Lektion 15

In der vorigen Lektion wurde vorgestellt, wie zu jedem regulären Ausdruck ein äquivalenter ε-NEA und ggf. dann auch ein DEA (mittels Teilmengenkonstruktion) gebildet werden kann. Eine Anwendung dafür sind sog. Scanner-Generatoren, die es erlauben, den Programmcode für die lexikalische Analyse, z.B. bei der Implementierung von Programmiersprachen, automatisch zu generieren.

Wir betrachten auch in umgekehrter Richtung den Zusammenhang zwischen regulären Ausdrücken und endlichen Automaten: Wie kann man von einem ggf. sehr "unstrukturierten" Automaten zu einem "wohlstrukturierten" regulären Ausdruck kommen? Es zeigt, sich dass dies immer möglich ist. Auch der Zusammenhang zwischen endlichen Automaten und Chomsky-Typ-3-Sprachen wird am Ende der Lektion noch untersucht.

9.1.3 Ausblick: Scanner und Scannergeneratoren

[Dieser Abschnitt ist für die Klausur nur im Überblick relevant, aber als Hintergrundwissen für Informatiker hilfreich.]

Im ersten Kapitel wurde schon der prinzipielle Aufbau eines Compilers vorgestellt (auch bei Interpretern ist das Frontend gleich). Reguläre Sprachen und die damit verbundenen Beschreibungs- und Analysemöglichkeiten bilden die Grundlage für die *lexikalische Analyse*.

Lexikalische Analyse im Compilern und Interpretern

Die lexikalische Analyse zerlegt den Programmtext in einzelne Teilabschnitte:

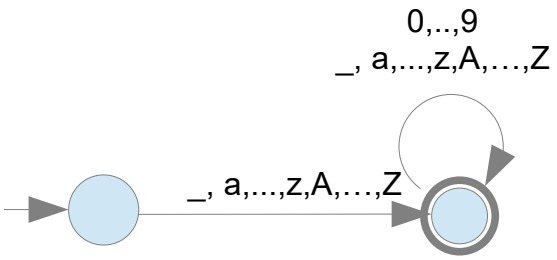
- sog. *Token* als grundlegende syntaktische Einheiten, z.B. Zahlen, Bezeichner, oder spezielle Symbole.
- sog. *White spaces*, die überlesen werden, z.B. Leerzeichen und Kommentare.

Da die lexikalische Analyse den möglicherweise sehr langen Programmquelltext Zeichen für Zeichen verarbeiten muss, muss die Verarbeitung sehr effizient erfolgen. Dies kann durch deterministische endliche Automaten (DEAs) realisiert werden.

Beispiel 9.12: Bezeichnern in C/C++ als DEA

Der zulässige Aufbau von Bezeichner (z.B. Namen von Variablen, Methoden etc.) ist für C/C++ folgendermaßen definiert: Ein Bezeichner kann mit einem Buchstaben oder mit einem Unterstrich beginnen. Danach können noch beliebig viele Ziffern, Buchstaben oder Unterstriche folgen.

Dies kann durch folgenden DEA beschrieben werden.

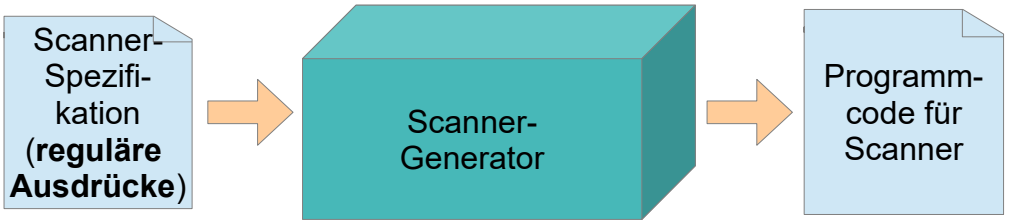


Ein entsprechender Automat könnte dann implementiert und als effizientes Verfahren zur Erkennung entsprechender Programmteile verwendet werden.

Allerdings wäre es sehr mühsam, solche Automaten direkt zu definieren. Viel bequemer ist es, den Aufbau von lexikalischen Einheiten durch reguläre Ausdrücke zu beschreiben.

Scanner-Generator

Die Theorie der formalen Sprachen erlaubt es nun, sog. *Scanner*-Generatoren zu realisieren. Ein Scanner-Generator (oder auch *Lexer*-Generator) ist ein Programm, das aus einer Spezifikation der lexikalischen Einheiten einer Sprache (d.h. der Token und White Spaces) den Programmcode für die lexikalische Analyse automatisch generiert. Bei der Scanner-Spezifikation werden reguläre Ausdrücke verwendet, um den Aufbau der Token und White-Spaces zu definieren.



Einige gängige Scanner-/Lexer-Generatoren sind z.B.:

- **lex**, **flex**: Seit Jahrzehnten der "Klassiker", generiert C-Code. *flex* ist die GNU-Variante des proprietären *lex*. Es sind auch portierte Versionen für viele andere Sprachen verfügbar, z.B. *jLex* und *JFlex* für Java oder *PLY* für Python.
- **ANTLR** (ANother Tool for Language Recognition): Kombinierter Scanner- und Parser-Generator, generiert u.a. Java, JavaScript, Python, C#, …
  Weitere Infos siehe https://www.antlr.org/
- **javacc**: Kombinierte Scanner- und Parser-Generator: generiert Java (wird auch für die Java-Implementierung selbst verwendet).
  Weitere Infos siehe https://github.com/javacc/javacc/releases/tag/7.0.5

Beispiel 9.13 - Eingabe für Scanner-Generator ANTLR

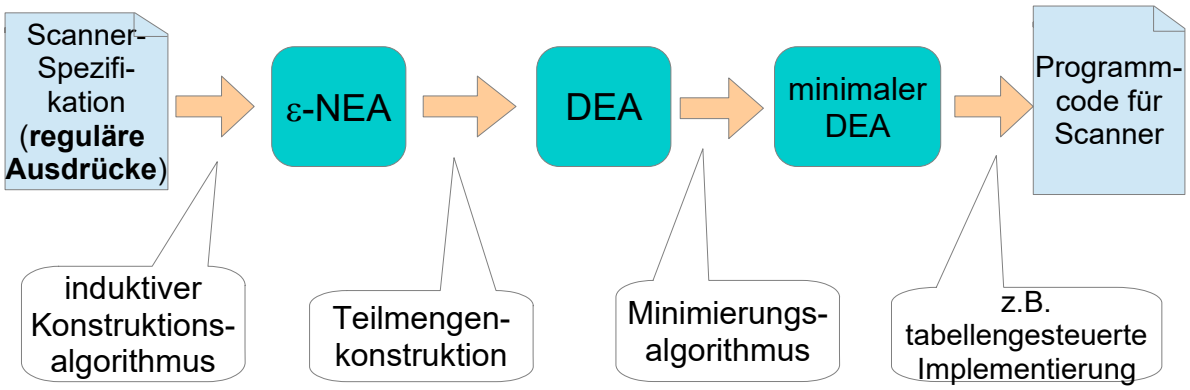
- Spezifikation einer Zahl in C

```
Number:( Digit )+ ( '.'( Digit )* Exponent)?| Exponent)
                      ( FloatSuffix | LongSuffix )?
        | '.' ( ( Digit )+ ( Exponent )?
                       ( FloatSuffix | LongSuffix )? )?
        | '0' ( '0'..'7' )*
                ( LongSuffix | UnsignedSuffix )?
        | '1'..'9' ( Digit )*
                ( LongSuffix | UnsignedSuffix)?
        | '0' ( 'x' | 'X' ) ( 'a'..'f' | 'A'..'F' | Digit )+
                ( LongSuffix| UnsignedSuffix)? ;

Exponent:       ( 'e' | 'E' ) ( '+' | '-' )? ( Digit )+ ;

FloatSuffix:    'f' | 'F' ;

LongSuffix:     'l' | 'L' ;

UnsignedSuffix: 'u' | 'U' ;
```

Die Operatoren `*`, `+`, `|` und `?` werden in bekannter Weise für Wiederholungen, Alternative und optionale Teile verwendet.

Funktionsweise eines Scanner-Generators

Intern verwendet der Scanner-Generator eine ganze Reihe der in der Vorlesung vorgestellten Verfahren:



Beispiel 9.14 - jLex Parsergenerators

An der folgenden Ausgabe des Scanner-Generators jLex ist der Verarbeitungsweg, ausgehend von regulären Ausdrücken (in der Ausgabe "lexical rules") über nichtdeterministische Automaten (NFA = nondeterministic finite state machine = ε-NEA), deterministische Automaten (DFA = deterministic finite state machine) und die Minimierung des Automaten bis zur Codegenerierung am Ende gut zu erkennen:

```
java -jar jlex.jar SampleLexer.lex

Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 254 states.
Working on character
classes.::::::::::::::::.:..:..:.:...::::.:::.....::::....:..
..:......:..::.......:..........:....:.........:..:...:.....
NFA has 44 distinct character classes.
Creating DFA transition table.
Working on DFA
states.......................................................
.............................................................
........
Minimizing DFA transition table.
91 states after removal of redundant states.
Outputting lexical analyzer code.
```

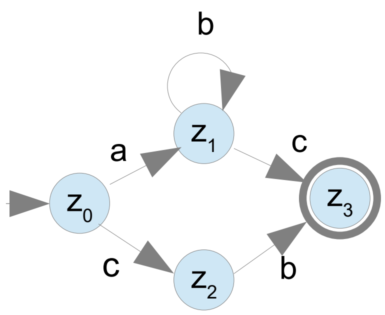
9.2 Umwandlung von ε-NEAs in reguläre Ausdrücke

Zu jedem **ε-NEA** *A* kann ein **regulärer Ausdruck** R_A konstruiert werden, so dass R_A genau die von *A* akzeptierte Sprache darstellt.

Bevor das Konstruktionsverfahren dafür vorgestellt wird, betrachten wir erst ein einfaches Beispiel, an dem die wesentlichen Grundideen dafür erkennbar werden.

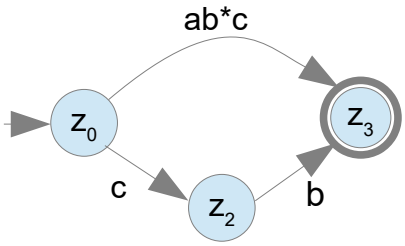
Grundidee Eliminationsverfahren

Wenn Sie folgenden Automaten betrachten,

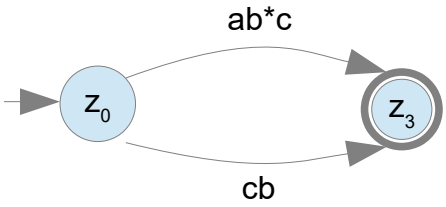


dann werden Sie vermutlich schnell zustimmen, dass der Automat die Sprache des regulären Ausdrucks ab*c | cb akzeptiert.

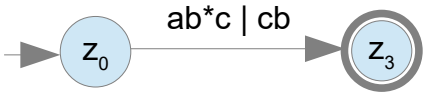
Warum ist das recht einfach zu erkennen? Wenn Sie zunächst Zustand z_1 betrachten, dann sieht man, dass man über diesen Zustand von z_0 aus den Zustand z_3 erreichen kann, und zwar mit einem a, beliebig vielen b und dann noch einem c. Abgekürzt könnte man diesen möglichen Weg über z_1 so darstellen:



In gleicher Weise könnte der Weg von z_0 über z_2 nach z_3 mittels c und b so zusammengefasst werden:



Die zwei parallelen Möglichkeiten, von z_0 nach z_3 zu kommen, "obenrum" mittels ab*c und "untenrum" mittels cb können nun noch als Alternative zusammengefasst werden.



Man sieht nun, dass man mit ab*c|cb vom Startzustand zum Endzustand kommen kann.

An diesem Beispiel sind die wesentlichen Grundideen für das folgende, allgemein anwendbare Konstruktionsverfahren schon erkennbar:

- Wir verwenden eine verallgemeinerte Form von Automaten, bei der Zustandsübergänge mit regulären Ausdrücken versehen sind. Ein Zustandsübergang

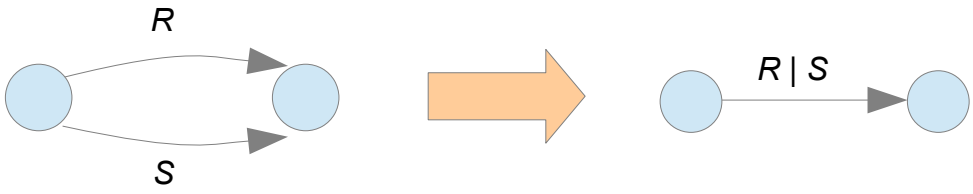
  

  bedeutet, dass man mit jedem Wort der Sprache des regulären Ausdrucks *R* von Zustand *z* nach Zustand *z'* kommen kann.
- Es werden nach und nach Zustände weggenommen ("eliminiert") und alle über einen eliminierten Zustand verlaufenden Kombinationen aus eingehenden und ausgehenden Transitionen werden durch direkte Zustandübergänge ersetzt.
- Parallele Weg zwischen zwei Zuständen können als Alternative im regulären Ausdruck zusammengelegt werden.

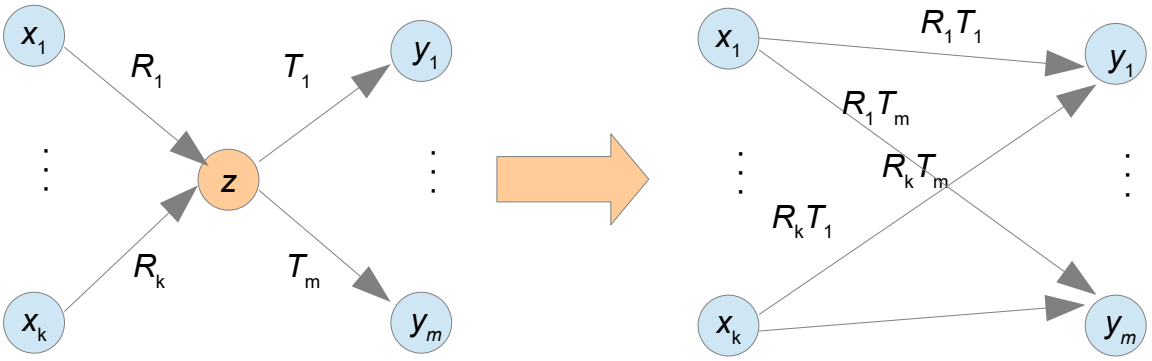
Es ergeben sich somit folgende Regeln für die Umformung von erweiterten Automaten:

Eliminationsregel 1: Parallele Zustandsübergänge zusammenlegen

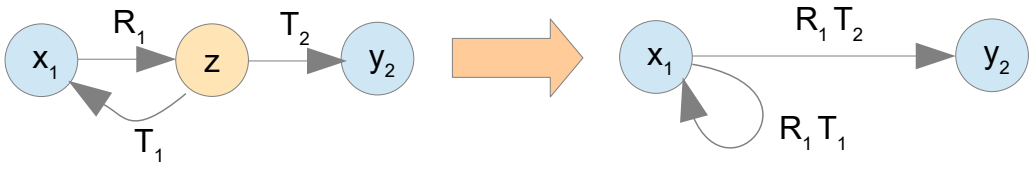
- Parallele Zustandsübergänge, die mit regulären Ausdrücken *R* bzw. *S* markiert sind, können zu einem Übergang mit R|S zusammengefasst werden.



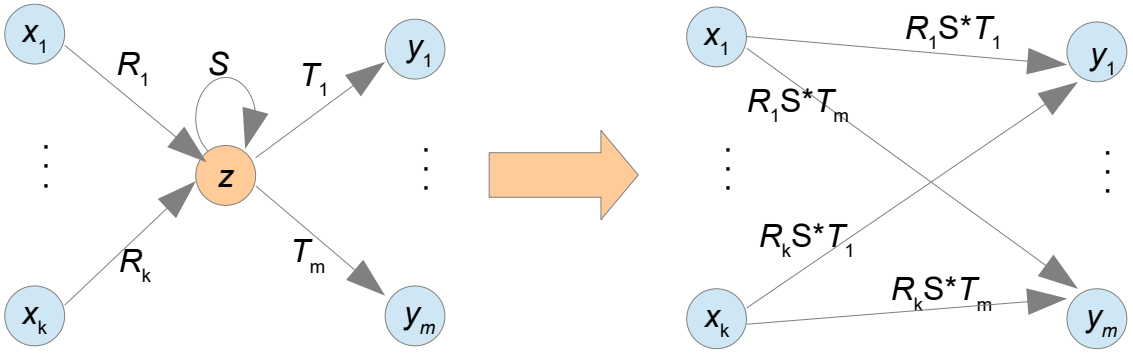
Eliminationsregel 2a: Zustand ohne Schleife eliminieren



- Jede Kombination von eingehendem und ausgehendem Zustandsübergang muss durch einen neuen Zustandsübergang abgedeckt werden, wenn z eliminiert wird.
- Ein Zielzustand y_j darf auch mit einen Ausgangszustand x_i identisch sein, z.B. wie hier:



Eliminationsregel 2b: Zustand mit Schleife eliminieren



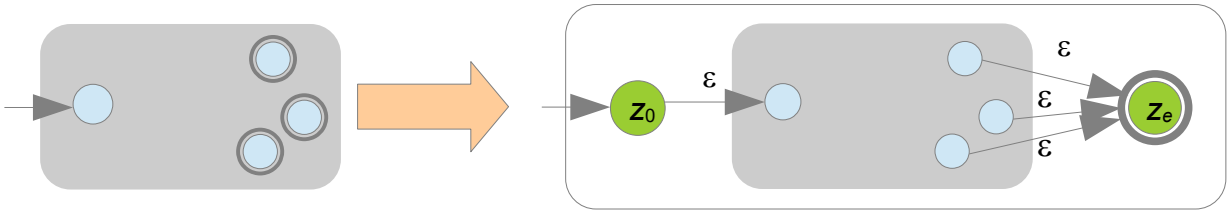
- Der einzige Unterschied zu Regel 2a besteht darin, dass der mit *S* markierte Schleifenübergang bei Zustand *z* beliebig oft durchlaufen werden kann, was im regulären Ausdruck durch $R_i S^* T_j$ entsprechend berücksichtigt wird.

Wie zu erkennen ist, wird bei jeder Anwendung einer dieser Regeln die Sprache des gesamten Automaten unverändert beibehalten.

Das komplette Konstruktionsverfahren, das im Kern auf diesen Eliminationsregeln basiert, funktioniert folgendermaßen

Eliminationsverfahren 9.15 - ε-NEA in regulären Ausdruck umwandeln

1. Vorbereitung:

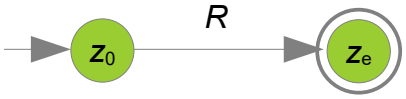


   - Der Automat wird um einen neuen Startzustand z_0 und einen neuen Endzustand z_e erweitert.
   - Ein ε-Übergang von z_0 zum ursprünglichen Startzustand wird eingeführt.
   - z_e wird einziger Endzustand. Es werden ε-Übergänge von jedem ursprünglichen Endzustand nach z_e eingeführt. Die ursprünglichen Endzustände sind nicht mehr Endzustände.

2. Die Eliminationsregeln 1, 2a und 2b (s.o.) werden solange angewendet, bis alle Zustände außer den neuen Zuständen *z* und *z'* eliminiert sind und keine parallelen Übergänge mehr zusammengelegt werden können.

Ergebnis: Es gibt nur zwei Möglichkeiten, wie der Automat am Ende aussehen kann:

- Es gibt einen Zustandsübergang, der mit einem regulären Ausdruck *R* markiert ist.



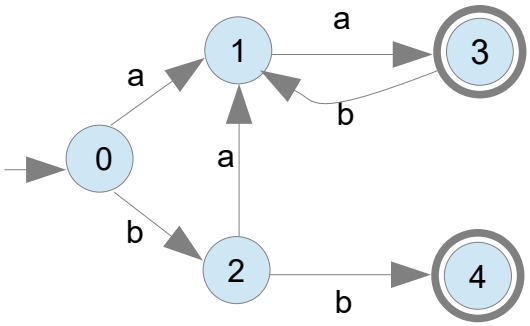
  Der reguläre Ausdruck *R* beschreibt dann die Sprache des Automaten.

- Es gibt keinen Zustandsübergang zwischen Start- und Endzustand:

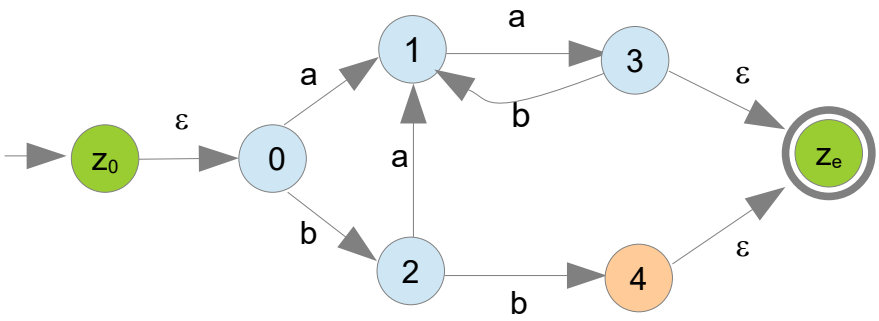
  Das ist dann der Fall, wenn der Automat gar nichts akzeptiert. In diesem Fall ist ∅ der reguläre Ausdruck, der die Sprache des Automaten darstellt.

Beispiel 9.16 - ε-NEA in regulären Ausdruck umwandeln

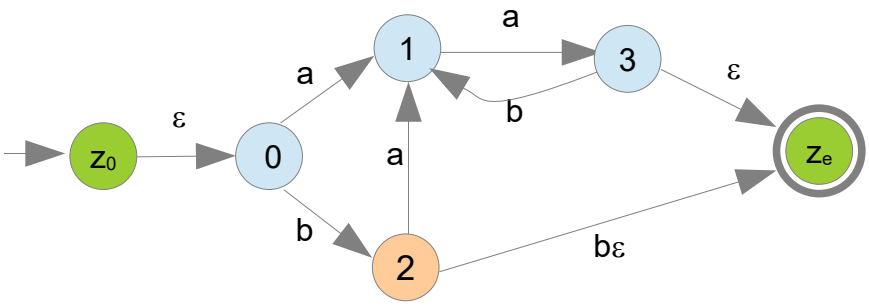
Es soll ein äquivalenter regulärer Ausdruck für folgenden endlichen Automaten konstruiert werden:



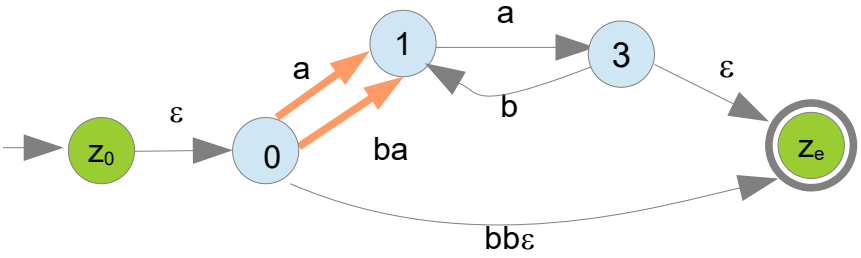
- Vorbereitung: neuen Startzustand z und Endzustand z' einfügen:



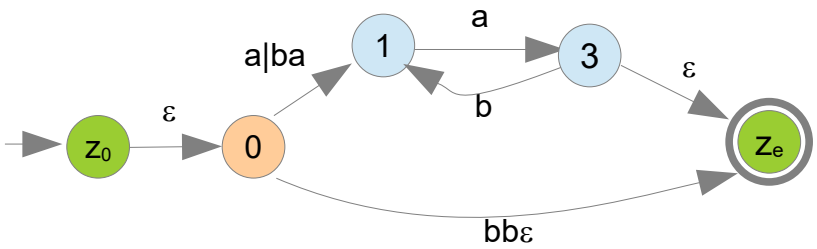
- Elimination von Zustand 4 mit Eliminationsregel 2a) Pfad 2→**4**→z_e muss ersetzt werden.



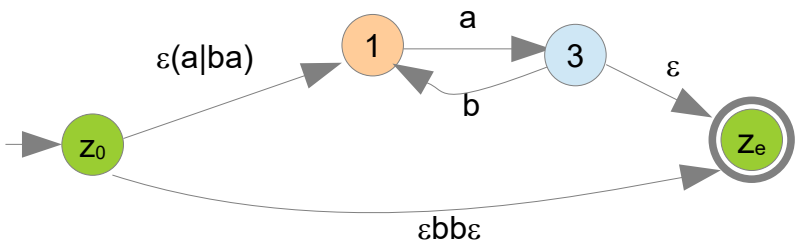
- Elimation von Zustand 2 (mit Eliminationsregel 2a). Pfade 0→2→1 und 0→2→z_e sind zu ersetzen.



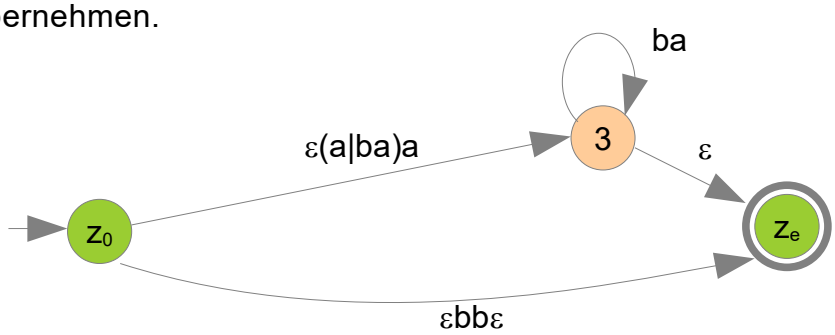
- Parallele Übergänge von 0 nach 1 zusammenlegen (Eliminationsregel 1):



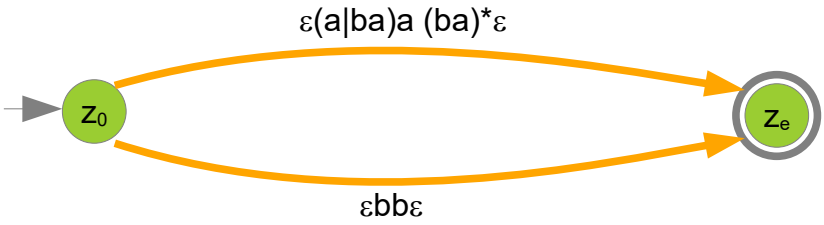
- Elimination von Zustand 0 (Eliminationsregel 2a): Pfade $z_0 \rightarrow \mathbf{0} \rightarrow 1$ und $z_0 \rightarrow \mathbf{0} \rightarrow z_e$ ersetzen.



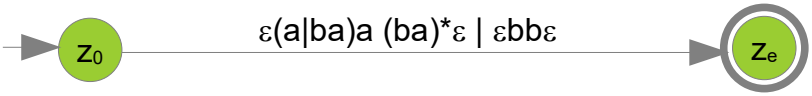
- Elimination von Zustand 1 (Eliminationsregel 2a): Pfade z0→**1**→3 und 3→**2**→3 übernehmen.



- Elimination von Zustand 3 (Eliminationsregel 2b): Pfad $z_0 \rightarrow \mathbf{3} \rightarrow z_e$ übernehmen.



- Parallele Übergänge von z_0 nach z_e zusammenlegen:



Das Ergebnis ist somit der reguläre Ausdruck

ε(a|ba)a (ba)*ε | εbbε

(das könnte natürlich noch vereinfacht werden zu (a|ba)a(ba)* | bb).

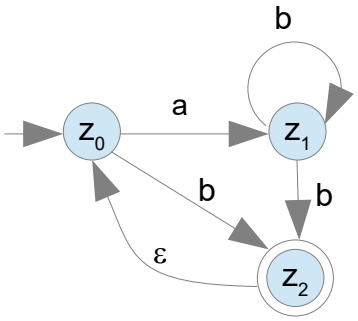
Anmerkungen

- In welcher Reihenfolge die Zustände eliminiert werden, ist beliebig. Je nach gewählter Reihenfolge wird ein syntaktisch anderer regulärer Ausdruck entstehen. Alle Ausdrücke sind aber äquivalent, d.h. sie beschreiben immer die gleiche Sprache - die des Automaten.
- Dadurch, dass am Anfang ein neuer Startzustand und ein neuer Endzustand eingeführt werden, ist sichergestellt, dass als Ergebnis nur die zwei oben angegebenen Endsituationen entstehen können und nicht noch andere Varianten berücksichtigt werden müssen.

  Dies liegt daran, dass folgende Eigenschaft des erweiterten Automaten bei jeder Anwendung einer Eliminationsregel erhalten bleibt: Beim (neuen) Startzustand z_0 gibt es nur ausgehende Zustandsübergänge, beim (neuen) Endzustand z_e gibt es nur eingehende Zustandsübergänge.
- Wie man an diesem Beispiel sieht, werden die regulären Ausdrücke meist sehr schnell komplex und unübersichtlich. Dies ist aber kein Problem, da es hier nicht um verständliche Ausdrücke geht, sondern nur darum, zu zeigen, dass es generell möglich ist, für jeden Automaten einen äquivalenten regulären Ausdruck zu bilden.
- Da jeder DEA auch als ε-NEA betrachtet werden kann, kann das Verfahren in gleicher Weise auch verwendet werden, um zu einem DEA einen regulären Ausdruck zu bilden

Aufgabe 9.17 - ε-NEA in regulären Ausdruck umwandeln

Konstruieren Sie mit dem vorgestellten Eliminationsverfahren einen regulären Ausdruck, der die Sprache des folgenden Automaten darstellt.



Eigenschaft 9.18 - Äquivalenz von Automat und konstruiertem Ausdruck

Der zu einem ε-NEA *A* mittels Eliminationsverfahren *9.15* konstruierte reguläre Ausdruck R_A beschreibt genau die Sprache, die der Automat akzeptiert, d.h. $L(A) = L(R_A)$.

Auf einen formalen Beweis hierfür wird verzichtet. Das Ergebnis sollte aber plausibel sein. Durch die Einführung der neuen Start- und Endzustände und bei jedem der einzelnen Teilschritte bleibt die Sprache des Automaten unverändert, so dass am Ende, wenn es nur noch einen Zustandsübergang gibt, der dort angegebene reguläre Ausdruck die ursprüngliche Sprache des Automaten angibt.

9.3 Chomksy-Typ-3-Grammatiken und endliche Automaten

Zwischen Chomsky-Typ-3-Grammatiken (rechtslineare Grammatiken) und endlichen Automaten besteht auch ein enger Zusammenhang:

- Zu jeder Chomsky-Typ-3-Grammatik kann ein ε-NEA konstruiert werden, der genau die Sprache der Grammatik akzeptiert.
- Zu jedem DEA kann eine Chomsky-Typ-3-Grammatik definiert werden, die genau die vom Automaten akzeptierte Sprache generiert.

  Da jeder ε-NEA mittels Teilmengenkonstruktion in einen DEA umgewandelt werden kann, kann somit auch indirekt zu jedem ε-NEA eine äquivalente Typ-3-Grammatik angegeben werden.

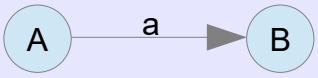
Konstruktion 9.19 - ε-NEA für Chomsky-Typ-3-Grammatik bilden

Gegeben sei eine Chomsky-Typ-3-Grammatik (rechtslineare Grammatik).
Der zugehörige ε-NEA wird in folgender Weise definiert:

- Zustände: Menge der nichtterminalen Symbole der Grammatik
- Startzustand: Startsymbol der Grammatik
- Für jede Produktion

  $A \rightarrow aB$

  wird ein entsprechender Zustandsübergang eingeführt:



- Gibt es eine Produktion $A \rightarrow \varepsilon$, dann ist der Zustand A ein Endzustand.

Anmerkungen

- Der gebildete Automat ist ggf. nichtdeterministisch, z.B. bei den Produktionen A → aB | aC. ε-Übergänge werden hier aber nicht benötigt.
- Der so konstruierter Automat akzeptiert genau die Sprache der Grammatik, da zu jeder Ableitung eins-zu-eins eine akzeptierende Folge von Zustandsübergängen im Automaten angegeben werden kann.

Beispiel 9.20 - ε-NEA für Chomsky-Typ-3-Grammatik bilden

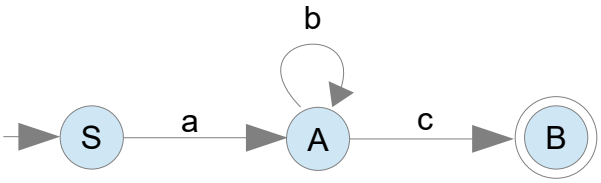
Folgende Chomsky-Typ-3-Grammatik ist gegeben:

$S \rightarrow aA$

$A \rightarrow bA \mid cB$

$B \rightarrow \varepsilon$

dann ergibt sich der folgende nichtdeterministische Automat, der die Sprache der Typ-3-Grammatik akzeptiert:



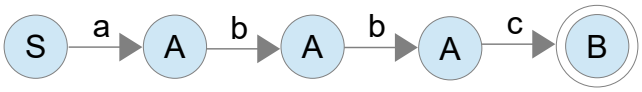
Dass Grammatik und Automat die gleiche Sprache beschreiben, wird plausibel, wenn man sich ein Beispiel für eine Ableitung betrachtet.

- Ableitung für das Wort abbc in der Grammatik:

  $S \Rightarrow aA \Rightarrow abA \Rightarrow abbA \Rightarrow abbcB \Rightarrow abbc$

  In jedem direkten Ableitungsschritt wird genau ein Zeichen produziert.

- Jede Ableitung kann eins-zu-eins in eine Folge von Zustandsübergängen übersetzt werden, die zu einem Endzustand führen:



  Bei jedem Zustandsübergang kann dann genau das entsprechende Zeichen verarbeitet werden.

Aufgabe 9.21 - ε-NEA für Chomsky-Typ-3-Grammatik

1. Erstellen Sie einen nichtdeterministischen Automaten, der die Sprache der folgenden Typ-3-Grammatik akzeptiert:

   $S \rightarrow aA \mid bS$

   $A \rightarrow bB \mid bS \mid \varepsilon$

   $B \rightarrow aS \mid \varepsilon$

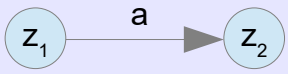
2. Ist mit der Grammatik das Wort *abab* ableitbar?
3. Wird das Wort *abab* vom erstellen Automaten akzeptiert?

In umgekehrter Weise kann in analoger Weise zu einem DEA auch eine Chomsky-Typ-3-Grammatik definiert werden:

Konstruktion 9.22 - Chomsky-Typ-3-Grammatik für einen DEA

Gegeben sei ein deterministischer endlicher Automat (DEA) A = (Z, Σ, δ, z_0, E). Die entsprechende Chomsky-Typ-3-Grammatik wird folgendermaßen definiert:

- Als Zustände des DEA werden als nichtterminale Symbole der Grammatik verwendet.
- Der Startzustand ist das Startsymbol.
- Für jeden Zustandsübergang $\delta(z_1, a) = z_2$



wird eine Produktion

$z_1 \rightarrow a\ z_2$

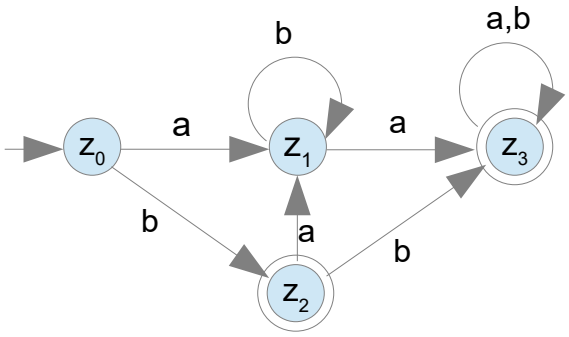
gebildet.

- Für jeden Endzustand *z* wird eine ε-Produktion eingeführt:

$z \rightarrow \varepsilon$.

Beispiel 9.23 Chomsky-Typ-3-Grammatik für DEA bilden

Folgender DEA ist gegeben:



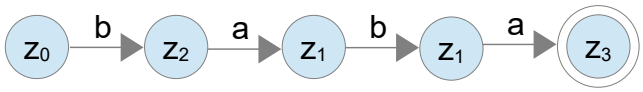
Dazu konstruierte Chomsky-Typ-3-Grammatik:

$$
\begin{aligned}
Z_0 &\rightarrow \quad aZ_1 \\
&\quad\ \mid bZ_2 \\
Z_1 &\rightarrow \quad aZ_3 \\
&\quad\ \mid bZ_1 \\
Z_2 &\rightarrow \quad aZ_1 \\
&\quad\ \mid \ bZ_3 \\
&\quad\ \mid \ \varepsilon \\
Z_3 &\rightarrow \quad aZ_3 \\
&\quad\ \mid \ bZ_3 \\
&\quad\ \mid \ \varepsilon
\end{aligned}
$$

Wie leicht zu sehen ist, kann jedes Wort, das vom DEA akzeptiert wird, auch durch

die Grammatik generiert werden.

Beispielsweise wird das Wort baba von dem Automaten akzeptiert:



Diese akzeptierende Folge von Zustandsübergängen lässt sich direkt in eine Ableitung der Grammatik übertragen:

$z_0 \Rightarrow bz_2 \Rightarrow baz_1 \Rightarrow babz_1 \Rightarrow babaz_3 \Rightarrow baba$

Anmerkungen

- Als Ausgangspunkt für die Umwandlung wird hier ein DEA vorausgesetzt und nicht ein ε-NEA, da es schwierig wäre, ε-Übergänge direkt in Typ-3-Produktionen zu übertragen. Da jeder ε-NEA aber mittels Teilmengenkonstruktion in einen DEA umgewandelt werden kann, ist der Bezug zwischen ε-NEAs und Typ-3-Grammatiken auch gegeben.

Zusammenfassung zu Lektion 15

Diese Fragen sollten Sie nun beantworten können

- Wie funktioniert prinzipiell ein Scanner-Generator? Welche Methoden aus der Theorie der formalen Sprachen werden dabei verwendet?
- Wie kann zu einem endlichen Automaten (ε-NEA oder DEA) ein regulärer Ausdruck erzeugt werden, der die Sprache des Automaten beschreibt?
- Welche Zusammenhänge bestehen zwischen Chomsky-Typ-3-Grammatiken und endlichen Automaten (ε-NEA bzw. DEA)?