988.

[10] Vgl. ebenda.

2.1. Das Programm starten

Um das Programm zu starten muß in der Kommandozeile lediglich das Schlüsselwort „java“ und der Name der Anwendung „RegExpr“ angegeben werden:

java RegExpr

2.2. Die Erlaubten Zeichen

- Die bei der Eingabe erlaubten *Symbole* sind die Buchstaben a,..,z.
- Für die Oder-Verknüpfung wird das Operatorzeichen ‘|‘ verwendet.[11] Soll beispielsweise klarwerden, daß entweder das *Wort* a oder das *Wort* b in den *regulären Ausdruck* einfließen soll, dann wird dies durch den *regulären Ausdruck* (a | b) dargestellt.
- Die Konkatenationsoperation verwendet kein Operationszeichen. Wenn zum Beispiel a und b *Worte* sind, dann ist das Ergebnis der Konkatenation von a und b das *Wort,* welches sich durch Anhängen des *Wortes* b an das *Wort* a ergibt.
- Der Kleene`sche Stern-Operator, ein Postfix-Operator mit nur einem Operanden, wird durch das Zeichen ‘*‘ repräsentiert. Der Stern-Operator angewendet auf das *Wort* a - geschrieben als a* - beschreibt die Menge der *Worte*, die aus null oder beliebig vielen a`s bestehen (ε,a,aa,aaa,...).

2.3. Beschreibung der Oberfläche

LabelRegExpr1

InputRegExpr1

LabelRegExpr2

InputRegExpr2

[11] Vgl. Aho, Sethi und Ullman, 1988. In der Literatur wird für die Oder-Verknüpfung meist das Operatorzeichen ‘+‘ verwendet. In Anlehnung an Aho et al., wovon der Algorithmus zum Vergleich zweier *regulärer Ausdrücke* auf Äquivalenz stammt, wird hier aber das Zeichen ‘|‘ verwendet.

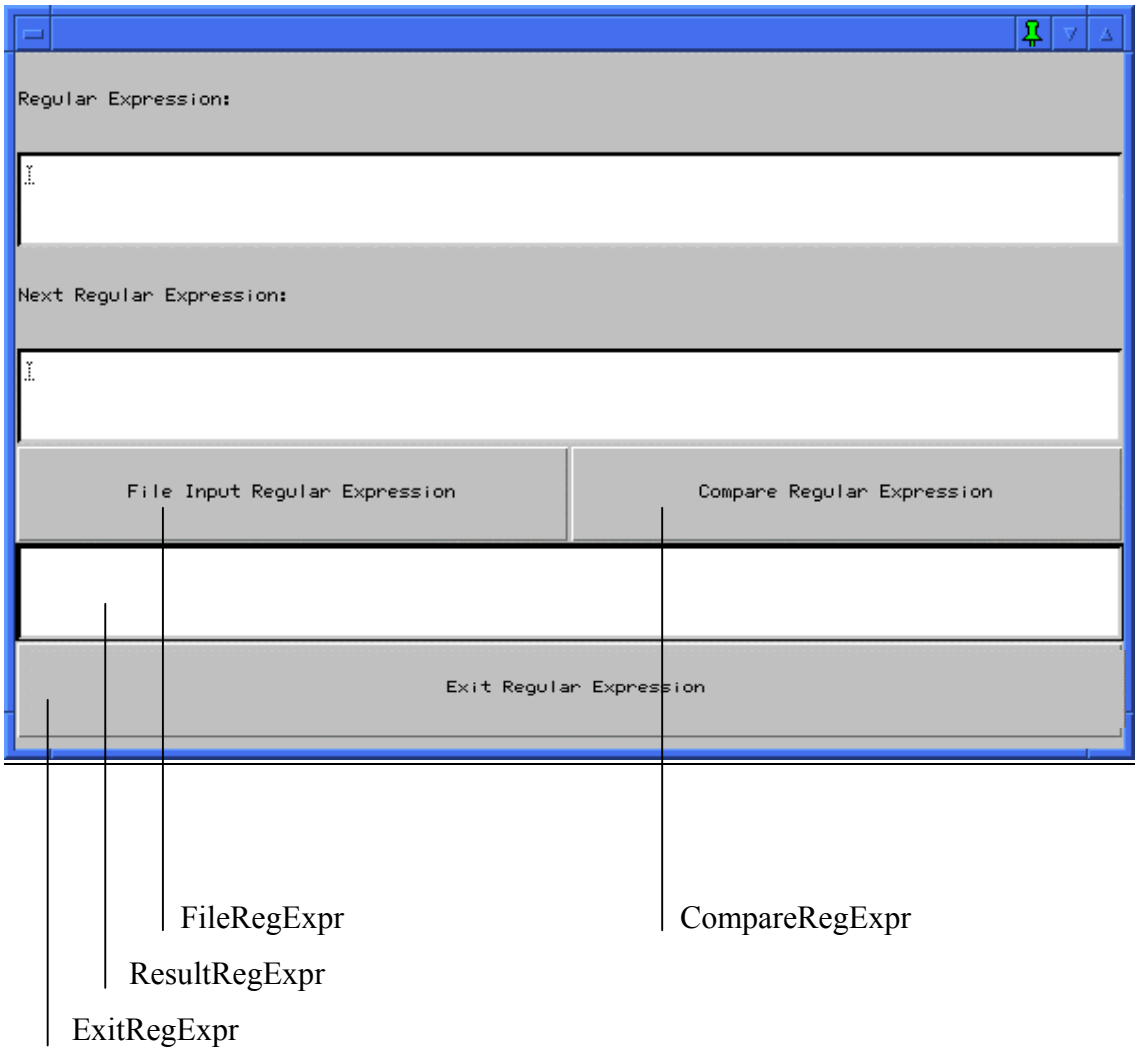


Abbildung2.1: Benutzeroberfläche der Anwendung.

2.4. Beschreibung des Dialogfensters

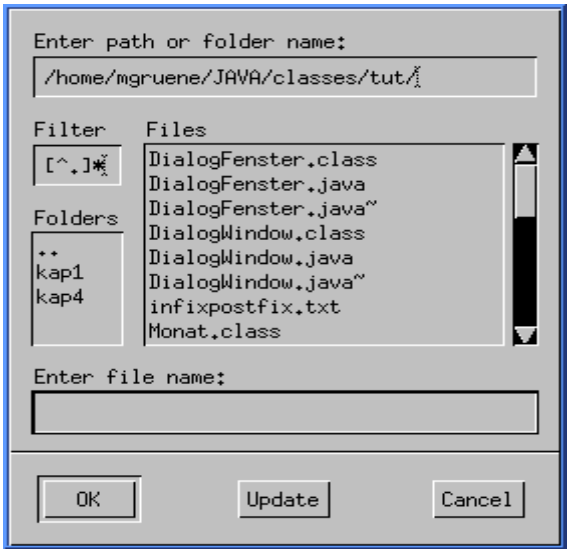


Abbildung2.2: Dialogfenster der Anwendung.

2.5. Beschreibung der Eingabe

- Es gibt mehrere Möglichkeiten der Eingabe eines *regulären Ausdrucks*. Eine besteht darin, beide *reguläre Ausdrücke* über die Eingabefelder „InputRegExpr1“ und „InputRegExpr2“ per Hand einzugeben.
- Weiterhin besteht die Möglichkeit der Eingabe beider *regulärer Ausdrücke* über das Einlesen einer Datei. Dazu kann man den Pushbutton „FileRegExpr“ verwenden. Klickt man diesen Button an, so wird ein Dialogfenster erzeugt. Hier kann man dann die einzulesende Datei und den zugehörigen Verzeichnispfad angeben.

  Die hierfür verwendeten Dateien werden zeilenweise eingelesen, d.h. in der ersten Zeile muß der erste einzulesende *reguläre Ausdruck* stehen und in der zweiten Zeile der zweite *reguläre Ausdruck*.

  Das Zeilenende der ersten Zeile ist das Trennungszeichen der beiden *regulären Ausdrücke*. Die Anwendung erkennt, daß hier der erste *reguläre Ausdruck* zu Ende ist und der zweite *reguläre Ausdruck* beginnt.
- Natürlich besteht auch die Möglichkeit beide Verfahren zu kombinieren. Beispielsweise kann die Eingabe des ersten *regulären Ausdrucks* über ein Eingabefeld erfolgen und die des zweiten *regulären Ausdrucks* über das Einlesen einer Datei bzw. umgekehrt.

  Die Anwendung erkennt von selbst, ob die Eingabefelder schon ausgefüllt sind oder nicht.

  Ist eines der beiden Eingabefelder schon beschriftet, so wird nur der erste *reguläre Ausdruck* aus der ausgewählten Datei ausgelesen und in das noch freie Eingabefeld geschrieben.

  Sind beide Eingabefelder schon beschriftet, so wird die ausgewählten Datei nicht ausgelesen.

- Das Drücken des Pushbottons „CompareRegExpr“ veranlaßt den Vergleich der beiden *regulären Ausdrücke* auf Äquivalenz.
- Das Ergebnis des Vergleichs wird im Ausgabefeld „ResultRegExpr“ angezeigt. Sind die beiden *regulären Ausdrücke* äquivalent, so wird der Text: „äquivalent“ im Ausgabefeld angezeigt, andernfalls erscheint der Text: „nicht äquivalent“.

2.6. Das Programm beenden

Zum Beenden der Anwendung existiert ein eigener Pushbotton „ExitRegExpr“. Klickt man diesen Button an, so wird das Programm beendet. Oberflächenfenster und ggf. Dialogfenster werden geschlossen.

3. Systemhandbuch

3.1. Der Algorithmus

Der hier verwendete Algorithmus von Aho et al. (1988) vergleicht zwei *reguläre Ausdrücke* auf Äquivalenz, indem er direkt, d.h. ohne den Umweg über einen *nichtdeterministischen endlichen Automaten (NEA)*, aus den jeweiligen *regulären Ausdrücken* einen *deterministischen endlichen Automaten (DEA)* erstellt, diesen reduziert und dann die beiden reduzierten *DEA's* auf Gleichheit überprüft.

Zunächst wird für den jeweiligen *regulären Ausdruck (R)* ein *Syntaxbaum (T)* konstruiert. Indem *T* in symmetrischer Reihenfolge durchlaufen wird, werden im Anschluß für jeden einzelnen Knoten von *T* die vier Methoden Nullable, Firstpos,

Lastpos und Followpos ausgeführt. Die Regeln zur Bestimmung dieser Methoden werden in den nächsten Abschnitten ausführlich behandelt.
Zum Schluß wird der *DEA* konstruiert und minimiert. Die beiden minimierten *DEA's* müssen nun identisch sein, wenn die *regulären Ausdrücke* äquivalent sind.

Bei der Eingabe wird ein *regulärer Ausdruck* als String-Objekt an das Programm übergeben. Der *reguläre Ausdruck* muß in Infixnotation sein. Obwohl für den Konkatenationsoperator bei der Eingabe kein Operationszeichen zur Verfügung steht, wird zur internen Verarbeitung innerhalb des Programms für die Konkatenation das Zeichen '+' verwendet. Unter Berücksichtigung folgender Konventionen lassen sich Klammern vermeiden:

- Der einstellige Stern-Operator '*' hat die höchste Priorität.
- Die Konkatenationoperation '+' hat die zweithöchste Priorität.
- Die Oder-Verknüpfung '|' hat die niedrigste Priorität.

Zur Auswertung des *regulären Ausdrucks* wird, iwe bereits erläutert, aus diesem ein *Syntaxbaum* konstruiert. Die Blätter des fertigen *Syntaxbaums*, der den *regulären Ausdruck* repräsentiert, sind mit den *Symbolen* des *Alphabets* markiert. Abbildung 3.1 zeigt den *Syntaxbaum* für den *regulären Ausdruck* ((a | b) * + a) .
Jedem Blatt wird außerdem eine eindeutige ganze Zahl zugeordnet, die die Position des Blattes und des zu diesem Blatt gehörenden *Symbols* im *Syntaxbaum* bezeichnet. So hat beispielsweise das *Symbol* a, welches als erstes *Symbol* in dem *regulären Ausdruck* ((a | b) * + a) steht, die Position 1 im *Syntaxbaum* in Abbildung 3.1, das zweite *Symbol* b die Position 2 und das dritte *Symbol* a die Position 3. Ein in einem *regulären Ausdruck* mehrfach vorkommendes *Symbol* kann demnach verschiedene Positionen im *Syntaxbaum* haben. Im *Syntaxbaum* von Abbildung 3.1 sind diese Positionen unter den *Symbolen* angegeben.
Darüberhinaus erhält der *reguläre Ausdruck* eine eindeutige Endemarkierung #. Damit wird in einfachster Weise für den *deterministischen endlichen Automat* die Menge der akzeptierenden Zustände festgelegt. Bei der späteren Konstruktion des *deterministischen endlichen Automats* ist jeder Zustand mit einem Übergang zu # ein akzeptierender Zustand. Das erspart umfangreiche Betrachtungen ob, ein Zustand akzeptierend sei oder nicht.

Die Endemarkierung # wird durch eine Konkatenation an den *regulären Ausdruck* angefügt ((a | b) * + a + #)[12]. Bei der weiteren Verarbeitung des *regulären Ausdrucks* wird die Endemarkierung # wie ein *Symbol* des *Alphabets* behandelt.
Die inneren Knoten des *Syntaxbaums* nehmen die Operatoren: Konkatenation (+), Oder-Verknüpfung (|) und Stern-Operator (*) ein. Operatoren werden keine Positionen zugeordnet. Sie unterteilen den *regulären Ausdruck* in *Teilworte* und den daraus konstruierten *Syntaxbaum* in Teilbäume. Die Wurzel eines Teilbaumes wird durch einen Operator bestimmt. Unter Anwendung einer vollständigen Klammerung für den *regulären Ausdruck* (((a | b) *) + a + #) entspräche jedes Klammernpaar in diesem *regulären Ausdruck* einem bestimmten Teilbaum im *Syntaxbaum* in Abbildung 3.1. So ist z.B. der dem *Teilwort* (a | b) entsprechende Teilbaum im *Syntaxbaum* in Abbildung 3.1. derjenige Unterbaum, der als Wurzel die Oder-Verknüpfung hat.

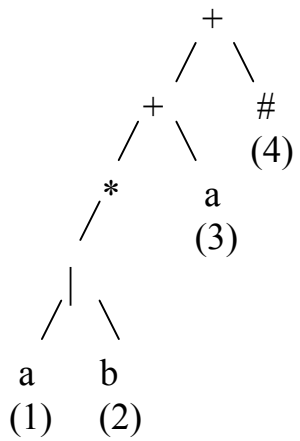


Abbildung3.1: Der aus dem regulären Ausdruck ((a | b) * + a + #) konstruierte Syntaxbaum.

Durch Depth-First-Abarbeitung des *Syntaxbaums* werden dann nacheinander die vier Methoden Nullable, Firstpos, Lastpos und Followpos ausgeführt. Ziel ist es zu ermitteln, wann eine Position auf eine andere Position folgen kann.

[12] Um den Anwender bei der Eingabe zu entlasten wird die Endemarkierung automatisch an den regulären Ausdruck angefügt.

Eine Position ’i’ paßt auf ein *Eingabesymbol*, wenn für diese Position ’i’ die Menge der Positionen, die ’i’ folgen können, gerade Followpos (i) sind. Die Methode Followpos (i) gibt folglich an, welche Positionen im *Syntaxbaum* auf die Position ’i’ folgen können.
Für die Berechnungen der Methode Followpos muß aber bekannt sein, welche Positionen auf das erste und auf das letzte *Symbol* eines *Wortes* passen, welches von einem bestimmten Teilausdruck eines *regulären Ausdrucks* erzeugt wurde. Hierzu werden für jeden Knoten ’n’ des fertigen *Syntaxbaums* eines *regulären Ausdrucks* die Methoden Firstpos(n) und Lastpos(n) definiert. Firstpos(n) liefert die Menge aller Positionen, die auf das erste *Symbol* eines *Wortes* passen können, der vom Teilausdruck mit Wurzel ’n’ erzeugt wird und Lastpos(n) liefert die Menge aller Positionen, die auf das letzte *Symbol* eines solchen *Wortes* passen können.

3.1.1. Die Methode Nullable

Zur Bestimmung der beiden Methoden Firstpos und Lastpos ist es jedoch nötig zu wissen, ob ein gegebener Knoten ’n’ die Wurzel eines Teilausdrucks ist, der eine Sprache mit dem leeren Wort {ε} erzeugen kann, d.h., daß das leere Wort {ε} in der Menge der möglichen *Teilworte* enthalten ist, die der Knoten ’n’ repräsentiert. Hierzu wird die Methode Nullable eingeführt. Wenn ein gegebener Knoten ’n’ das leere Wort {ε} erzeugen kann, dann liefert Nullable(n) den Wert ‘true‘ zurück, ansonsten den Wert ‘false‘.
Die nachfolgenden Regeln definieren die Möglichkeiten, die Wahrheitswerte für die Methode Nullable zu bestimmen.

(1) Nullable liefert den Wert ‘false‘, falls der Knoten ’n’ ein Blatt im *Syntaxbaum* ist, das mit einem *Symbol* des *Alphabets* markiert ist. In diesem Fall entspricht nämlich jedes solche Blatt einem einzelnen Eingabesymbol (a,..,z) und kann deswegen das leere Wort {ε} nicht erzeugen.

(2) Falls der Knoten ’n’ ein Kleene`scher Stern-Operator ’*’ ist, dann liefert Nullable den Wert ‘true‘. Es spielt dabei keine Rolle, welche Werte für das *Teilwort* seines Vorgängers im linken Unterbaum ermittelt wurden. Auf die Bestimmung der Wahrheitswerte für die Methode Nullable hat der Operand keinen Einfluß. Die Anwendung des Stern-Operators auf einen beliebigen Operanden bedeutet ja gerade, daß dieser Operand nullmal (also ε) oder beliebig oft wiederholt wird.
Der Kleene`sche Abschluß ist idempotent. D.h., auch bei mehrmaligem Anwenden des Stern-Operators auf einen beliebigen Operanden bleibt die Lösungsmenge dieser Operation immer gleich (a* = a** = a*** =...). Aufgrund dieser Eigenschaft ändert sich das Ergebnis auch nicht, wenn der Operand selbst das leere Wort {ε} erzeugen kann. Für den in Abbildung 3.2 dargestellten *Syntaxbaum* für den *regulären Ausdruck* (a*) erzeugt der Stern-Operator ’*’ die Menge aller *Worte*, die aus null oder mehreren a´s bestehen (ε,a,aa,aaa,...). Nullable(n) liefert demzufolge den Wert ‘true‘ zurück.

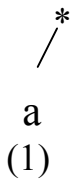
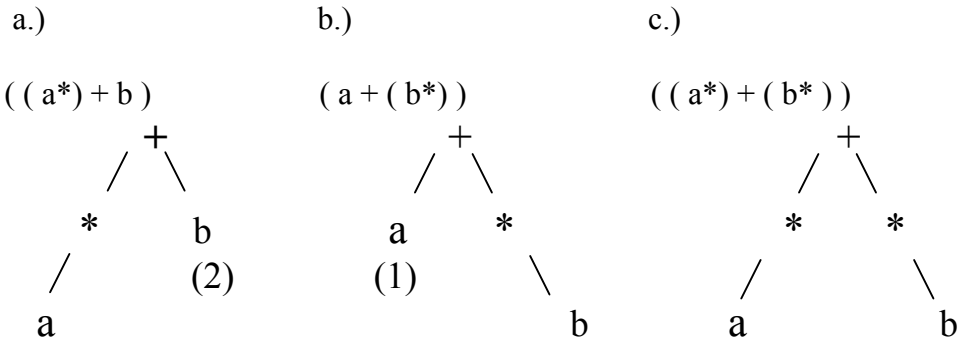


Abbildung3.2: Syntaxbaum für den regulären Ausdruck (a*) .

(3) Falls der Knoten ’n’ ein Konkatenationsoperator mit einem Vorgänger *n.vorgänger* und einem Nachfolger *n.nachfolger* ist, dann ergibt sich als Ergebnis für Nullable(n) der bool´sche Wert der UND-Verknüpfung der beiden Werte Nullable(*n.vorgänger*) und Nullable(*n.nachfolger*). Das liegt daran, daß an der Konkatenationsoperation beide Operanden gleichberechtigt Teilnehmen. Die Konkatenation kann nicht, wie zum Beispiel der Kleene´sche Abschluß, das leere Wort {ε} erzeugen.
Abbildung 3.3 zeigt drei Beispiele, wie eine Konkatenation für Nullable zustandekommen kann. Im ersten Fall, Abbildung 3.3 a.), erzeugt das *Teilwort* des

Vorgängers im linken Unterbaum das leere Wort {ε}, d.h. Nullable(*n.vorgänger*) liefert den Wert 'true' zurück. Das *Teilwort* des Nachfolgers im rechten Unterbaum aber erzeugt das leere Wort nicht. Nullable(*n.nachfolger*) hat somit den Wert 'false'. Da das leere Wort { ε } das Identitätselement bezüglich der Konkatenation ist, d.h. daß ein beliebiges *Wort* a konkateniert mit dem leeren Wort { ε } gerade wieder dieses *Wort* a ergibt (a + ε = a), wird das Ergebnis des linken *Teilwortes*, welches ε erzeugen kann, von dem des rechten *Teilwortes*, welches ε nicht erzeugen kann, überdeckt. Der Knoten 'n' kann demzufolge das leere Wort { ε } ebenfalls nicht erzeugen und liefert für Nullable(n) den Wert 'false'.

Ganz änhlich ist die Situation im zweiten Fall in Abbildung 3.3 b.). Das *Teilwort* des Vorgängers im linken Unterbaum kann das leere Wort {ε} nicht erzeugen, dafür aber das *Teilwort* des Nachfolgers im rechten Unterbaum. Genauso wie im obigen Beispiel wird das Ergebnis des *Teilwortes,* welches das leere Wort erzeugen kann, von dem Ergebnis des *Teilwortes,* welches das leere Wort nicht erzeugen kann, überdeckt. Auch das Ergebnis ist das gleiche. Der Knoten 'n' hat für Nullable(n) den Wert 'false'.
In Abbildung 3.3 c.) wird der *Syntaxbaum* für den *regulären Ausdruck* ((a*) + (b*)) gezeigt. Hier ist es so, daß beide *Teilworte,* sowohl das des Vorgängers im linken Unterbaum (a*) als auch das des Nachfolgers im rechten Unterbaum (b*) das leere Wort {ε} erzeugen können. Für die Ausführung der Methode Nullable bezüglich der Konkatenationsoperation des Knoten 'n' bedeutet dies, daß beide Operanden, Nullable(*n.vorgänger*) und Nullable(*n.nachfolger*) den Wert 'true' zurückliefern. Somit ist auch Nullable(n) wahr.



(1) (2) (1) (2)

Abbildung3.3: Syntaxbäume für die regulären Ausdrücke: ((a*) + b), (a + (b*)) und ((a*) + (b*)) .

3.1.2. Die Methode Firstpos

Firstpos(n) liefert also, wie weiter oben schon erwähnt, die Menge aller Positionen eines *Teilwortes*, die an erster Stelle dieses *Teilwortes* möglich sind. Die Methode Firstpos(n) wird für jeden Knoten des *Syntaxbaums* aufgerufen. Im folgenden werden die Regeln zur Bestimmung der Methode Firstpos angegeben:

(1) Ist der aktuell betrachtete Knoten ’n’ ein mit der Position ’i’ markiertes Blatt, dann liefert Firstpos(n) gerade die Menge { i } zurück. Das *Teilwort* des *regulären Ausdrucks* besteht an dieser Stelle ja nur aus einem einzigen *Symbol.* Die erste Position dieses *Teilwortes* kann demnach nur das *Symbol* selbst einnehmen. In Abbildung 3.4 zum Beispiel nimmt das Symbol b im *Syntaxbaum* die Position zwei ein. Für das *Teilwort*, welches nur aus dem *Symbol* b besteht, ist die Menge der Positionen, die an erster Stelle dieses *Teilwortes* möglich sind, gerade die Menge { 2 }.

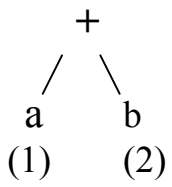


Abbildung 3.4: Syntaxbaum für den regulären Ausdruck (a + b) .

(2) Wenn der gerade betrachtete Knoten ’n’ durch einen Stern-Operator ’*’ dargestellt ist, dann enthält Firstpos(n) lediglich die Werte seines Vorgängers im linken Unterbaum, nämlich Firstpos(*n.vorgänger*). Das liegt daran, daß der Stern-Operator ein

Postfixoperator mit nur einem Operanden ist. Dadurch steht der Operand automatisch immer vor dem Operator, und die Menge aller Positionen, die auf das erste *Symbol* dieses *Wortes* passen können, wird somit auch immer durch den Operanden bestimmt. Der in Abbildung 3.5 dargestellte *Syntaxbaum* für den *regulären Ausdruck* (a*) liefert als Firstposwerte für den Teilausdruck a nach Regel (1) die Menge { 1 } zurück. Der einstellige Postfixoperator '*' sagt nun aus, daß der davor stehende *reguläre Ausdruck* nullmal oder beliebig oft vorkommt, d.h. die Menge aller *Worte*, die aus null oder mehreren a´s bestehen (ε,a,aa,aaa,...). Also paßt auch nullmal oder beliebig oft die Position 1 auf das erste *Symbol* dieses *Wortes.* Es existiert keine andere Lösungsmenge für (a*). Firstpos(n) liefert also ebenfalls die Menge {1} zurück.

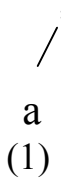


Abbildung 3.5: Syntaxbaum für den regulären Ausdruck (a*) .

(3) Falls der Knoten 'n' ein Konkatenationsoperator mit einem Vorgänger *n.vorgänger* und einem Nachfolger *n.nachfolger* ist, dann erhält Firstpos(n) die Werte seines Vorgängers Firstpos(*n.vorgänger*).

Diesen Fall zeigt Abbildung 3.6 und läßt sich so erklären: Bei einer Konkatenation zweier *regulärer Ausdrücke* wird die Menge aller Positionen, die auf das erste *Symbol* dieses gemeinsamen *Wortes* passen können normalerweise - bis auf die Ausnahme, die im nächsten Abschnitt erläutert wird - nur aus Positionen des ersten *Teilwortes* bestehen können. Das liegt daran, daß die Konkatenationsoperation nicht kommutativ ist, d.h., die Reihenfolge der Operanden ist fest. Zuerst wird das *Teilwort* des Vorgängers im linken Unterbaum ausgewertet und dann erst wird das *Teilwort* des Nachfolgers im rechten Unterbaum ausgewertet. Für den in Abbildung 3.6 gezeigten *Syntaxbaum* des

regulären Ausdrucks (a + b) liefert die Methode Firstpos(n) für den Knoten 'n' mit dem Konkatenationsoperator '+' die Menge { 1 }.

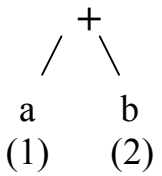


Abbildung3.6: Syntaxbaum für den regulären Ausdruck (a + b) .

Falls aber der Knoten 'n' ein Konkatenationsoperator mit einem Vorgänger *n.vorgänger* und einem Nachfolger *n.nachfolger* ist und wenn der Vorgänger ein mit dem leeren Wort{ε} markierter Knoten sein kann, dann ändert sich die Ergebnismenge der Methode Firstpos gravierend. Denn wie schon in den vorherigen Regeln erläutert, ist das leere Wort {ε} das neutrale Element bezüglich der Konkatenation. Wenn das *Teilwort* des Vorgängers tatsächlich nur aus dem leeren Wort besteht, erlaubt ε dem dahinterliegenden *Teilwort,* durch sich hindurch sichtbar zu werden. In diesem Fall kann die erste Position des Nachfolgers im rechten Teilbaum die erste Position im gesamten *regulären Ausdruck* sein.
Da es nicht vorherbestimmbar ist, ob und wann dieser Fall eintritt, muß er in jedem Fall berücksichtigt werden.
So ergibt sich nun als Ergebnismenge für Firstpos(n) die Vereinigung der beiden Mengen Firstpos(*n.vorgänger*) und Firstpos(*n.nachfolger*).
Dieser Fall ist in Abbildung 3.7 dargestellt. Hier ist es nun so, daß (a*), das *Teilwort* des Vorgängers im linken Unterbaum, aus dem leeren Wort {ε} bestehen kann (ε,a,aa,aaa,...). Dies muß bei der Auführung der Methode Firstpos für den Knoten 'n', der die Konkatenationsoperation enthält, berücksichtigt werden. Die Ergebnismenge des *Teilwortes* des Nachfolgers im rechten Unterbaum ist somit Teilmenge der Ergebnismenge des Knotens 'n'. Für Firstpos(n) ergibt sich demzufolge die Vereinigung der beiden Mengen Firstpos(*n.vorgänger*) und Firstpos(*n.nachfolger*). Diese Menge ist gerade die Menge { 1 , 2 }.

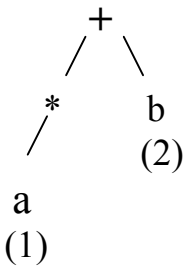


Abbildung 3.7: Syntaxbaum für den regulären Ausdruck ((a*) + b) .

(4) Zuletzt wird der Fall betrachtet, daß der Knoten ’n’ durch eine Oder-Verknüpfung ’|’ dargestellt wird, mit einem Vorgänger *n.vorgänger* im linken Unterbaum und einem Nachfolger *n.nachfolger* im rechten Unterbaum. Als Ergebnis für Firstpos(n) ergibt sich dann die Vereinigung der beiden Mengen Firstpos(*n.vorgänger*) und Firstpos(*n.nachfolger*). Abbildung 3.8 zeigt den aus dem *regulären Ausdruck* (a | b) konstruierten *Syntaxbaum*. Dieser *reguläre Ausdruck* bezeichnet die Menge der *Worte*, für die ein *Wort* entweder ein einzelnes a ist oder aus einem b besteht. Die Oder-Verknüpfung ist aber kommutativ, d.h. (a | b) = (b | a), deshalb spielt es auch keine Rolle, in welcher Reihenfolge die beiden Operanden angeordnet sind. Sowohl das *Teilwort* des Vorgängers im linken Unterbaum als auch das *Teilwort* des Nachfolgers im rechten Unterbaum kann die erste Position im *regulären Ausdruck* liefern. Die Ergebnismenge für Firstpos(n) ist in diesem Fall die Menge $\{ 1 \} \cup \{ 2 \} = \{ 1 , 2 \}$.

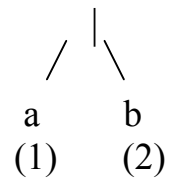


Abbildung 3.8: Der aus dem regulären Ausdruck (a | b) konstruierte Syntaxbaum.

3.1.3. Die Methode Lastpos

Die Methode Lastpos(n) liefert die Menge aller Positionen, die auf das letzte *Symbol* eines *Wortes* passen können, der von einem Teilausdruck mit Wurzel 'n' erzeugt wird. Wie schon die Methode Firstpos wird auch die Methode Lastpos für jeden Knoten des *Syntaxbaums* aufgerufen. Im folgenden werden die Regeln zur Bestimmung der Methode Lastpos angegeben. Da diese Regeln ganz ähnlich denen der Methode Firstpos sind, wird bei den Erklärungsansätzen meist auf die entsprechende Regel der Methode Firstpos verwiesen.

(1) Ist der aktuell betrachtete Knoten ein mit der Position 'i' markiertes Blatt, dann liefert Lastpos(n) gerade die Menge { i } zurück. Abbildung 3.4 zeigt den Syntaxbaum für den regulären Ausdruck (a + b) . Das *Symbol* a nimmt im *Syntaxbaum* die Position eins ein. Für dasjenige *Teilwort* aber, welches nur aus dem *Symbol* a besteht, ist die Menge der Positionen, die an letzter Stelle dieses *Teilwortes* möglich sind, gerade die Menge { 1 }.

(2) Wenn der gerade betrachtete Knoten 'n' durch einen Stern-Operator '*' dargestellt ist, dann bekommt Lastpos(n) die Werte des *Teilwortes* seines Vorgängers im linken Unterbaum, der die Menge der Positionen, die auf das letzte Symbol dieses *Wortes* passen können enthält, nämlich gerade Lastpos(*n.vorgänger*). Wie schon in Regel (2) der Methode Firstpos ausgeführt wurde, liegt das an den Eigenheiten des einstelligen Postfix-Operators '*', der auch bei der Bestimmung der Lösungsmenge für Lastpos immer durch seinen einen Operanden bestimmt wird. Anhand des in Abbildung 3.5 dargestellten *Syntaxbaums* für den *regulären Ausdruck* (a*) kann man auch die Bestimmung der Methode Lastpos nachvollziehen. Nach Regel (1) liefert der Teilausdruck a für Lastpos die Menge { 1 } zurück. Die Anwendung des Stern-Operators auf a erzeugt keine neuen Positionen für den *regulären Ausdruck*, sondern kann a lediglich beliebig oft wiederholen. Es existiert demzufolge keine weitere Lösungsmenge für (a*). Lastpos(n) liefert demnach ebenfalls die Menge {1} zurück.

(3) Falls der Knoten ’n’ ein Konkatenationsoperator mit einem Vorgänger *n.vorgänger* und einem Nachfolger *n.nachfolger* ist und wenn der Nachfolger ein mit {ε} markierter Knoten sein kann, dann ergibt sich als Ergebnis für Lastpos(n) die Vereinigung der beiden Mengen Lastpos(*n.vorgänger*) und Lastpos(*n.nachfolger*).
Diese Regel unterscheidet sich von der Regel (3) für Firstpos lediglich dadurch, daß Vorgänger und Nachfolger vertauscht sind.
Da die Konkatenationsoperation nicht kommutativ ist, kann die Menge der Positionen, die auf das letzte *Symbol* eines *Wortes* passen können, welches von einem Teilausdruck mit der Wurzel ’n’ erzeugt wird, normalerweise nur aus den Positionen des *Teilwortes* des Nachfolgers im rechten Unterbaum zusammengesetzt sein. Das leere Wort {ε}, welches das neutrale Element der Konkatenation ist, erlaubt es nun, dem davorliegenden *Teilwort* durch sich hindurch sichtbar zu werden. Folglich kann in diesem Fall eine letzte Position des Vorgängers im linken Teilbaum die letzte Position im gesamten *regulären Ausdruck* sein. Die Methode Lastpos berücksichtigt das durch die Vereinigung der beiden Mengen Lastpos(*n.vorgänger*) und Lastpos(*n.nachfolger*).
Der in Abbildung 3.9. kontruierte *Syntaxbaum* für den *regulären Ausdruck* (a + (b*)) läßt sich zur Veranschaulichung der Methode Lastpos verwenden.
Für den Teilausdruck a des Vorgängers im linken Teilbaum liefert Lastpos die Menge {1} zurück. Für den Teilausdruck (b*) des Nachfolgers im rechten Teilbaum liefert Lastpos gemäß Regel (2) die Menge {2}. Damit ergibt sich als Lösungsmenge von Lastpos für den Knoten mit dem Konkatenationsoperator die Vereinigung der beiden Mengen Lastpos(*n.vorgänger*) und Lastpos(*n.nachfolger*). Diese Menge ist die Menge { 1 , 2 }.

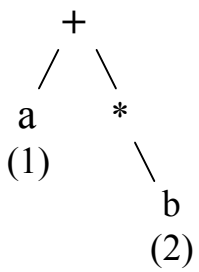


Abbildung 3.9: Syntaxbaum für den regulären Ausdruck (a + (b*)) .

Kann der Nachfolger im rechten Unterbaum das leere Wort {ε} nicht erzeugen, so enthält Lastpos(n) nur die Werte des Nachfolgers Lastpos(*n.nachfolger*). Für diesen Fall kann, die schon für die Methode Firstpos verwendete, obige Abbildung 3.7 herangezogen werden, die den *Syntaxbaum* für den *regulären Ausdruck* ((a*) + b) zeigt. Hierbei ist es so, daß das *Teilwort* des Nachfolgers im rechten Unterbaum niemals aus dem leeren Wort {ε} bestehen kann und sich somit auch immer hinter dem *Teilwort* des Vorgängers im linken Unterbaum befindet. Das bedeutet, daß sich die Menge der Positionen, die auf das letzte Symbol dieses *Wortes* passen können, der vom Teilausdruck mit der Wurzel 'n' erzeugt wird, nur aus Positionen des Nachfolgers zusammensetzen kann. Die Lösungsmenge für die Methode Lastpos für den Knoten 'n' ist somit die Menge { 2 }. Für dieses Beispiel in Abbildung 3.7 wird auch deutlich, daß es für die Lösungsmenge der Methode Lastpos überhaupt keine Rolle spielt, daß das *Teilwort* des Vorgängers im linken Unterbaum das leere Wort{ε} erzeugen kann. Relevant für Lastpos ist nur der Nachfolger im rechten Unterbaum.

(4) Der für die Methode Lastpos letzte zu betrachtende Fall ist der, daß der Knoten 'n' durch eine Oder-Verknüpfung '|', mit einem Vorgänger *n.vorgänger* im linken Unterbaum und einem Nachfolger *n.nachfolger* im rechten Unterbaum, dargestellt wird. Die Ergebnismenge für Lastpos(n) ist dann die Vereinigung der beiden Mengen Lastpos(*n.vorgänger*) und Lastpos(*n.nachfolger*). Abbildung 3.8 zeigt den aus dem *regulären Ausdruck* (a | b) konstruierten *Syntaxbaum*. Wie schon bei der Besprechung der Regel (4) der Methode Firstpos deutlich wurde, ist die Oder-Verknüpfung kommutativ. Deshalb kann sowohl das *Teilwort* des Vorgängers im linken Unterbaum als auch das *Teilwort* des Nachfolgers im rechten Unterbaum die letzte Position im *regulären Ausdruck* liefern. Die Ergebnismenge für Lastpos(n) ist in diesem Fall die Menge { 1 } ∪ { 2 } = { 1 , 2 }.

3.1.4. Die Methode Followpos

Die Methode Followpos wird immer dann aufgerufen, wenn der gerade betrachtete Knoten entweder ein innerer Knoten mit dem Konkatenationsoperator '+' ist oder wenn ein Stern-Operator '*' vorliegt.

(1) Falls der gerade betrachtete Knoten eine Oder-Verknüpfung '|' darstellt, so kann keine Aussage über einen potentiellen Nachfolger gemacht werden, da die Oder-Verknüpfung kommutativ ist, d.h. (a | b) = (b | a), und somit in diesem Fall auch die Alternative berücksichtigt werden muß, daß es keinen eindeutigen Nachfolger gibt.
(2) Liegt nun eine Konkatenation vor, so ist klar, daß nur Positionen aus dem rechten Teilbaum den Positionen aus dem linken Teilbaum folgen können. Für den linken Teilbaum werden deshalb alle möglichen letzten Positionen in der Methode Lastpos ermittelt und für alle diese letzten Positionen alle ihre möglichen Nachfolger bestimmt. Diese Nachfolger sind dann gerade alle die in der Methode Firstpos berechneten möglichen ersten Positionen des rechten Teilbaums.
Abbildung 3.10 zeigt den *Syntaxbaum* für den *regulären Ausdruck* ((a | b) + a). Für den linken Teilbaum, der dem *Teilwort* (a | b) entspricht, ergibt sich für Lastpos die Menge { 1 , 2 }. Der rechte Teilbaum liefert für Firstpos die Menge { 3 }. Nach obiger Regel werden nun jeder der Lastpos-Positionen des linken Teilbaums jeweils alle Firstpos-Positionen des rechten Teilbaums zugewiesen. Somit ergibt sich Followpos(1) = { 3 } und Followpos(2) = { 3 }.

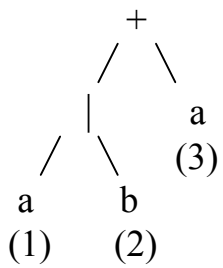


Abbildung 3.10: Der aus dem regulären Ausdruck ((a | b) + a) konstruierte Syntaxbaum.

Die Ergebnisse der Methode Followpos werden in einer Matrix zusammengefaßt. Somit ergeben sich nach Ausführung der Methode Followpos für den *regulären Ausdruck* ((a | b) + a) die in Tabelle 3.1 gezeigten Resultate.

Knoten	Followpos

1	{ 3 }
2	{ 3 }
3	---

Tabelle 3.1: Ergebnisse der Methode Followpos bei der Auswertung des Knotens '+' für den regulären Ausdruck ((a | b) + a).

(3) Wenn nun der gerade betrachtete Knoten ein Stern-Operator ist, dann werden für den linken Teilbaum wie schon beim Konkatenationsoperator alle möglichen letzten Positionen in der Methode Lastpos ermittelt und für alle diese letzten Positionen alle ihre möglichen Nachfolger bestimmt. Da der Stern-Operator ein Postfixoperator ist, existiert kein rechter Teilbaum. Die Nachfolger sind dann alle die in der Methode Firstpos berechneten möglichen ersten Positionen des betrachteten Knotens. Das liegt daran, daß der Operand des Stern-Operators beliebig oft wiederholt werden und sich somit auch beliebig oft selbst folgen kann.

In Abbildung 3.11 wird der *Syntaxbaum* für den *regulären Ausdruck* (a | b)* dargestellt. Tabelle 3.2 zeigt die zugehörige Ergebnismatrix nach Anwendung der Methode Followpos.

Für den Teilbaum, der dem *Teilwort* (a | b) entspricht, ergibt sich sowohl für Firstpos, als auch für Lastpos die Menge { 1 , 2 }. Nach der eben besprochenen Regel (3) werden nun jeder Lastpos-Position dieses Teilbaums alle Firstpos-Positionen des selben Teilbaums zugewiesen. Folglich ergibt sich für Followpos(1) = { 1 , 2 } und für Followpos(2) = { 1 , 2 }.

*

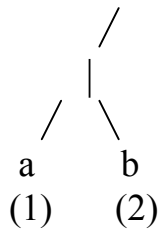


Abbildung 3.11: Der aus dem regulären Ausdruck ((a | b)*) konstruierte Syntaxbaum.

Knoten	Followpos
1	{ 1 , 2 }
2	{ 1 , 2 }
3	---

Tabelle 3.2: Ergebnisse der Methode Followpos bei der Auswertung des Knotens '*' für den regulären Ausdruck ((a | b)*).

3.1.5. Der deterministische endliche Automat

Ausgehend von den Ergebnissen der Methode Followpos kann nun der *deterministische endliche Automat* erstellt werden. Dazu müssen zuerst *Dstates* und *Dtran* konstruiert werden. *Dstates* definiert die Menge der Zustände des *DEA*. *Dtran* ist die Übergangstabelle für den *DEA*.

(1) *Dstates***:** Die Zustände in *Dstates* sind aus den Mengen der Positionen, welche *Symbole* im *regulären Ausdruck* repräsentieren, hergeleitet. In jedem Zustand sind diejenigen Positionen des *regulären Ausdrucks* zusammengefaßt, die auf Positionen des Zustands vor diesem Zustand folgen können.
Der erste Zustand in *Dstates* kann keinem anderen Zustand folgen. Demnach wird dieser Zustand durch die Menge Firstpos der Wurzel des zugehörigen *Syntaxbaums* dargestellt, d.h. die Ergebnismenge, die sich ergibt, wenn die Methode Firstpos für den Knoten ausgeführt wird, der die Wurzel des *Syntaxbaums* bildet. Folglich sind in dieser Menge die Positionen der *Symbole* enthalten, die an erster Stelle des *Wortes* möglich sind, welches durch den gegebenen *regulären Ausdruck* beschrieben wird. Der erste

Zustand in *Dstates* ist dementsprechend auch der Startzustand für den *deterministischen endlichen Automat.*

Alle weiteren Zustände werden dann aus der Methode Followpos hergeleitet. Dazu wird jeder vorhandene Zustand in *Dstates* einzeln aufbereitet, indem für alle Positionen dieses Zustands, die ein bestimmtes *Symbol* des *regulären Ausdrucks* repräsentieren, die Menge der Folgepositionen, also Followpos(i), bestimmt wird. Die Vereinigung aller dieser Mengen von Folgepositionen eines Zustands ergibt dann einen neuen Zustand, den Folgezustand des gerade betrachteten Zustands für dieses bestimmte *Symbol*. Ist dieser Zustand noch nicht in der Menge *Dstates* enthalten, so wird er *Dstates* hinzugefügt.

Der aus dem *regulären Ausdruck* ((a | b) + a) konstruierte *Syntaxbaum* in Abbildung 3.10 hat als Wurzelknoten den Operator '+'. Die Methode Firstpos angewendet auf den Wurzelknoten liefert die Menge { 1 , 2 } zurück. Folglich ist der erste Zustand in Dstates der aus der Menge{ 1 , 2 } hergeleitete Zustand (1,2). Für das *Symbol* a des *regulären Ausdrucks*, welches durch die Position 1 im Zustand (1,2) repräsentiert wird, wird sodann die Menge der Folgepositionen ermittelt. Wie in Abschnitt 3.1.4. für Tabelle 3.1 schon berechnet, ist dies die Menge { 3 }. Weitere Mengen von Folgepositionen gibt es nicht für das *Symbol* a. Folglich entfällt auch die Vereinigung aller Mengen von Folgepositionen zu einem neuen Zustand. Folgezustand des Zustands (1,2) für das *Symbol* a ist demnach der Zustand (3).

Wenn die ganze Prozedur auf das *Symbol* b des *regulären Ausdrucks*, welches für die Position 2 im Zustand (1,2) steht, angewandt wird, dann hat dies ebenfalls den Zustand (3) zum Ergebnis. Dstates hat in diesem Fall nur zwei Zustände: { (1 , 2) , (3) }.

(2) *Dtran***:** *Dtran* ist die Übergangstabelle für den *deterministischen endlichen Automat* mit einer Zeile für jeden Zustand von *Dstates* und einer Spalte für jedes *Eingabesymbol* des *regulären Ausdrucks*.

Für jedes dieser *Eingabesymbole*, welche durch eine Position in einem Zustand von *Dstates* repräsentiert wird, werden alle Übergänge zu anderen Zuständen bestimmt und zu *Dtran* hinzugefügt.

Demnach ergeben sich, für das schon oben besprochene Beispiel des *regulären Ausdrucks* ((a | b) + a) aus Abbildung 3.10, folgende zwei Übergänge: Ein Übergang

von Zustand (1 , 2) für das Eingabesymbol a zu dem Zustand (3) und ebenfalls ein Übergang von Zustand (1 , 2) für das Eingabesymbol b zu dem Zustand (3). Tabelle 3.3 zeigt die Übergangstabelle des *deterministischen endlichen Automats* für den *regulären Ausdruck* ((a | b) + a).

Dtran		
Dstate	a	B
(1 , 2)	(3)	(3)
(3)	–––	–––

Tabelle 3.3: Übergangstabelle des deterministischen endlichen Automats für den regulären Ausdruck ((a | b) + a).

Zustände mit gleichen Übergängen für alle *Eingabesymbole* zu einem bestimmten Zustand können zu einem Zustand zusammengefaßt werden, indem die Referenzen dieser Zustände in *Dtran* für jedes *Eingabesymbol* angeglichen werden.
Ein komplettes Beispiel, wie ein *regulärer Ausdruck* in einen *deterministischen endlichen Automat* überführt wird, wird in Abschnitt 3.4. „Beispielverarbeitung eines regulären Ausdrucks“ gezeigt.

3.2. Verwendete Klassen

3.2.1. Die Benutzerschnittstelle

Unter einer Benutzerschnittstelle (user interface) versteht man die Art, wie sich ein Terminal dem Benutzer darstellt. Im einfachsten Fall werden die Kommandos zeilenweise eingetippt und nach dem Drücken der Return-Taste ausgeführt. Diese Art der Eingabe heißt Kommandozeileneingabe. Neuere Anwendungen arbeiten kaum noch

mit der Kommandozeileneingabe. Sie stellen dem Benutzer stattdessen eine umfangreiche Sammlung graphischer Schnittstellen (graphical user interface kurz GUI genannt) zur Verfügung, die es diesem erlauben, mittels graphischer Elemente wie Fenster, Schaltknöpfe oder Menüs, auf einfache Art und Weise mit dem Programm zu kommunizeren. Javaprogramme, die solche modernen Fensterumgebungen erstellen, verwenden eine spezielle Klassenbibliothek, genannt Abstract Window Toolkit, kurz AWT. Die AWT ist eine Standardbibliothek, die für alle Javaversionen verfügbar ist [13].

Auch das hier vorliegende Programm verwendet die AWT um dem Benutzer die Bedienung zu erleichtern. Die folgende Abbildung 3.12 zeigt die Vererbungshirarchien der AWT-Klassen in einem Ausschnitt aus dem Paket java.awt, wie bei Deitel und Deitel (1998) verwendet. Aufgeführt sind nur diejenigen Klassen der AWT, die das Progamm auch tatsächlich benutzt.

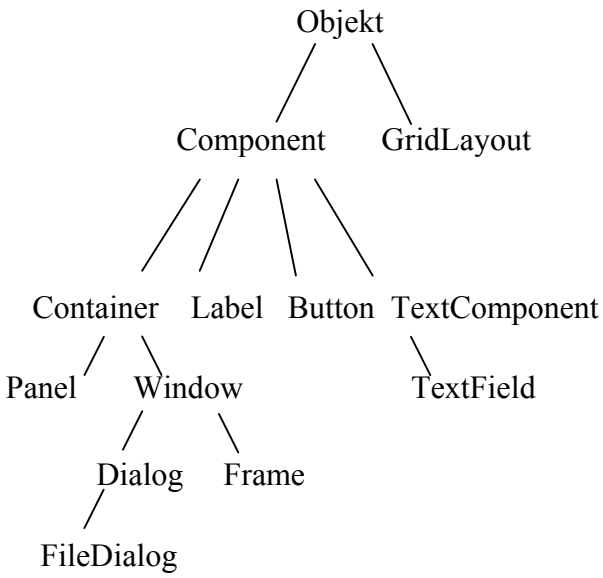


Abbildung 3.12: Vererbunghirarchie der verwendeten AWT-Klassen.

[13] Vgl. Savitch 1999.

Im Folgenden werden die verwendeten AWT-Klassen kurz beschrieben. Eine ausführlichere Behandlung aller AWT-Komponenten sowie allgemeine Regeln zu deren Nutzung bieten Campione und Walrath (1997) sowie Schader und Schmidt-Thieme(1999).

- `Container` stellen eine rechteckige Fläche zur Verfügung, in der andere `Component`-Objekte dargestellt werden können. Dies ist deshalb notwendig, weil jedes `Component`-Objekt zur Darstellung auf dem Bildschirm einem `Container`-Objekt hinzugefügt werden muß. Dieser `Container` aber ist wiederum selbst ein `Component`-Objekt und kann wieder einem anderen `Container` hinzugefügt werden.

  Die am häufigsten verwendete `Container`-Methode ist die Methode add. Sie wird zur

  Aufnahme der Komponenten in den `Container` benötigt. Die verwendeten `Container`-Unterklassen sind `Panel, Window, Dialog` und `Frame`.

  Hier eine kurze Übersicht:

  - Die Klasse `Panel` kann dazu verwendet werden, Komponenten aufzubewahren oder Unterklassen zu definieren, um für diese besondere Aufgaben, wie zum Beispiel die Ereignisbehandlung, durchzuführen.
  - `Window`-Objekte sind sogenannte top-level `Container`. Sie sind die einzigen Komponenten, die keinem `Container` hinzugefügt werden. `Window`−Objekte sind die Klassen `Frame` und `Dialog`.
    - Die Klasse `Frame` stellt umrahmte Anwendungsfenster für Anwendungen zur Verfügung. Jede Anwendung benötigt mindestens ein `Frame`-Objekt.

      Neben dem argumentlosen `Frame`-Konstruktor, bietet die Klasse `Frame` auch noch einen einargumentigen Konstruktor `Frame`(String) an. Das String-Argument soll die Titelzeile des umrahmten Fensters angeben.

      Weitere durch `Frame` zur Verfügung gestellte Methoden sind z.B.: getTitle( ) und setTitle(String), die die Titelzeile des umrahmten Fensters liefern bzw. neu setzen, oder getIconImage( ) und setIconImage(Image), die das Bild, welches als Symboldarstellung für das Fenster dienen soll, zurückliefern bzw. festlegen.

- Mit der Klasse `Dialog` bietet AWT Unterstützung für Komponenten an, die eine Benutzereingabe erwarten, sogenannte Dialogfenster. Dialogfenster können modal erzeugt werden, d.h., daß keine anderen Benutzeraktivitäten möglich sind, solange das Dialogfenster aktiv ist. Der Unterschied zwischen Dialogfenstern und üblichen Anwendungsfenstern, die durch ein `Frame`-Objekt implementiert sind, ist der, daß Dialogfenster von einem anderen Anwendungsfenster abhängig sind. Wird das Anwendungsfenster entfernt, so wird auch das von ihm abhängige Dialogfenster entfernt. Wird das Anwendungsfenster in seine Symboldarstellung weggeblendet, so verschwindet auch das von ihm abhängige Dialogfenster vom Bildschirm. Wird das Anwendungsfenster wieder in seinen alten Zustand gebracht, so erscheint auch das Dialogfenster wieder auf dem Bildschirm.
    - Die Klasse `FileDialog` ist eine Unterklasse der Klasse `Dialog` und dient der Interaktion mit dem Dateisystem. `FileDialog`-Fenster sind immer modal.
- Das `GridLayout`-Objekt ist ein Layoutverwalter. Ein Layoutverwalter ist ein Objekt, welches die Größe und die Position einer Komponente in einem `Container` verwaltet. Ein `GridLayout` plaziert die verwendeten Komponenten in einem Zellengitter, in dem alle Zellen gleich groß sind. Jeder der benutzten Komponenten wird eine komplette Zelle im Zellengitter zugewiesen.

  Für die Klasse `GridLayout` gibt es zwei Konstruktoren: `GridLayout`(int, int) und `GridLayout`(int, int, int, int). Der Konstruktor mit zwei int-Argumenten verlangt die Anzahl der Zeilen und Spalten. Der zweite Konstruktor erwartet zusätzlich noch die Angaben über den horizontalen bzw. den vertikalen Abstand, d.h., die Angabe wie viele Bildpunkte zwischen den einzelnen Zellen liegen sollen.

  Die wichtigste Methode ist die Methode add. Mithilfe dieser Methode werden die zu verwaltenden Komponenten von links nach rechts und zeilenweise von oben nach unten hinzugefügt.
- Die Klasse `Label` bietet eine einfache Möglichkeit, einzeiligen, nicht selektierbaren Text in eine Anwendung zu setzen. `Labels` werden meist dazu verwendet, Eingabefelder zu markieren beziehungsweise Informationen über die gerade ausgewählte Komponente anzuzeigen.

Für die Klasse `Label` gibt es drei Konstruktoren: `Label`( ), `Label`(String) und `Label`(String, int). Der erste Konstruktor erwartet gar kein Argument, der zweite Konstruktor erwartet als Argument den Labeltext und der dritte zusätzlich noch einen Parameter zur Positionierung.

Mittels der beiden Methoden getText bzw. setText kann der Labeltext gelesen bzw. verändert werden.

- Die Klasse `Button` stellt eine voreingestellte Implementierung einer Schaltfläche zur Verfügung, deren einzige Aufgabe es ist, ein bestimmtes Ereignis auszulösen, sobald der Anwender diese Schaltfläche anklickt.

  Der Klasse `Button` stehen zwei Konstruktormethoden zur Verfügung: `Button`( ) und `Button`(String). Mit dem String-Parameter kann man den `Button` beschriften.

  Ferner sind die Methoden getLabel und setLabel(String) deklariert, die den angezeigten `Button`-Text zurückliefern bzw. setzen.
- Die Klasse `TextField` wird dazu benutzt, um einzeiligen, auswählbaren Text darzustellen. Je nach auszuführender Funktion kann es dem Anwender erlaubt werden, den Text selbst zu änderen oder nicht.

  Die vier Konstruktoren der Klasse `TextField` sind `TextField`( ), `TextField`(int), `TextField`(String) und `TextField`(String, int). Das int-Argument des zweiten und vierten Konstruktors wird dazu benutzt, die Breite des Textfeldes anzugeben. Mit dem String-Parameter des dritten und vierten Konstruktors wird ein voreingestellter Text spezifiziert.
- Die Klasse `TextField` ist eine Unterklasse der Klasse `TextComponent`. Von dieser Klasse erbt sie unter anderem die Methode setEditable, die es ihr erlaubt, das Ändern der Texte zuzulassen bzw. zu verbieten, oder die Methode setText, die den darzustellenden Text festlegt.

3.2.2. Die Klasse `RegExpr`

Die Klasse `RegExpr` ist das Kernstück des Programms. Sie enthält die Methode main, die aufgerufen wird, wenn die Anwendung gestartet wird. In ihr ist die graphische Benutzeroberfläche definiert und für jeden der beiden zu vergleichenden *regulären*

Ausdrücke r und s werden hier die Objekte der Klassen `InfixPostfix`, `BinTree` und `Automat` erzeugt und verarbeitet.
Zuletzt werden dann die beiden referenzierten Übergangstabellen des `Automat`-Objekts auf Gleichheit überprüft. Sind diese gleich, so kann man daraus schließen, daß die ihnen zugrundeliegenden *regulären Ausdrücke* r und s äquivalent sind.

3.2.2.1. Die Oberfläche

Die Oberfläche wird in der Klasse RegExpr definiert, die von der Klasse Frame abgeleitet ist. Als Layoutverwalter für die Oberfläche wurde ein GridLayout-Objekt benutzt.

Folgende Komponenten werden im Oberflächenfenster verwendet:

LabelRegExpr1: Nicht anwählbares und nicht editierbares `Label`-Objekt. Die Beschriftung des `Labels` enthält die Aufforderung zur Eingabe des ersten zu bearbeitenden regulären Ausdrucks.

InputRegExpr1: Eingabefeld der Klasse TextField. Dieses TextField-Objekt ist editierbar.

LabelRegExpr2: Textfeld der Klasse `Label`. Die Beschriftung dieses `Labels` enthält die Aufforderung zur Eingabe des zweiten zu bearbeitenden regulären Ausdrucks.

InputRegExpr2: Eingabefeld der Klasse `TextField`. Dieses `TextField`-Objekt ist editierbar.

FileRegExpr : Schaltfläche der Klasse `Button`. Erzeugt ein Dialogfenster der Klasse `FileDialog`.

CompareRegExpr: Schaltfläche der Klasse `Button`.

FileRegExpr und CompareRegExpr teilen sich eine GridLayout-Komponente und sind dazu in einem Panel-Objekt zusammengefaßt. Um das TextField-Objekt und das Button-Objekt trennen zu können, benötigt das Panel-Objekt zwei Komponenten und verwendet dazu wiederum ein GridLayout.

ResultRegExpr: Ausgabefeld der Klasse `TextField`. Dieses `TextField`-Objekt ist nicht editierbar.

ExitRegExpr: Schaltfläche der Klasse `Button`.

3.2.2.2. Das Dialogfenster

Das Dialogfenster der Klasse `FileDialog` ist eine Unterklasse der Klasse `Dialog`

Das Dialogfenster verwendet folgende Komponenten:

LabelPath : Textfeld der Klasse `Label`. Die Beschriftung dieses `Labels` enthält die Aufforderung zur Eingabe des Verzeichnispfades.

InputPath : Eingabefeld der Klasse `TextField`. Dieses `TextField`-Objekt ist editierbar und erwartet die Angabe des Verzeichnispfades.

LabelFilter : Textfeld der Klasse `Label`. Die Beschriftung dieses `Labels` enthält die Aufforderung zur Eingabe des Filters.

InputFilter : Eingabefeld der Klasse `TextField`. Dieses `TextField`-Objekt ist editierbar und erwartet die Angabe des Filters.

LabelFileText : Textfeld der Klasse `Label`. Die Beschriftung dieses `Labels` enthält die Aufforderung zur Eingabe der zu öffnenden Datei.

InputFileText : Eingabefeld der Klasse `TextField`. Dieses

`TextField`-Objekt ist editierbar und erwartet die Angabe des Dateinamens.

LabelFileList : Textfeld der Klasse `Label`. Die Beschriftung dieses `Labels` enthält die Aufforderung zum anklicken der jeweiligen Datei in der Liste.

InputFileList : Eingabefeld der Klasse `List`. Erwartet ein ActionEvent, ausgelöst durch ein Doppelklick in der Zeile der jeweiligen auszuwählenden Datei.

LabelFolderList : Textfeld der Klasse `Label`. Die Beschriftung dieses `Labels` enthält die Aufforderung zum anklicken des jeweiligen Folders in der Liste.

InputFolderList : Eingabefeld der Klasse `List`. Erwartet ein ActionEvent, ausgelöst durch ein Doppelklick in der Zeile des jeweiligen auszuwählenden Folders.

ButtonOk : Schaltfläche der Klasse Button.

ButtonUpdate : Schaltfläche der Klasse Button.

ButtonCancel : Schaltfläche der Klasse Button.

3.2.3. Die Klasse BinTree

Mit der Ausführung der Klasse `BinTree` wird für den jeweiligen *regulären Ausdruck* ein *Syntaxbaum* konstruiert. Für jeden einzelnen Knoten des *Syntaxbaums* werden dann die weiter oben in Abschnitt 3.1. schon besprochenen vier Methoden Nullable, Firstpos, Lastpos und Followpos ausgeführt.

Die Konstruktormethode BinTree(char[ ] post, int[ ] blattpos) erwartet zwei Argumente.

- Das char[ ]-Argument post erwartet einen Array mit dem *regulären Ausdruck* in Postfix-Notation.
- Dem int[ ]-Argument blattpos wird ein Array zugewiesen, welches die Positionen speichert, die die einzelnen *Symbole* im *regulären Ausdruck* einnehmen. Diese

Positionen werden bei der Konstruktion des *Syntaxbaums* den Knoten zugeordnet, die die Blätter im *Syntaxbaum* einnehmen. Zu Beginn des Abschnitts 3.1. wird dieser Sachverhalt ausführlich behandelt.

Wichtige Methoden der Klasse `BinTree` sind die Methoden Built, Traverse, Nullable, Firstpos, Lastpos und Followpos.

- Die Methode Built ist die Methode, die den Aufbau des *Syntaxbaums* veranlaßt. Die Methode Built wird jedesmal rekursiv aufgerufen, wenn beim Auslesen des Array´s post, welches den *regulären Ausdruck* in Postfix-Notation speichert, ein Operator gelesen wird.

  Das Array post wird in umgekehrter Reihenfolge, von hinten nach vorne ausgelesen. Der erste durch Built erzeugte Knoten, die Wurzel des *Syntaxbaums,* ist demnach der letzte Operator im Postfix-Array. Alle weiteren Knoten werden im Anschluß daran solange durch rekursive Aufrufe als Wurzel eines Unterbaums in den rechten Teilbaum eigefügt, wie Operatoren aus dem Array post ausgelesen werden. Falls *Symbole* ausgelesen werden starten die rekursiven Aufrufe für den linken Teilbaum.
- In der Methode Traverse wird der *Syntaxbaum* in Postorderreihenfolge durchlaufen. Postorderreihenfolge heißt, daß zuerst der linke Teilbaum besucht wird, dann wird der rechte Teilbaum besucht und zuletzt wird die Wurzel besucht. Dabei werden nacheinander die Methoden Nullable, Firstpos, Lastpos und Followpos, wie im Abschnitt 3.1. beschrieben, abgearbeitet. Die Struktur eines Knotens im *Syntaxbaum* wird in Abschnitt 3.3.1. ausführlich erläutert.

  Die Boolean-Werte von Nullable sowie die Integer-Array`s von Firstpos und Lastpos werden den jeweiligen Komponenten in den Knoten des *Syntaxbaums* zugewiesen.

  Die Methode Followpos ist nicht auf den Knoten des *Syntaxbaums*, sondern auf der Menge der Positionen des Syntaxbaums definiert. Aus diesem Grund ist für Followpos auch keine Komponente in den Knoten des *Syntaxbaums* vorgesehen. Die Ergebnisse von Followpos müssen daher in einer anderen Datenstruktur gespeichert werden. Dazu wird die Matrix followpos [ ][ ] definiert.

3.2.4. Die Klasse Automat

Mithilfe der Klasse `Automat` kann man nun umgehend einen *deterministischen endlichen Automat* erstellen.

Für die Klasse `Automat` steht der Konstruktor Automat(int[] first, int[ ][ ] follow, char[ ] blatt, char[ ] symbol) zur Verfügung, der vier Argumente erwartet.

- Das int[ ]-Argument first dient zur Aufnahme des Array`s firstpos[ ], der die Ergebnismenge der Methode Firstpos enthält, wenn diese für den Knoten der Wurzel des zugehörigen *Syntaxbaums* ausgeführt wird. Aus first wird dann erste Zustand des *DEA*`s kontruiert
- Der Parameter int[ ][ ] follow erwartet die in Abschnitt 3.1.4. beschriebene Ergebnismatrix der Methode Followpos. In follow sind für jedes *Symbol* des *regulären Ausdrucks* bzw. des *Syntaxbaums* alle möglichen Folgepositionen enthalten, die auf diese *Symbole* im zugehörigen *Syntaxbaum* folgen können. Aus diesen Folgepositionen können, wie in Abschnitt 3.1.5. erläutert, die Zustandsübergänge für den *DEA* hergeleitet werden.
- An den dritten Parameter char[ ] blatt werden alle *Symbole*, die die Blätter des zugehörigen *Syntaxbaums* repräsentieren, übergeben.
- Der vierte Parameter char[ ] symbol erwartet ebenfalls die Menge der *Symbole,* die im zugehörigen *Syntaxbaum* Blätter darstellen, aber vermindert um deren Duplikate.

Die wesentlichen Methoden der Klasse `Automat` sind die Methode Tab und die Methode TMKonstrukt.

- In der Methode Tab wird *Dstates,* die Menge der Zustände des *deterministischen endlichen Automats* konstruiert und darauf aufbauend kann *Dtran,* die Übergangsttabelle des *DEA,* bestimmt werden
- Die Methode TMKonstrukt, führt die Teilmengenkonstruktion für den *deterministischen endlichen Automat* durch, infolgedessen die Zustandsmenge des *DEA`s* minimiert wird.

Die Vorgehensweise der beiden Methoden wurde bereits in Abschnitt 3.1.5. beschrieben.

3.3. Verwendete Datenstrukturen

3.3.1. Syntaxbaum

Die Datenstruktur *Syntaxbaum* wird dazu verwendet, den *regulären Ausdruck* in seine einzelnen Bestandteile (*Symbole* und Operatoren) zu zerlegen. Dazu ist für jedes *Symbol* und jeden Operator des *regulären Ausdrucks* genau ein Knoten im *Syntaxbaum* reserviert. Ein Knoten im *Syntaxbaum* enthält aber nicht nur die Information darüber, ob ein *Symbol* oder ein Operator vorliegt, er beinhaltet vielmehr einen ganzen Satz an Komponenten, der das Verhalten des *Syntaxbaums* während der Ausführung bestimmt.

Nachfolgend wird die Struktur eines Knotens im *Syntaxbaum* aufgezeigt:

```
-------------------------------------------------------------------------------------------------
--
char info;                                  // Symbol des Alphabets bzw. Operator
int position;                  // Position dieses Symbols / Operators im Syntaxbaum
Node parent;                                // Referenz zum Vaterknoten
Node vorgänger;                             // Referenz zum Vorgänger
Node nachfolger;                            // Referenz zum Nachfolger
boolean nullable;                  // Rückgabewert der Methode Nullable
int[] firstpos;                     // Rückgabewert der Methode Firstpos
int[] lastpos;                      // Rückgabewert der Methode Lastpos
-------------------------------------------------------------------------------------------------
--
```

- Die erste Komponente deklariert die Variable *info* des Typs Character. Diese Variable wird zur Aufnahme der Daten verwendet, die während der Programmausführung vom *regulären Ausdruck* gelesen werden. Das sind eben die *Symbole* (a,..,z) und Operatoren des *regulären Ausdrucks*.
- Die zweite Komponente *position* ist eine Variable vom Typ Integer und spezifiziert die Positionen, welche die *Symbole* im *regulären Ausdruck* einnehmen. Da die

Werte für Positionen und *Symbole* bei der Konstruktion des *Syntaxbaums* miteinfließen, wird durch *position* auch die Position eines Blattes und des zu diesem Blatt gehörenden *Symbols* im *Syntaxbaum* bezeichnet.

Ein *Symbol*, welches im *regulären Ausdruck* die Position 1 einnimmt, nimmt auch im *Syntaxbaum* die Position 1 ein.

Operatoren werden keine Positionen zugewiesen. Enthält etwa die Variable *info* einen der Operatoren '+', '*' oder '|', so wird der Variablen *position* der Wert 0 zugewiesen.

- Die nächsten drei Komponenten sind vom Typ Node und definieren drei weitere Knoten in *Syntaxbaum*. Dabei repräsentiert *parent* eine Referenz zum Vaterknoten des gerade betrachteten Knotens im *Syntaxbaum, vorgänger* und *nachfolger* sind jeweils Referenzen zum Vorgänger- und zum Nachfolgerknoten dieses Knotens im *Syntaxbaum.*
- Die Komponente *nullable* des Typs boolean repräsentiert den Rückgabewert der Methode Nullable. Die Methode Nullable wird für jeden Knoten des *Syntaxbaums* aufgerufen und testet, ob dieser Knoten die Wurzel eines Teilausdrucks ist, der eine Sprache mit dem leeren Wort {ε} erzeugen kann. Wenn der Knoten ε erzeugen kann, dann ist der Rückgabewert von Nullable 'true', sonst 'false'.
- Die beiden letzten Komponenten *firstpos* und *lastpos* sind vom Typ Array und haben nur Elemente des Typs Integer. *firstpos* und *lastpos* enthalten die Rückgabewerte der Methoden Firstpos und Lastpos. Firstpos liefert die Menge aller Positionen, die auf das erste *Symbol* eines *Wortes* passen können, der vom Teilausdruck mit Wurzel *info* erzeugt wird. Lastpos liefert die Menge all der Positionen, die auf das letzte *Symbol* eines solchen *Wortes* passen können.

3.3.2. Array:

Die Datenstruktur *Array* wird sehr häufig verwendet. Die für die Anwendung wesentlichen *Array's* werden hier kurz beschrieben.

Die Methode InfixPostfix verwendet einen *Array* als Rückgabewert. In dieser Methode wird der *reguläre Ausdruck* als Eingabestring, der in Infixnotation an die Methode übergeben wird, ausgelesen und dann in Postfixnotation in das Array *infixpostfix*

hineingeschrieben. Das *infixpostfix*-Array dient später als Übergabewert für den *Syntaxbaum*.
Die Methoden Firstpos und Lastpos, die im obigen Abschnitt schon eingeführt wurden, verwenden beide jeweils einen *Array* als Rückgabewert während des Baumdurchlaufes. Die Datentypen der Elemente der beiden verwendeten *Array´s* sind Integerwerte.

---

char[ ] infixpostfix;
int[ ] firstpos;
int[ ] lastpos;

3.3.3. Matrix:

Die Datenstruktur *Matrix* wird verwendet, um *Dstates* und *Dtran* darzustellen. *Dstates* definiert die Menge der Zustände des DEA, *Dtran* ist die Übergangstabelle für den DEA. Die Datentypen der Elemente beider verwendeter *Matrizen* sind Integerwerte.
Die Methode Followpos verwendet ebenfalls eine *Matrix,* und zwar die *Matrix followpos*. Die Methode Followpos gibt an, welche Positionen im *Syntaxbaum* auf die aktuelle Position *info* folgen können. Die Datentypen der Elemente der *Matrix* sind ebenfalls Integerwerte.

---

int[ ][ ] Dstates;
int[ ][ ] Dtran;
int[ ][ ] followpos;

---

4. Beispielverarbeitung eines regulären Ausdrucks

Gegeben ist der *reguläre Ausdruck* mit der eindeutigen Endemarkierung # .

((a | b) * + a + b + #)

(1) Im ersten Schritt wird der *reguläre Ausdruck*, der in Infix-Notation vorliegt, in Postfix-Notation überführt.

a b | * a + b + # +

(2) Im zweiten Schritt werden für die im *regulären Ausdruck* vorkommenden *Symbole* deren Positionen bestimmt

a b | * a + b + # +
1 2 3 4 5

(3) Im nächsten Schritt wird der *Syntaxbaum* für den *regulären Ausdruck* erstellt. Die Bätter des *Syntaxbaum* sind mit den *Symbolen* des *Alphabets* markiert. Die Positionen der *Symbole* stehen im *Syntaxbaum* in Abbildung 4.1 unter den *Symbolen*. Die Operatoren Konkatenation(+), Oder-Verknüpfung(|) und Stern-Operator(*) nehmen die inneren Knoten des *Syntaxbaums* ein.

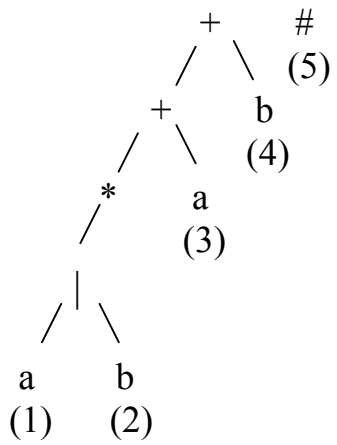


Abbildung 4.1: Syntaxbaum mit den Positionen der Symbole für den regulären Ausdruck ((a | b)* + a + b + #).

Nullable, Firstpos, Lastpos und Followpos werden dann durch Depth-First-Abarbeitung des Syntaxbaums ausgeführt.

(4) Nullable:

Wenn ein gegebenen Knoten das leere Wort { ε } erzeugen kann, dann liefert Nullable den Wert ‘true‘ zurück, ansonsten den Wert ‘false‘.

- Nach Regel (1) liefert Nullable den Wert ‘false‘, falls der betrachtete Knoten ein mit einem Symbol des Alphabets markiertes Blatt ist. Die Blätter des Syntaxbaums erhalten deshalb alle den Wert ‘false‘.
- Für die Oder-Verknüpfung(|) liefert Nullable nach Regel(4) den Wert ‘false‘ zurück, wenn einer der Operanden ‘false‘ ist. In diesem Fall sind beide Operanden ‘false‘. Der Oder-Knoten liefert den Wert ‘false‘.
- Der Stern-Operator(*) darüber kann das leere Wort { ε } erzeugen. Er liefert den Wert ‘true‘.
- Die drei Konkatenationsoperatoren liefern den Wert ‘false‘, wenn mindestes einer ihrer Operatoren den Wert ‘false‘ liefert. Für die erste Konkatenation liefert zwar der Teilausdruck (a|b)* des Vorgängers im linken Teilbaum den Wert ‘true‘, aber das Blatt a des Nachfolgers im rechten Teilbaum liefert den Wert ‘false‘. Damit ist auch der Nullable-Wert für diesen Knoten ‘false‘.

- Die beiden folgenden Konkatenationsoperationen haben somit nur Operanden die den Wert ‘false‘ liefern und sind somit ebenfalls ‘false‘.

In der folgenden Abbildung 4.2 des *Syntaxbaums* für den *regulären Ausdruck* ((a | b)* + a + b + #) sind die Ergebnisse der Methode Nullable an die Knoten des *Syntaxbaums* geschrieben.

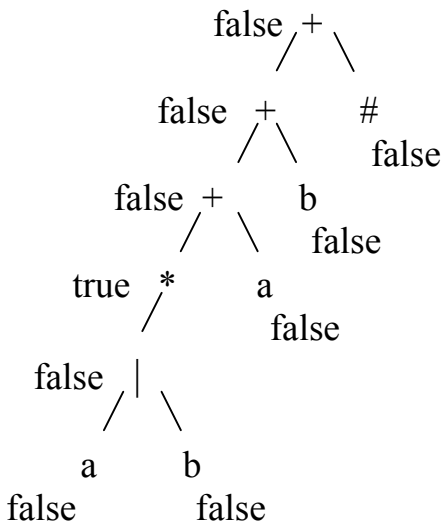


Abbildung 4.2: Syntaxbaum mit Nullable-Werten für den regulären Ausdruck ((a | b)* + a + b + #).

(5) Firstpos:

Firstpos liefert also, wie in Abschnitt 3.1.2. erwähnt, die Menge aller Positionen eines *Teilwortes*, die an erster Stelle dieses *Teilwortes* möglich sind. Abbildung 4.3 zeigt die Ergebnisse der Methode Firstpos.

- Die mit einem *Symbol* des *Alphabets* markierten Blätter des *Syntaxbaums* übernehmen nach Regel (1) für Firstpos alle den Wert ihrer Position.
- Nach Regel (4) von Firstpos übernimmt der Oder-Knoten sowohl die Firstpos-Werte, die Menge { 1 }, des Vorgängers im linken Teilbaum (a) als auch die Firstpos-Werte, die Menge { 2 }, des Nachfolgers im rechten Teilbaum (b). Die Menge Firstpos dieses Knotens ist die Menge { 1,2 }.

- Der Stern-Operator erhält nach Regel (2) die Firstpos-Menge seines Vorgängers im linken Unterbaum{ 1,2 }.
- Für die erste Konkatenationsoperation liefert der Vorgänger im linken Unterbaum für Nullable den Wert ‘true‘. Somt ist nach Regel (3b) die Menge Firstpos dieses Knoten die Vereinigungsmenge der beiden Firstpos-Mengen des Vorgängers im linken Teilbaum{ 1,2 } und des Nachfolgers im rechten Teilbaum{ 3 }. Firstpos der Konkatenation ist also { 1,2,3 }.
- Die zweite und die dritte Konkatenation übernehmen nach Regel (3a) die Firstpos-Werte des Vorgängers im linken Teilbaum{ 1,2,3 }.

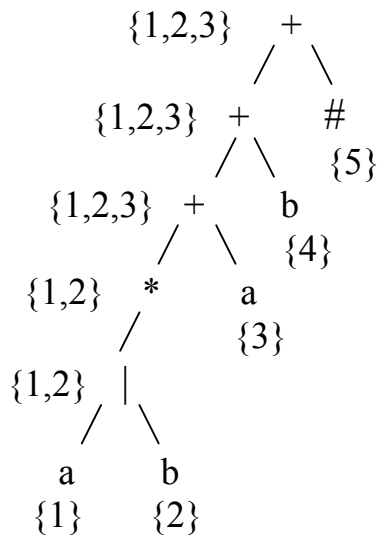


Abbildung 4.3: Syntaxbaum mit Firstpos-Werten für den regulären Ausdruck ((a | b)* + a + b + #).

(6) Lastpos:

Lastpos liefert, wie in Abschnitt 3.1.3. beschrieben, die Menge aller Positionen eines *Teilwortes*, die an letzter Stelle dieses *Teilwortes* möglich sind.

- Somit haben die mit einem *Symbol* des *Alphabets* markierten Blätter des *Syntaxbaums* nach Regel (1) von Lastpos für ihre Position im *Syntaxbaum* und für die Menge Lastpos immer den gleichen Wert.
- Nach Regel (4) von Lastpos übernimmt der Knoten, mit der Oder- Verknüpfung als Lösungsmenge für Lastpos die Lastpos-Mengen seiner beiden Operanden. Es ergibt sich die Menge { 1,2 }.
- Nach Regel (2) von Lastpos erhält der Stern-Operator die Lastpos-Menge seines Vorgängers im linken Unterbaum{ 1,2 }.
- Die drei Konkatenationsoperatoren übernehmen nach Regel (3a) von Lastpos jeweils die Lastpos-Werte der jeweiligen Nachfolger im rechten Teilbaum. Für die erste Konkatenation ergibt sich so die Menge { 3 } für die zweite die Menge { 4 } und für die dritte die Menge { 5 }.

Die Ergebnisse der Methode Lastpos sind in Abbildung 4.4 an die Knoten des *Syntaxbaums* geschrieben.

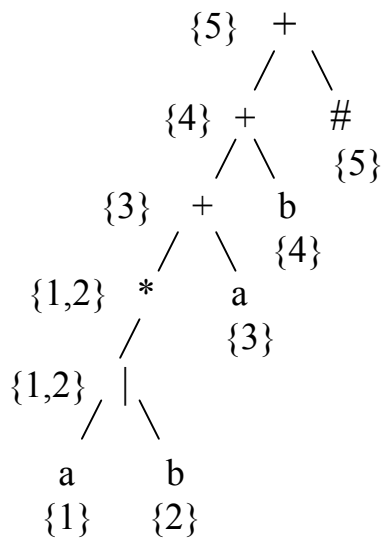


Abbildung 4.4: Syntaxbaum mit Lastpos-Werten für den regulären Ausdruck ((a | b)* + a + b + #).

(7) Followpos:

Followpos bestimmt, wann ein *Symbol* auf ein anderes folgen kann und tritt immer dann in Aktion, wenn ein Stern-Operator oder eine Konkatenationsoperation vorliegt.

- Während der Depth-First Abarbeitung des *Syntaxbaums* ist der erste zu betrachtende Knoten der Stern-Operator(*). Bei der Stern-Operation werden gemäß Regel (3) von Followpos jeder Position in Lastpos alle Positionen in Firstpos zugewiesen. Das bedeutet, daß zu followpos[1] und zu followpos[2] die Positionen 1 und 2 hinzugefügt werden.
- Beim Vaterknoten des Stern-Operators, dem ersten Konkatenationsoperator, werden nach Regel (2) von Followpos jeder Position von Lastpos des Vorgängers im linken Unterbaum, also Lastpos von dem Stern-Operator, alle Positionen in Firstpos des Nachfolger im rechten Unterbaum, also Firstpos von dem Blatt (a), zugewiesen Die Menge Lastpos vom Stern-Operator ist die Menge { 1,2 } und die Menge Firstpos von dem Blatt (a) ist die Menge {3}. Also wird zu followpos[1] und zu followpos[2] die Position 3 hinzugefügt.
- Beim zweiten Konkatenationsoperator wird genau so vorgegangen. Die Menge Lastpos des Vorgängers ist die Menge { 3 }und die Menge Firstpos des Nachfolger ist die Menge { 4 }. Das bedeutet, zu followpos[3] wird die Position 4 hinzugefügt.
- Ebenso beim dritten Konkatenationsoperator. Die Menge Lastpos des Vorgängers ist die Menge { 4 }und die Menge Firstpos des Nachfolger ist die Menge { 5 }. Das bedeutet, zu followpos[4] wird die Position 5 hinzugefügt.

In der nachfolgenden Tabelle 4.1 werden die Ergebnisse der Methode Followpos zusammengefaßt.

Knoten	Followpos
1	{ 1,2,3 }
2	{ 1,2,3 }
3	{ 4 }
4	{ 5 }
5	-----

Tabelle 4.1: Ergebnisse der Methode Followpos.

(8) Der *deterministische endliche Automat*:

Im letzten Schritt wird jetzt der *deterministische endliche Automat* erstellt. Dazu müssen zuerst die beiden Mengen *Dstates* und *Dtran* konstruiert werden.

Dstates definiert die Menge der Zustände des *DEA* und *Dtran* ist die Übergangstabelle für den *DEA*.

Wie in Abschnitt 3.1.5. nachzulesen ist, sind die Zustände in *Dstates* aus den Mengen der Positionen, welche *Symbole* im *regulären Ausdruck* repräsentieren, hergeleitet.

Um *Dstates* zu erstellen, wird zuerst die Menge Firstpos der Wurzel des zugehörigen *Syntaxbaums* bestimmt. Die Positionen dieser Menge erzeugen den ersten Zustand in *Dstates.*

Für diesen und für jeden neu hinzukommenden Zustand von *Dstates* werden nun, für alle Positionen dieses Zustands, die ein bestimmtes *Symbol* des *regulären Ausdrucks* repräsentieren, die Menge der Folgepositionen mittels Followpos bestimmt und miteinander vereinigt. Daraus ergibt sich dann ein neuer Zustand, der Folgezustand des gerade betrachteten Zustands für dieses bestimmte *Symbol*.

Ist dieser Zustand noch nicht in der Menge *Dstates* enthalten, so wird er *Dstates* hinzugefügt.

Zur Konstruktion von *Dtran* werden für jedes dieser *Eingabesymbole*, welche durch eine Position in einem Zustand von *Dstates* repräsentiert wird, alle Übergänge zu anderen Zuständen bestimmt und zu *Dtran* hinzugefügt.

Dstates und Dtran werden auf folgende Weise bestimmt:

- Der erste Zustand in *Dstates* wird aus der Menge Firstpos der Wurzel des zugehörigen *Syntaxbaums* hergeleitet. Die Wurzel des *Syntaxbaums* wird durch den zweiten Konkatenationsoperator repräsentiert und enthält für Firstpos die Menge {1,2,3}.
- Für jedes *Symbol*, welches durch eine Position in diesem Zustand repräsentiert ist, werden nun alle Übergänge zu anderen Zuständen bestimmt.
- Zuerst wird das *Symbol* a betrachtet. Das *Symbol* a wird durch die Positionen 1 und 3 repräsentiert. Der Zustand { 1,2,3 } hat also für das *Symbol* a den Übergang followpos[1] $\cup$ followpos[3] = {1,2,3 } $\cup$ { 4 } = { 1,2,3,4 }. *Dtran* erhält so seinen ersten Übergang: *Dtran*[{1,2,3 } ; a] = { 1,2,3,4 }.
- Der Zustand { 1,2,3,4 } ist noch nicht in *Dstates* enthalten. Also wird er als neuer Zustand zu *Dstates* hinzugefügt.
- Das nächste zu betrachtende *Symbol* für den Zustand { 1,2,3 }ist das *Symbol* b. Das *Symbol* b wird durch die Position 2 repräsentiert. D.h., der Zustand {1,2,3} hat für das *Symbol* b den Übergang followpos[2] ={1,2,3}. Also einen Zustand zu sich

selbst. Da der Zustand {1,2,3} schon in *Dstates* vorhanden ist, wird er auch nicht mehr zu *Dstates* hinzugefügt. *Dtran* erhält seinen zweiten Übergang:

Dtran[{1,2,3 } ; b] = { 1,2,3 }.

- Jetzt sind alle Positionen im Zustand { 1,2,3 } abgearbeitet. Da *Dstates* einen weiteren Zustand, den Zustand { 1,2,3,4 } enthält, werden alle Positionen für diesen Zustand betrachtet.
- Zunächst wieder *Symbol* a , welches durch die Positionen 1 und 3 repräsentiert wird. Der Zustand { 1,2,3,4 } hat also für das *Symbol* a den Übergang

  followpos[1] ∪ followpos[3] = {1,2,3 } ∪ { 4 } = { 1,2,3,4 }.

  Dies ist wieder ein Übergang zu sich selbst. Der dritte Übergang für *Dtran* ist also:

  Dtran[{1,2,3,4 } ; a] = { 1, 2,3,4 }.
- Das *Symbol* b wird durch die Positionen 2 und 4 repräsentiert. Also hat der Zustand { 1,2,3,4 } für das *Symbol* b den Übergang

  followpos[2] ∪ followpos[4] = {1,2,3 } ∪ { 5 } = {1,2,3,5}.

  Dieser Zustand existiert noch nicht in *Dstates,* wird also *Dstates* hinzugefügt. Der nächste Übergang für *Dtran* ist dann:

  Dtran[{1,2,3,4 } ; b] = { 1,2,3,5 }.
- Jetzt sind wieder alle *Symbole* in {1,2,3,4 } abgearbeitet. Der nächste noch zu betrachtende Zustand ist der Zustand { 1,2,3,5 }.
- Für *Symbol* a ergibt sich der Übergang

  followpos[1] ∪ followpos[3] = {1,2,3 } ∪ { 4 } = { 1,2,3,4 }.

  Dieser Zustand ist schon in *Dstates*. *Dtran* erhält den Übergang:

  Dtran[{1,2,3,5 } ; a] = { 1,2,3,4 }.
- Für *Symbol* b ergibt sich der Übergang followpos[2] = { 1,2,3 }. Dieser Zustand ist ebenfalls schon in *Dstates. Dtran* erhält den Übergang:

  Dtran[{1,2,3,5 } ; b] = { 1,2,3 }.
- Das erreichen der Endmarkierung # stellt einen Übergang zu einem akzeptierenden Zustand dar. Auf die Endmarkierung kann kein weiteres *Symbol* mehr folgen. Also gibt es auch keinen Folgezustand mehr, der nach dem Zustand {1,2,3,5 } folgen könnte.
- Alle Zustände in *Dstates* sind jetzt abgearbeitet. Damit ist der *deterministische endliche Automat* fertig.

Tabelle 4.2 zeigt *Dtran,* die Übergangstabelle des *deterministischen endlichen Automats*, für den *regulären Ausdruck* ((a | b)* + a + b + #). Die Werte von *Dstates,* der Menge der Zustände des *DEA,* sind in der ersten Spalte von *Dtran* abgetragen.

Dtran		
Dstate	a	b
{ 1,2,3 }	{ 1,2,3,4 }	{ 1,2,3 }
{ 1,2,3,4 }	{ 1,2,3,4 }	{ 1,2,3,5 }
{ 1,2,3,5 }	{ 1,2,3,4 }	{ 1,2,3 }

Tabelle 4.2: Übergangstabelle des deterministischen endlichen Automats für den regulären Ausdruck ((a | b)* + a + b + #).

5. Abschließende Bemerkung

In der vorliegenden Studienarbeit wurde eine mögliche Implementierung der Überprüfung zweier *regulärer Ausdrücke* r und s auf Äquivalenz aufgezeigt. Dabei wird für jeden der beiden zu vergleichenden *regulären Ausdrücke* r und s direkt, d.h. ohne den Umweg über einen *nichtdeterministischen endlichen Automat*, der zugehörige *deterministische endliche Automat* R bzw. S konstruiert und reduziert.

Bei Gleichheit der beiden reduzierten DEA´s R und S kann man folgern, daß die beiden *regulären Ausdrücke* r und s, aus denen sie hergeleitet wurden äquivalent sind.

Für jeden der beiden *deterministischen endlichen Automaten* wird die Übergangsfunktion durch eine Übergangstabelle, und zwar *Dtran,* implementiert. Das bedeutet, daß für die jeweilige Eingabe der *regulären Ausdrücke* r und s eine

Laufzeit des DEA`s bestimmt werden kann, die sich proportional zur Länge des jeweiligen zugrundeliegenden *regulären Ausdrucks* verhält und damit unabhängig von der Anzahl der Zustände des DEA`s ist[14].

Bei der Ermittlung des Platzbedarfs muß allerdings berücksichtigt werden, daß es *reguläre Ausdrücke* gibt, für welche die Anzahl der Zustände des *deterministischen endlichen Automaten* im ungünstigsten Fall exponentiell mit der Größe des regulären Ausdrucks wächst. Dieser Fall tritt dann auf, wenn jeder Zustand des betreffenden DEA`s für jedes Symbol des Eingabealphabets einen Übergang zu einem anderen Zustand erzeugen muß. Die Übergangsfunktion des jeweiligen DEA`s stellt sodann eine Abbildung von der Menge der Zustände des DEA verknüpft mit allen Symbolen des Eingabealphabets in die Potenzmenge der Menge der Zustände des DEA dar[15].

[14] Vgl. Aho et al. 1988.

[15] Vgl. ebenda.