Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Rekursionsfreie Muster und endliche Automaten

- Muster, Musterbeschreibungen und Mustererkennung
- Endliche Automaten: NEA und DEA
- Äquivalenz deterministischer und nichtdeterministischer Automaten

Muster (engl. Pattern)

Muster

Ein Muster ist eine Menge von Objekten, die durch eine bestimmte Eigenschaft charakterisiert sind.

Beispiel: „gestreifter Hut“ : die Menge aller Hüte mit der Eigenschaft gestreift zu sein.

Textmuster

In unserem Kontext sind Text-Muster relevant: Ein Textmuster ist eine Menge von Texten (Zeichenketten / Strings), die eine bestimmte Eigenschaft haben.

Beispiel: die Menge aller Worte die jeden Vokal genau einmal enthalten.

Textmuster sind in der Informatik im Allgemeinen sehr wichtig und im Compilerbau sind sie ganz besonders wichtig.

Im Folgenden beschäftigen wir uns nur noch mit Textmustern.

Thematik 1: Musterdefinition

Ein Muster beschreibt eine Menge, die oft unendlich groß ist. Das Muster kann also nicht durch Aufzählung definiert werden.

Notationen für die Definition von Mustern werden benötigt.

Thematik 2: Mustererkennung

Ein Text kann zu einem Muster gehören oder nicht. Die Klärung dieser Frage wird Mustererkennung (*Pattern-Matching*) genannt.

Übersicht

In diesem Abschnitt werden folgende Themen behandelt:

Endliche Automaten

als Notation für die Definition von Mustern in zwei Varianten:

- Deterministische endliche Automation (DEA)
- Nicht-deterministische endliche Automaten (NEA)

Implementierung deterministischer endlicher Automaten

Die Implementierung eines DEA ist einer Erkennungsmechanismus für das Muster, das der DEA beschreibt

Nicht-deterministische endliche Automaten

NEAs vereinfachen die Definition von Mustern

Implementierung eines NEA

- Als Suche
- Durch Umwandlung in einen äquivalenten DEA

Reguläre Ausdrücke

als eine „Algebra“ zur Definition von Mustern

Übersicht

Rekursive und nicht-rekursive Muster

Die (Text-) Muster in diesem Abschnitt sind *nicht rekursiv definiert* !

Sie können darum

- **„algebraisch“, also durch Ausdrücke („reguläre Ausdrücke“) beschrieben werden (statt durch Gleichungen) und**
- **ihre Erkennung ist einfach und schnell**

Im nächsten Kapitel geht es um Muster, die rekursiv (definiert) sein können. Sie werden

- **durch *kontext-freie* Grammatiken (als Gleichungssystem) definiert,**
- **und können nur mit wesentlich größerem Aufwand erkannt werden.**

Übersicht

Reguläre Ausdrücke

Ein Muster wird durch einen Ausdruck beschrieben

Reguläre Sprache

Eine Menge von „Worten“ die durch einen regulären Ausdruck beschrieben wird,

oder beschrieben werden könnte

Reguläre Grammatik

Eine Grammatik (Gleichungssystem) mit bestimmten Eigenschaften

Reguläre Grammatiken sind äquivalent zu regulären Ausdrücken

Reguläre und kontextfreie Sprachen

Reguläre Sprache: Eine Menge an Texten (Zeichenketten, Strings) die durch ein nicht-rekursives Muster – einen regulären Ausdruck – definiert werden kann

Kontextfreie Sprache: Menge an Texten die durch ein (eventuell) rekursives Muster definiert werden kann

Übersicht

Scanner

Comiler- / Interpreter-Komponente, die die „Worte“ einer Programmiersprache erkennt.

Meist werden diese durch nicht-rekursive Muster definiert.

Parser

Compiler-- / Interpreter-Komponente, die die „Sätze“ einer Programmiersprache erkennt. Meist werden diese als rekursive Muster definiert.

Musterdefinition und -Erkennung

Beispiel **Muster: Alle Worte welche die Vokale a, e, i, o, u in dieser Reihenfolge enthalten**

```
import scala.io.Source

object Match {

  def checkAeiou(w: String) : Boolean = {
    var i = 0
    def checkChar(c: Char) : Boolean = {
      var found = false
      while (i < w.length() && ! found) {
        if (w.charAt(i) == c) {found = true}
        i = i+1
      }
      found
    }
    checkChar('a') && checkChar('e') && checkChar('i') && checkChar('o') && checkChar('u')
  }
}

object FindWords_Main extends App {

  val wordList = Source.fromURL("http://wortschatz.uni-leipzig.de/Papers/top10000de.txt",
                                "windows-1252").getLines()
  for (word <- wordList) {
    if (Match.checkAeiou(word.toLowerCase())) println(word)
  }
}
```

```
Arbeitslosenquote
```

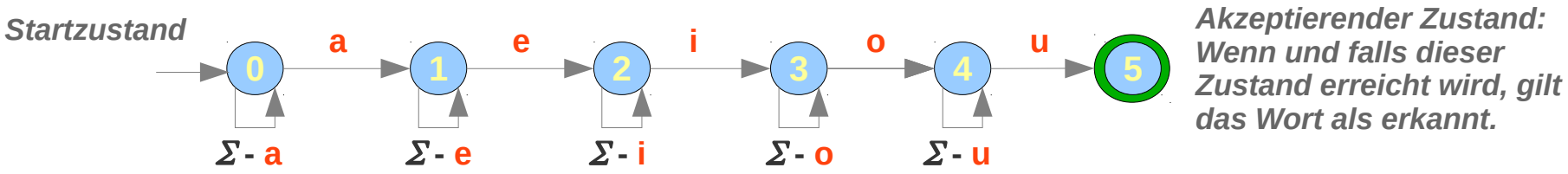
Musterdefinition und -Erkennung

Beispiel **Muster: Alle Worte welche die Vokale a, e, i, o, u in dieser Reihenfolge enthalten**

Prüf-Algorithmus für ein Wort:

- **Die Funktion durchläuft alle Buchstaben des Worts,**
  und ist dabei in verschiedenen Zuständen:
  - **kein Vokal erkannt,**
  - **a erkannt,**
  - **a, e erkannt,**
  - **…**
  - **a, e, i, o, u erkannt**

  Wenn alle Zustände durchlaufen wurden, ist das Wort (als Element des Musters) erkannt;
  ansonsten wird es nicht (als Element des Musters) erkannt.
- **Symbolisch als Automat:**

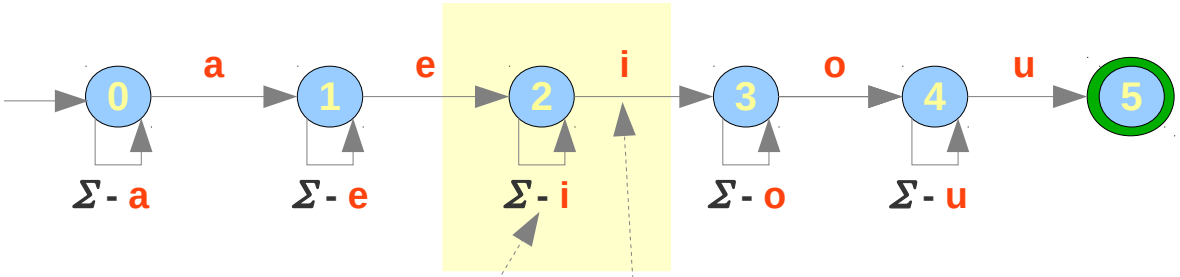


Algorithmus zur Erkennung von Worten, die a,e,i,o,u in dieser Reihenfolge enthalten.
Dargestellt als Zustandsautomat.
Σ Ist die Menge aller Zeichen (Buchstaben).

Musterdefinition und -Erkennung

Erkennungsalgorithmus als Funktion und Automat

```
checkChar('a') && checkChar('e') && checkChar('i') && checkChar('o') && checkChar('u')
```



Kein i: Weitersuchen

i gefunden: Fertig. Weiter zum nächsten.

```
def checkAeiou(w: String) : Boolean = {
    var i = 0
    def checkChar(c: Char) : Boolean = {
      var found = false
      while (i < w.length() && ! found) {
        if (w.charAt(i) == c) {found = true}
        i = i+1
      }
      found
    }
```

Der Automat ist eine kurze und übersichtliche Beschreibung des Algorithmus'.

i : (Aktuelles Zeichen ist) i.

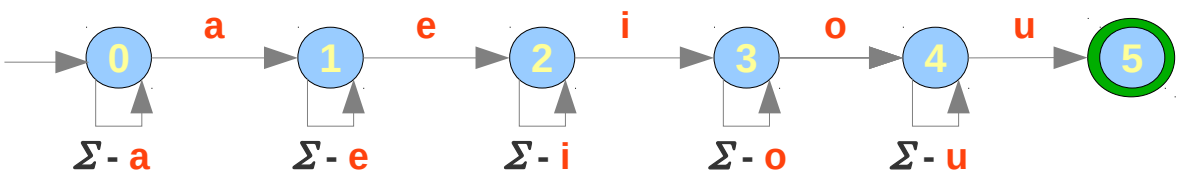
Σ - i : (Aktuelles Zeichen ist) alles ausser i.

Σ : „(Groß-) Sigma“, Alle Zeichen / alle Terminale / alle Symbole

Musterdefinition und -Erkennung

Erkennungsalgorithmus als Funktion und Automat

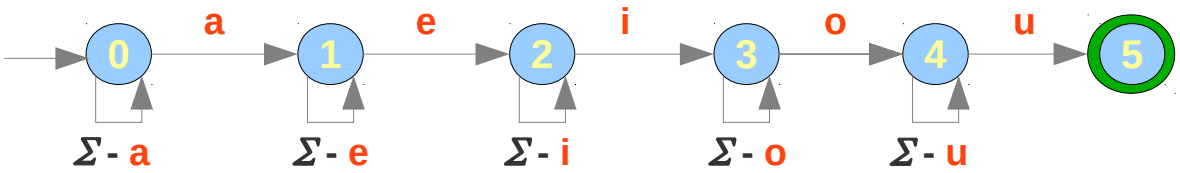
```
def checkAeiou(w: String) : Boolean = {
  var i = 0
  def checkChar(c: Char) : Boolean = {
    var found = false
    while (i < w.length() && ! found) {
      if (w.charAt(i) == c) {found = true}
      i = i+1
    }
    found
  }
  checkChar('a') && checkChar('e') && checkChar('i') && checkChar('o') && checkChar('u')
}
```



Der Automat ist eine kurze und übersichtliche Beschreibung des Algorithmus'.

Musterdefinition und -Erkennung

Erkennungsalgorithmus als Automat und explizit umgesetzter Automat



Zwei Notationen, ein Algorithmus

Der Automat ist eine kurze und übersichtliche Beschreibung des Algorithmus'.

```
object Match {

  def checkAeiou(w: String) : Boolean = {
    var i = 0
    def checkChar(c: Char) : Boolean = {
      var found = false
      while (i < w.length() && ! found) {
        if (w.charAt(i) == c) {found = true}
        i = i+1
      }
      found
    }
    checkChar('a') && checkChar('e') && checkChar('i') && checkChar('o') && checkChar('u')
  }
}
```

Endlicher Automat

Abstrakte Definition eines Erkennungs-Algorithmus'

Warum „endlich“: Ein endlicher Automat hat eine fixe endliche Menge von Zuständen

Liefert Entscheidung

- **Ja / akzeptiert**

  Es gibt einen Pfad, der mit den Zeichen des Wortes markiert ist, und dieser endet in einem akzeptierenden Zustand.
- **Nein / nicht akzeptiert**

  Es gibt keinen solchen Pfad: Es existiert entweder kein Pfad der mit den Zeichen markiert ist, oder ein solcher Pfade endet nicht in einem akzeptierenden Zustand.

In einem akzeptierenden Zustand

- **kann verlangt sein, dass alle Zeichen des Wortes „konsumiert“ sein müssen,**
- **oder auch nicht**

Dazu gibt es keine Regel, das ist anwendungsabhängig (kann beliebig festgelegt werden).

Endliche Automaten sind ein vielseitig eingesetzter Formalismus der Informatik

- **Generell: Ereignis x Zustand → Zustand**
- **Syntaxanalyse: „Ereignis“ = nächstes Zeichen**
- **Ein „Zeichen“ muss nicht unbedingt ein Char sein**

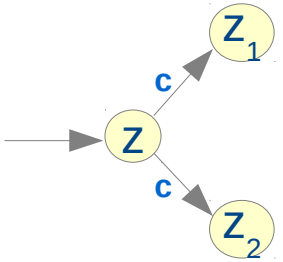
Nicht-deterministischer Endlicher Automat, NEA

Ein endlicher Automat heisst deterministisch wenn

- **für einen Zustand und ein Symbol / Ereignis**
- **höchsten ein Folgezustand definiert ist.**

Bei einen nicht-deterministischen Automat ist es erlaubt, dass

- **für einen Zustand und ein Symbol**
- **mehr als ein Folgezustand definiert ist.**



Der Nicht-Determinismus eines NEA besteht darin,
dass der Folgezustand gelegentlich nicht eindeutig festgelegt (nicht determiniert) ist.

Die von einem nicht-deterministischen Automat akzeptierten Worte sind
(wie beim deterministischen) die Worte

- **für die es einen Pfad durch die Zustände zu einem akzeptierenden Zustand gibt,**
- **der mit den Symbolen des Wortes markiert sind.**

Man beachte:

- **Jeder deterministische Automat ist auch ein nicht-deterministischer Automat.**

  Es ist in einen NEA erlaubt, aber nicht zwingend notwendig, dass es einen Zustand gibt, der mehr als einen Folgezustand hat, der mit dem gleichen Symbol erreicht werden kann.
- **I. A. redet man aber nur dann von einem nicht-deterministischen Automat, wenn er wirklich nicht-deterministisch ist.**

Nicht-deterministischer Endlicher Automat NEA

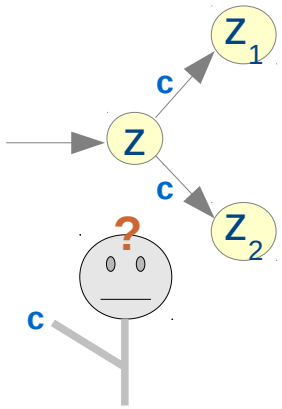
Vorteil eines NEA

Viele Muster können wesentlich **einfacher definiert** werden wenn
die Folgezustände nicht eindeutig festgelegt sind.

Nachteil eines NEA

- Eine Implementierung des deterministischen Automaten ergibt sich von selbst:
  Die „Ereignis-Verarbeitung“:
  Zustand x Zeichen => Folgezustand
  ist direkt implementierbar
- **Die Implementierung eines NEA ist nicht so einfach**
  Bei nicht-deterministischen Automaten sind mehrere „Ereignis-Verarbeitungen“ möglich
  Zustand x Zeichen => Folgezustand$_1$ *oder* **Folgezustand$_2$** *oder* **… Folgezustand$_n$**
  Es ist nicht klar, welcher Folgezustand
  - zu einem akzeptierenden Pfad gehört und gewählt werden sollte
  - und welcher nicht.

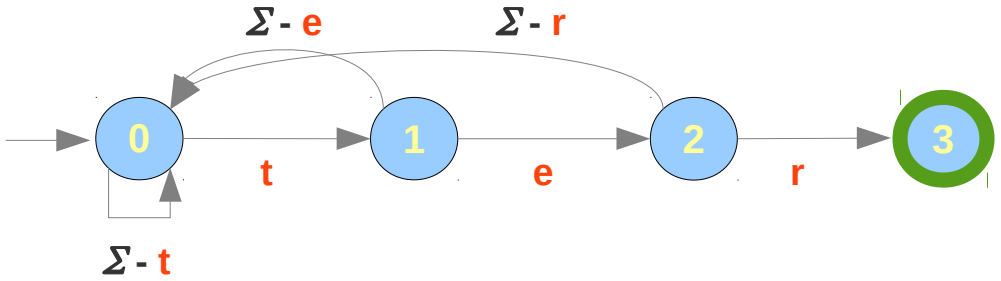
  Ein Wort wird ja akzeptiert wenn
  - seine Zeichen zu *irgendeinem* Pfad durch den Automat gehören,
  - der zu einem akzeptierenden Zustand führt.
  - Dieser Pfad kann nur durch eine erfolgreiche Suche gefunden werden.



NEA als Mittel zur vereinfachten Musterdefinition

Beispiel – 1: Deterministischer Automat

- **Gesucht: Muster aller Worte, die „ter“ enthalten.**
- **Naiver deterministischer Automat:**



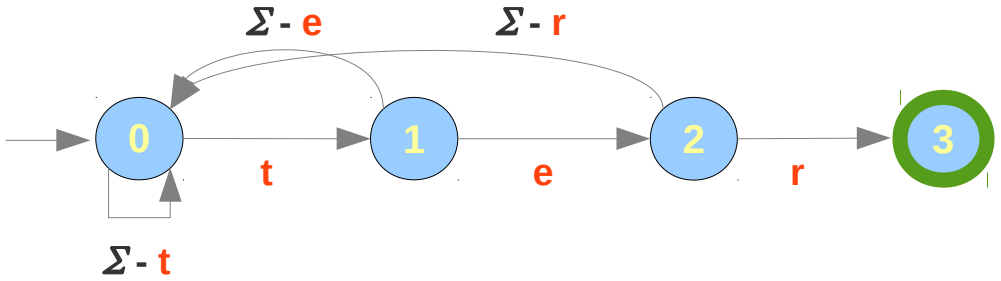
DEA

Funktioniert nicht: Welche Worte, die ter enthalten, werden nicht erkannt?

NEA als Mittel zur vereinfachten Musterdefinition

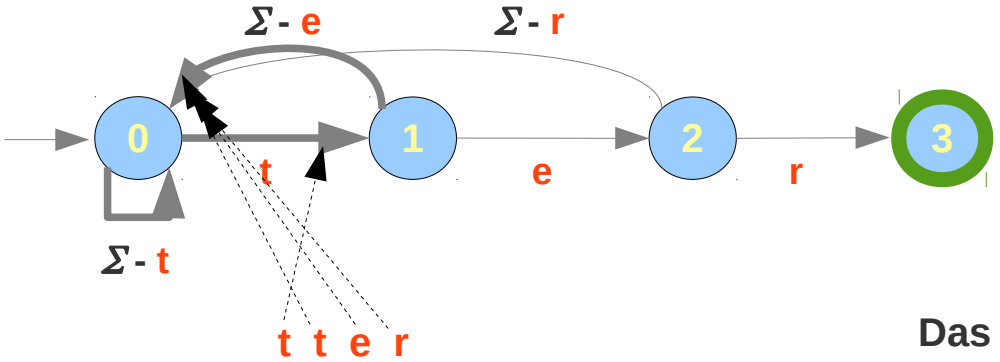
Beispiel – 1: Deterministischer Automat

- **Gesucht: Muster aller Worte, die „ter“ enthalten.**
- **Naiver deterministischer Automat:**



Es ist oft gar nicht so einfach einen korrekten deterministischen Automat anzugeben.

tter wird nicht erkannt:

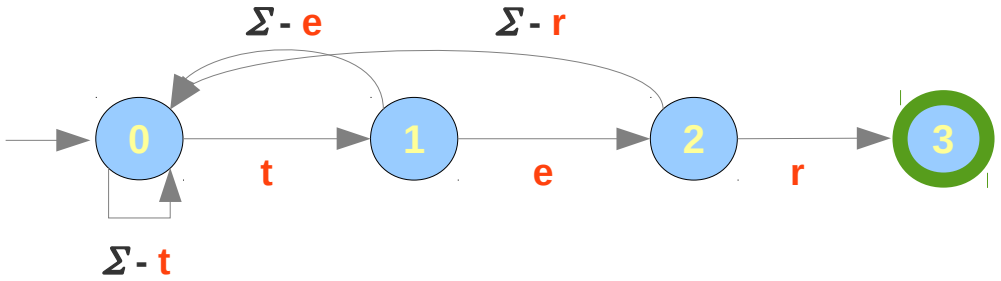


Das erste t hätte ignoriert werden müssen.

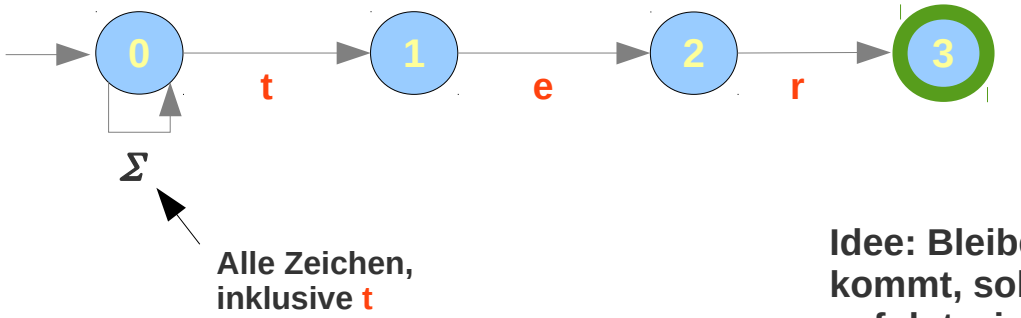
NEA als Mittel zur vereinfachten Musterdefinition

Beispiel – 1: NEA

- Gesucht: Muster aller Worte, die „ter“ enthalten.
- Vom deterministischen (falschen) zum Nicht-deterministischer Automat:



DEA (nicht korrekt)



NEA

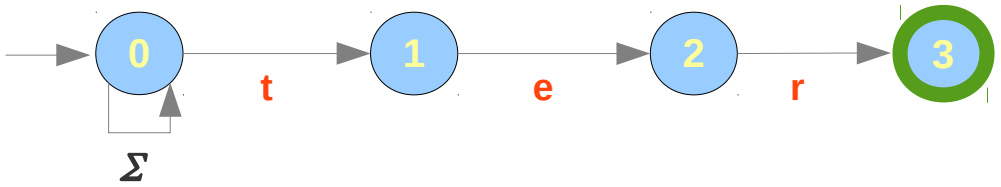
Alle Zeichen, inklusive t

Idee: Bleibe in Zustand 0 – auch wenn ein t kommt, solange bis ein t kommt, das von e r gefolgt wird.

NEA als Mittel zur vereinfachten Musterdefinition

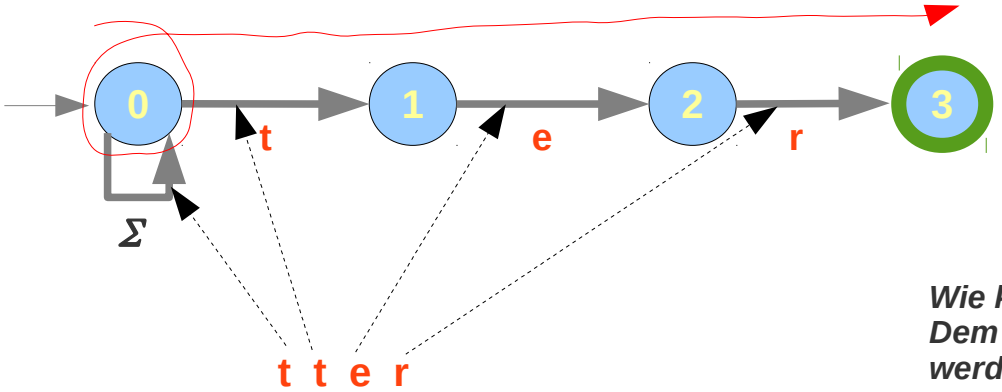
Beispiel – 1: NEA

- **Gesucht: Muster aller Worte, die „ter“ enthalten.**



NEA

tter wird erkannt (es gibt einen Pfad):

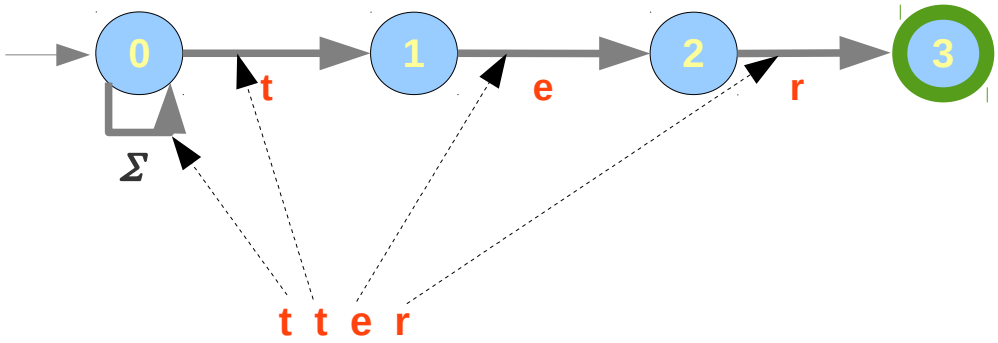


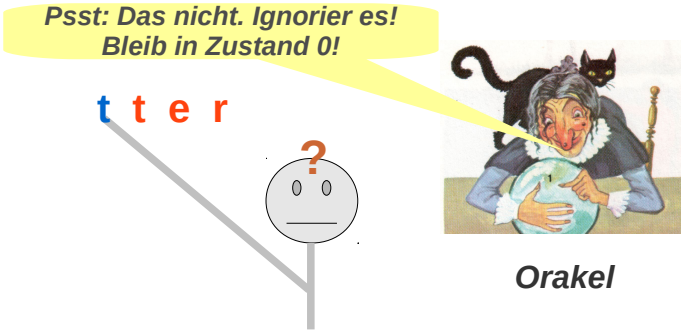
Wie kann ein solcher Automat implementiert werden? Dem ersten t sieht man ja nicht an, dass es ignoriert werden muss.

NEA als Mittel zur vereinfachten Musterdefinition

Beispiel – 1: NEA

- **Implementierung**





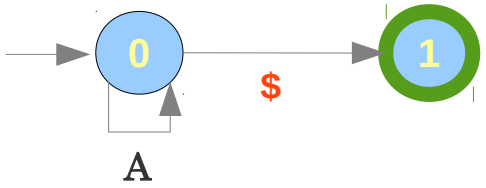
Orakel

Wie kann ein solcher Automat implementiert werden?
Die Spezifikation eines NEAs basiert auf einem „Orakel“.
Das Orakel entspricht einer erschöpfenden Suche. –
Eventuell mit Backtracking optimiert

NEA als Mittel zur vereinfachten Musterdefinition

Beispiel – 2: DEA 1

- **Gesucht: Muster aller Worte, die nur Buchstaben einer Menge A enthalten.**



Statt Σ nennen wir die Symbolmenge zur Abwechslung mal A.

DEA (korrekt)

$: Sonderzeichen, das an jedes Wort angehängt wird, es wird genutzt, um das Ende eines Wortes zu markieren und damit dem Automat verfügbar zu machen.

NEA als Mittel zur vereinfachten Musterdefinition

Multimenge: Menge mit Wiederholungen.

Beispiel – 2: DEA1 und DEA 2

- **Gesucht: Alle Worte, die nur Buchstaben einer Multi-Menge Menge M enthalten und zwar höchstens so oft, wie sie in M vorkommen. (Groß- und Kleinbuchstaben werden nicht unterschieden)**

  Beispiel:

  M = { w a s s e r }

  Passende Worte: Was, Wer, Wasser, … ? ….

- **Lösung zweistufig Ein Positiv-Filter kombiniert mit Negativ-Filtern:**

  1) Positiv-Filter: Nur Zeichen aus M

  Alle Worte ausfiltern, die Buchstaben enthalten, die nicht in M sind.

  Ein einfacher DEA kann das erledigen.

  2) Negativ-Filter: Zeichen tritt nicht öfter auf als in M

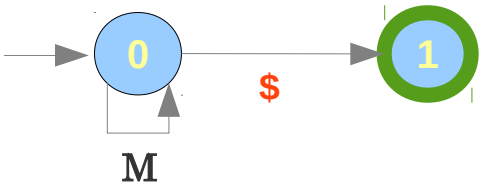
  Alle Worte ausfiltern, die Buchstaben aus M enthalten, aber öfter als in M.

  Pro Buchstaben ein DEA. Ein Wort muss alle passieren um akzeptiert zu werden.

NEA als Mittel zur vereinfachten Musterdefinition

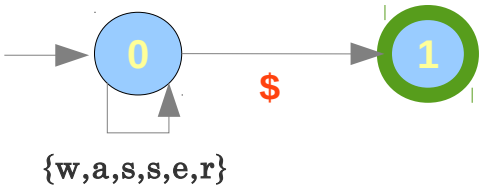
Beispiel – 2: DEA1 und DEA 2

Stufe 1 Positiv-Filter: Alle Worte ausfiltern, die Buchstaben enthalten, die nicht in M sind.



Jedes Wort muss von diesem Automat erkannt werden.

Stufe 1 – Ausfiltern: DEA akzeptiert Worte, die nur Zeichen aus M enthalten, die also nicht ausgefiltert werden.



Beispiel: 'wer' *(genauer* 'wer$'*) wird von diesem Automat akzeptiert.*

Der Automat akzeptiert aber auch Worte wie 'waaasr' *(zu viele* a's*) diese werden in Stufe 2 ausgefiltert.*

NEA als Mittel zur vereinfachten Musterdefinition

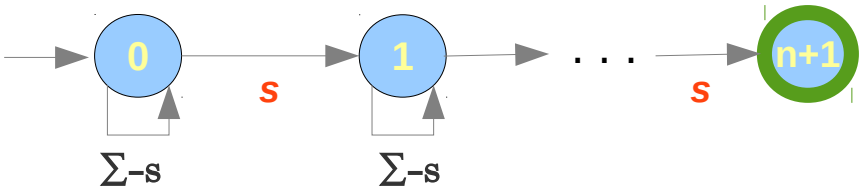
Multimenge: Menge mit Wiederholungen.

Beispiel – 2: DEA1 und DEA 2

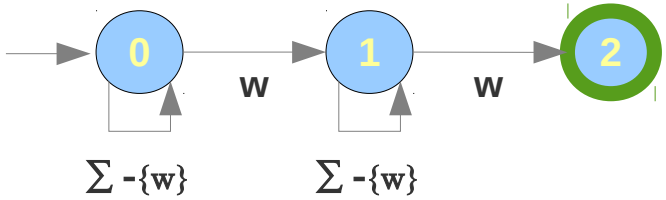
Stufe 2. Negativfilter:

Alle Worte ausfiltern, die Buchstaben aus M enthalten, aber öfter als in M.

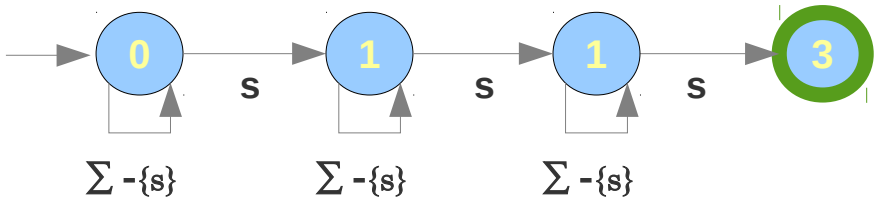
Pro Buchstaben ein DEA. Ein Wort muss alle passieren und darf von keinem akzeptiert werden.



Stufe 2 – **ein DEA für jedes(!) *s***, das in M ***n***-mal vorkommt. Jedes Wort muss einen solchen Automat für jedes s in A durchlaufen und von ihm akzeptiert werden.



Beispiel: Der Automat, der Worte mit zu vielen 'w'-s akzeptiert

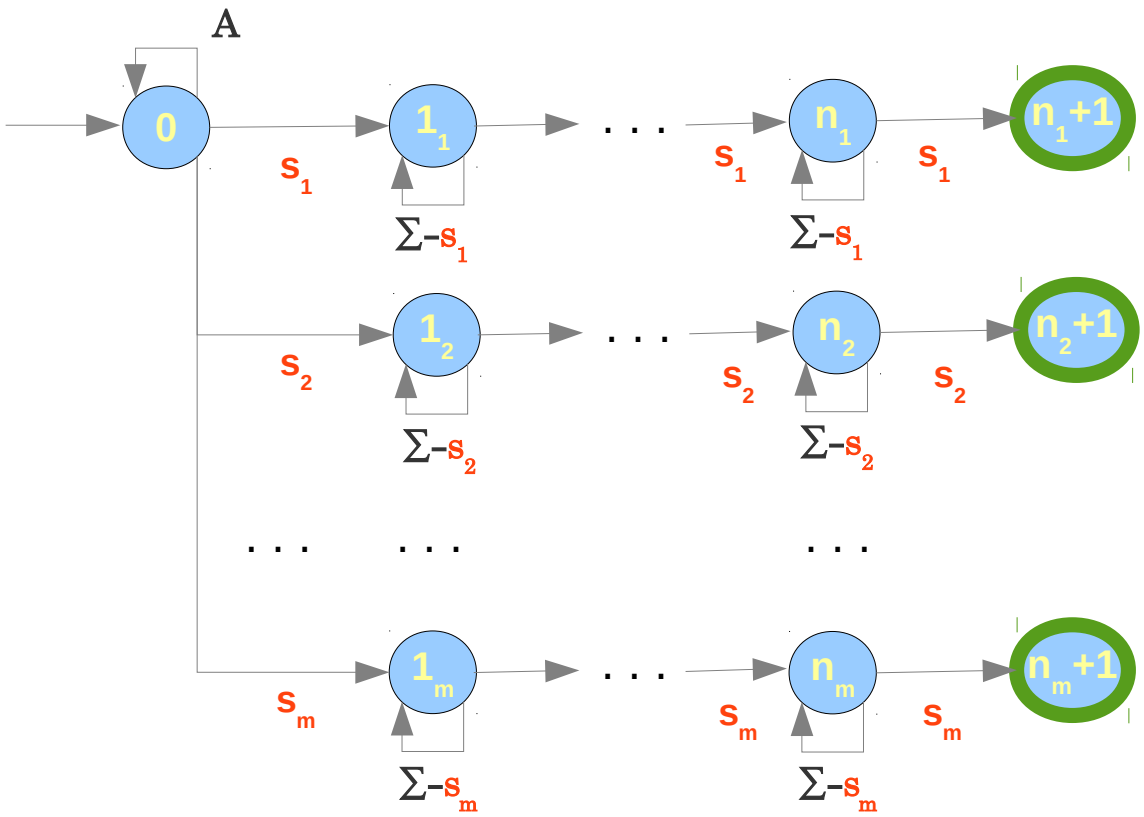


Beispiel: Der Automat, der Worte mit zu vielen 's' akzeptiert

NEA als Mittel zur vereinfachten Musterdefinition

Beispiel – 2: DEA 1 und viele DEA 2 in einem NEA kombinieren

- **Gesucht: Alle Worte, die nur Buchstaben einer Multi-Menge Menge M enthalten und zwar höchstens oft, wie sie in M vorkommen. 2–stufiges Verfahren.**
- **Stufe 2 mit einem (!) NEA**

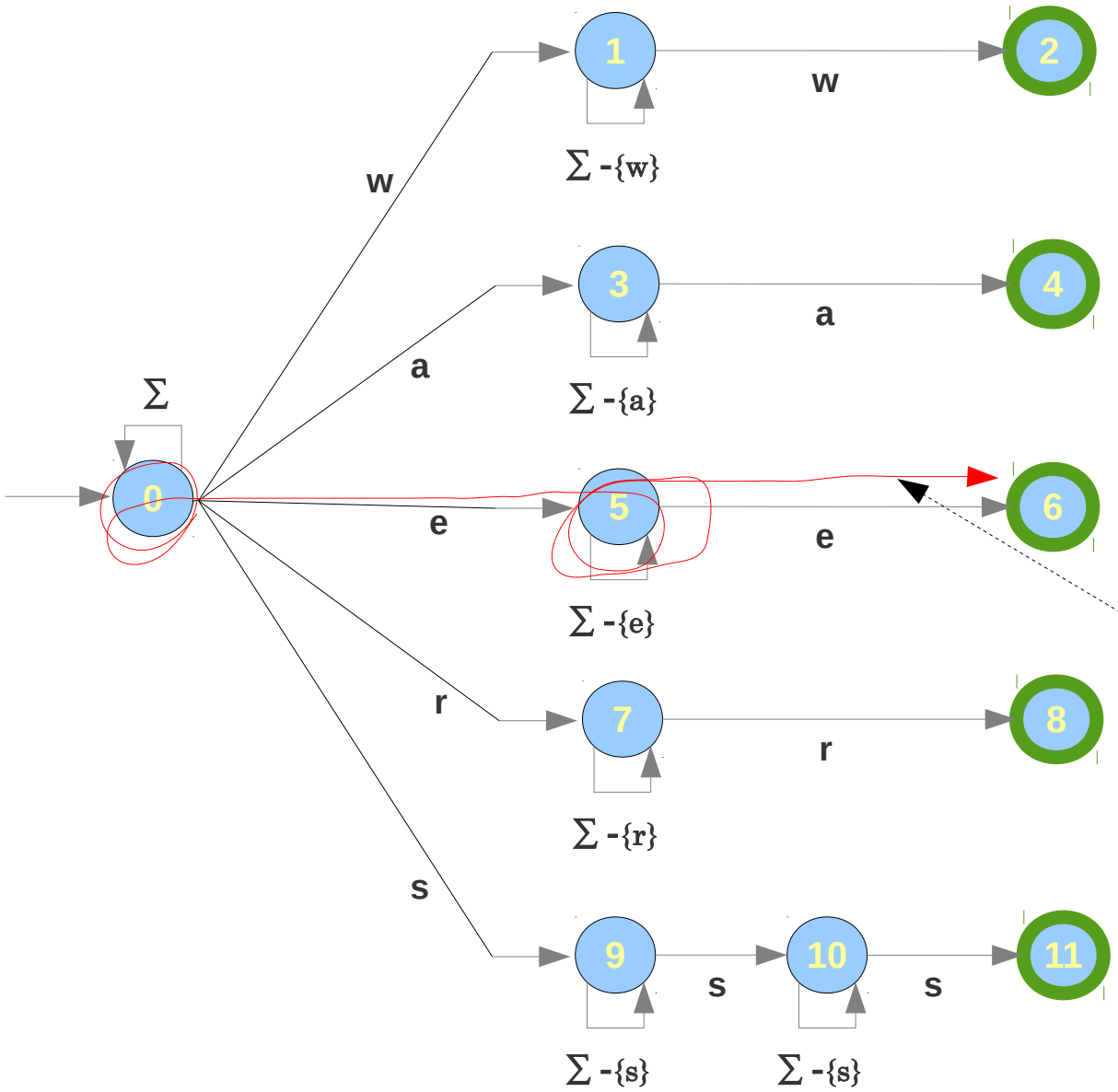


Stufe 2 – ein NEA.

Akzeptiert alle Worte, die nicht (!) die richtige Anzahl von Vorkommen eines Buchstaben haben, sondern zu viele. Die also in Stufe 2 ausgefiltert werden müssen.

Das Orakel entscheidet in Zustand 0, ob das aktuelle Zeichen eines von denen ist, die zu oft vorkommen.

Nicht-Deterministische endliche Automaten



Beispiel
Stufe 2 für M = {w, a, s, s, e, r}

Pfad für aeste

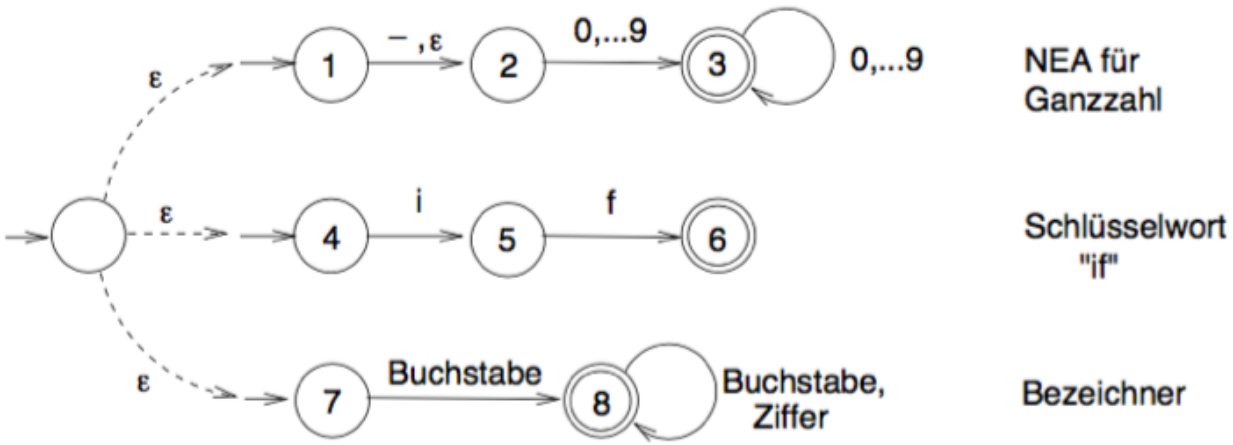
Das Wort wird von diesem Automat akzeptiert, weil es ein e mehr enthält als wasser.

Es ist ein „falsches Wort“, das von diesem Automat (der Worte mit zu vielen Buchstaben akzeptiert) erkannt wird.

NEA als Mittel zur vereinfachten Musterdefinition

Beispiel – NEAs im Compilerbau

Genutzt zur Definition der Syntax von Tokens: elementaren „Worten“ eines Programms



ε: „Epsilon-Transition“: Der Automat geht in einen neuen Zustand über, ohne ein Zeichen zu verbrauchen.
ε ist das „leere Zeichen“.

Aus M. Jäger: *Compilerbau – Eine Einführung*, Seite 70

Nicht-deterministischer Endlicher Automat

Formale Definition*

A ***nondeterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta: Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

Bemerkung – gilt auch für deterministische endliche Automaten:

Die Transitionsfunktion ist nicht zwingend total. D.h. es ist erlaubt, Automaten zu definieren, in denen zu einem Zeichen und einem Zustand kein Folgezustand definiert ist.

Das Verhalten des Automaten ist in diesem Fall undefiniert. In der Praxis sollte dies vermieden werden: etwa durch Angabe einer totalen Transitionsfunktion, oder durch eine Erklärung, dessen, was eine undefinierte Transition bedeuten soll. Die naheliegende und hier allgemein verwendete Interpretation ist: Der Automat akzeptiert die Eingabe nicht.

Für tiefere Betrachtungen und Diskussionen der endlichen Automation und ihrer formalen Grundlagen sei an die Veranstaltung „Automaten und formale Sprachen“ verwiesen.

*Siehe: W. Sipser *Introduction to the Thory of Computation*, Thomson, 2006, Seite 55

NEA Implementierung

Der Nichtdeterminismus beim Akzeptieren im NEA bedeutet

Es gibt einen Pfad der mit den Symbolen des Wortes markiert ist

Eine Implementierung eines NEA

prüft ob es einen Pfad gibt, der mit den Symbolen des Wortes markiert ist

Mit welcher algorithmischen Strategie könnte ein Algorithmus entwickelt werden, der prüft, ob es zu einem Wort einen Pfad gibt, der in einem akzeptierenden Zustand endet?

NEA Implementierung

Orakel

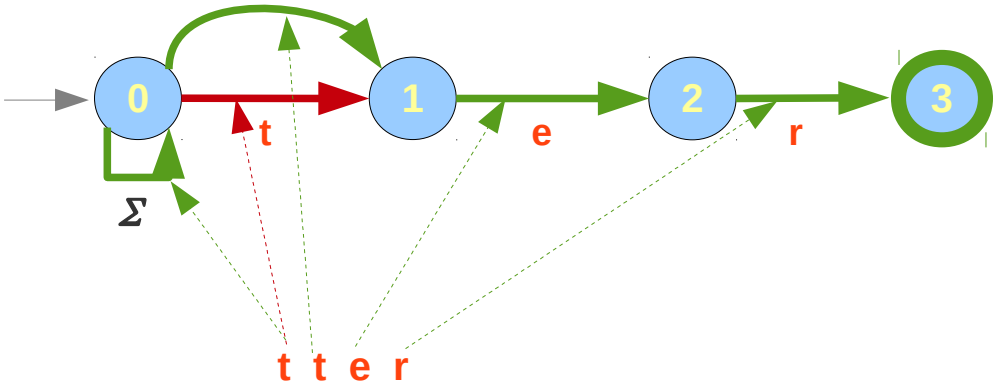
- Sind mehrere Zustandsübergänge möglich, dann entscheidet ein Orakel, welcher zu wählen ist.
- Das Orakel trifft immer die/eine richtige Wahl, denn es kann in die Zukunft sehen.

Erschöpfende Suche

- **Suchraum:** Alle Pfade die mit den Symbolen des Wortes markiert sind
- **Erfolg:** Pfad der in einem akzeptierenden Zustand endet

Organisation der erschöpfenden Suche

- ***Backtracking*** (ein Pfad, der in einer Sackgasse endet, muss nicht weiter untersucht werden)
- ***Parallel***: Alle möglichen Pfade werden gleichzeitig verfolgt



NEA Implementierung – Backtracking

Beispiel „ter“-Match mit *Backtracking*

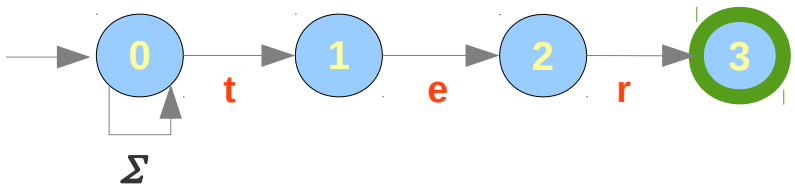
```
def checkTer(w: String) : Boolean = {
  def check(i: Int, state: Int) : Boolean =
    if (i >= w.length()) false
    else state match {
      case 0 =>
        if (w.charAt(i) == 't') check(i+1, 0) || check(i+1, 1)
        else check(i+1, 0)
      case 1 => w.charAt(i) == 'e' && check(i+1, 2)
      case 2 => w.charAt(i) == 'r'
    }
  check(0, 0)
}
```

t gesehen: bleibe in Zustand 0

oder

gehe über in Zustand 1.

Implementiert als:
Versuche es erst mit
„in Zustand 0 bleiben“,
wenn das nicht funktioniert versuche es mit
„in Zustand 1 übergehen.“



NEA Implementierung

Theorem: Zu jedem NEA gibt es einen äquivalenten DEA

Die Standardmethode zur Spezifikation eines Musters ist die Angabe eines NEA

Die Standardmethode zur Implementierung eines NEAs ist

- **die Konstruktion eines äquivalenten DEAs**
- **Und dessen Implementierung**

Reguläre Ausdrücke: Definition eines NEA

Viele Programmiersprachen unterstützen reguläre Ausdrücke
Mit regulären Ausdrücken können NEAs definiert werden
Bibliotheksfunktionen übersetzen diese in äquivalente DEAs,
die dann durch weitere Bibliotheksfunktionen implementiert werden

Äquivalenz von Automaten

Zwei Automaten *A* und *B* sind äquivalent,

wenn sie die gleichen Sequenzen von Zeichen akzeptieren.

Etwas ausführlicher: Die Automaten *A* und B sind äquivalent genau dann, wenn gilt:

- **Falls *A* einen Pfad p_A enthält der**
  - **mit dem Startzustand beginnt, und**
  - **der mit $z_1, z_2, z_3, \ldots z_n$ markiert ist, und**
  - **in einem akzeptierenden Zustand endet,**

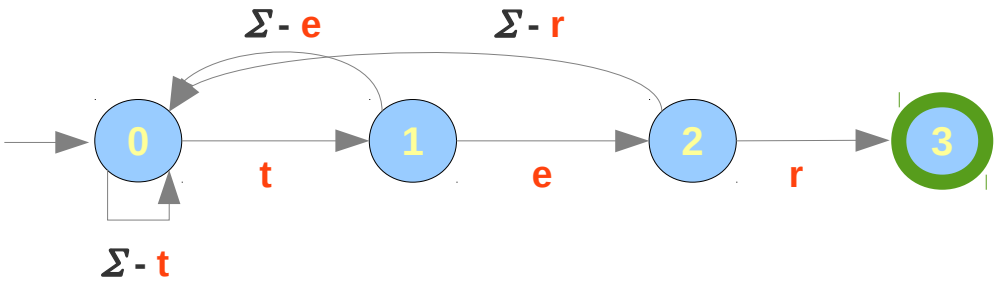
  dann enthält *B* einen ebensolchen Pfad (gleiche Markierungen, eventuell andere Zustände)
- **Falls *B* einen Pfad p_B enthalt der**
  - **mit dem Startzustand beginnt,**
  - **und mit $z_1, ,z_2, z_3, \ldots z_n$ markiert ist**
  - **und in einem akzeptierenden Zustand endet,**

  dann enthält *A* einen ebensolchen Pfad

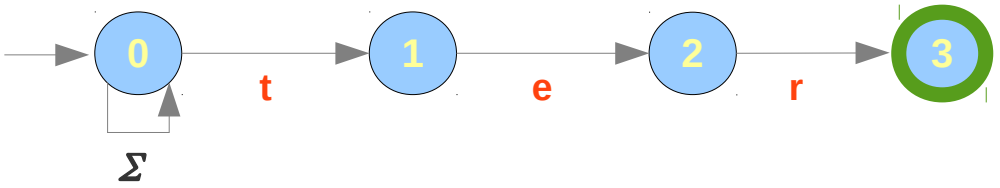
Äquivalenz von Automaten

Beispiel : Die Automaten A_1 und A_2 sind *nicht* äquivalent:

- **Für jeden Pfad in A_1, der in einem akzeptierenden Zustand endet, gibt es einen entsprechenden Pfad in A_2, aber**
- **Es gibt aber Pfade in A_2, die in einem akzeptierenden Zustand enden, für die es keinen entsprechenden Pfad in A_1 gibt.**



A_1 DEA



A_2 NEA

Konstruktion eines DEA zu einem NEA

Teilmengen-Konstruktion / Subset Construction

Zu jedem NEA kann ein äquivalenter DEA konstruiert werden.

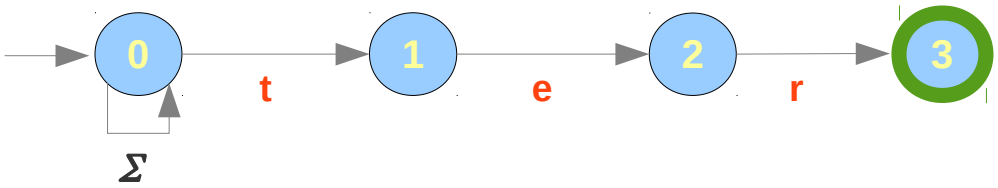
Ein einsprechender Algorithmus ist die Teilmengen-Konstruktion oder engl. *Subset Construction*

Idee der Teilmengen-Konstruktion

- **Ein Wort kann zu vielen Pfaden durch einen NEA führen,**
  einer oder mehrere dieser Pfade führt eventuell zu einen akzeptierenden Zustand
- **Die Prüfung eines Wortes bedeutet, dass ein solcher Pfad gesucht werden muss.**
- **Die Suche kann mit einem *Backtracking*-Algorithmus ausgeführt werden.**
- **Alternativ könnten alle möglichen Pfade gleichzeitig verfolgt werden,**
  der Automat müsste dann zu einem Zeitpunkt in mehreren Zuständen sein
- **Da ein NEA nur endlich viele Zustände hat, kann er auch gleichzeitig nur in endlich vielen Zuständen sein. (Die Potenzmenge einer endlichen Menge ist endlich.)**
- **Mit der Teilmengen-Konstruktion wird**
  - **zu einem NEA der DEA konstruiert,**
  - **dessen Zustände die Zustände sind, in denen**
  - **ein – parallel alle Pfade verfolgender – Durchlauf eines NEA gleichzeitig sein kann.**

Konstruktion eines DEA zu einem NEA

Beispiel

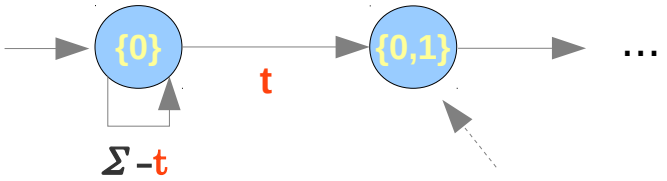


NEA

Bei Zeichen t in Zustand 0 hat der Automat die Wahl in Zustand 0 zu bleiben oder nach 1 zu wechseln

Am Anfang kann der Automat nur in Zustand 0 sein.

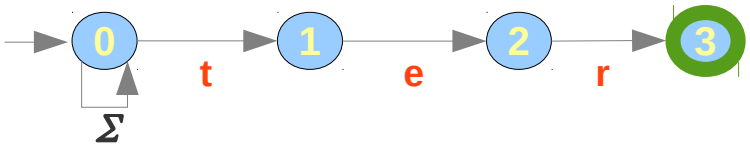
In Zustand 0 kann er mit t nach 0 oder 1 gehen, mit jedem anderen Zeichen bleibt er in 0:



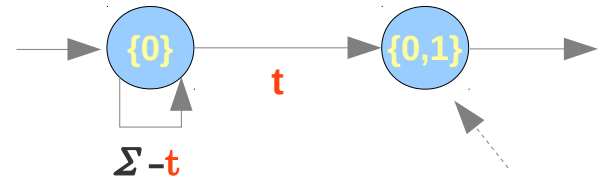
DEA

Der NEA oben ist in 0 oder 1:
Ein möglicher Durchlauf für ein Wort ist in 0, oder in 1

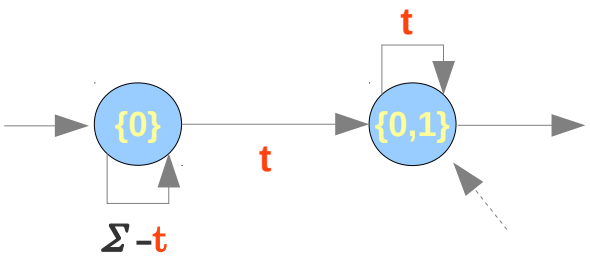
Konstruktion eines DEA zu einem NEA



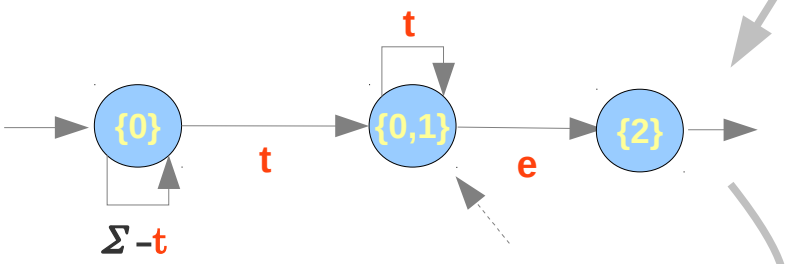
Beispiel – fortgesetzt



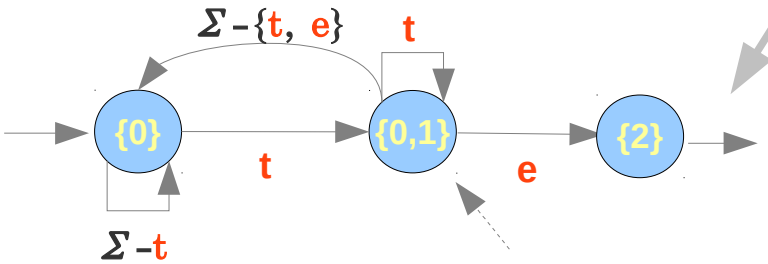
Betrachten wir jetzt den Zustand {0, 1}: Wir haben ein t gesehen und sind damit nach 1 gegangen oder in 0 geblieben.



Mit einem weiteren t können wir wieder nach 1 gehen oder in 0 bleiben:

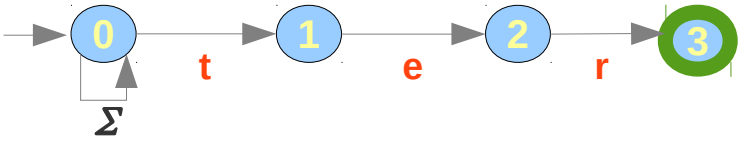


Mit einem e können wir, wenn wir in 1 sind, nach 2 gehen

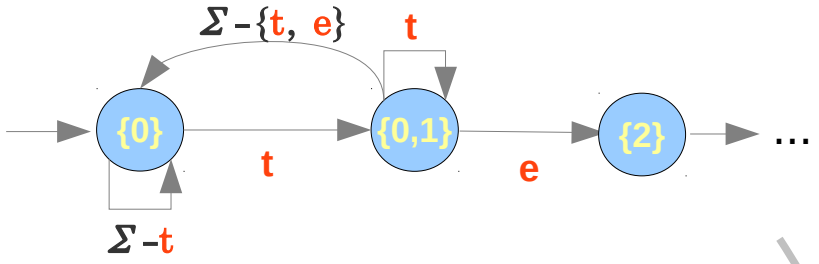


Folgt nach einem t ein Zeichen, das weder t noch e ist, (Eingabe z.B. „tx…“) dann sind wir nur dann noch „im Spiel“, wenn das erste t ignoriert wurde und wir in 0 geblieben sind. Wir bleiben dann weiter in 0.

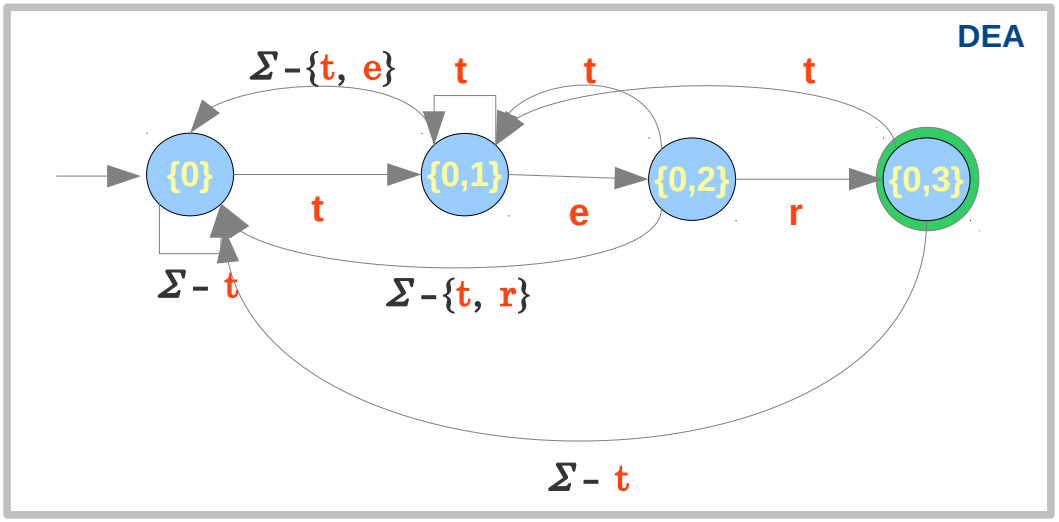
Konstruktion eines DEA zu einem NEA



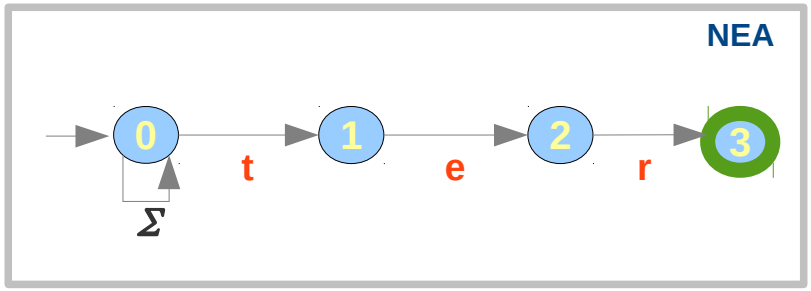
Beispiel – fortgesetzt



Nach weiterem verwickeltem Räsonieren über mögliche Übergänge die durch mögliche Worte ausgelöst werden könnten, kommen wir schließlich zu



≃



Hmm, geht das auch systematisch?

Konstruktion eines DEA zu einem NEA

Ein DEA kann systematisch konstruiert werden

1. Die Zustände des DEA sind Teilmengen der Menge aller Zustände des NEA

in der Regel besteht der DEA nur aus einer kleinen Teilmenge der Potenzmenge der Zustände des NEA .

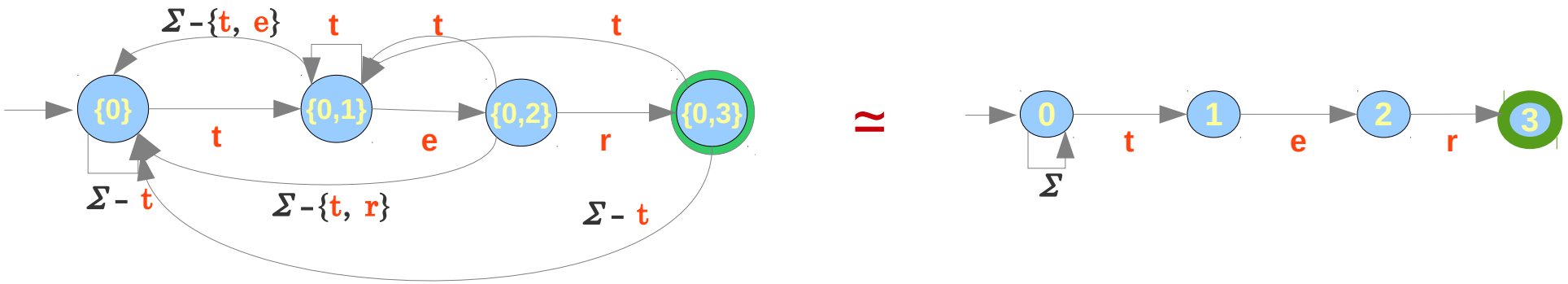
Beispiel oben:

- Der „ter“-NEA hat die Zustände 1, 2, 3
- Der „ter“-DEA“ hat die Zustände {0}, {0, 1}, {0, 3}

2. Die akzeptierenden Zustände des DEA enthalten einen akzeptierenden Zustand des NEA

Beispiel oben:

- Der „ter“-NEA hat den akzeptierenden Zustand 3
- Der „ter“-DEA“ hat den akzeptierenden Zustand {0, 3}



Konstruktion eines DEA zu einem NEA

Ein DEA kann systematisch konstruiert werden

3. Der DEA kann systematisch generiert werden

Dazu werden induktiv die erreichbaren Zustände des DEA berechnet:

– **Basis**

Der Zustand {s}, wobei s der Startzustand des NEAs ist, ist ein (erreichbarer) Zustand, der Startzustand, des DEA

Beispiel oben:

- Der „ter“-NEA hat den Startzustand 0
- Der „ter“-DEA“ hat den (Start-) Zustand {0}

– **Induktionsschritt**

Angenommen, wir haben S als Menge der erreichbaren Zustände des DEA berechnet.

Für jedes Symbol a werden Folgezustände für jedes $z \varepsilon S$ berechnet:

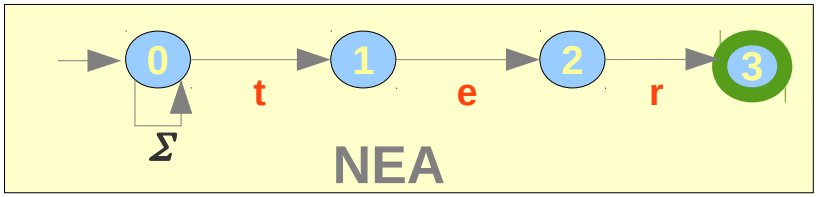
Dazu sei $z = \{s_1, \ldots s_n\}$ s_i sind dabei Zustände im NEA (ein DEA-Zustand ist eine Menge von NEA-Zuständen)

Für jeden Zustand $s_i \varepsilon \{s_1, \ldots s_n\} = z$ berechnen wir die Menge $\delta_{NEA}(s_i, a)$

der möglichen Folgezustände bei Eingabe a im DEA. Deren Vereinigung ist der Folgezustand von z bei Eingabe a

$$\delta_{DEA}(z, a) = \bigcup \delta_{NEA}(s_i, a) \qquad s_i \in z$$

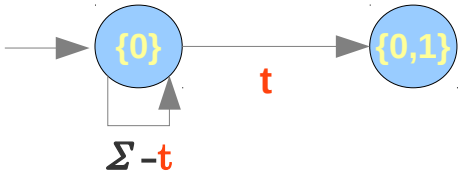
Konstruktion eines DEA zu einem NEA

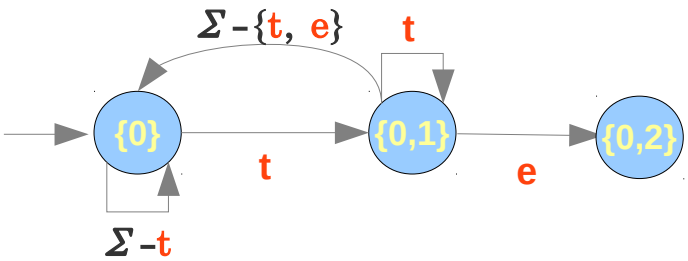


Ein DEA kann systematisch konstruiert werden

Induktionsschritt

Beispiel, angenommen wir haben bereits konstruiert:
und berechnen jetzt die Folgezustände von {0, 1}





– Folgezustände von ({0,1}, e)
  NEA:
  - 0,e ~> 0
  - 1,e ~> 2

  also: ({0,1}, e) ~> {0,2}
– Folgezustände von ({0,1}, t)
  NEA:
  - 0,t ~> 0
  - 0,t ~> 1

  also: ({0,1}, t) ~> {0,1}
– Folgezustände von ({0,1}, x) x ∉ {t, e}
  NEA:
  - 0,x ~> 0

  also: ({0,1}, x) ~> {0}

Konstruktion eines DEA zu einem NEA

Konstruktion deterministischer endlicher Automaten (Teilmengenkonstruktion)

Eingabe: Ein NEA $A = (S, \Sigma, next, s_0, F)$

Ausgabe: Ein DEA $A' = (S', \Sigma, next', s_0, F')$ mit L(*A*) = L(A').

Idee: Der nichtdeterministische Ausgangsautomat A kann mit einer Eingabe w ggf. unterschiedliche Zustände aus S erreichen. Der Zustand, den der äquivalente DEA A' mit der Eingabe w erreicht, repräsentiert die Menge aller Zustände, die A mit der Eingabe w erreichen kann. Jeder Zustand T des DEA A' ist also eine Teilmenge von S.

Basis-Operationen:

zu jedem Zustand $s \in S$, zu jeder Zustandsmenge $T \in 2^S$ und jedem $a \in \Sigma$ sei definiert:

- ε-Abschluss(s)=$\{s' \in S \mid s' = s$ oder s' ist von s aus durch einen oder mehrere ε-Übergänge zu erreichen $\}$
- ε-Abschluss(T)=$\bigcup_{s \in T}$ ε-Abschluss(s)
- $move(T, a) = \{s' \in S \mid \exists_{s \in T} : (s, a, s') \in next\}$

Aus M. Jäger: *Compilerbau – Eine Einführung*, Seite 70, 71

Konstruktion eines DEA zu einem NEA

Algorithmus:

Die Zustandsmenge *S* und die Übergangsfunktion *next* werden sukzessive erweitert, alle in S übernommenen Zustände seien anfangs unmarkiert.

S := ε-Abschluss(s_0)
while (S enthält unmarkierten Zustand T) **do**
  markiere T
  for $a \in \Sigma$ **do**
    U:=ε-Abschluss(*move(T,a)*);
    if not $U \in S$ **then**
      $S := S \cup \{U\}$
    end if;
    next(T,a) := U;
  end for;
end while;

Aus M. Jäger: *Compilerbau – Eine Einführung*, Seite 72