Magisterarbeit

Scannergenerator mit benannten regulären Teilausdrücken

ausgeführt am

Institut für Computersprachen
Arbeitsbereich Programmiersprachen und Übersetzer

Argentinierstraße 8, 1040 Wien, Österreich

der Technischen Universität Wien

Karlsplatz 13, 1040 Wien, Österreich

unter Anleitung von

M. Anton Ertl

anton@mips.complang.tuwien.ac.at
Tel.: (+43-1) 58801 4459
Fax.: (+43-1) 505 78 38

durch

Christian Judt Bakk. techn.

Feldkellergasse 20, 1130 Wien, Österreich
e0125844@student.tuwien.ac.at

Datum

Unterschrift (Student)

Kurzfassung:

Die üblichen Scannergeneratoren wie z.B. Lex und Flex haben einen Nachteil. Die Scanner liefern nur den jeweils erkannten String, sowie welche Regel ihn beschreibt.

Sie liefern keine Information über die innere Struktur des Strings (z.B. bei einer Gleitkommazahl die Ziffern der Mantisse vor und nach dem Komma sowie den Exponenten), sodass dieser in vielen Fällen von handgeschriebenem scannerähnlichen Code nachbearbeitet werden muss.

Diese Magisterarbeit beschreibt die Möglichkeiten einen Scannergenerator zu implementieren, der diesen Nachteil nicht hat, sowie die dabei auftretenden allgemeinen Probleme und dazugehörigen Lösungen bzw. Lösungsansätze.

Außerdem wird eine Erweiterung des Scannergenerators Flex beschrieben, welche ein derartiges System implementiert.

Abstract:

Common scanner generators like Lex and Flex share a disadvantage. The scanners only determine the matched string and which rule matched it.

They don’t supply information about the string’s internal structure (for example the digits before and after a floating point number’s dot and the corresponding exponent). Afterwards, in many cases the string has to be refined by hand written, scanner like code.

This master thesis describes possibilities to implement a scanner generator without this disadvantage, arising common problems and corresponding solutions.

In addition it describes an extension of the scanner generator Flex which implements such a system.

Inhaltsangabe

1 Einleitung

Ziel dieser Magisterarbeit ist es einen Scannergenerator zu implementieren, welcher nicht nur einzelne Regeln erkennen kann, sondern auch die Möglichkeit bietet, nach einem Regeltreffer auf bestimmte benannte Teile des erkannten Strings zuzugreifen.

Dies ist am besten mit einem Beispiel zu verdeutlichen. Es handelt sich dabei um das Eingabedateiformat für die Scannergeneratoren lex bzw. flex. Bei beiden Beispielen erfolgt eine Regeldarstellung in EBNF Notation. Dabei geht es um das Erkennen einer Gleitkommazahl inklusive Basis.

Eingabedatei für Standard - flex:

```
ALNUM [0-9a-zA-Z]

BASIS [0-9]+
MANTISSE [+-]?(({ALNUM}+(,{ALNUM}*)?)|(({ALNUM}*,)?{ALNUM}+))
EXPONENT [+-]?{ALNUM}+

%%
({BASIS}#)?{MANTISSE}(^{EXPONENT})? {
     char *basis = 0, *mantisse = 0, *exponent = 0;

     int i;
     for(i=0; i<yyleng; i++) {
          if(yytext[i] == '#') {
               yytext[i] = 0;
               basis = yytext;
               yytext += ++i;
               yyleng -= i;
               i = 0;
               break;
          }
          else if(yytext[i] == '^') break;
     }

     mantisse = yytext;

     for(;i<yyleng; i++) {
          if(yytext[i] == '^') {
               yytext[i] = 0;
               exponent = yytext + i + 1;
               break;
          }
     }

     number_scanned(basis,mantisse,exponent);
}
%%
```

Hier ist es notwendig in der zur Regel gehörenden Aktion weitere scannerähnliche Prozeduren zu verwenden, um die einzelnen Zahlstrings aus dem Gesamtstring extrahieren. Dies ist eigentlich unnötig, da diese Arbeit genau genommen auch vom Scanner selbst erledigt werden könnte.

Code mit einer entsprechenden flex - Erweiterung:

```
%%
({basis,{BASIS}}#)?{mantisse,{MANTISSE}}(^{exponent,{EXPONENT}})? {
     number_scanned(basis,mantisse,exponent);
}
%%
```

In diesem Fall werden die relevanten Bereiche im Regulären Ausdruck der Regel explizit mit benannt und stehen so nach einem Regeltreffer als einzelne Strings zur Verfügung.

Der im Rahmen dieser Magisterarbeit geschaffene Scannergenerator soll genau diese Art von Funktonalität bieten. Davor ist es allerdings notwendig herauszufinden, auf welche Art und Weise dieser Scannergenerator überhaupt implementiert werden kann. Dies umfasst sowohl die Frage wie eine Benennung von regulären Ausdrücken theoretisch zu machen ist, als auch auf welche Weise die konkrete Implementierung erfolgen sollte. Beispielsweise als Erweiterung eines bereits bestehenden Scannergenerators oder als, eher klein gehaltene, Neuimplementierung.

2 Problemstellung

Das grundsätzliche Problem wurde bereits in der Einleitung vorgestellt. Weitere Probleme ergeben sich daraus auf welche Art und Weise die in dem Regulären Ausdruck erfolge Benennung über die verschiedenen Vereinigungs-, Determinierungs-, Optimierungs- und Minimierungsschritte erhalten werden kann, sodass der resultierende Scanner in der Lage ist die einzelnen benannten Bereiche aus einem erkannten String automatisch herauszulesen.

Bei den meisten Scannergeneratoren werden aus den einzelnen regulären Ausdrücken (RAs) nichtdeterministische endliche Automaten (NEAs) generiert. Diese werden dann zu einem großen NEA vereinigt, der anschließend zu einem deterministischen endlichen Automaten (DEA) umgewandelt wird. Zuletzt findet unter Umständen noch eine Minimierung des DEA statt. Dieser wird dann, auf welche Art und Weise auch immer, in Code abgebildet.

Es stellt sich also auch die Frage, wie ein spezieller Teil eines regulären Ausdrucks über die Transformationen

RAs→ NEAs
NEAs → vereinigter NEA
vereinigter NEA → vereinigter DEA (NEA → DEA)
vereinigter DEA → minimaler vereinigter DEA (DEA → minimaler DEA)

erhalten werden kann, um bei der Verwendung des (minimalen) vereinigten DEAs die realen Vorkommen der benannten Teilausdrücke zu erkennen.

3 Idee

Die Markierung von Teilausdrücken (Benennung) an sich ist, wie bereits in der Einleitung beschrieben, nicht besonders aufwendig. Ein Missbrauch der geschwungenen Klammern (in Flex eigentlich für Makros und begrenzte Vielfachheit gedacht), der zu benennende reguläre Ausdruck und der Name selbst ist, wie in der Einleitung skizziert, ausreichend.

Nun ist noch zu klären, welche Aktionen durchgeführt werden sollen, wenn einer dieser benannten Teilausdrücke im gescannten String erkannt wird. Die eine Möglichkeit ist, jedem Namen eine Aktion zuzuordnen, welche beim Erkennen eines Substrings mit diesem Namen ausgeführt wird. Eine weitere Möglichkeit ist diese Substrings zwischenzuspeichern und nach erkanntem Gesamtstring einer einzigen Aktion zur Verfügung zu stellen.

Diese Arbeit konzentriert sich auf die letzte der beiden Möglichkeiten, da auf diese Art und Weise eine größere Ähnlichkeit mit bereits bestehenden Scannergeneratoren wie Lex bzw. Flex beibehalten werden kann. Außerdem ist es für den Scanner erst nach dem Erkennen des Gesamtstrings möglich zu sagen, welche Regel, und damit welche Benennungen, bezüglich der Substrings relevant sind.

Weiters können die dabei gebildeten Datenstrukturen in der Aktion außerdem schrittweise durchgegangen werden, um so denselben Effekt wie mit einer Aktion pro Vorkommen zu erzielen.

Bei der Bildung dieser Ergebnisdatenstrukturen kommt es bei mehr oder weniger liberaler Anwendung des Benennungskonstrukts zu Phänomenen, welche im Folgenden genauer betrachtet werden.

Die Beispiele werden ab hier im Stil von Lex bzw. Flex gehalten.

3.1 Listenbildung

Falls ein benannter Ausdruck im RA nur einmal vorkommt, im konkreten String dagegen mehrmals, ist es notwendig die der Menge der erkannten Strings in eine möglicherweise mehrdimensionales Datenstruktur umzuwandeln. Diese Datenstruktur dient dann als gut strukturierter Behälter für die einzelnen Strings.

Beispiel 3.1.1: Listendefinition in Prolog

```
ELEMENT = ([a-zA-Z0-9]+)
({ELEMENT}“.“)*“[]“
```

Erweiterung:

```
({element,{ELEMENT}}.)*“[]“
```

Folgen:

```
"a.[]"                  => element=['a']
"a.b.c.d.[]"            => element=['a','b','c','d']
"[]"                    => element=[]
```

3.2 Nichtvorkommen

Ein benannter regulärer Teilausdruck kann im konkreten String natürlich nicht nur mehrfach, sondern auch überhaupt nicht vorkommen.

Beispiel 3.2.1: optionales Vorzeichen #1

```
NUMBER = [0-9]+
(“+“|“-“)?NUMBER
```

Erweitert:

```
{vorzeichen,(“+“|“-“)}?{NUMBER}
```

Folgen:

```
"+250"                  => vorzeichen='+'
"-2"                    => vorzeichen='-'
"28"                    => vorzeichen=NULL
```

Hier stellt sich die Frage ob der String `vorzeichen` im Fall eines Nichtvorkommens NULL oder einen leeren String beinhalten sollte, was zum Teil aber auch auf die Zielsprache des Generators ankommt. Im obigen Beispiel kann der Teilausdruck für sich genommen keinen leeren String beschreiben. NULL ist daher angemessener.

Beispiel 3.2.2: optionales Vorzeichen #2

```
{vorzeichen,(“+“|“-“)?}{NUMBER}
```

Folgen:

```
"+250"                  => vorzeichen='+'
"-2"                    => vorzeichen='-'
"28"                    => vorzeichen=’’
```

In dieser Schreibweise kann der Teilausdruck für sich genommen sehr wohl einen leeren String beschreiben. Hier ist aus diesem Grund im Falle eines Nichtvorkommens ein leerer String angemessener.

3.3 Mehrfachnennungen

Unter Umständen kann es sinnvoll sein im regulären Ausdruck einen Namen mehr als einmal zu verwenden.

Beispiel 3.3.1: Integer Tupel

```
“(“{NUMBER},{NUMBER}“)“
```

Erweiterung:

```
“(“{element,{NUMBER}},{element,{NUMBER}}“)“
```

Folgen:

```
"(325,33)"            =>  element=['325','33']
```

Hier ist das Element `element` eine Liste bestehend aus zwei Strings, welche den beiden Vorkommen im regulären Ausdruck entsprechen.

Beispiel 3.3.2: verschiedene Klammerungen

```
(“[“{NUMBER}“]“)|(“{“{NUMBER}“}“)
```

Erweiterung:

```
(“[“{inhalt,NUMBER}“]“)|(“{“{inhalt,NUMBER}“}“)
```

Folgen:

```
"[45]"                  => inhalt='45'
"{258}"                 => inhalt='258'
```

Hier dagegen ist das Element `element` ein String da niemals beide Vorkommen in einem erkannten String auftauchen können. Es kann immer nur entweder jener String innerhalb der eckigen Klammern ODER jener innerhalb der geschwungenen Klammern vorkommen.

3.4 Listenbildung & Mehrfachnennungen

Wesentlich sinnvollere Möglichkeiten und interessantere Probleme ergeben sich aus der Kombination von Listenbildung und Mehrfachnennungen.

Beispiel 3.4.1: eindimensionales Integer-Array

```
“[“({NUMBER}(,{NUMBER})*)?“]“
```

Erweiterung:

```
“[“({element,{NUMBER}}(,{element,{NUMBER}})*)?“]“
```

Folgen:

```
"[]"                    => element =[]
"[8]"                   => element =['8']
"[23,5]"                => element =['23','5']
"[123,430,988]"         => element =['123','430','988']
```

Durch die Kombination von Listenbildung und Mehrfachnennungen ergibt sich allerdings das Problem, festzustellen, wie in welchen Fällen genau vorzugehen ist. Beispielsweise ist es ohne Probleme möglich zwei n-dimensionale Vorkommen zu einem n+1-dimensionalen Vorkommen zusammenzufassen. Dasselbe gilt für die Zusammenfassung eines n-dimensionalen und eines n+1-dimensionalen Vorkommens zu einem (um ein n-dimensionales Element größeren) n+1-dimensionalen Vorkommen.

Ebenfalls möglich ist eine indirekte Zusammenfassung eines n-dimensionalen und eines n+2-dimensionalen Vorkommens. In diesem Fall kann das n-dimensionale Vorkommen zu einem aus einem Element bestehenden n+1-dimensionalen Vorkommen erweitert werden. Dieses kann dann mit dem n+2-dimensionalen Vorkommen zu einem um eins größeren n+2-dimensionalen Vorkommen zusammengefasst werden.

Falls nun ein n-dimensionales Vorkommen mit einem n+m-dimensionalen Vorkommen zusammengefasst werden soll, muss das n-dimensionale Vorkommen zuerst zu einem m-1-dimensionalen Vorkommen erweitert werden. Dieses kann dann mit dem m-dimensionalen Vorkommen zu einem um eins größeren m-dimensionalen Vorkommen zusammengefasst werden.

3.5 Mehrfach benannte Ausdrücke

Mehrfach benannte Ausdrücke, also benannte Ausdrücke als Teile von anderen benannten Ausdrücken sind ebenfalls möglich.

Beispiel 3.5.1: Mehrfachbenennung

```
({AB,{A,a*}-{B,b*}}-)*
```

Dabei stellt sich die Frage ob hier der äußere Name AB eine spezielle Beziehung zu den inneren Namen A und B haben soll. Dabei könnte die Ergebnisdatenstruktur von AB die Ergebnisdatenstrukturen von A und B enthalten sowie jedes Element von AB je ein Vorkommen von A und B enthalten.

Beispiel 3.5.2: Fortsetzung von 2.5.1 mit Beziehung in den Ergebnisdatenstrukturen

```
"aa-bbb-a-b"              => AB = ["aa-bbb","a-b"]
                             A = ["aa","a"]
                             B = ["bbb","b"]
                             AB.A = ["aa","a"]
                             AB.B = ["bbb","b"]
                             AB[1].A = "aa"
                             AB[1].B = "bbb"
                             AB[2].A = "a"
                             AB[2].B = "b"
                             AB.A[1] = "aa"
                             AB.B[1] = "bbb"
                             AB.A[2] = "a"
                             AB.B[2] = "b"
```

Diese Möglichkeit die Ergebnisdatenstrukturen miteinander zu verknüpfen hat aber den Nachteil, dass es im Fall einer Mehrfachnennung eines der beteiligten Namen zu komplizierten Problemen kommen kann.

Beispiel 3.5.3: Probleme mit der Beziehung in den Ergebnisdatenstrukturen #1

```
({AB,{A,a*}-{B,b*}}-{B,b*}-)*
```

In diesem Fall kommt die bezüglich AB innere Benennung B auch außerhalb der Benennung AB vor.

Folgen:

```
"aa-bbb-b-a-b-bbb"        => AB = ["aa-bbb","a-b"]
                             B = ["bbb","b","b","bbb"]
                             AB.B = ["bbb","b"]
```

Beispiel 3.5.4: Probleme mit der Beziehung in den Ergebnisdatenstrukturen #2

```
({AB,{A,a*}-{B,b*}}-{AB,{A,a}-b}-)*
```

In diesem Fall kommt die äußere Benennung AB auch ohne B vor.

Folgen:

```
"aa-bbb-a-b-a-b-a-b"      => AB = ["aa-bbb","a-b","a-b","a-b"]
                             B = ["bbb","b"]
                             AB.B = ["bbb",null,"b",null]
```

In beiden Fällen tritt das Problem auf, dass die Ausdrücke B und AB.B unterschiedliche Listen ergeben. Aus diesem Grund habe ich mich entschlossen diese Möglichkeit nicht weiter in Betracht zu ziehen.

Die Regeln bezüglich mehrfach benannter Ausdrücke beschränken sich deswegen auf das Verbot der Verwendung eines bestimmten Namens innerhalb eines regulären Ausdrucks welcher bereits diesen Namen trägt. Ein derartig benannter Ausdruck sollte immer zu einer Fehlermeldung führen.

4 Algorithmen und Datenstrukturen

Dieses Kapitel behandelt alle Algorithmen und Datenstrukturen welche eine Benennung vom regulären Teilausdruck in einen minimalen deterministischen Automaten überführen.

4.1 Einleitung

Bei der Umwandlung des regulären Ausdrucks in einen nichtdeterministischen Automaten stellt sich die Frage, wo die Markierungsinformation im erzeugten NEA am besten platziert wird.

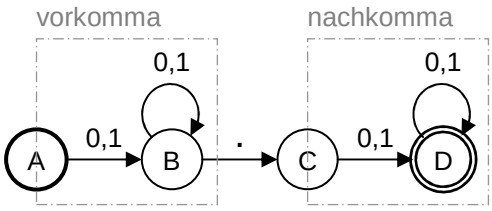
Beispiel 4.1.1: binäre Kommazahl

```
[01]+"."[01]+
```

Erweitert:

```
{vorkomma,[01]+}"."{nachkomma,[01]+}
```

Entsprechender endlicher Automat:



4.1.1 Aktionen

Die Lösung ergibt sich aus der Einführung so genannter Aktionen. Eine „Aktion“ in diesem Zusammenhang ist ein abstrakter Ausdruck für eine eindeutig bestimmte Prozedur, welche in einem Automaten bei einem eindeutig bestimmten Zustandsübergang durchgeführt wird. Sie können grundsätzlich auf jedem Zustandsübergang platziert werden.

Zu unterscheiden sind diese Aktionen von jenen Elementen, welche in vielen Scannergeneratoren ebenfalls als Aktionen bezeichnet werden, nämlich die vom Benutzer definierbaren Programmteile welche beim Erkennen einer bestimmten Regel ausgeführt werden.

Hier besteht Aktion im einfachsten Fall aus dem bei der Benennung verwendeten Namen und einem Vorzeichen „+“ bzw. „-“, je nachdem ob der benannte Bereich an der Aktionsposition beginnt oder endet.

Aktionen können wiederum zu Aktionslisten zusammengefasst werden, falls es notwendig sein sollte an einer Stelle des NEAs mehrere Aktionen unterzubringen. Diese Stellen können prinzipiell Übergänge zwischen zwei Zuständen, der Startzustand oder ein Zielzustand sein.

Falls es sich dabei um einen Übergang handelt der ein Zeichen konsumiert, kann eine Aktion auf sowohl vor als auch nach der Konsumierung dieses Zeichens ausgeführt werden (Vor- bzw. Nachaktionen). Bei ε-Übergängen ist diese Unterscheidung irrelevant, da hier kein Zeichen konsumiert wird (ε-Aktionen). Die Reihenfolge der Aktionen in der Aktionsliste an einer Stelle im endlichen Automaten ist natürlich auch von Bedeutung.

Die Aktionen im Startzustand werden ausgeführt, wenn der Automat zu Beginn über den Startzustand „betreten“ wird (Startaktionen). Jene in einem Endzustand werden ausgeführt, wenn der Automat über diesen Endzustand verlassen wird (Endaktionen).

Beispiele für Aktionen: +a, -element, +vorzeichen, -b

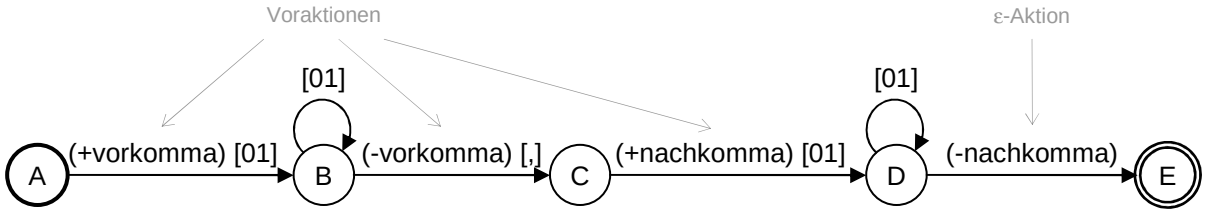
Beispiele für Aktionslisten: (+a-element), (+vorzeichen-b)

4.1.2 Aktionen zur Bennennung von Teilausdrücken

Aktionen können, falls sie richtig platziert werden, zum Benennen von Teilausdrücken in einem regulären Automaten benutzt werden. Die relevanten Stellen im NEA werden dabei mit Aktionslisten besetzt, welche im nächsten Beispiel allerdings nur je eine Aktion beinhalten.

Im NEA ist es dabei völlig ausreichend, ausschließlich ε-Aktionen zu verwenden. Um die Diagramme so klein und übersichtlich wie möglich zu halten, werden in diesem Artikel aber auch Voraktionen verwendet.

Beispiel: binäre Kommazahl



Folgen:

```
"010,11"            => (+vorkomma)0()1()0(-vorkomma),
                       (+nachkomma)1()1(-nachkomma)
                    => vorkomma='010'
                       nachkomma='11'
```

4.1.3 Behandlung von Strukturbildung

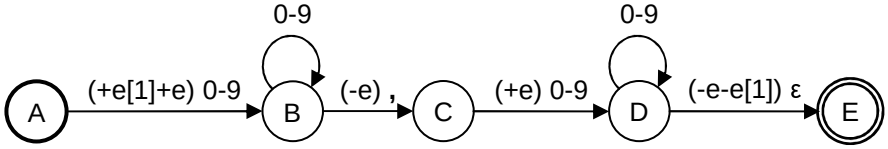
Strukturbildung kann sowohl durch Listenbildung als auch durch Mehrfachnennungen auftreten. Der Aufbau dieser Strukturen wird im NEA ebenfalls mit Aktionen beschrieben. Diese Aktionen beinhalten dann nicht nur den Namen des strukturbildenden Ausdrucks und ein Vorzeichen, sondern auch die dazugehörige Dimension. Die Dimension ist dabei jene Zahl, welche die Dimensionalität jener Datenstruktur beschreibt, welche im Falle einer Durchführung der Aktion begonnen bzw. abgeschlossen wird. Die Dimension wird in eckigen Klammern nach dem Namen geschrieben und im Fall von Level 0 weggelassen.

Beispiele für Aktionen mit Dimension > 0: +a[2], -element[1], +vorzeichen[3], -b[16]

Beispiel 4.1.2: Integer Tupel

`“(“{e,{NUMBER}},{e,{NUMBER}}“)“`

Entsprechender endlicher Automat:



Folgen:

```
"(325,33)"              => (+e[1]+e)3()2()5(),(+e)3()3(-e-e[1])
                        => e=['325','33']
```

Dabei ist noch zu erwähnen, dass sämtliche beendenden Aktionen, (also jene mit einem „-“ als Vorzeichen) außer jenen mit der Dimension 0 für die Erstellung der finalen Datenstruktur nicht erforderlich sind und daher genauso gut weggelassen werden könnten (siehe Kapitel 4.6.3). Der Übersicht halber werde ich sie in dieser Arbeit aber beibehalten.

4.1.4 Behandlung von Mehrdeutigkeiten

Bei einem nichtdeterministischen endlichen Automaten gibt es, wie sein Name schon sagt, unter Umständen mehrere Möglichkeiten, einen String zu akzeptieren. Bei NEAs welche keine Aktionen beinhalten ist dies bloß ein Effizienzproblem, welches sich durch die Umwandlung in einen DEA lösen lässt, da im NEA jede Möglichkeit einen String zu akzeptieren gleichwertig ist.

Sollte der NEA dagegen Aktionen beinhalten, ist dies nicht mehr der Fall, was zu einigen schwer zu lösenden Problemen führt.

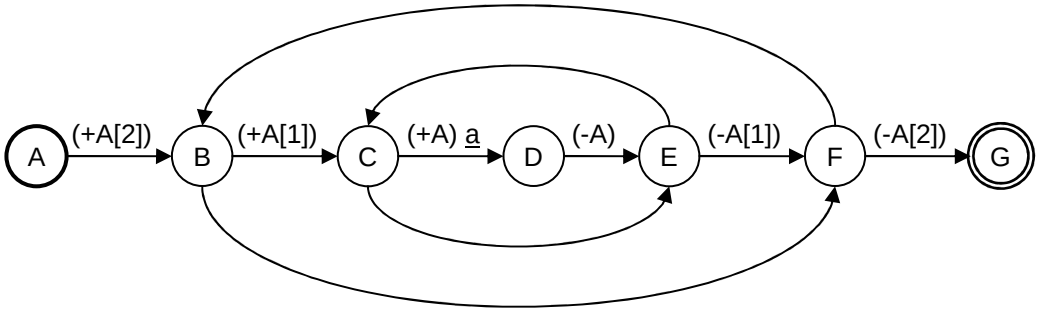
Beispiel 4.1.3: Idempotenz des Kleene Sterns

```
a**
```

Erweitert:

```
{A,a}**
```

Entsprechender endlicher Automat:



Folgen:

```
""          => (+A[2]-A[2])
               => element=[]

 oder       => ([+A[2]+A[1]-A[1]-A[2])
               => element=[[]]

 oder       => ([+A[2]+A[1]-A[1]+A[1]-A[1]-A[2])
               => element=[[],[]]

 ...

"a"         => (+A[2]+A[1]+A)a(-A-A[1]-A[2])
               => element=[['a']]

 oder       => (+A[2]+A[1]+A)a(-A-A[1]+A[1]-A[1]-A[2])
               => element=[['a'],[]]
```

```
 oder      => (+A[2]+A[1]+A)a(-A-A[1]-A[2])
              => element=[[],['a']]

 ...

"aa"       => (+A[2]+A[1]+A)a(-A+A)a(-A-A[1]-A[2])
              => element=[['a','a']]

 oder      => (+A[2]+A[1]+A)a(-A-A[1]+A[1]+A)a(-A-A[1]-A[2])
              => element=[['a'],['a']]

 ...
```

Diese Indeterminismen können grundsätzlich auf zwei Arten behandelt werden. Entweder der Indeterminismus wird als Fehler angesehen, oder eine der sich aus ihm ergebenden Möglichkeiten wird bevorzugt.

Da es im Bereich der Scannergeneratoren üblich ist, gewisse Indeterminismen zu tolerieren und dem Programmierer die „am ehesten erwartete“ Lösung zu bieten, wird auch diese Arbeit in diesem Sinne geschrieben.

Alle auftretenden Indeterminismen müssen also irgendwo in der Umwandlung vom regulären Ausdruck zum DEA beseitigt werden um zu verhindern, dass diese auch noch in den Ergebnisdatenstrukturen auftauchen.

4.2 Der reguläre Ausdruck

Bevor der NEA aus dem regulären Ausdruck erstellt werden kann muss dieser erst entsprechend vorbereitet werden. Dies ist hauptsächlich wegen der Strukturbildung und der Möglichkeit von Mehrfachnennungen notwendig. Dabei werden jedem regulären Ausdruck ein Namensraum und ein Aktionsraum zugeordnet.

4.2.1 Namensraum

Der Namensraum ist eine Multimenge bestehend aus Paaren (n,d) von jeweils einem Namen und einer Dimension. Sie beschreibt die Namensverhältnisse innerhalb des zugehörigen regulären Ausdrucks.

Beispiel 4.2.1: Mehrfachnennungen

```
{A,a}{A,a}                => {(A,0),(A,0)}
```

Beispiel 4.2.2: Strukturbildung

```
{A,a}*                    => {(A,1)}
```

Beispiel 4.2.3: Strukturbildung und Mehrfachnennungen

```
{A,a}{A,a}*               => {(A,0),(A,1)}
```

Beispiel 4.2.4: Disjunktion

```
{A,a}|{A,a}               => {(A,0)}
```

Beispiel 4.2.5: Disjunktion mit unterschiedlichen Dimensionen

```
{A,a}|{A,a}*              => {(A,1)}
```

4.2.2 Aktionsraum

Der Aktionsraum ist eine Abbildung $n \rightarrow (d_{min} ... d_{max})$ welche jedem Namen einen zugehörigen Dimensionsbereich zuordnet. Dieser beschreibt welche Aktionsliste vor bzw. nach jenem aus dem zugehörigen regulären Ausdruck gebildeten NEA platziert werden soll um diesen zu komplettieren.

Beispiel 4.2.6: Benennung

```
{A,a}                    => {(A->0..0)}
```

Beispiel 4.2.7: Mehrfachnennungen

```
{A,a}{A,a}               => {(A->1..1)}
```

Beispiel 4.2.8: Strukturbildung

```
{A,a}*                   => {(A->1..1)}
```

4.2.3 Bestimmung vom Namens- und Aktionsraum

Der Namensraum eines regulären Ausdrucks ist ausschließlich von ihm selbst abhängig. Sein Aktionsraum dagegen hängt von seinem Namensraum und den ihn „umgebenden“ regulären Ausdrücken zum nächsten übergeordneten rekursiven regulären Ausdruck bzw. bis zur Wurzel des regulären Ausdrucks. Ein rekursiver regulärer Ausdruck kann entweder ein Kleene Stern, ein Plus, eine begrenzte Vielfachheit oder ein Minimum sein.

Aus diesem Grund ist es am besten, beide Räume in zwei Schritten rekursiv zu berechnen. Die Namensräume werden „bottom up“ berechnet, und die Aktionsräume ausgehend von einem rekursiven regulären Ausdruck bzw. der Wurzel „top down“.

<u>Berechnung der Namensräume</u> (bottom up)

- Epsilon - ε
  Der Namensraum ist leer.
- Symbol - {a}
  Der Namensraum ist leer.
- Verkettung - A · B
  Der Namensraum ist die Multimengenvereinigung beider Unternamensräume.
- Disjunktion - A ∪ B
  Der Namensraum enthält alle Elemente der Unternamensräume, es sei denn ein Name kommt in beiden Unternamensräumen vor. In diesem Fall ist eine spezielle, auf die vielen verschiedenen Strukturierungsmöglichkeiten abgestimmte, spezielle Vereinigung durchzuführen. Diese wird im nächsten Kapitel genauer erläutert.
- Benennung - {n, A}
  Dem Namensraum werden alle Elemente des Unternamensraums und einmal (n,0) hinzugefügt.
- Kleene Stern - A*
  Der Namensraum des Kleene Sterns (und aller anderen eventuell rekursiven regulären Ausdrücke) enthält pro enthaltenem Namen n ein Paar (n,d*) wobei der Wert von d* von vielen verschieden Faktoren abhängt, welche ich im nächsten Kapitel genauer schildern werde.

Berechnung der Aktionsräume (bottom up)

Die Berechnung der Aktionsräume beginnt bei jedem eventuell rekursiven regulären Ausdruck und der Wurzel. Dabei muss jeder reguläre Ausdruck bezüglich seines Aktionsraumes einer extra erzeugen Anpassungsfunktion d_A genügen. Diese ist eine Funktion $n \rightarrow d$ wobei d die Namensanpassungsdimensionen ist.

- Epsilon - ε
  Nichts wird getan.
- Symbol - {a}
  Nichts wird getan.
- Verkettung - $A \cdot B$
  Die Verkettung ist jenes Element welches neben den rekursiven Elementen für den Großteil des komplexen Aktionsbestimmungsalgorithmus verantwortlich ist. Aus diesem Grund verweise ich hier auf das nächste Kapitel.
- Disjunktion - $A \cup B$
  Die Anpassungsfunktion wird an beide Unternamensräume weitergegeben.
- Benennung - {n, A}
  Dem Aktionsraum wird das Element $(n \rightarrow (0 \ldots d_A(n)))$ hinzugefügt. Außerdem wird die Anpassungsfunktion an den Ausdruck A weitergegeben. $d_A(n)$ ist dabei immer definiert.
- Kleene Stern - A*
  Hier gibt es für jeden relevanten Namen n eine eindeutige Dimension d im Namensraum. Wenn diese Dimension kleiner oder gleich $d_A(n)$ ist wird zum lokalen Aktionsraum das Element $(n \rightarrow (d \ldots d_A(n)))$ hinzugefügt. Ansonsten wird nichts getan.

4.2.4 Die Behandlung der rekursiven Elemente sowie der Verkettung und der Disjunktion

Sowohl die Disjunktion und Verkettung von Elementen als auch die rekursiven Elemente sind von essenzieller Bedeutung für die Komplexität des Namenssystems. Aus diesem Grund sind sie hier separat in einem eigenen Kapitel beschrieben. Zur Einleitung in diese spezielle Problematik möchte ich einige Beispiele anführen, wobei es jeweils darum geht ein passendes Gesamtelement zu finden.

Dabei gilt folgender Schlüssel:

nulldimensionales Element	Singular
eindimensionales Element	Liste
zweidimensionales Element	Feld
dreidimensionales Element	Raum

Weiters gilt X_Y = `[...`$_{XY}$`]`

Beispiel 4.2.9: Zwei Singulare (trivial)

`{A,a}{A,a}`

Lösung: eine Liste `['a','a']`

Beispiel 4.2.10: Zwei Listen L_1 und L_2 (interessanter)

`{A,a}*-{A,a}*`

Lösung 1: eine Liste welche die beiden Listen vereinigt `[...L1,...L2]`

Lösung 2: ein Feld mit den beiden Listen als erstes bzw. zweites Element `[L1,L2]`

Hier ist klar, dass Lösung 2 besser als Lösung 1 ist, da mehr von der strukturellen Aufteilung der einzelnen Elemente erhalten bleibt.

Beispiel 4.2.11: Ein Element und eine Liste L (wichtig)

`{A,a}-{A,a}*`

Lösung 1: eine Liste welche das Element als Kopf und die Liste als Rest beinhaltet `['a',...L]`

Lösung 2: ein Feld mit einer einelementigen Liste als erstes Element und der Liste als zweites Element `[['a'],L]`

Hier ist nicht offensichtlich welche Lösung die bessere ist, daher sind beide Möglichkeiten von Relevanz.

Beispiel 4.2.12: Ein Element und ein Feld F (ähnlich wichtig)

`{A,a}-{A,a}**`

Lösung 1: ein Feld welches eine einelementige Liste als Kopf und das Feld als Rest beinhaltet `[['a'],...F]`

Lösung 2: ein Raum mit einem einelementigen Feld als erstes Element und dem Feld als zweites Element `[['a']],F]`

Auch hier ist nicht offensichtlich welche Lösung die bessere ist.

Beispiel 4.2.13: Zwei Elemente und ein Feld F (komplex)

`{A,a}-{A,a}-{A,a}**`

Lösung 1: ein Feld welches eine zweielementige Liste als Kopf und das Feld als Rest beinhaltet `['a','a'],...F]`

Lösung 2: ein Feld welches am Beginn zwei einelementige Listen enthält und danach das Feld als Rest `['a'],['a'],...F]`

Lösung 3: ein Raum mit einem zweielementigen Feld als erstes Element und dem Feld als zweites Element. Dabei kann das zweielementige Feld folgendermaßen ausfallen.

Lösung 3a: als zwei Listen mit je einem Element: `[['a'],['a']],F]`

Lösung 3b: als eine Liste mit zwei Elementen: `[['a','a']],F]]`

Hier ist das Problem wirklich komplex, der Unterschied zwischen den Lösungen 1 bzw. 2 und 3a bzw. 3b ähnelt jenen Unterschieden in den letzten beiden Beispielen 4.2.11 und 4.2.12. Der Unterschied zwischen den Lösungen 1 und 2 bzw. 3a und 3b ist dagegen irgendetwas anderes.

Schlussfolgerungen:

Falls eine Reihe von Elementen vereinigt werden soll und jenes Element mit der höchsten Dimension d_{max} nur einmal vorkommt ist es möglich alle anderen Elemente in dieses einzubinden ODER ein neues Element mit der Dimension $d_{max} + 1$ zu definieren, welches alle bereits existierenden Elemente beinhaltet. Dies werde ich ab hier als mit bzw. ohne Einbindung bezeichnen.

In beiden Fällen ist es beim Definieren der internen Struktur des obersten Elements möglich nach zwei unterschiedlichen Methoden vorzugehen.

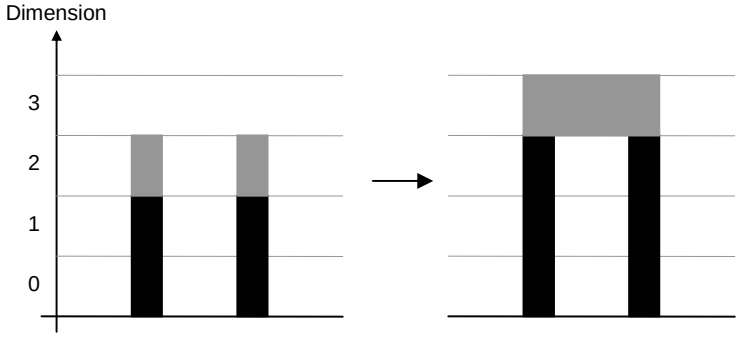
Im Einbindungsfall kann für jedes zusätzliche andere Element ein eigenes Element mit der Dimension $d_{max} - 1$ definiert werden ODER die anderen Elemente werden so gut es geht miteinander verschmolzen, um die Anzahl der zusätzlichen Elemente mit der Dimension $d_{max} - 1$ so gering wie möglich zu halten. Dies werde ich ab hier als ohne bzw. mit Zusammenschluss bezeichnen.

Im Falle eines neuen Elements mit der Dimension $d_{max} + 1$ gilt dasselbe, nur mit der Dimension d_{max} statt $d_{max} - 1$.

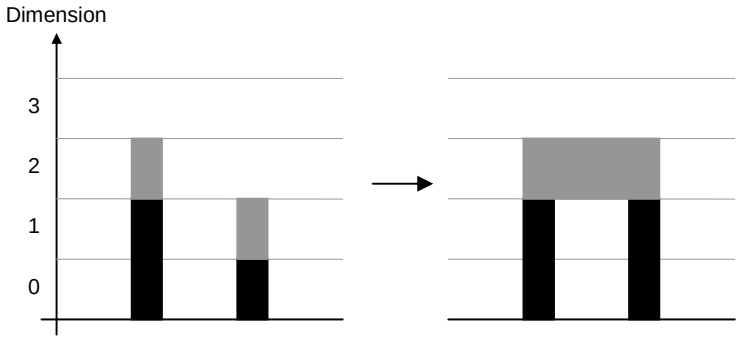
Visualisierung:

nur im Namensraum fixiert im Aktionsraum fixiert

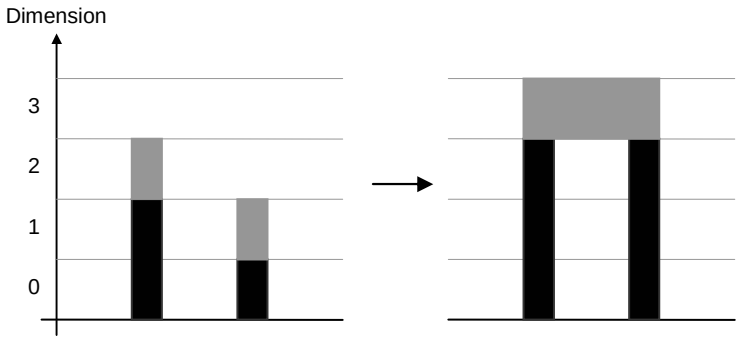
- Vereinigung zweier gleichdimensionaler Vorkommen:



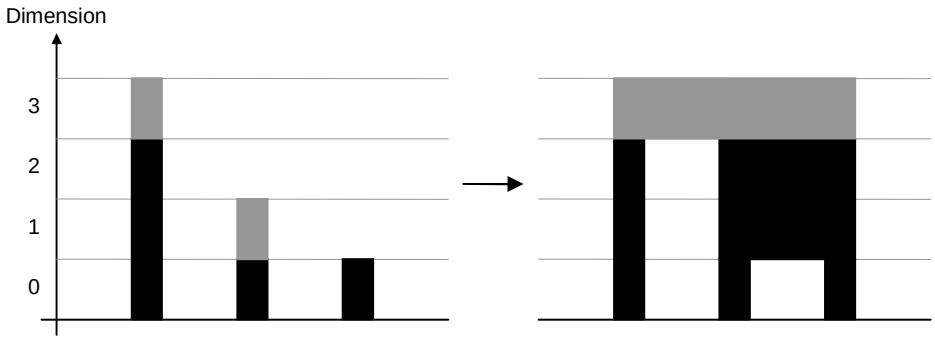
- Vereinigung zweier unterschiedlichdimensionaler Vorkommen mit Einbindung:



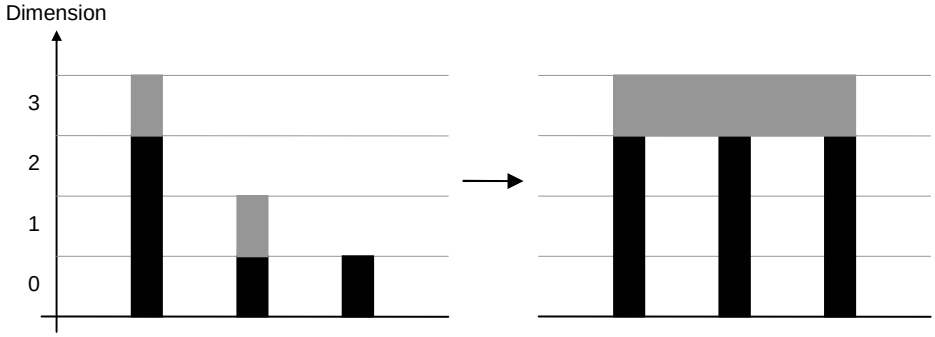
- Vereinigung zweier unterschiedlichdimensionaler Vorkommen ohne Einbindung:



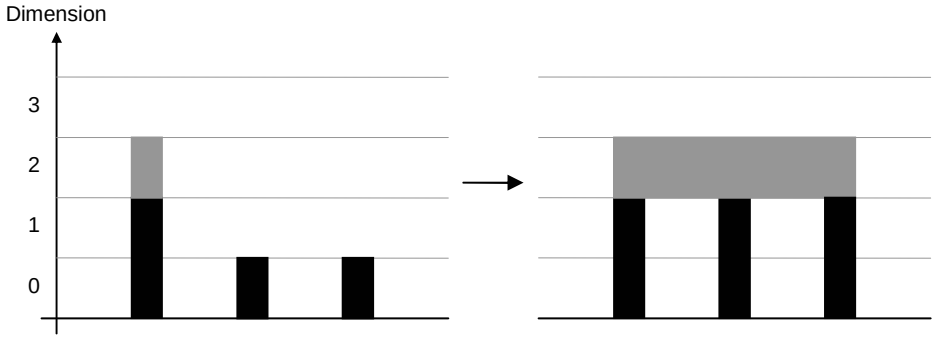
- Vereinigung von drei unterschiedlichdimensionalen Vorkommen mit Einbindung:



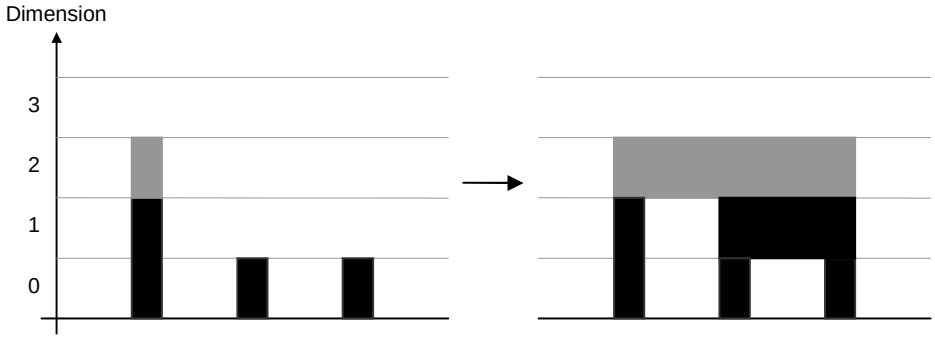
- Vereinigung von drei unterschiedlichdimensionalen Vorkommen ohne Einbindung:



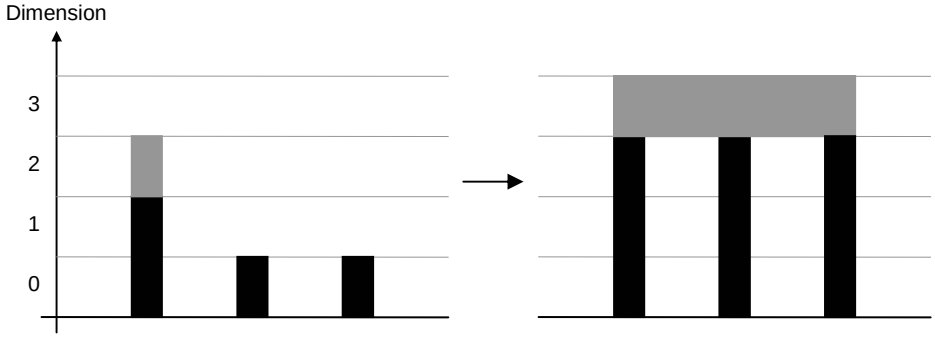
- Vereinigung von zwei null- und einem zweidimensionalen Vorkommen mit Einbindung aber ohne Zusammenschluss:



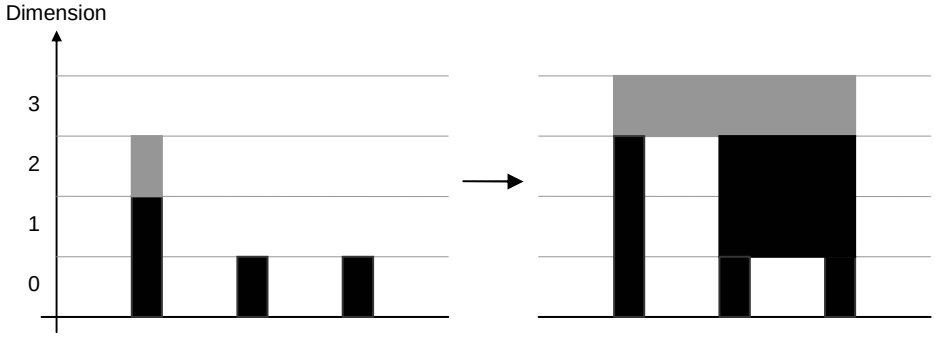
- Vereinigung von zwei null- und einem zweidimensionalen Vorkommen mit Einbindung und mit Zusammenschluss:



- Vereinigung von zwei null- und einem zweidimensionalen Vorkommen ohne Einbindung und ohne Zusammenschluss:



- Vereinigung von zwei null- und einem zweidimensionalen Vorkommen ohne Einbindung aber mit Zusammenschluss:



Bedeutung für die Berechung der Namens- und Aktionsräume:

Ein rekursives Element wie beispielsweise der Kleene Stern kann pro Namen nur ein einzelnes Namenselement behandeln. Dasselbe gilt auch für das Wurzelelement an der Spitze des Baumes. Aus diesem Grund ist es in beiden Fällen notwendig, alle darunter liegenden und durch Verkettung verbundenen Elemente nach einer der im letzten Abschnitt erklärten Methoden zu einem einzigen zu vereinigen.

- Kleene Stern - A* (Namensraum & Aktionsraum)
  Der Namensraum des Kleene Sterns (und aller anderen eventuell rekursiven regulären Ausdrücke) enthält pro enthaltenem Namen n und dem zugehörigen (eventuell vereinigten) Element mit der Dimension d_v ein Paar $(n, d_v + 1)$. Weiters wird, falls eine Vereinigung notwendig war, zum Aktionsraum des Unterausdrucks das Paar $(n, d_v ... d_v)$ hinzugefügt. Weiters wird zur Anpassungsfunktion für den Ausdruck A das Element $(n \rightarrow d_v - 1)$ hinzugefügt. Schließlich wird die Anpassungsfunktion zwecks Initialisierung dem Ausdruck A übergeben.
- Wurzelement (Aktionsraum)
  Das Wurzelement macht zusätzlich, also nach der Erstellung seines Namensraumes, dasselbe mit sich selbst wie der Kleene Stern mit seinem Unterausdruck. Die sich dabei pro Namen ergebenden Dimensionen sind aber keinem Namensraum hinzuzufügen, sondern bilden die Dimensionen der finalen Datenstrukturen.

Weiters sind die Verkettung und die Disjunktion von großer Bedeutung.

- Verkettung - A · B (Aktionsraum)
  Falls ein anzupassender Name n in nur einem der beiden Teilausdrücke vorkommt ist wird die entsprechende Anpassung einfach an den betroffenen Ausdruck weitergeleitet. Falls der Name jedoch in beiden Elementen vorkommt sind die entsprechenden Maximaldimension d_{Amax} bzw. d_{Bmax} der beiden Unternamensräume sowie der Strukturierungstyp (also Einbinden ja/nein bzw. Zusammenschluss ja/nein) ausschlaggebend. Die lokale Maximaldimension d_{max} ist dabei das Maximum von d_{Amax} und d_{Bmax} und die Kardinalität ihres Vorkommens c_{max}. Außerdem ist die lokale Anpassungsfunktion $d_{A,L}$ und die Subanpassungsfunktion $d_{A,S}$ wichtig.

  Falls Anhängen UND $d_{max} = d_{A,L}(n)$ UND $d_{Amax} \neq d_{Bmax}$ UND $c_{max} = 1$ dann

  lokaler Aktionsraum += $(n \rightarrow (d_{max} ... d_{max}))$
  $d_{A,S}$ += $(n \rightarrow d_A(n) - 1)$

  ansonsten falls $d_{max} < d_{A,L}(n)$ UND Anhängen UND $c_{max} = 1$ dann

  Falls es im lokalen Namensraum Elemente (n,d) mit $d < d_{max}$ gibt dann

  lokaler Aktionsraum += $(n \rightarrow (d_{max} ... d_{A,L}(n)))$
  $d_{A,S}$ += $(n \rightarrow d_{max} - 1)$

  ansonsten falls Verketten dann

  lokaler Aktionsraum += $(n \rightarrow (d_{max} + 1 ... d_{A,L}(n)))$
  $d_{A,S}$ += $(n \rightarrow d_{max})$

  Nachdem alle Namen des Namensraumes wie oben beschrieben abgearbeitet wurden, werden beide Teilausdrücke mit der Anpassungsfunktion $d_{A,S}$ initialisiert.

- Disjunktion - $A \cup B$ (Namensraum)
  Falls ein Name n in nur einem der beiden Teilausdrücke vorkommt werden alle zugehörigen Elemente des Namensraumes des Teilausdrucks einfach in den lokalen Namensraum übernommen. Falls der Name jedoch in beiden Elementen vorkommt sind bezüglich dieses Namens die Dimensionen d_{Amax} und d_{Bmax} sowie die zugehörigen Kardinalitäten c_{Amax} und c_{Bmax} von Relevanz. Die beiden Unternamensräume werden als N_A und N_B bezeichent.

  Falls $d_{Amax} < d_{Bmax}$ dann

  N_A wird dem lokalen Namensraum hinzugefügt.

  Falls $d_{Amax} > d_{Bmax}$ dann

  N_B wird dem lokalen Namensraum hinzugefügt.

  Falls $d_{Amax} = d_{Bmax}$ dann

  Falls $c_{Amax} > c_{Bmax}$ dann

  Dem lokalen Namensraum wird c_{Amax} mal das Paar (n, d_{Amax}) hinzugefügt.

  Falls $c_{Amax} < c_{Bmax}$ dann

  Dem lokalen Namensraum wird c_{Bmax} mal das Paar (n, d_{Bmax}) hinzugefügt.

  Falls $c_{Amax} = c_{Bmax}$ dann

  Falls es in N_A ein (n,d) mit $d < d_{Amax}$ gibt dann

  lokaler Namensraum += $\{(n, d_{Amax}), (n, d)\}$

  ansonsten falls es in N_B ein (n,d) mit $d < d_{Bmax}$ gibt dann

  lokaler Namensraum += $\{(n, d_{Bmax}), (n, d)\}$

  ansonsten

  lokaler Namensraum += (n, d_{Amax})

4.2.5 Prioritäten

Wie bereits im Kapitel 4.1.4 erwähnt, gibt kommt es beim Durchlaufen des NEA zu Mehrdeutigkeiten. Um diese zu beseitigen, müssen bestimmte Wege anderen vorgezogen werden. Dies geschieht mittels Prioritäten. Die Prioritäten werden, genauso wie die Aktionsräume, im Baum der regulären Ausdrücke berechnet.

Damit Prioritäten die Mehrdeutigkeit der NEAs auf ein Minimum reduzieren können, muss jeder Übergang im NEA eine von einem wohldurchdachten Prioritätssystem festgelegte Priorität erhalten. Das in dieser Magisterarbeit verwendete System basiert auf den einzelnen regulären Ausdrücken und versucht die jeweils einfachste Lösung möglichst lokal zu bevorzugen.

Die Prioritäten werden pro Aktionsliste vergeben und sind additiv. Wenn also ein NEA mehrere Möglichkeiten bietet, einen String zu akzeptieren, dann sind jene Möglichkeiten auszuwählen, welche eine höhere inkrementelle Priorität besitzt. Die Priorität einer Aktionsliste wird ab hier vor der schließenden runden Klammer und nach einem Rufzeichen geschrieben. Eine Priorität von 0 wird nicht dargestellt.

Beispiel 4.2.14: zu Demonstrationszwecken aus der Luft gegriffene Ergebnisse

1) `()`**`a`** `(+ab!1)`**`b`**`(-ab!2)`
2) `(+ab)`**`a`** `(-ab!2)`**`b`** `(!1)`

In diesem Fall wird die zweite Möglichkeit bevorzugt, da ersten Elemente beider Aktionslistenlisten dieselbe Priorität (nämlich 0) haben, aber die zweiten beiden Elemente der zweiten Möglichkeit den Vorzug geben. Dass die dritten Elemente in diesem Fall der ersten Möglichkeit den Vorzug geben würden und dass die Gesamtpriorität beider Aktionslistenlisten identisch ist, spielt keine Rolle.

Beispiel 4.2.15: NEA aus `{a,a}**` scannt `"aa"`

1) `(+a[2]+a[1]+a)` **a** `(-a+a!-1)` **a** `(-a-a[1]-a[2])`
2) `(+a[2]+a[1]+a)` **a** `(-a-a[1]+a[1]+a!-2)` **a** `(-a-a[1]-a[2])`

In diesem Fall wird die erste Möglichkeit bevorzugt, da ersten Elemente beider Aktionslistenlisten dieselbe Priorität haben, aber die zweiten beiden Elemente der ersten Möglichkeit den Vorzug geben. Dass die Gesamtpriorität ebenfalls der ersten Möglichkeit den Vorzug geben würde, spielt keine Rolle.

Auf den genauen Zeitpunkt dieser Selektion nach inkrementeller Priorität wird im Kapitel 4.6.2 genauer eingegangen.

Bei der Auswahl der Prioritäten wird nach folgenden Kriterien vorgegangen:

1) Als entscheidungsrelevant gilt die Summe von Prioritäten von nach der Konsumierung des letzten Symbols (bzw. nach dem Start) bis vor der Konsumierung nächsten Zeichens (bzw. nach der Akzeptanz einer Zeichenkette).
2) Ein Zeichen welches im regulären Ausdruck weiter „vorn“ steht wird eher konsumiert als ein Zeichen weiter „hinten“ (Dazu wird eine Totalordnung aller Symbolvorkommen definiert).
3) Falls das System während einem Schritt die Wahl hat, die aktuelle Datenstruktur fortzusetzen oder sie abzuschließen und eine neue Datenstruktur zu erstellen, wird die aktuelle Datenstruktur fortgesetzt.
4) Falls das System während einem Schritt die Wahl hat, eine neue Datenstruktur zu erstellen und diese danach sofort wieder zu beenden oder nichts zu tun, so wird nichts getan.

Unglücklicherweise war es mir nicht möglich aus diesen Kriterien ein in sich stimmiges Modell zu entwickeln, welches auf Basis der regulären Ausdrücke für jeden Übergang im NEA eine Priorität festlegt. Aus diversen Experimenten an der Implementierung dieses Systems hat sich jedoch ein durchaus akzeptabler Algorithmus herauskristallisiert. Dieser wird im Weiteren beschrieben.

Berechnung der Prioritäten (top down)

Der Prioritätsalgorithmus durchläuft den Baum der regulären Ausdrücke rekursiv „top down“ wobei eine ganze Zahl durch den Baum durchgereicht und dabei sukzessive verringert wird. Dieser Wert wird während des Algorithmus sukzessive verringert. Der Name der zugehörigen rekursiven Methode ist hier

int set_priorities(RegExp element, int p)

Je nachdem welchen Typ *element* hat wird eine der unten beschriebenen Prozeduren durchgeführt. Dabei ist der Header vom folgenden Format:

<Typ von *element*> - <Formel, eventuell mit Teilausdrücken> - (<Liste der vom *element*-Typ später benötigten Prioritäten>)

Falls ein Typ in der folgenden Auflistung nicht vorkommt besteht die Prozedur daraus, dass nichts getan wird. Leere Listen von später benötigten Prioritäten werden nicht dargestellt.

Begonnen wird der Algorithmus indem *set_priorities(Wurzelement, 0)* aufgerufen wird.

- Symbol - {a} - (Symbolpriorität p_S)
  $p_S = p$
  return p - 1
- Verkettung - A · B
  p = set_priorities(A, p)
  p = set_priorities(B, p)
  return p
- Disjunktion - A ∪ B
  p = set_priorities(A, p)
  p = set_priorities(B, p)
  return p
- Benennung - {n, A}
  return set_priorities(A, p)
- Kleene Stern - A* - (Rekursivpriorität p_R)
  p_R = set_priorities(A, p) - p
  return p + p_R - 1
- Plus - A^+ - (Rekursivpriorität p_R)
  p_R = set_priorities(A, p) - p
  return p + p_R - 1
- Begrenzte Vielfachheit - A {min,max} - (Rekursivprioritätsbasis p_{RB}, Startpriorität p_S)
  $p_S = p$
  p_{RB} = set_priorities (A, p) - p
  return p + p_{RB} · max - 1
- Minimum - A {min,} - (Rekursivprioritätsbasis p_{RB}, Startpriorität p_S)
  $p_S = p$
  p_{RB} = set_priorities (A, p) - p
  return p + p_{RB} · min - 1

Dieser Algorithmus sorgt unter anderem dafür, dass:

- sich die Symbolprioritäten p_S der einzelnen Symbole alle unterschiedlich sind und bei jenen Symbolen, welche weiter „vorn“ stehen, größer sind als bei jenen, die eher weiter „hinten“ stehen (Kriterium 2).
- die Rekursivprioritäten p_R bzw. die Rekursivprioritätsbasen p_{RB} der „inneren“ regulären Ausdrücke größer sind als jene der „äußeren“ regulären Ausdrücke (Kriterium 3 und 4).

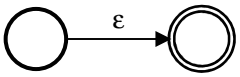
Die Beschreibung der genauen Platzierung der einzelnen Prioritäten im NEA, und damit der Sinn und Zweck der gerade geschilderten Zusammenhänge, erfolgt im nächsten Kapitel.

4.3 Erstellung eines NEAs aus einem regulären Ausdruck

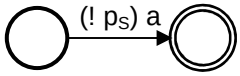
Für jeden bereits beschriebenen regulären Ausdruck gibt es eine Regel wie dieser in einen nichtdeterministischen endlichen Automaten umgewandelt werden kann. Das hinzufügen von Aktionslisten auf Basis der einzelnen Aktionsräume erfolgt für jede Regel auf dieselbe Weise und wird zuletzt beschrieben.

Verwendet werden dabei ausschließlich Vor- bzw. ε-Aktionen. Die entsprechenden Aktionslisten (z.B. „(+a[1]+a ! -13)“ wobei „+a[1]+a“ die Aktionen sind und -13 die zugehörige Priorität) werden also stets vor dem eigentlichen Zeichen des Übergangs konsumiert.

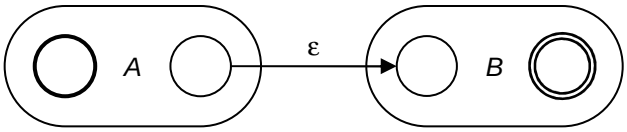
- Epsilon - ε



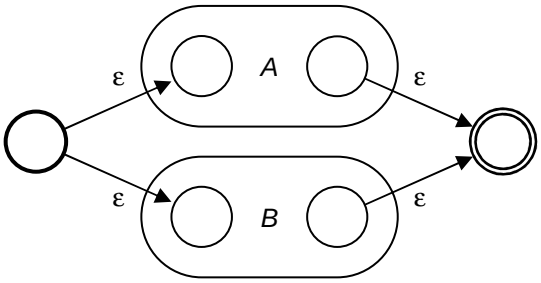
- Symbol - {a} - (Symbolpriorität p_S)



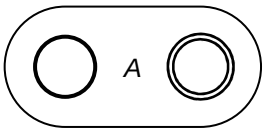
- Verkettung - A · B



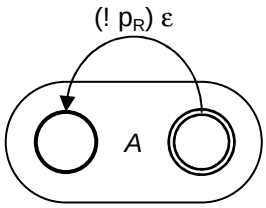
- Disjunktion - A ∪ B



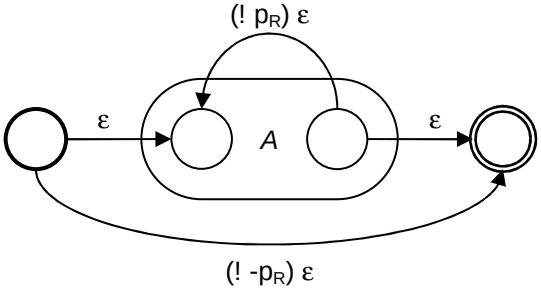
- Benennung - {n, A}



- Plus - A^+ - (Rekursivpriorität p_R)



- Kleene Stern - A* - (Rekursivpriorität p_R)

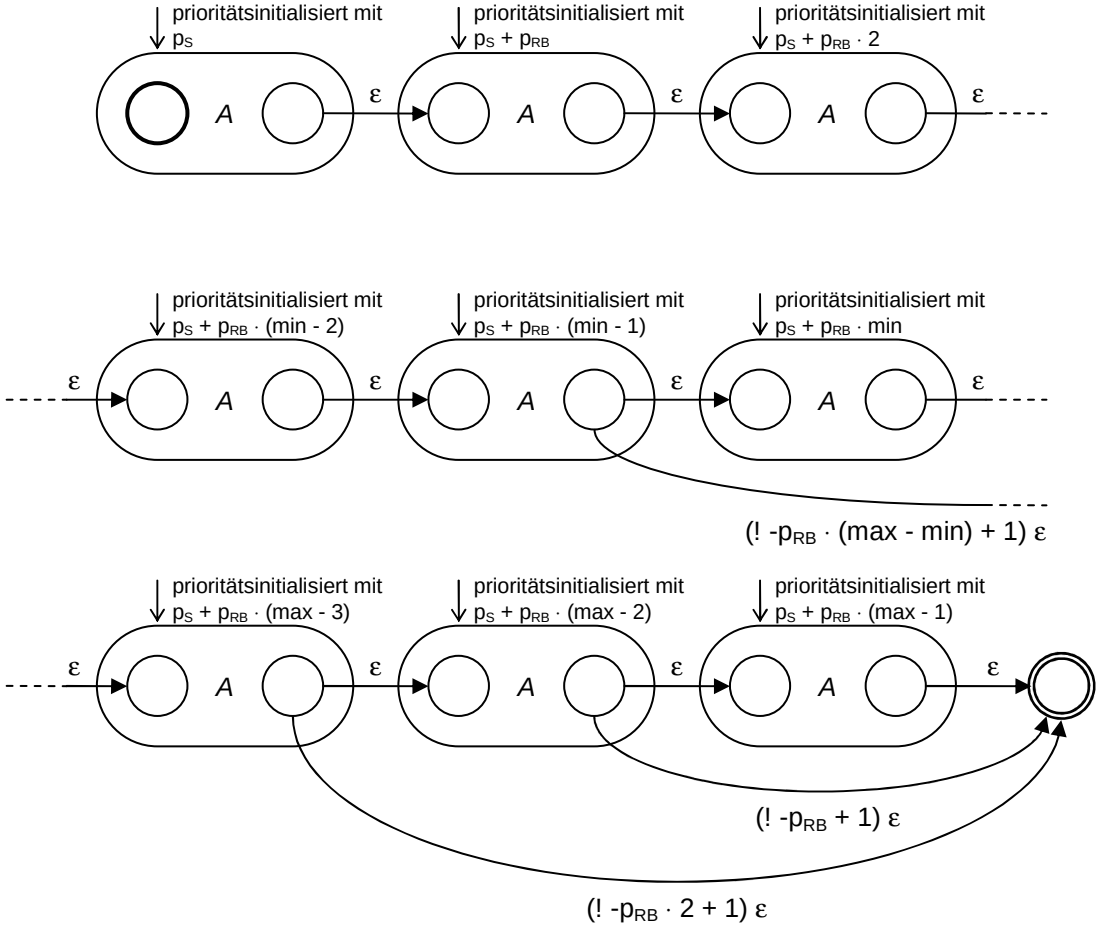


- Begrenzte Vielfachheit - A {min,max} - (Rekursivprioritätsbasis p_{RB}, Startpriorität p_S)

  Hier müssen aus dem regulären Ausdruck A mehrere, bezüglich der enthaltenen Prioritäten verschiedene, NEAs erzeugt und verbunden werden. Jener Wert von *p*, welcher der im Kapitel 4.2.5 beschriebenen Funktion

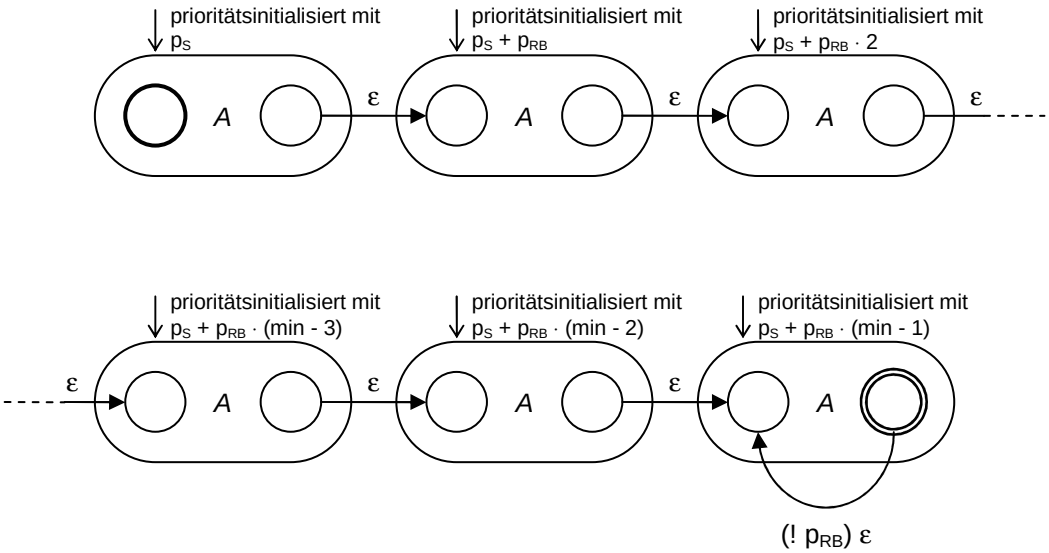
  int set_priorities(RegExp element, int p)

  zusammen mit dem regulären Ausdruck A als *element* übergeben wird, um danach aus ihm den NEA *A* zu erstellen, ist hier für jeden NEA *A* explizit angegeben.



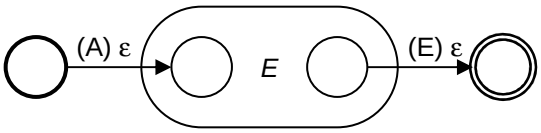
- Minimum - A {min,} - (Rekursivprioritätsbasis p_{RB}, Startpriorität p_S)

  Auch hier müssen aus dem regulären Ausdruck A mehrere, bezüglich der enthaltenen Prioritäten verschiedene, NEAs erzeugt und verbunden werden. Details dazu: siehe letzter Abschnitt.



- Jedes Element mit nicht leerem Aktionsraum

  Falls das erzeugte Element E über einen nicht leeren Aktionsraum verfügt, müssen die dabei anfallenden Aktionen noch vor und nach den betroffenen (Sub)NEA in der richtigen Reihenfolge platziert werden. In der Anfangsaktionsliste A müssen sämtliche Aktionen eines Namens in absteigender Reihenfolge mit positivem Vorzeichen platziert werden. In der Endaktionsliste E gelten die aufsteigende Reihenfolge und das negative Vorzeichen.



  Allerdings können im Fall der Endaktionsliste sämtliche Aktionen mit einer Dimensionalität größer als 0 vernachlässigt werden da sie ganz am Schluss für die Erstellung der Ergebnisdatenstruktur nicht erforderlich sind (siehe Kapitel 4.6.2).

Diese Verteilungen der einzelnen Prioritäten sorgt dafür, dass die Rekursivprioritäten ineinander verschachtelter rekursiver Elemente so platziert sind, dass:
(für die einzelnen Kriterien siehe Kapitel 4.2.5)

- bei der Wahl der rückwärtsgerichteten Kante die innerste mögliche Kante genommen wird (Kriterium 3).
- bei der Wahl der vorwärtsgerichteten Kante die äußerste mögliche Kante genommen wird (Kriterium 4).

Außerdem wird dafür Sorge getragen, dass die Kriterien 2, 3 und 4 meist auch dann erfüllt sind, falls mehrere platzierte Prioritäten miteinander in Konflikt geraten (Kriterium 1).

4.4 Konstruktion eines deterministischen Automaten

Bei der Umwandlung eines NEA mit Aktionslisten in einen entsprechenden DEA kann die Potenzmengenkonstruktion verwendet werden. Mehr Informationen zu diesem und ähnlichen Algorithmen finden sich im „Drachenbuch“ [AhSU86].

Bezüglich der allgemeinen Verwendung von NEAs und DEAs im Bereich der Lexikalischen Analyse empfehle ich Ihnen ein Buch von Klaus Brouwer, Wolfgang Gellerich und Erhard Plödereder [BrGePl98], sowie ein Buch von J. Van Leeuwen [Le94].

Die Anwesenheit von Aktionen hat keinerlei Folgen für die Struktur des resultierenden DEAs. Allerdings müssen die im NEA vorkommenden Aktionslisten natürlich irgendwie im DEA repräsentiert werden.

Dies erfolgt mittels Aktormengen auf jedem DEA-Übergang sowie einer Reihe von Regeln, die beschreiben welche Aktoren aufeinander folgen bzw. am Beginn und am Ende eines erkannten Strings stehen können.

Aktoren:

Aktoren sind Referenzen auf Aktionslisten, welche eine eigene einzigartige Identität bekommen haben. Dies kann beispielsweise eine bestimmte Position im NEA sein. So können beispielsweise alle Aktionslisten in einem NEA durchnumeriert werden. Diese Nummern wären dann die Aktoren. Zwei Aktoren sind identisch wenn sie dieselbe Aktionsliste referenzieren. Aktoren treten auch in Mengen auf, den Aktormengen.

Aktorlisten:

Diese Listen stellen eine Zwischenrepräsentation der Aktionslisten im Übergang von den NEA Aktionslisten und den DEA Aktormengen dar. Eine Aktorliste beinhalten eine Reihe von Aktoren sowie einen zugehörende NEA Basis- und Folgezustand. Auch sie treten in Mengen, den Aktorlistenmengen, auf. Dabei gelten zwei Aktorlisten als identisch wenn sie dieselben Aktoren in derselben Reihenfolge enthalten, sowie denselben NEA Basis- und Folgezustand teilen. Auf Aktorlistenmengen können notwendige Optimierungen durchgeführt werden.

4.4.1 Die Potenzmengenkonstruktion

Wie bereits erwähnt, kann die Determinierung mit Hilfe der Potenzmengenkonstruktion durchgeführt werden. Dabei wird zuerst vom Startzustand ausgehend die erste Potenzmenge bestimmt. Diese ist dabei die Menge all jener NEA-Zustände, welche vom Startzustand ausgehend ohne Konsumierung eines Symbols erreicht werden können (die ε-Umgebung des Startzustandes). Ausgehend von dieser Startpotenzmenge werden nun die Folgepotenzmengen berechnet.

Dabei wird, ausgehend von der jeweils betrachteten Potenzmenge, pro konsumierbares Symbol, eine Menge von Folgezuständen erzeugt. Falls ein Zustand der betrachteten Potenzmenge über einen das Symbol s konsumierenden Übergang zum Zustand F verfügt, so findet sich der Zustand F in der Menge der Folgezustände bezüglich s der betrachteten Potenzmenge. Die Folgepotenzmenge bezüglich s ist dann die Vereinigung der ε-Umgebungen aller Zustände der Folgemenge bezüglich s.

Dieser Vorgang muss für alle neuen Potenzmengen wiederholt werden, bis alle Potenzmengen und ihre Folgepotenzmengen bekannt sind.

Der DEA ergibt sich dann folgendermaßen: Jeder Potenzmenge P wird ein DEA Zustand Z_S zugeordnet. Dann wird für jede Potenzmenge A für jedes Symbol s mit existenter Folgepotenzmenge B, ein Übergang von Z_A zu Z_B, welcher s konsumiert, zum DEA hinzugefügt.

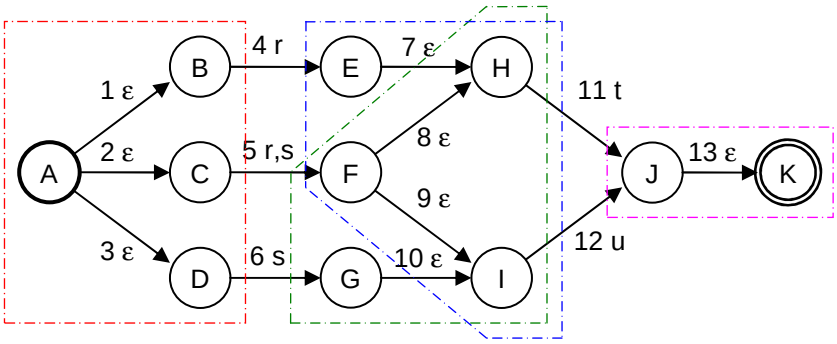
Jener der Startpotenzmenge S zugeordnete DEA-Zustand Z_S wird zum Startzustand des DEAs. Falls eine Potenzmenge P einen NEA-Endzustand beinhaltet, so ist Z_P ein Endzustand.

Sollen allerdings Aktionslisten mitverarbeitet werden sind gewisse Aspekte zu beachten. Bezüglich der Bestimmung der Aktorlisten ist es notwendig jede Potenzmenge nicht als Gesamtheit zu sehen sondern als Menge jener NEA-Zustände, durch welche die Potenzmenge von „außen“ ohne ε-Übergang erreicht werden kann.

Visualisierung:

Beispiel 4.4.1: Demonstrationsbeispiel

Die Aktionslisten wurden hier schon durch durchnummerierte Aktoren (1 - 13) ersetzt.



1 → (+x)
2 → (+x)
3 → (+x)
4 → (+r)
5 → (+rs)
6 → (+s)
7 → (-r)
8 → (-rs)
9 → (-rs)
10 → (-s)
11 → ()
12 → ()
13 → (-x)

In diesem Beispiel ergibt sich aus dem Startzustand zuerst die Potenzmenge PM_1 {A,B,C,D}. Bezüglich der Symbole „r“ und „s“ kommen wir in einem Schritt zu PM_2 {E,F,H,I} und PM_3 {F,G,H,I}. Zuletzt wird, ausgehend von den letzten beiden PMs, über die Symbole „t“ bzw. „u“ die PM_4 {J,K} erzeugt.

Diese Potenzmengen sind gute Repräsentanten, aber für die Bestimmung der Aktorlisten pro Symbol ist es notwendig, jene Zustände zu kennen durch welche diese Potenzmenge erreichbar ist (Also die Teilmenge jener Zustände welche einen eingehenden Symbolübergang haben oder Startzustand sind).

Das wären in für dieses Beispiel für PM_1 {A}, für PM_2 {E,F}, für PM_3 {F,G} und für PM_4 {J}. Ausgehend von diesen Basiszuständen werden die einzelnen Aktorlisten berechet.

4.4.2 Die Berechung der Aktorlisten

Die Aktorlisten werden pro Potenzmenge pro Symbol bzw. Endzustand berechnet und in Mengen zusammengefasst.

Bei der Berechnung der Aktorlisten wird ähnlich vorgegangen wie bei der Berechnung der kompletten Potenzmenge. Ausgehend von den Basiszuständen wird die Epsilonumgebung mit Tiefensuche durchlaufen. Dabei werden die jeweils passierten Aktoren mitgeschrieben. Dabei wird ein bereits besuchter Zustand wieder als „noch nicht besucht“ markiert, wenn die Tiefensuche diesen Zustand wieder vom Stack entfernt. Auf diese Art und Weise werden auch mehrere Wege zwischen zwei Zuständen erkannt.

Falls während der Suche ein Zustand erreicht wird, von dem ein Symbolübergang mit dem Symbol s ausgeht, wird der mitgeschriebenen Aktorliste noch jener dem Symbolübergang zugehörigen Aktor hinzugefügt. Die resultierende Liste wird dann zusammen mit dem aktuellen Basiszustand und dem Zielzustand des Symbolübergangs in der Aktorlistenmenge des Symbols s der aktuellen Potenzmenge gespeichert.

Sollte ein Endzustand erreicht werden, wird die mitgeschriebene Aktorliste mit dem aktuellen Basiszustand und einem für alle Endzustände desselben Typs gleichem Pseudozustand in der Aktorlistenmenge des erreichten Endzustandes der aktuellen Potenzmenge gespeichert.

Bezüglich des Beispiels 4.4.1 ergeben sich damit die folgenden Aktorlistenmengen:

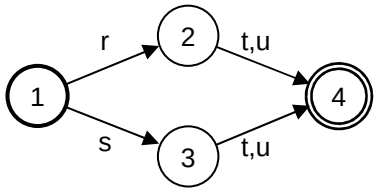
PM_1:
„r“: PM_2 {A→[1,4]→E, A→[2,5]→F}
„s“: PM_3 {A→[2,5]→F , A→[3,6]→G}

PM_2:
„t“: PM_4 {E→[7,11]→J , F→[7,11]→J}
„u“: PM_4 {F→[9,12]→J}

PM_3:
„t“: PM_4 {F→[8,11]→J}
„u“: PM_4 {F→[9,12]→J , G→[10,12]→J}

PM_4:
Ende: {J→[13]→„Ende“}

Woraus sich wiederum folgende DEA-Struktur ergibt:
(Aktorlistenmengen nicht eingezeichnet)



4.4.3 Optimierungen auf Aktorlistenmengen

Nachdem alle einer Potenzmenge zugehörigen Aktorlistenmengen berechnet wurden, können diese optimiert werden. Dies ist sehr empfehlenswert, da die Berechnung der finalen Ergebnisstrukturen im Scanner ansonsten in bestimmten Fällen zu einem exponentiellen Aufwand führt.

Bei der Optimierung wird die Aktorlistenmenge in Teilmengen von Aktorlisten eingeteilt, welche alle denselben NEA Basis- und Folgezustand haben.

Pro Teilmenge werden jene Elemente bestimmt deren Gesamtpriorität (die Summe aller in der Aktorliste referenziell enthaltenen Aktionslisten) am höchsten ist. Diese verbleiben in der ursprünglichen Gesamtmenge. Alle anderen werden aus derselben entfernt. Die zu jeder Aktorliste gehörenden Basis- und Folgezustände sind ab dieser Optimierung ebenfalls nicht mehr notwendig.

Beispiel 4.4.2: Optimierung einer aus der Luft gegriffenen Aktorlistenmenge

Ausgangsmenge:

```
{ A→  [(-a),(!1),(+a)]  →C,
  A→  [(!1),(!1)]       →C,
  A→  [(-a),(!2),(+b)]  →C,
  A→  [(-a),(!1),(+b)]  →D,
  A→  [(-a),(!2),(+c)]  →D,
  B→  [(-a),(+d)]       →E  }
```

Teilmengen mit Aktorlistengesamtprioritäten:

```
A→C: {  [(-a),(!1),(+a)],   Priorität 1
        [(!1),(!1)],        Priorität 2 (max)
        [(-a),(!2),(+b)]    Priorität 2 (max)  }

A→D: {  [(-a),(!1),(+b)],   Priorität 1
        [(-a),(!2),(+c)]    Priorität 2 (max)  }

B→E: {  [(-a),(+d)]         Priorität 0 (max)  }
```

Optimierte Aktorlistenmenge:

```
{ [(!1),(!1)],
  [(-a),(!2),(+b)],
  [(-a),(!2),(+c)],
  [(-a),(+d)]  }
```

4.4.4 Komprimierung zu Aktormengen

Nach der Optimierung können die einzelnen Aktorlistenmengen zu Aktormengen komprimiert werden. Die während der Optimierung noch benötigten NEA Basis- und Folgezustände können dabei ignoriert werden.

Die einzelnen Aktoren einer Aktorliste werden dabei zu einem einzigen verschmolzen indem die zugehörigen Aktionslisten aneinandergehängt und die von ihnen verwendeten Prioritäten aufsummiert werden. Die resultierende Aktionsliste hat dabei nur in trivialen Fällen einen Platz im NEA. Jeder neuen oder alten Aktionsliste muss daher eine eindeutige Nummer zugeordnet werden damit sie als Aktor in eine Aktormenge eingefügt werden kann.

Einen Sonderfall stellt eine leere Aktorliste dar. Aus ihr kann kein Aktor gebildet werden. Aus diesem Grund ist es notwendig in jeder Aktormenge extra zu vermerken ob ein derartiges Element in der ursprünglichen Aktorlistenmenge enthalten war (Aktormenge enthält Epsilon) oder nicht (Aktormenge enthält kein Epsilon).

Beispiel 4.4.3: Komprimierung einer künstlichen Aktorlistenmenge

Optimierte Aktorlistenmenge:

```
{ [(!1),(!1)],
  [(-a),(!2),(+b)],
  [(-a),(!2),(+c)],
  [(-a),(+d)]  }
```

Zugehörige Aktormenge:

```
{ (!2),
  (-a+b!2),
  (-a+c!2),
  (-a+d)  }
```

4.4.5 Bestimmung der Folgeregeln

Die Folgeregeln sind ein Regelwerk welches aufgrund des NEAs festlegt, welcher Aktor auf welchen anderen Aktor folgen kann. Außerdem wird festgelegt welche Aktoren am Start und am Ende stehen können, sowie ob das Ende ohne Aktoren dazwischen auf den Start folgen kann.

Die Folgeregeln kapseln all jene benennungsrelevanten Strukturen des NEAs welche nicht im DEA repräsentiert werden können. Mit ihrer Hilfe können aus der Menge der während eines Scannvorgangs aufgesammelten Aktoren jene für die finalen Datenstrukturen relevanten Aktorlisten gebildet werden.

Diese Regeln beziehen sich nur auf relevante Aktoren, also auf jene welche in den im letzten Kapitel erzeugen Aktormengen enthalten sind. Die darin enthaltenen Aktoren sind aber nur in trivialen Fällen im NEA enthalten. Aus diesem Grund ist es notwendig diesen Aktoren je zwei Aktoren zuzuweisen, welche im NEA enthalten sind. Der eine heißt erster NEA-Aktor und ist das erste Element jener Aktorliste, aus der der betreffende relevante Aktor hervorgegangen ist. Der andere heißt letzter NEA-Aktor und ist das letzte Element jener Aktorliste. Beide NEA-Aktoren sind logischerweise identisch, falls die ursprüngliche Aktorliste nur ein Element enthält.

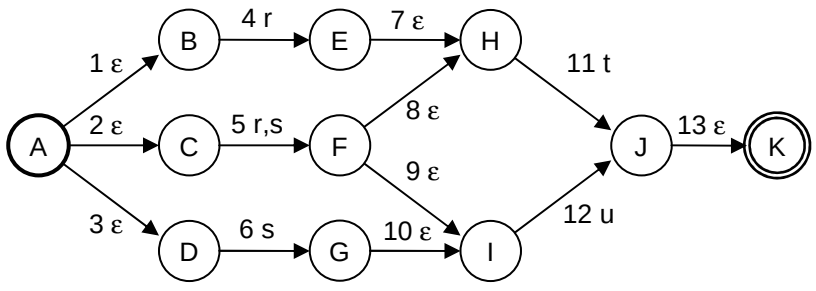
Die Bestimmung der Folgeregeln läuft so ab, dass für jeden Aktor des NEAs eine Tiefensuche bis zu den jeweils nächsten Aktoren durchgeführt wird. Wenn also ein Aktor A_2 auf einen Aktor A_1 folgen kann, müssen jene Paare von relevanten Aktoren (RA_1, RA_2) eruiert werden, bei denen das letzte NEA - Element von RA_1 identisch mit A_1 und das erste NEA - Element von RA_2 identisch mit A_2 ist. Für alle diese Paare gilt: RA_2 kann auf RA_1 folgen.

Für die Sonderfälle Startzustand und Endzustand ist die Vorgehensweise prinzipiell identisch, nur eben mit dem Startzustand als Ausgangspunkt der Tiefensuche bzw. mit einem Endzustand als Ergebnis einer Tiefensuche.

Beispiel 4.4.4: Folgeregeln von Beispiel 4.4.1

<u>NEA mit Aktoren aus Beispiel 4.4.1:</u>

Die „Inhalte“ der Aktoren, also die Aktionslisten, sind für die Bestimmung der Folgeregeln ohne Relevanz. Sie werden nur der Übersicht halber dargestellt.



1 → (+x)	8 → (-rs)
2 → (+x)	9 → (-rs)
3 → (+x)	10 → (-s)
4 → (+r)	11 → ()
5 → (+rs)	12 → ()
6 → (+s)	13 → (-x)
7 → (-r)	

Potenzmengen, Folgepotenzmengen und Aktorlistenmengen aus Beispiel 4.4.1:

PM_1:
„r“: PM_2 {A→[1,4]→E, A→[2,5]→F}
„s“: PM_3 {A→[2,5]→F , A→[3,6]→G}

PM_2:
„t“: PM_4 {E→[7,11]→J , F→[8,11]→J}
„u“: PM_4 {F→[9,12]→J}

PM_3:
„t“: PM_4 {F→[8,11]→J}
„u“: PM_4 {F→[9,12]→J , G→[10,12]→J}

PM_4:
Ende: {J→[13]→„Ende“}

Nach der Komprimierung: (Optimierung wird hier weggelassen)

Alle hier auftretenden Aktoren sind relevante Aktoren. Weiters werden zusätzliche „zusammengesetzte“ Aktoren (14 - 20) eingeführt.

PM_1:
„r“: PM_2 {14,15}
„s“: PM_3 {15,16}

PM_2:
„t“: PM_4 {17,18}
„u“: PM_4 {19}

PM_3:
„t“: PM_4 {18}
„u“: PM_4 {19,20}

PM_4:
Ende: {13}

Eine hilfreiche Tabelle mit den Eigenschaften der „zusammengesetzten“ Aktoren:

Nummer	Ursprungs-aktorliste	erster NEA-Aktor	letzter NEA-Aktor	Inhalt
14	[1,4]	1	4	(+x+r)
15	[2,5]	2	5	(+x+rs)
16	[3,6]	3	6	(+x+s)
17	[7,11]	7	11	(-r)
18	[8,11]	8	11	(-rs)
19	[9,12]	9	12	(-rs)
20	[10,12]	10	12	(-s)

Ausnahmsweise haben hier alle „zusammengesetzten“ Aktoren zwei Elemente. Das muss aber nicht immer so sein.

NEA-Folgeregeln für NEA-Aktoren (ohne „zusammengesetzte“ Aktoren):

direkt aus dem NEA abgeleitet

„Start“ → 1,2,3
1 → 4
2 → 5
3 → 6
4 → 7
5 → 8,9
6 → 10
7 → 11
8 → 11
9 → 12
10 → 12
11 → 13
12 → 13
13 → „Ende“

sowie: „Ende“ kann *nicht* auf „Start“ folgen

Folgeregeln für die relevanten Aktoren:

aus NEA-Folgeregeln und dem jeweils ersten und letzten NEA-Aktor abgeleitet

Hier sind ausschließlich Beziehungen zwischen relevanten Aktoren (13 - 20) vertreten.

„Start“ → 14,15,16	16 → 20	19 → 13
14 → 17	17 → 14	20 → 13
15 → 18,19	18 → 14	13 → „Ende“

sowie: „Ende“ kann *nicht* auf „Start“ folgen

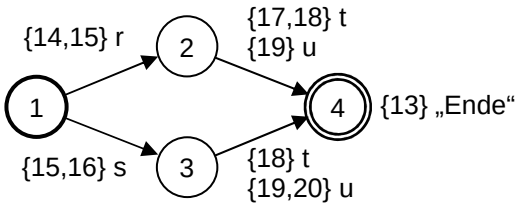
4.5 Optimierung eines deterministischen Automaten

Die Optimierung eines deterministischen Automaten mit Aktormengen kann mit denselben Algorithmen erfolgen wie die Optimierung eines normalen deterministischen Automaten. Falls im Rahmen der Optimierung zwei identische Zustände entdeckt und zusammengelegt werden, müssen die jeweiligen Aktormengen pro Symbol bzw. Endzustand nur miteinander vereinigt werden.

Dabei geht keine Information verloren, da die Folgeregeln des DEAs das über den DEA gespannte Geflecht von Aktoren weiterhin „in Form“ halten. Nur zwei Knoten eben dieses Geflechts werden übereinander gelegt.

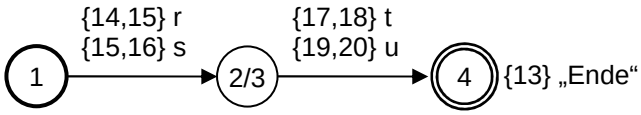
Beispiel 4.5.1: Optimierung des DEAs aus Beispiel 4.4.4

DEA:



Hier sind die Zustände 2 und 3 offensichtlich ununterscheidbar, da sie bezüglich jedes Symbols denselben Folgezustand haben.

Optimierter DEA:



4.6 Der Scanvorgang

Während des Scannens werden die Aktormengen der jeweils durchgeführten Übergänge mitgeschrieben. Die Aktormenge des erreichten Endzustandes wird ebenfalls notiert. Letztendlich ergibt der Scanvorgang den erkannten String der Länge L und eine Aktormengenliste der Länge L+1. Diese beiden Elemente werden anschließend zu den finalen Ergebnisstrukturen weiterverarbeitet.

4.6.1 Lösungssuche

Aus der Aktormengenliste können durch Auswahl der darin enthaltenen Elemente viele Kombinationen der gleichen Länge gewonnen werden. Diese Kombinationen enthalten dabei an jeder Position ein Element jener Aktormenge an selben Position in der Aktormengenliste oder ein Epsilon falls diese Menge ein solches enthält.

Nun gilt es aus allen möglichen Kombinationen möglichst effizient jene Listen herauszusuchen deren Elemente in dieser Reihenfolge den Folgeregeln genügen. Dabei leitet ein Epsilon die entsprechenden Regeln einfach an das nächste Element weiter, da es aus einer leeren Aktorliste erstellt wurde.

Dies geschieht am Besten ausgehend vom letzten Element der Liste und jenen dem zugehörigen Endzustand zugeordneten Folgeregeln rekursiv nach vorne (Eine rekursive Prozedur von vorne nach hinten ist ebenfalls möglich, aber in den meisten praktischen Fällen wäre das suboptimal).

Wenn eine Aktormenge erreicht wird, werden jene Aktoren, welche von dem letzten Aktor bzw. vom Endzustand gefolgt werden können herausgesucht und pro Aktor mit ihm selbst als letztem Aktor weitergemacht. Falls die Aktormenge außerdem noch ein Epsilon enthält, wird dieser Index zusätzlich noch übersprungen. Sollten keine passenden Aktoren gefunden werden und die Aktormenge kein Epsilon enthalten, so ist die aktuelle Teilkombination ungültig. Wenn das Ende der Liste erreicht wird, muss noch überprüft werden, ob der letzte Aktor bzw. der Endzustand auf den Start folgen kann. Falls dem so ist, dann wurde eine gültige Kombination gefunden.

Jede gültige Kombination ist eine (eventuell Epsilons enthaltende) Aktorliste der Länge L+1. Zusammen bilden sie eine Aktorlistenmenge.

Beispiel 4.6.1: Scannen mit dem DEA aus Beispiel 4.5.1

```
"rt"   =DEA=>           [{14,15},{17,18},{13}]
        =Folgeregeln=>  {[14,17,13],[15,18,13]}
         =Referenzen=>  {[(+x+r),(-r),(-x)],[(+x+rs),(-rs),(-x)]}
          =entspricht=> (+x+r)r(-r)t(-x) bzw. (+x+rs)r(-rs)t(-x)

"ru"   =DEA=>           [{14,15},{19,20},{13}]
        =Folgeregeln=>  {[15,19,13]}
         =Referenzen=>  {[(+x+rs),(-rs),(-x)]}
          =entspricht=> (+x+rs)r(-rs)u(-x)

"st"   =DEA=>           [{15,16},{17,18},{13}]
        =Folgeregeln=>  {[15,18,13]}
         =Referenzen=>  {[(+x+rs),(-rs),(-x)]}
          =entspricht=> (+x+rs)s(-rs)t(-x)

"su"   =DEA=>           [{15,16},{19,20},{13}]
        =Folgeregeln=>  {[15,19,13],[16,20,13]}
         =Referenzen=>  {[(+x+rs),(-rs),(-x)],[(+x+s),(-s),(-x)]}
          =entspricht=> (+x+rs)s(-rs)u(-x) bzw. (+x+s)s(-s)u(-x)
```

4.6.2 Lösungsauswahl

Aus der im letzten Kapitel bestimmten Aktorlistenmenge werden nun, wie in Kapitel 4.2.5 beschrieben, jene Aktorlisten ausgewählt, welche die höchste inkrementelle Priorität haben. Epsilons haben dabei die Priorität 0 (für Beispiele für Selektionen nach inkrementeller Priorität siehe ebenfalls Kapitel 4.2.5).

4.6.3 Aufbau der finalen Strukturen

Nun wird pro übrig gebliebener Aktorliste pro Name eine zugehörige Datenstruktur erstellt. Dabei fällt für jede Aktorliste pro vorkommenden Namen eine entsprechende Datenstruktur an.

Dabei wird die Aktorliste für jeden vorkommenden Namen Element für Element von vorn nach hinten durchgegangen. Die erste enthaltene Aktion mit diesem Namen bestimmt die Dimension der Datenstruktur. Für jede weitere „+“ Aktion wird eine entsprechend dimensionale Teilstruktur an der „richtigen“ Stelle in der Datenstruktur erstellt.

Diese „richtige“ Stelle ist beim Lesen einer n-dimensionalen Aktion die Stelle hinter dem letzten n-dimensionalen Element des allerletzten n+1-dimensionalen Elements der Ergebnisdatenstruktur.

Falls eine „-“ Aktion der Dimension null gefunden wird, ist der Substring ausgehend von der zugehörigen Position in der Aktorliste bis zu der Position der letzten „+“ Aktion desselben Namens mit der Dimension 0 der Inhalt der dabei erstellten nulldimensionalen Struktur.

„-“ Aktionen mit einer Dimension größer als 0 können ignoriert werden.

Beispiel 4.6.2: Erstellen von Ergebnisdatenstrukturen (aus der Luft gegriffen)

```
(+a[2]+a[1]+a+as)a(-a+a)a(-a-a[1]-a[2]-as)

1.1)  +a[2]  =>   a-2-[]
1.2)  +a[1]  =>   a-2-[[]]
1.3)  +a     =>   a-2-[[?]]
1.4)  +as    =>   a-2-[[?]], as-0-?

2.1)  -a     =>   a-2-[["a"]], as-0-?
2.2)  +a     =>   a-2-[["a",?]], as-0-?

3.1)  -a     =>   a-2-[["a","a"]], as-0-?
3.2)  -a[1]  =>   a-2-[["a","a"]], as-0-?    (keine Veränderung!)
3.3)  -a[2]  =>   a-2-[["a","a"]], as-0-?    (keine Veränderung!)
3.4)  -as    =>   a-2-[["a","a"]], as-0-"aa"
```

4.6.4 Lösungsvereinigung

Möglicherweise entstehen bei dem oben beschriebenen Algorithmus bei mehreren verarbeiteten Aktorlisten für einen Namen mehrere Datenstrukturen (also mehrere Ergebnisse). Dies ist von der Effizienz des Prioritätssystems abhängig. Falls dies passiert auftritt, muss eine dieser Datenstrukturen ausgewählt werden. Welche Datenstruktur ausgewählt wird ist eigentlich egal, solange der verwendete Vergleichsoperator transitiv ist.

5 Implementierung

Eine Implementierung des bisher vorgestellten Systems erfolgte als Erweiterung des Scannergenerators Flex (Version 2.5.4). Die Eigenschaften dieses erweiterten Scannergenerators werden im Weiteren beschrieben. Zum Einarbeiten in Flex empfehle ich die zugehörige Bedienungsanleitung [Pax90].

5.1 Interface

5.1.1 Benannte reguläre Ausdrücke

In der Flex Erweiterung kann prinzipiell jedes Element benannt werden, welches ein regulärer Ausdruck ist. Das gilt sowohl für die Beschreibung der gesuchten Strings als auch für den „Trailing Context“. Die zugehörige Syntax lautet:

```
singelton : '{' NAME ',' re '}'
```

Unabhängig von der Benennung hat dieses Konstrukt auch eine Klammerwirkung bezüglich der inneren EBNF Notation.

Beispiel 5.1.1: Flex-Ausdrücke mit benannten Teilausdrücken

Verkettung: `{A,a}{A,a}`

Mehrfache Benennung: `{A1,{A2,a}}`

Kleene-Sterne: `{A,a*}*`

Klammerwirkung: `{AB,a|b}c  <=>  {AB,(a|b)}c`

5.1.2 Ergebnisstrukturen

Für jede vorkommende Dimensionalität wird ein eigener Datentyp definiert. Die entsprechenden Instanzen werden am Heap erzeugt und beim Scannen des nächsten Token gelöscht. Sollte die Dimensionalität eines Ergebnisses 0 sein, dann wird nicht der entsprechende Datentyp, sondern der enthaltene String direkt zur Verfügung gestellt. Als Variablennamen werden die in der Regel verwendeten Bezeichner verwendet.

5.2 Berechung

Die Berechnung des Scanners sowie der Scanvorgang selbst erfolgt prinzipiell genau so wie in Kapitel 3 beschreiben.

Bei der Erstellung der Aktorlisten werden allerdings alle trivialen Aktionslisten (keine Aktionen, Priorität 0) einfach ignoriert. Dies ist ausschließlich in der dadurch verbesserten Performance begründet und hat keine Folgen für das Ergebnis.

Weiters werden die „-“ Aktionen mit einer Dimensionalität größer als 0, wie in Kapitel 4.1.3 bzw. 4.6.3 empfohlen, nicht in den NEA geschrieben, sondern ignoriert.

Da Flex bei mehreren miteinander verketteten Elementen eine implizite Klammerung vorsieht, kommt es in bestimmten Fällen nicht zu den gewünschten Ergebnisstrukturen. Eine explizite Klammerung löst das Problem allerdings sofort wieder.

Beispiel 5.2.1: Probleme mit impliziter Klammerung

```
abcd  <=>  ((ab)c)d
```

```
{A,a}**-{A,a}{A,a}  <=>  (({A,a}**-){A,a}){A,a}
```

Daraus ergibt sich ein Problem, falls der Name A mit Zusammenschluss ausgewertet werden soll (siehe Kapitel 4.2.4).

Problem: `{A,a}**-{A,a}{A,a}` (A mit Einbindung und mit Zusammenschluss)

```
"a-aa"  =>  (+A[2]+A[1]+a)a(-A)-(+A[1]+A)a(-A+A[1]+A)a(-A)
         => A-2-[["a"],["a"],["a"]]
```

statt wegen des „mit Zusammenschluss“ erwartetem: `A-2-[["a"],["a","a"]]`

Lösung: `{A,a}**-({A,a}{A,a})`

```
"a-aa"    =>  (+A[2]+A[1]+a)a(-A)-(+A[1]+A)a(-A+A)a(-A)
      => A-2-[["a"],["a","a"]]
```

5.3 Wichtige Änderungen an Flex

5.3.1 Notwendige architekturelle Änderung

Die Regel-NEAs werden nun nicht mehr während des Parsens der einzelnen Regeln der Eingabe erstellt. Stattdessen wird ein Baum des regulläreren Ausdrucks aufgebaut. Auf diesem werden dann bei Bedarf diverse, zur Bestimmung der Aktionsräume, Namensräume sowie der Prioritäten notwendige, Algorithmen ausgeführt. Danach wird aus dieser Datenstruktur der Regel-NEA generiert.

5.3.2 Einschränkung der Defaultimplementierung des DEAs im Scanner

Flex benutzt zur Komprimierung der DEA-Tabelle des generierten Scanners so genannte „Protos“ und „Templates“. Falls Aktionen verarbeitet werden, sind diese Komprimierungen aus Gründen der Einfachheit deaktiviert.

5.3.3 Einschränkung der anderen Scannerimplementierungen

Flex beherrscht außer der Default-Tabellenstruktur noch zwei weitere Tabellenstrukturen, „full table“ und „full speed“. Beide Tabellenstrukturen werden von der programmierten Flex Erweiterung nicht unterstützt. Das Verhalten des Programms bei der Verwendung der zugehörigen Optionen „-f“ bzw „-F“ ist undefiniert.

5.4 Performance

Diese Erweiterung ist aufgrund des hinzugefügten Benennungssystems natürlich etwas langsamer als das Original. Das gilt sowohl für die Erweiterung selbst, als auch für die von ihr generierten Scanner.

Sollte die Erweiterung ohne die Verwendung von Aktionen benutzt werden, so ist die Performanceverschlechterung der Erweiterung selbst vernachlässigbar und die von ihr generierten Scanner sind genau so schnell wie jene des Originals.

In Extremfällen kann es bei der Verwendung von Aktionen jedoch zu ernsthaften Performanceeinbrüchen kommen.

5.4.1 Laufzeitmessungen

Laufzeitmessungen wurden nur auf Basis einer beschränkten Menge von relativ einfachen regulären Ausdrücken und Eingabedaten durchgeführt. Dies hatte den Grund, dass es für Flex noch keine ausreichend realistischen, die Erweiterung nutzenden, Eingabedateien gibt. Und das Verhalten der Flex Erweiterung bei konventionellen Eingabedateien ist nahezu identisch mit dem von Flex 2.5.4.

Die Messung erfolgte auf dem folgenden System:

Prozessor: AMD Athlon 1.0 GHz
Arbeitsspeicher: 256 MB RAM
Betriebssystem: SuSE Linux 9.0
GCC Version: 3.3.1 (SuSE Linux)

Für die Laufzeitmessungen wurde die Flex Erweiterung mit den C-Compileroptionen „-O2“ kompiliert. Die Kompilierung des Scanners erfolgte mit den Optionen „-O3“. Keinerlei Debuginformationen jeglicher Art wurden generiert.

Gemessen werden immer jeweils 3 Zeiten; jene Zeit, welche die Flex Erweiterung für das Generieren des Scanners benötigt, der Zeitaufwand für das Kompilieren des Scanners, sowie der Zeitaufwand für das Scannen selbst. Außerdem wird die Größe des Scanners als Quellcode und als ausführbare Datei in Byte angegeben.

Beispiel aus der Einleitung:

Zum einen habe ich das Beispiel aus der Einleitung einem Performancetest unterzogen. Die beiden Eingabedateien entsprechen den beiden in der Einleitung vorgestellten Beispielen. Die Eingabe für die generierten Scanner sind 1 Million zufällig generierte Fließkommazahlen mit nicht mehr als 10 Zeichen für Basis, Vorkommastellen, Nachkommastellen sowie Exponenten.

Test	Flex	Quellcode	GCC	Scanner	Scanvorgang
ohne Benennung	0s	37.192	0.63s	17.138	1.02s
mit Benennung	0s	57.013	1.06s	24.500	7.07s

Durch die Verwendung von Benennungen statt handgeschriebenem Code wächst der Scanner selbst etwa auf das Doppelte an, die Dauer des Scanvorgangs verachtfacht sich in etwa. Jener in der Eingabedatei verwendete Code schrumpft allerdings auf etwa ein Zwölftel.

Extreme Ausdrücke:

Außerdem wurden verschiedene reguläre Ausdrücke einzeln getestet. Die dabei verwendete Inputdatei für die Flex Erweiterung, wobei <ausdruck> jeweils durch den getesteten Ausdruck zu ersetzen ist, ist:

```
%%
<ausdruck>
.
"\n"
%%
```

Der die Eingabe für alle generierten Scanner ist ein String bestehend aus 100.000 ‚a’s.

Test	Regulärer Ausdruck	Flex	Quellcode	GCC	Scanner	Scanvorgang
1	a{1000}a*	0.01s	81.343	0.70s	30.505	0.02s
2	{A,a}{1000}{A,a}*	0.06s	196.352	1.33s	68.971	0.15s
3	a*a{1000}	0.78s	81.301	0.70s	30.473	0.02s
4	{A,a}*{A,a}{1000}	8.51s	3.455.080	9.11s	1.071.979	1.07s
5	a{2000}a*	0.03s	126.913	0.80s	44.521	0.02s
6	{A,a}{2000}{A,a}*	0.20s	339.572	1.65s	115.147	0.15s
7	a*a{2000}	5.86s	126.871	0.79s	44.521	0.02s
8	{A,a}*{A,a}{2000}	63.56s	13.368.147	38.35s	4.121.131	2.01s
9	a{3000}a*	0.06s	172.483	0.93s	58.505	0.02s
10	{A,a}{3000}{A,a}*	0.51s	482.792	2.01s	160.971	0.16s
11	a*a{3000}	19.81s	172.441	0.91s	58.473	0.02s
12	{A,a}*{A,a}{3000}	208.60s	29.791.622	Fehler*		

* Die Kompilierung des Scanners wurde wegen eines GCC-internen Fehlers abgebrochen.

Diese Messungen zeigen, dass der Zeitaufwand für die Generierung eines Scanners mit Aktionen um etwa den Faktor 10 aufwändiger ist, als die Generierung eines Scanners ohne Aktionen. Bei den Tests 1-2, 5-6 bzw. 9-10 ist zu erkennen, dass sich dieser Faktor auch auf die Laufzeit des Scanners übertragen lässt.

Die Tests 3-4 bzw. 7-8 zeigen allerdings, dass dieser Faktor beim Scannen nur von Relevanz ist, falls der zugehörige reguläre Ausdruck so formuliert ist, dass keine „großen“ Umwälzungen beim Erzeugen des DEAs aus dem NEA notwendig sind. Im Fall von Test 4 bzw. 8 kam es nämlich zu einem quadratischen Zusammenhang von Aktionslisten im NEA zu Aktoren im DEA, was das Zeitverhalten des resultierenden Scanners äußerst ungünstig beeinflusste.

Test 12 (äquivalent zu Test 4 bzw. 8) erzeugte (mit einer Laufzeit von fast 4 Minuten!) sogar einen Scanner bei dessen Kompilierung der GCC nach einem nicht näher beschriebenen internen Fehler die Kompilierung abbrach.

5.5 Akquirierung

Die hier beschriebene Flex Erweiterung kann zum Zeitpunkt der Veröffentlichung dieser Arbeit von der Internetseite http://stud4.tuwien.ac.at/~e0125844/ herunter geladen werden.

Die dort zu findenden Archive enthalten den entsprechend erweiterten Source Code von Flex (Version 2.5.4).

6 Verwandte Arbeiten

In diesem Kapitel möchte ich das in dieser Arbeit vorgestellte System mit jenen anderer Scannergeneratoren bzw. Scannern vergleichen. Die Vergleichssysteme sind der Parsergenerator ANTLR und die Module für das Erkennen von regulären Ausdrücken der Sprachen Python und PHP (Die relevanten Informationen hierzu stammen aus einer Taschenreferenz [St03]). Außerdem werde ich das System mit einer Kombination aus Flex und Bison (äquivalent zu Lex & Yacc) vergleichen.

Bei allen Systemen wird ausschließlich das Interface des Markierungs- bzw. Benennungssystems betrachtet und verglichen.

6.1 ANTLR

Dieser Parsergenerator verfügt über einen integrierten Scanner in dessen Regeln reguläre Teilausdrücke benannt werden können um im Regelkörper über eine Variable auf den entsprechenden Substring des erkannten Strings zuzugreifen.

Diese Variable enthält allerdings immer nur einen Substring und zwar immer nur das jeweils letzte Vorkommen im erkannten String. Mehrfachnennungen, also derselbe Namen für verschiedene Ausdrücke, sind nicht möglich.

Beispiel 6.1.1: ANTLR Integer Array

```
class L extends Lexer;

INDEX     :    '[' (head:INT (',' tail:INT)* )? ']'
          {
               System.out.println(head.getText());
               System.out.println(tail.getText());
          }
     ;

INT  :    ('0'..'9')+ ;
```

Folgen:

```
"[]"                    => head=null, tail=null
"[12]"                  => head='12', tail=null
"[12,34]"               => head='12', tail='34'
"[12,34,56]"            => head='12', tail='56'
```

6.2 Python

In der Sprache Python können reguläre Ausdrücke sehr einfach erzeugt und benutzt werden. Dabei kommt es auf der obersten Ebene automatisch zu einer Durchnummerierung der geklammerten Ausdrücke. Außerdem können diese benannt werden wobei Mehrfachnennungen nicht möglich sind. Ausdrücke innerhalb eines rekursiven Elements werden nicht durchnummeriert, können aber benannt werden. Jeder Namen kann allerdings wie im Scanner von ANTLR nur einem Substring Platz bieten, welcher ebenfalls immer nur das jeweils letzte Vorkommen ist.

Beispiel 6.2.1: Python Integer Array

```
import re
pattern = r"\[((?P<head>\d+)(\,(?P<tail>\d+))*)?\]"
matched = re.match(pattern,string)
```

Folgen:

```
string = "[]"           => matched.lastindex = None
                           matched.group('head') = None
                           matched.group('tail') = None

string = "[12]"         => matched.lastindex = 1
                           matched.group(0) = '[12]'
                           matched.group(1) = '12'
                           matched.group('head') = '12'
                           matched.group('tail') = None

string = "[12,34]"      => matched.lastindex = 1
                           matched.group(0) = '[12,34]'
                           matched.group(1) = '12,34'
                           matched.group('head') = '12'
                           matched.group('tail') = '34'

string = "[12,34,56]"   => matched.lastindex = 1
                           matched.group(0) = '[12,34,56]'
                           matched.group(1) = '12,34,56'
                           matched.group('head') = '12'
                           matched.group('tail') = '56'
```

6.3 PHP

In der Sprache PHP können reguläre Ausdrücke ebenfalls sehr einfach erzeugt und benutzt werden. Dabei kommt es automatisch zu einer Durchnummerierung aller geklammerten Ausdrücke. Diese können nicht benannt werden. Jede Nummer kann allerdings wie im Scanner von ANTLR und wie in Python nur einem Substring Platz bieten, welcher auch hier immer nur das jeweils letzte Vorkommen ist.

Beispiel 6.3.1: PHP Integer Array

```
$pattern = "/\[((\d+)(\,(\d+))*)?\]/";
preg_match($pattern,$string,$matches);
```

Folgen:

```
$string = "[]"          => count($matches) = 1
                           matches[0] = '[]'

$string = "[12]"        => count($matches) = 3
                           matches[0] = '[12]'
                           matches[1] = '12'
                           matches[2] = '12'
```

```
$string = "[12,34]"      => count($matches) = 5
                            matches[0] = '[12,34]'
                            matches[1] = '12,34'
                            matches[2] = '12'
                            matches[3] = ',34'
                            matches[4] = '34'
$string = "[12,34,56]"   => count($matches) = 5
                            matches[0] = '[12,34,56]'
                            matches[1] = '12,34,56'
                            matches[2] = '12'
                            matches[3] = ',56'
                            matches[4] = '56'
```

6.4 Flex & Bison

Bei der Implementierung mit Flex und Bison sind zwei Dateien erforderlich.

Beispiel 6.4.1: Flex & Bison Integer Array

Scannerdatei: scanner.l

```
%%
[0-9]+ {
     element = atoi(yytext);
     return INTEGER;
     }

[,\[\]] {
     return yytext[0];
     }

.
[\n\t\f\c]
%%
```

Parserdatei: parser.y (nicht komplett)

```
%token INTEGER
%start Array

%{
int element;
%}

%%
Array : '[' ']' { finish_list(); }
      | '[' Integer Integers ']' { finish_list(); }

Integers : ',' Integer Integers
         |

Integer : INTEGER { add_element(element); }
%%

#include "lex.yy.c"
```

Folgen:

```
"[]"                    => finish_list()

"[12]"                  => add_element(12)
                           finish_list()

"[12,34]"               => add_element(12)
                           add_element(34)
                           finish_list()

"[12,34,56]"            => add_element(12)
                           add_element(34)
                           add_element(56)
                           finish_list()
```

6.5 Flex Erweiterung (zum Vergleich)

Mit der in dieser Arbeit beschriebenen Flex Erweiterung können reguläre Ausdrücke explizit benennt werden. Mehrfachnennungen sind möglich. Mehrfache Vorkommen bilden nach beeinflussbaren Kriterien Listen, Felder, Räume und, falls nötig, höherdimensionale Strukturen.

Beispiel 6.5.1: Flex Erweiterung Integer Array

```
NUMBER=[0-9]+
%%
“[“({element,{NUMBER}}(,{element,{NUMBER}})*)?“]“ {
                      YYList list = element;
                      int length = list.length;
                      YYSingular singulars[] = list.yysingulars;
                   }
%%
```

Folgen:

```
"[]"                    => length = 0

"[12]"                  => length = 1
                           singulars[0].string ='12'

"[12,34]"               => length = 2
                           singulars[0].string ='12'
                           singulars[1].string ='34'

"[12,34,56]"            => length = 3
                           singulars[0].string ='12'
                           singulars[1].string ='34'
                           singulars[2].string ='56'
```

7 Zusammenfassung

Das in dieser Arbeit vorgestellte System ermöglicht es in einem regulären Ausdruck bestimmte Teilausdrücke zu benennen. Diese Benennungen finden sich in entsprechend angepasster Form in dem aus dem regulären Ausdruck erstellten deterministischen Automaten wieder. Während des Scanvorgangs werden diese Elemente mitprotokolliert. Nach einem Match kann die dabei entstehende Liste genutzt werden um herauszufinden, welche Teilstrings welchen benannten Teilausdrücken entsprechen.

Dabei sind Mehrfachnennungen von Namen im regulären Ausdruck möglich. Möglicherweise mehrfache Vorkommen von Teilstrings mit demselben Namen führen zur Strukturbildung (Stringlisten, Listen von Stringlisten, usw.).

Die dieses System im Rahmen dieser Magisterarbeit verwirklichende Implementierung ist eine Erweiterung des Scannergenerators „Flex“ (Version 2.5.4). Hier werden die Ergebnisstrukturen des Scanners dem Programmierer der Aktionen als Variablen zur Verfügung gestellt. Das ermöglicht eine einfachere, robustere und weniger wartungsintensivere Programmierung von Aktionen komplexer Regeln.

8 Danksagung

Ich danke meinem Betreuer M. Anton Ertl für seine Zeit, seine Geduld und seine guten Ratschläge.

Abkürzungsverzeichnis

RE ..... regulärer Ausdruck
NEA ..... nichtdeterministischer endlicher Automat
DEA ..... deterministischer endlicher Automat
PM ..... Potenzmenge

Literatur

[AhSU86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman; Compilers - Principles, Techniques, and Tools; Addison-Wesely Publishing Company; 1986

[Pax90] Vern Paxson; Flex, version 2.5 - A fast scanner generator; Manual; University of California; 1990; http://www.gnu.org/software/flex/manual

[BrGePl98] Klaus Brouwer, Wolfgang Gellerich, Erhard Plödereder; Myths and Facts about the Efficient Implementation of Finite Automata and Lexical Analysis; Lecture Notes In Computer Science; Vol. 1383; Springer Verlag; Conference on Compiler Construction; 1998

[Le94] J. Van Leeuwen; Handbook of Theoretical Computer Science, Volume A (Kapitel 5); MIT Press; 1994

[St03] Tony Stubblebine; Regular Expression Pocket Reference - Regular Expressions for Perl, C, PHP, Python, Java, and .NET; O'Reilly; 2003