

Compiler

Scanner

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik

Hochschule München

Letzte Änderung: 27.01.2020 19:45

Inhaltsverzeichnis

Scanner	3
Wörter erkennen — Beispiel	3
Wörter erkennen — Beispiel	5
Verschiedene Wörter erkennen	8
Ein Formalismus für Recognizer	9
Ein Endlicher Automat (EA)	9
Beispiel	11
Beispiel in Haskell	12
Positive Zahlen erkennen	14
Reguläre Ausdrücke	15
Reguläre Ausdrücke	15
Formalisierung regulärer Ausdrücke	15
Operationen	15
Definition regulärer Ausdrücke	16
Vorrang und Intervalle,	16
Beispiele	16
PCRE - Perl Compatible Regular Expressions	16
Reguläre Ausdrücke in der Praxis	16
Sonderzeichen in PCRE	17
Quantifikatoren	17
Backtracking	17

Zu gierige REs	17
Kein Backtracking	18
Zeichenklassen	18
Gruppierung und Referenzierung	18
und viel viel mehr	18
Von regulären Ausdrücken zu Scannern	19
Konstruktionszyklus	19
Konstruktion von EAs	19
Nichtdeterministischer Endlicher Automat	25
Äquivalenz von NEAs und DEAs	29
RE nach NEA: Thompson's Construction	33
RE nach NEA: Thompson's Construction (2)	37
Anwendung von Thompson's Construction	40
Haskell-Datentyp für Reguläre Ausdrücke	41
Beispiel: RE $(aa^*a (c d)^*b)^*e$ in Haskell	41
Haskell-Datentyp für NFAs	41
Beispiel: NFA in Haskell für ab	42
Beispiel: NFA in Haskell für $a b$	42
Thompson's Construction in Haskell	42
NEA nach DEA: Die Teilmengenkonstruktion	43
Algorithmus zur Teilmengenkonstruktion	43
Von Q nach D	43
Beispiel	45
Haskell-Datentyp für DFAs	46
Teilmengenkonstruktion in Haskell	46
FixPunkt-Berechnungen	46
Erzeugen eines minimal DFA aus einem beliebigen DFA: Hopcroft's Algorithmus	46
Hopcroft's Algorithmus	47
Beispiel	48
Hopcroft's Algorithmus in Haskell	48
Vom DEA zum Recognizer	48
Eine andere Art zu erkennen	48
Implementierung von Scannern	49
Table-Driven Scanner	49
Beispiel	49
Exzessiven Rollback vermeiden	50
Direct-Coded Scanners	50
Overhead des Tabellen-Lookups	50
Ersatz für die while-Schleife des Table-Driven Scanners	51
Beispiel	51
Hand-coded Scanner	51

Scanner

- erster Schritt des Prozesses der das Eingabeprogramm verstehen muss
- Scanner = lexical analyzer
- ein Scanner
 - liest eine Zeichen-Strom
 - produziert einen Strom von Wörtern
- aggregiert Zeichen um Wörter zu bilden
- wendet eine Menge von Regeln an um zu entscheiden ob ein Wort akzeptiert wird oder nicht
- weist dem Wort eine syntaktische Kategorie zu, wenn es akzeptiert wurde

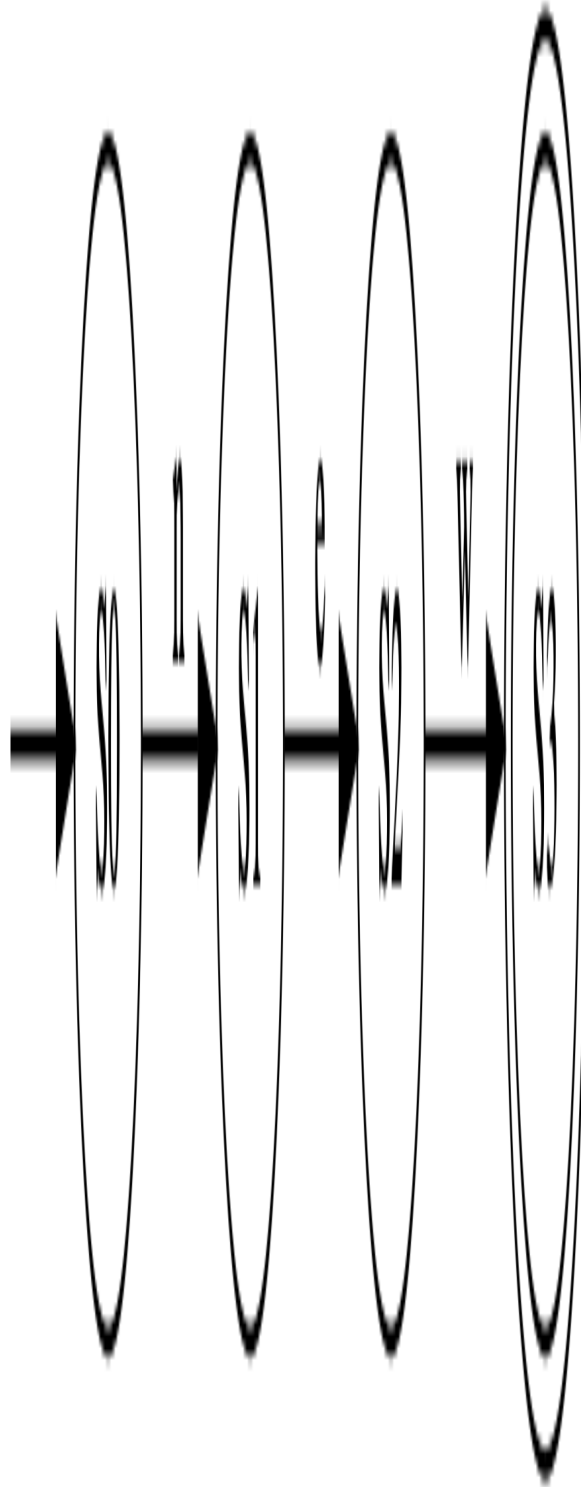
Wörter erkennen — Beispiel

new erkennen

Pseudo Code

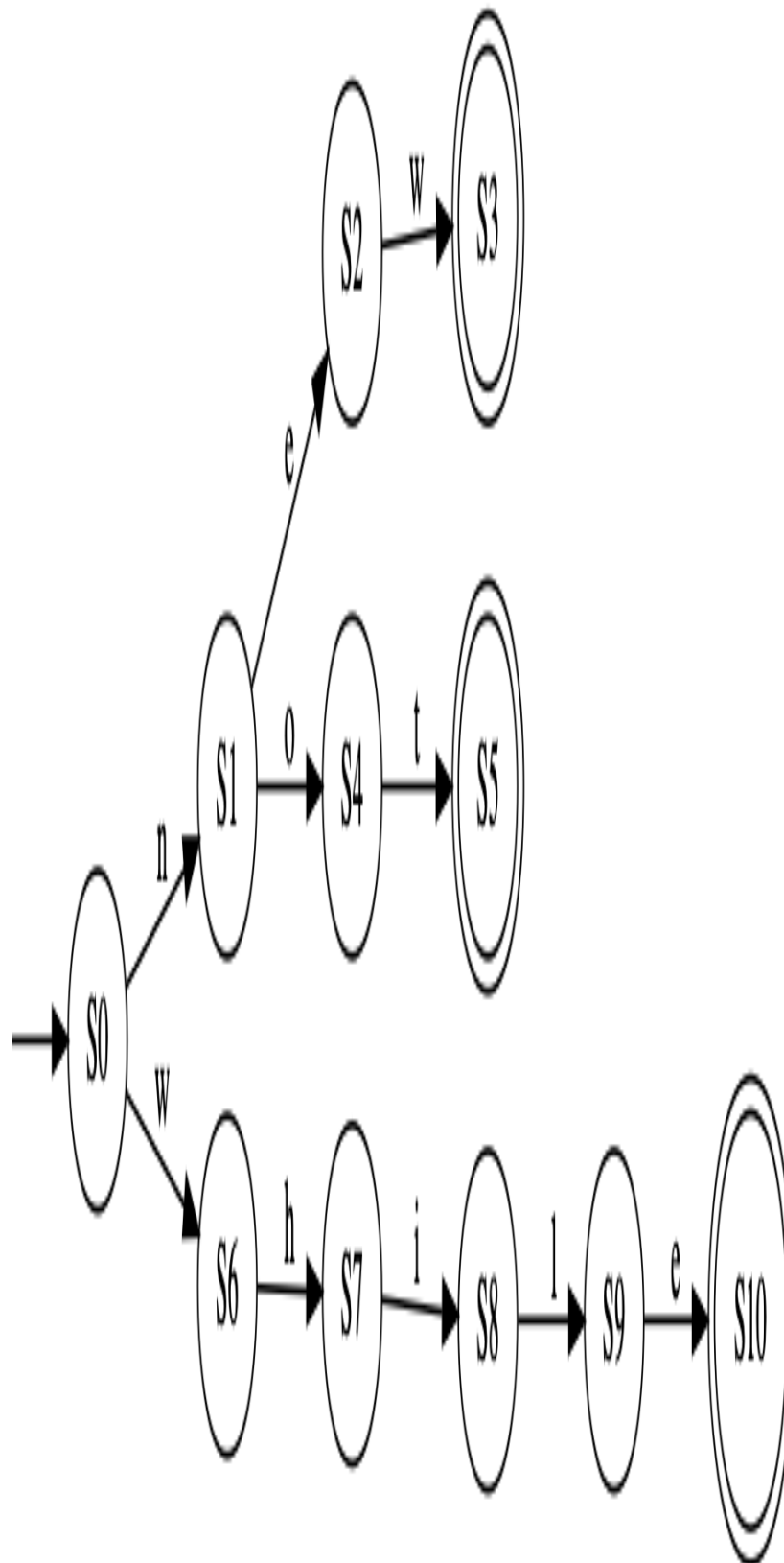
```
c = nextChar();
if (c == 'n')
  then begin;
    c = nextChar();
    if (c == 'e')
      then begin;
        c = nextChar();
        if (c == 'w')
          then report success;
          else try something else;
        end;
      else try something else;
    end;
  else try something else;
```


Wörter erkennen — Beispiel



Haskell:

```
recognizeNew :: String -> Bool
recognizeNew ('n':'e':'w':_) = True
recognizeNew _                = False
```


Verschiedene Wörter erkennen

Ein Formalismus für Recognizer

Zustandsübergangsdiagramme können als mathematische Objekte betrachtet werden, sog. **Endliche Automaten** (*finite automata*)

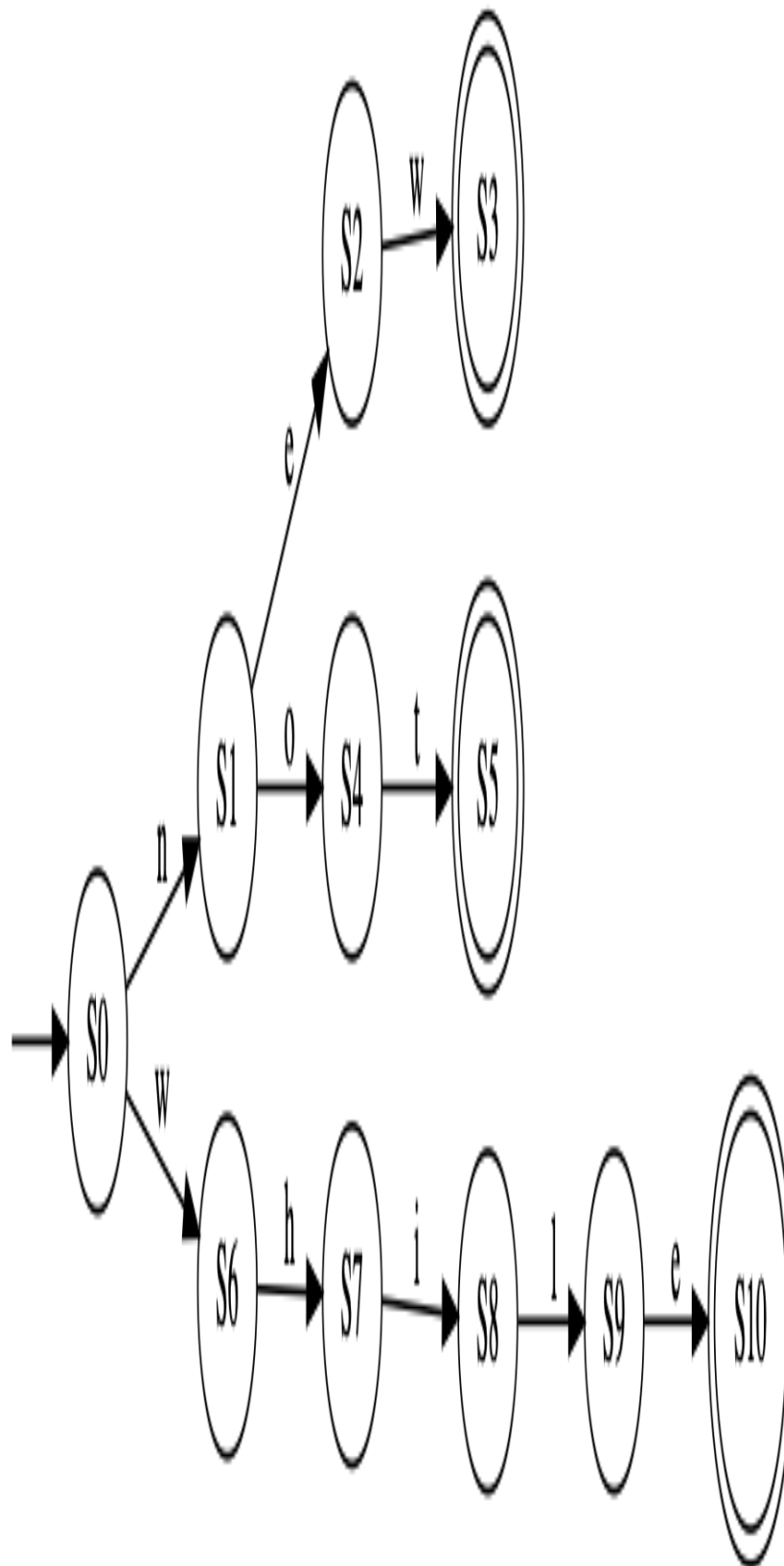
Ein Endlicher Automat (EA)

(engl. *finite automaton (FA)*)

ist ein Tupel $(S, \Sigma, \sigma, s_0, S_A)$ mit

- S ist die endlichen Menge von Zuständen im EA, sowie ein Fehlerzustand s_e .
- Σ ist das vom EA genutzte Alphabet. Typischerweise ist es die Vereinigungsmenge der Kantenbezeichnungen im Zustandsübergangsdiagramm.
- $\sigma(s, c)$ ist die Zustandsübergangsfunktion. Es bildet jeden Zustand $s \in S$ und jedes Zeichen $c \in \Sigma$ auf den Folgezustand an. Im Zustand s_i mit dem Eingabezeichen c , nimmt der EA den Übergang $s_i \xrightarrow{c} \sigma(s_i, c)$.
- $s_0 \in S$ ist der ausgewählte Startzustand.
- S_A ist die Menge von akzeptierenden Zuständen, mit $S_A \subseteq S$. Jeder Zustand in S_A wird als doppelt umrandeter Kreis im Zustandsübergangsdiagramm dargestellt.

Beispiel



$$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$$

$$\Sigma = \{e, h, i, l, n, o, t, w\}$$

$$\sigma = \{s_0 \xrightarrow{n} s_1, s_0 \xrightarrow{w} s_6, s_1 \xrightarrow{e} s_2, \dots\}$$

$$s_0 = s_0$$

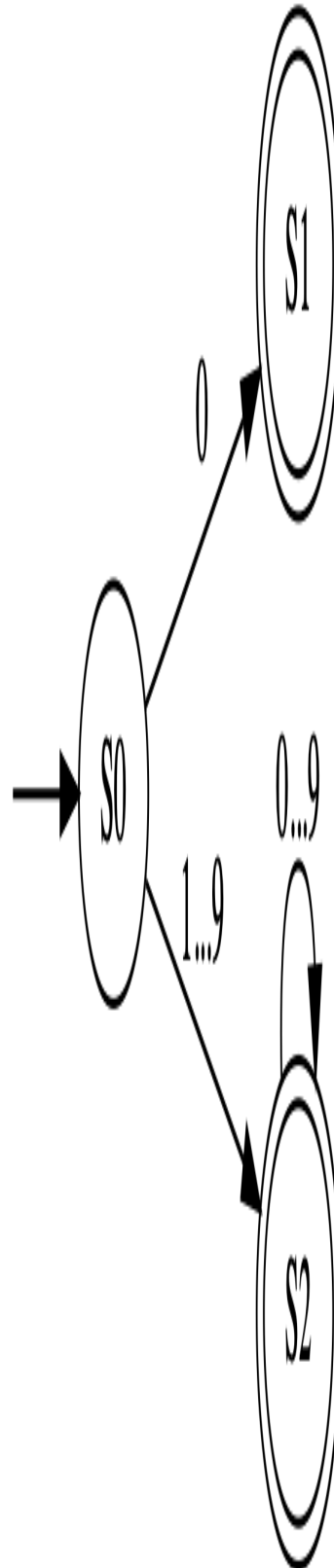
$$S_A = \{s_3, s_5, s_{10}\}$$

Übung: Ergänzen Sie σ durch die fehlenden Abbildungen.

Beispiel in Haskell

[Code auf GitHub](#)

<https://github.com/ob-cs-hm-edu/compiler-ea1.git>

Positive Zahlen erkennen

Übung: Geben Sie den endlichen Automaten als Tupel an.

Reguläre Ausdrücke

Reguläre Ausdrücke

- die Menge der Wörter die von einem endlichen Automaten \mathcal{F} akzeptiert wird, bildet eine Sprache, die $L(\mathcal{F})$ bezeichnet wird
- das Zustandsübergangsdiagramm des EA spezifiziert diese Sprache
- intuitiver ist die Spezifikation mit **regulären Ausdrücken** (*regular expressions* (REs))
- die Sprache die durch einen RE beschrieben wird, heisst **reguläre Sprache**
- REs sind äquivalent zu EAs

Formalisierung regulärer Ausdrücke

Ein regulärer Ausdruck r beschreibt

- eine Menge von Zeichenketten, genannt *Sprache*, bezeichnet mit $L(r)$,
- bestehend aus Zeichen aus einem Alphabet Σ
- erweitert um ein Zeichen ϵ das die leere Zeichenkette repräsentiert

Operationen

Ein regulärer Ausdruck wird aus drei Grundoperationen zusammengesetzt

Alternative Die Alternative oder Vereinigung von zwei Mengen von Zeichenketten R and S , wird geschrieben $R|S$ und ist $\{x|x \in R \text{ or } x \in S\}$.

Verkettung Die Verkettung zweier Mengen R and S , wird geschrieben RS und enthält alle Zeichenketten, die entstehend wenn an ein Element aus R ein Element aus S angehängt wird, also $\{xy|x \in R \text{ and } y \in S\}$.

Kleenesche Hülle Die Kleenesche Hülle, oder Kleene-Stern, einer Menge R , wird geschrieben R^* und ist $\bigcup_{i=0}^{\infty} R^i$. Das sind also alle Verkettungen von R mit sich selbst, null bis unendlich mal.

Zusätzlich wird oft genutzt

Endliche Hülle R^i , für ein positives i

Positive Hülle R^+ , als Kurzschreibweise für RR^*

Definition regulärer Ausdrücke

Die Menge der REs über einem Alphabet Σ ist definiert durch

1. Wenn $a \in \Sigma$, dann ist a ein RE der die Menge beschreibt, die nur a enthält.
2. Wenn r und s REs sind die $L(r)$ and $L(s)$ beschreiben, dann gilt:
 - $r|s$ ist ein RE
 - rs ist ein RE
 - r^* ist ein RE
3. ϵ ist ein RE der die Menge beschreibt, die nur die leere Zeichenkette enthält.

Vorrang und Intervalle, ...

Reihenfolge des Vorrangs (vom höchsten):

- Klammern
- Hülle
- Verkettung
- Alternative

Zeichenintervalle können durch das erste und letzte Element verbunden mit drei Punkten umschlossen von eckigen Klammern beschrieben werden, z.B. $[0\dots9]$.

Komplementbildungs-Operator $\hat{}$ ist die Menge $\Sigma - c$
Escape Sequenzen wie in Zeichenketten, z.B. $\backslash n$

Beispiele

- Bezeichner in manchen Programmiersprachen
 $([A\dots Z]|[a\dots z])([A\dots Z]|[a\dots z]|[0\dots9])^*$
- positive ganze Zahlen
 $0|[1\dots9][0\dots9]^*$
- positive reelle Zahlen
 $(0|[1\dots9][0\dots9]^*)(\epsilon|[0\dots9]^*)$

PCRE - Perl Compatible Regular Expressions

Reguläre Ausdrücke in der Praxis

- es gibt verschiedene "Geschmacksrichtungen" regulärer Ausdrücke

- wir verwenden im Praktikum und in der Klausur PCRE (Perl Compatible Regular Expressions)

Sonderzeichen in PCRE

Zeichen mit besonderer Bedeutung in PCRE

Sonderzeichen Bedeutung

\ um das folgende Sonderzeichen zu maskieren ^ Zeilenanfang . ein beliebiges Zeichen (außer Zeilenumbruch) \$ Zeilenende oder Ende der Zeichenkette | Alternative () Gruppierung [] Umschließt eine Zeichenklasse

Quantifikatoren

Quantifikator Bedeutung

* matche null- oder mehrmals + matche ein- oder mehrmals ? matche null- oder einmal {n} matche genau n-mal {n,} matche mindestens n-mal {n,m} matche mindestens n- und höchstens m-mal

Backtracking

- wenn ein quantifiziertes Teilpattern dazu führen würde, dass der Rest nicht mehr matched, wird Backtracking verwendet
- Beispiel: Zeichenkette aaaa
- Regulärer Ausdruck: a+a
 - a+ würde schon die gesamte Zeichenkette matchen
 - durch Backtracking matched a+ auf die ersten drei as und das zweite a im RE auf das vierte in der Zeichenkette

Zu gierige REs

- durch Hintanstellen von ? wird nur auf das Minimum gematcht
- Beispiel: Zeichenkette aaab
- Regulärer Ausdruck a+(a|b) matcht auf aaab
- Regulärer Ausdruck a+?(a|b) matcht auf aa

Kein Backtracking

- durch Hintanstellen von + wird das Backtracking ausgeschaltet
- Beispiel: Zeichenkette aaaa
- Regulärer Ausdruck a+a matcht auf aaaa
- Regulärer Ausdruck a++a matcht gar nicht

Zeichenklassen

Sequenz Bedeutung

[...] eines der statt ... enthaltenen Zeichen, auch [a-z] möglich [^...] keines der enthaltenen Zeichen [[:...:]] Posix-Klassen, z.B. `digit`, `upper` \w alphanumerisches Zeichen oder `_` \W kein alphanumerisches Zeichen oder `_` \s Whitespace \S kein Whitespace \d Dezimalziffer \D keine Dezimalziffer ...

Gruppierung und Referenzierung

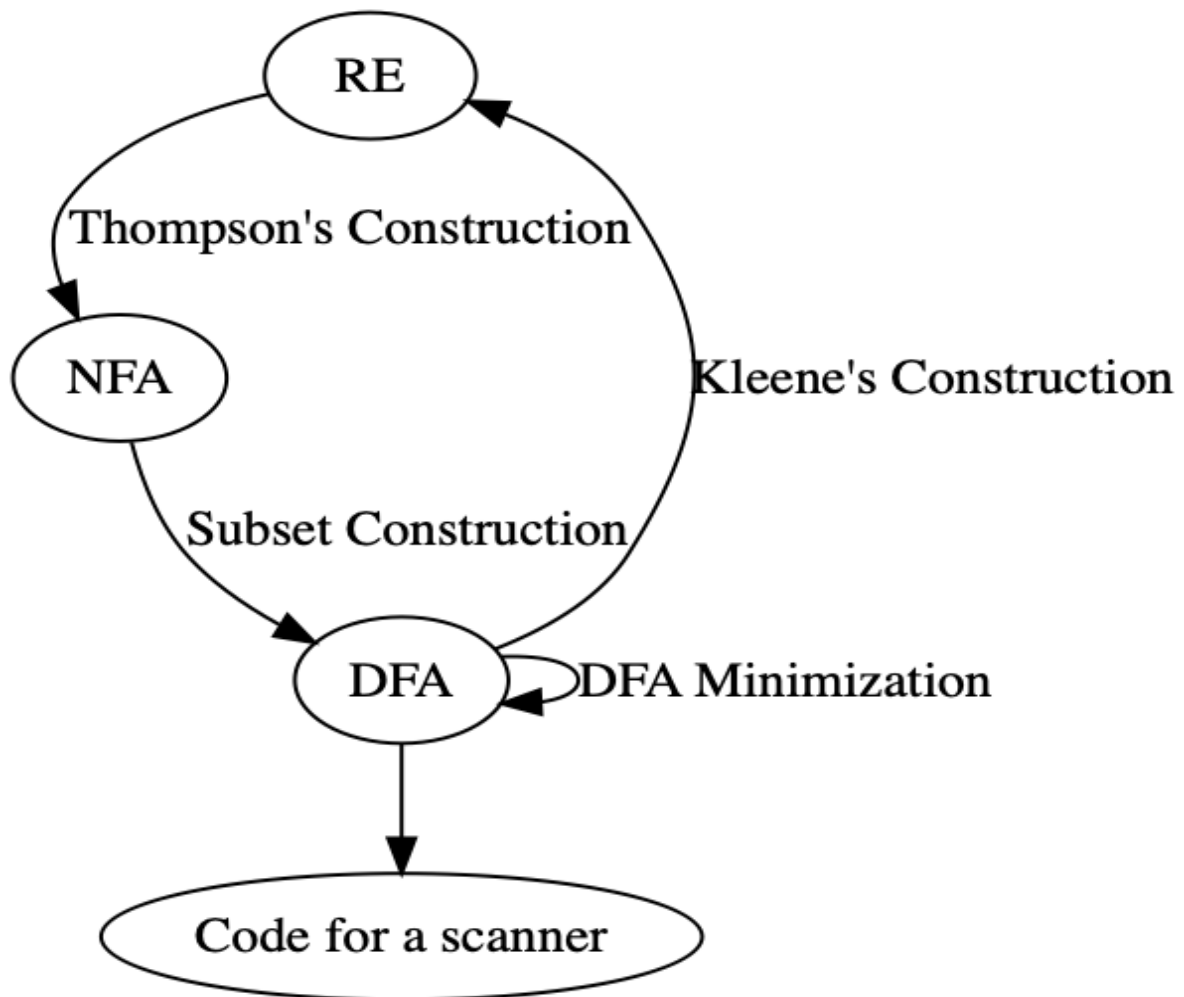
- Teilausdrücke in Klammern werden erfasst
- und können referenziert werden
- Beispiel: Zeichenkette Hallo Hans
- Regulärer Ausdruck `(..)*\1` matcht auf Hallo Ha

und viel viel mehr

- Lookaround Assertions
 - z.B. matche ein Wort auf das ein Tabulator folgt: `\w+(?=\t)`
- Rekursive Subpattern
 - z.B. matche geklammerte Ausdrücke: `\((?>[^(]+|(?R))*\)`
 - `(?>S)` ist eine non-backtracking-group und verhindert daher zeitraubendes Backtracking
- ...
- Doku z.B. unter <http://perldoc.perl.org/perlre.html>

Von regulären Ausdrücken zu Scannern

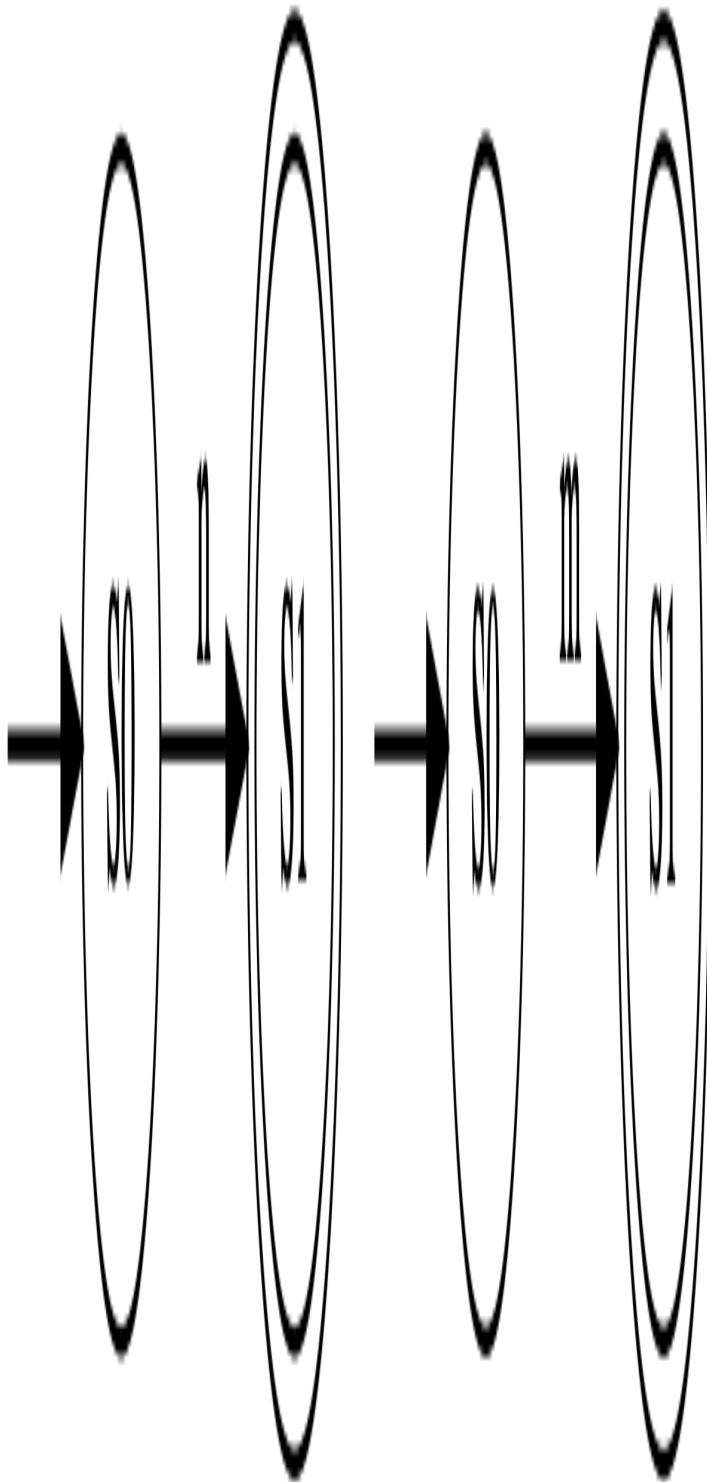
Konstruktionszyklus



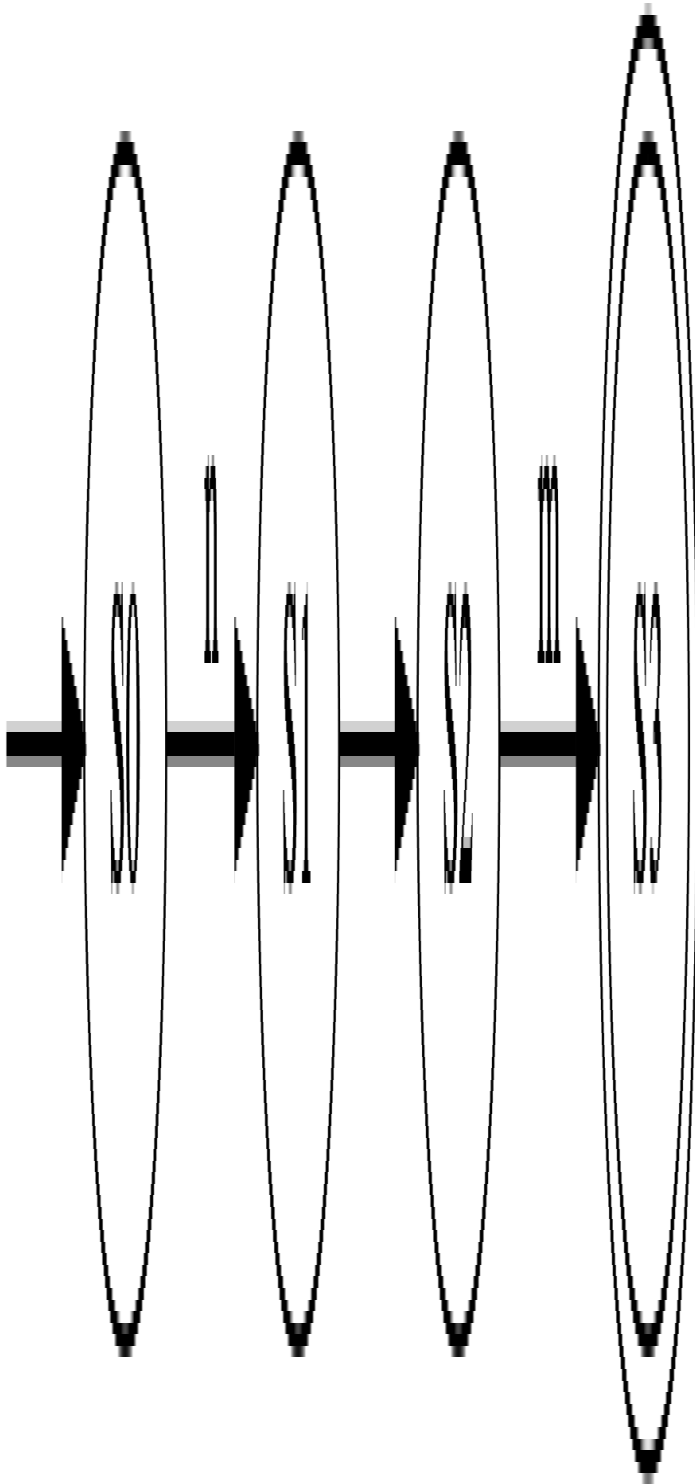
Konstruktionszyklus

Konstruktion von EAs

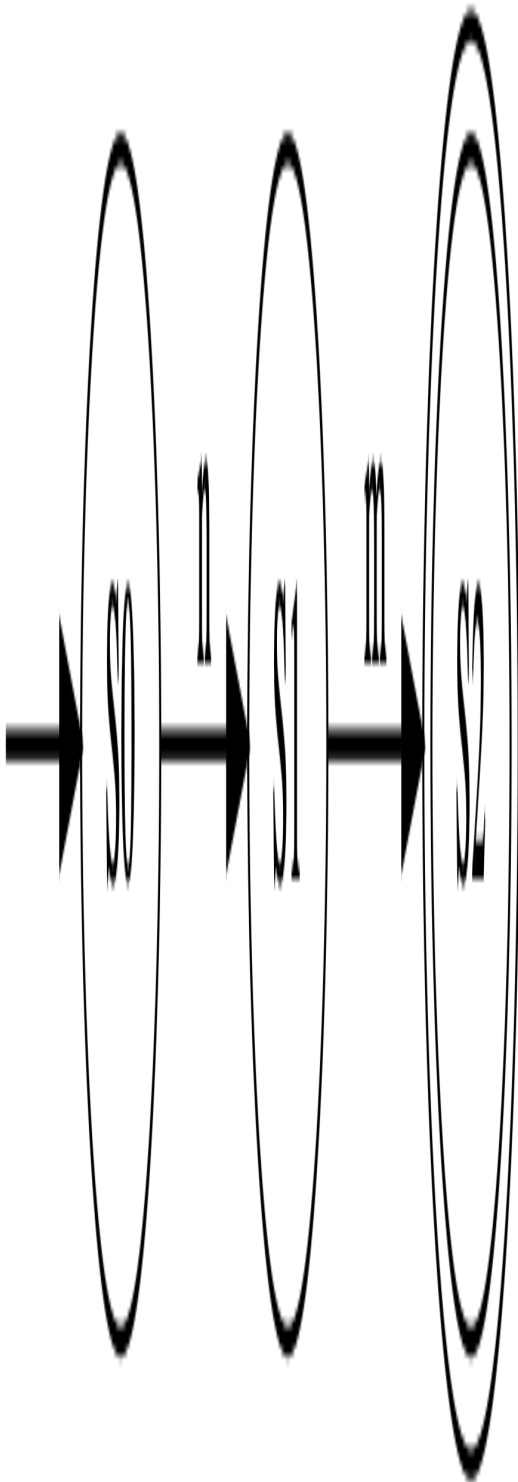
- gegeben seien die beiden EAs



- Wir können einen ϵ -Übergang, der die leere Zeichenkette akzeptiert, einfügen und so einen EA für nm konstruieren.

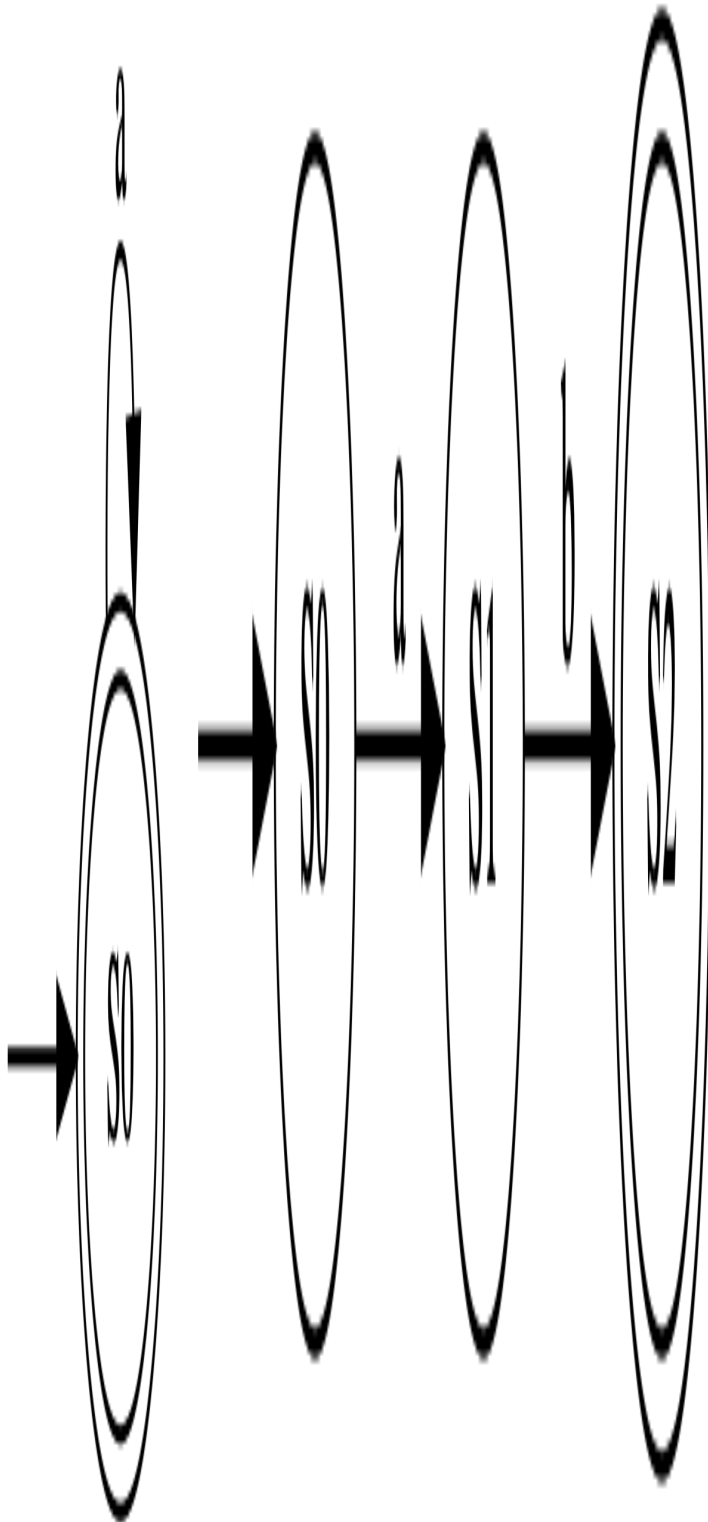


- Im zweiten Schritt können wir den ϵ -Übergang eliminieren.

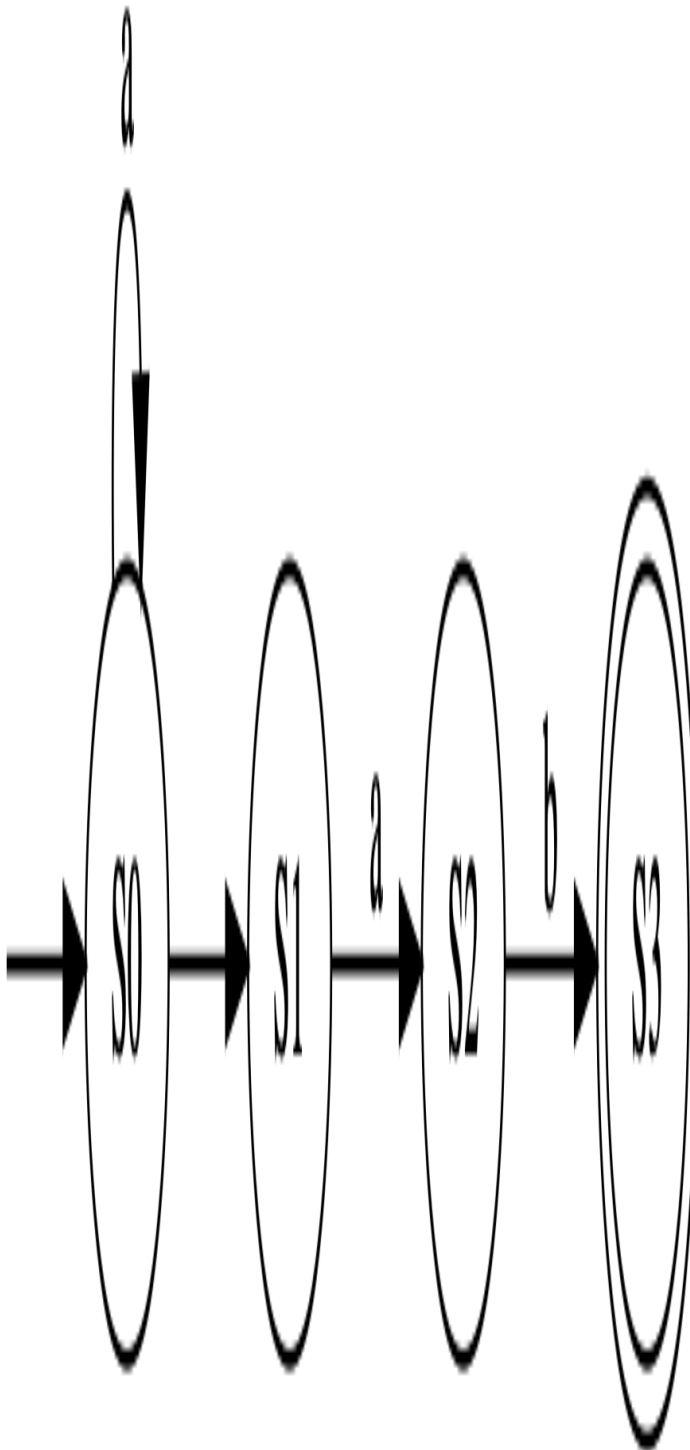


Nichtdeterministischer Endlicher Automat

- angenommen wir wollen die folgenden beiden EAs konkatenieren



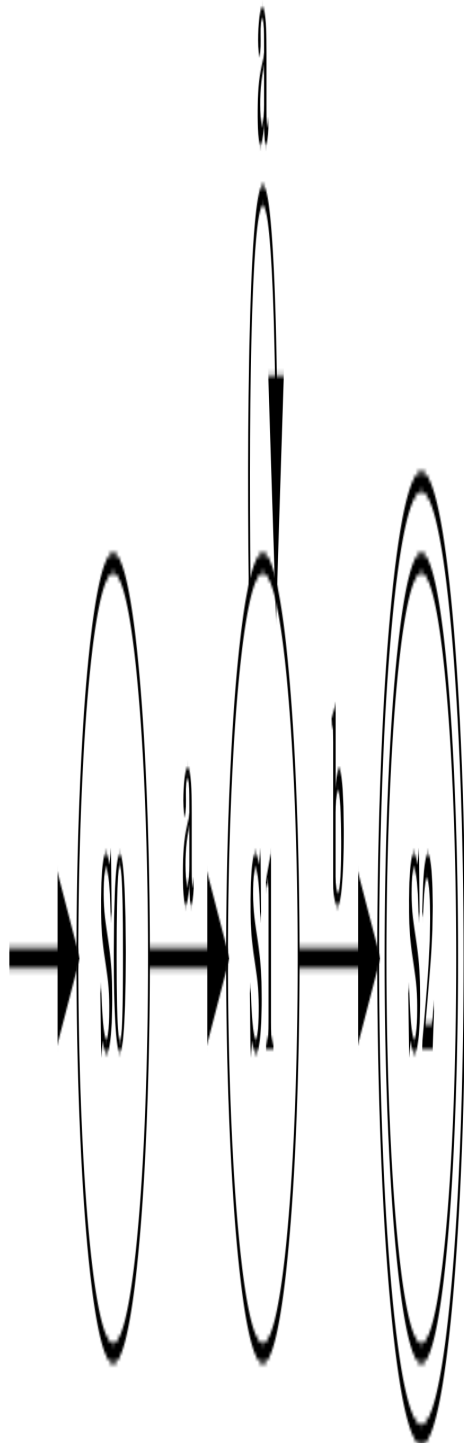
- mit einem ϵ -Übergang bekommen wir



Das ist ein **NEA**, weil es von einem Zustand mehrere Übergänge mit einem Zeichen gibt.

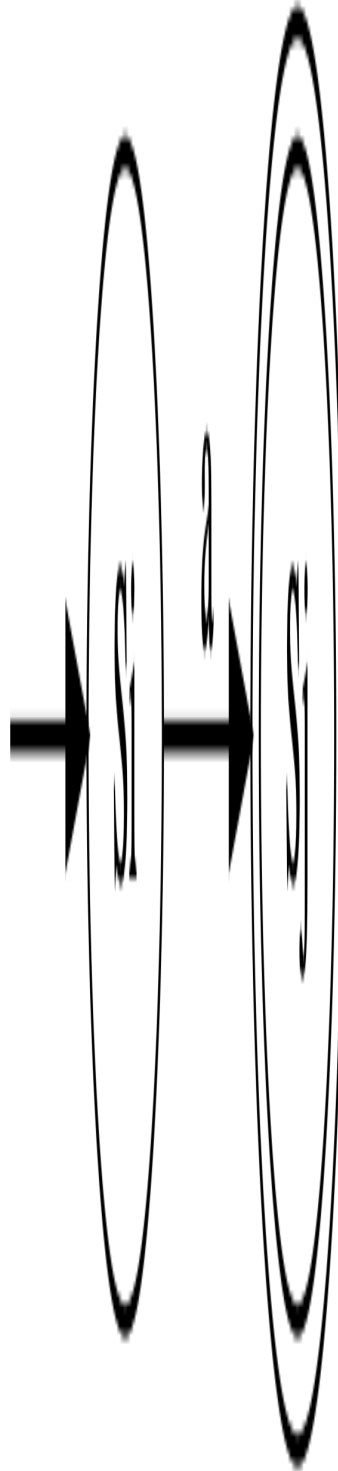
Äquivalenz von NEAs und DEAs

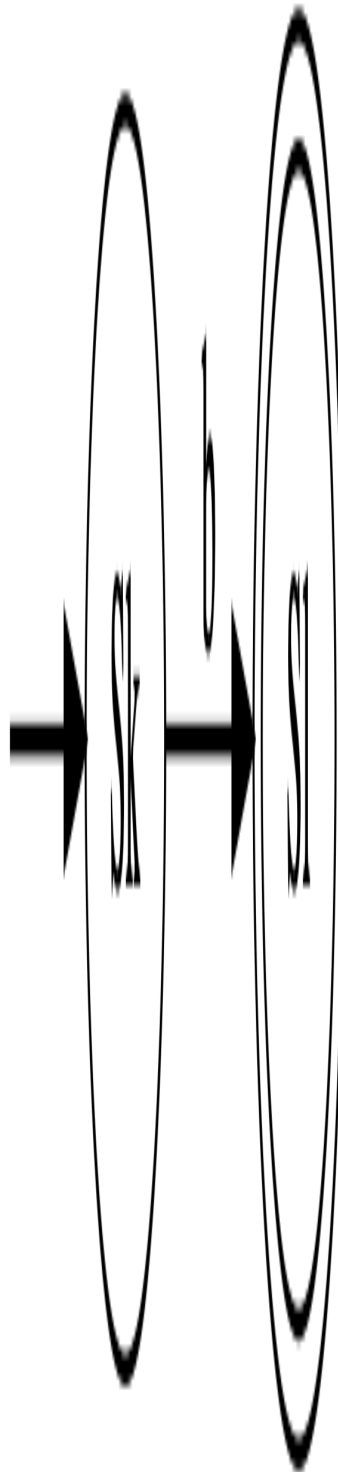
- NEAs und DEAs sind äquivalent bzgl. ihrer Ausdruckskraft
- jeder NEA kann durch einen DEA simuliert werden
- DEA für a^*ab ist



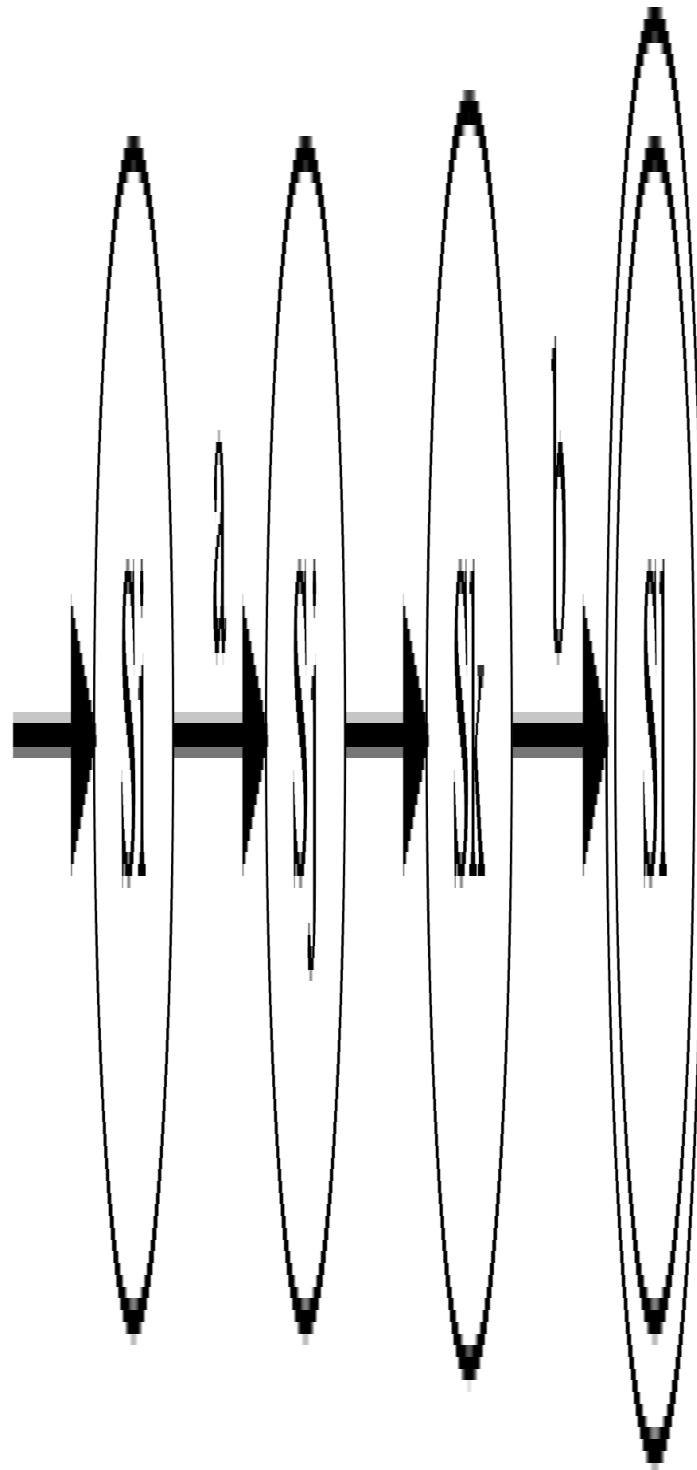
das ist der selbe wie für aa^*b

RE nach NEA: Thompson's Construction



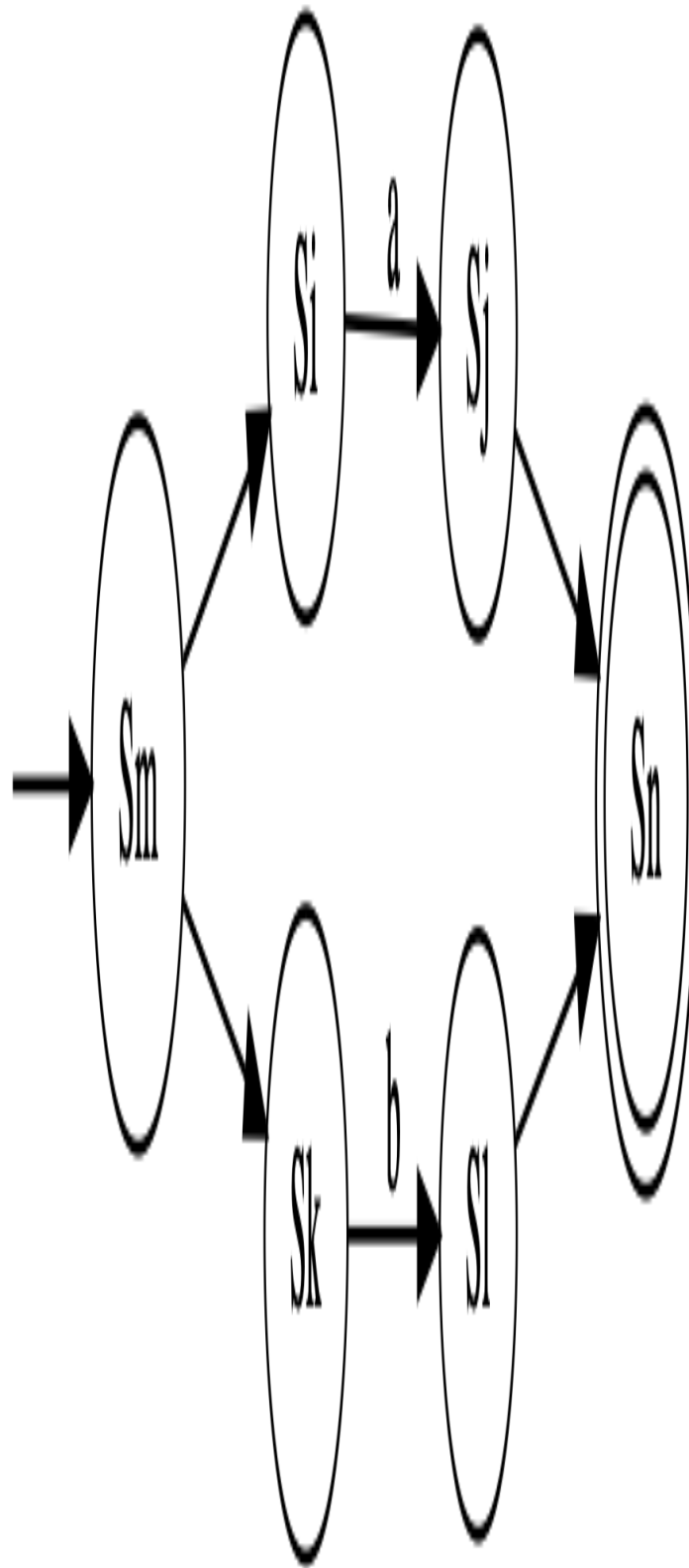


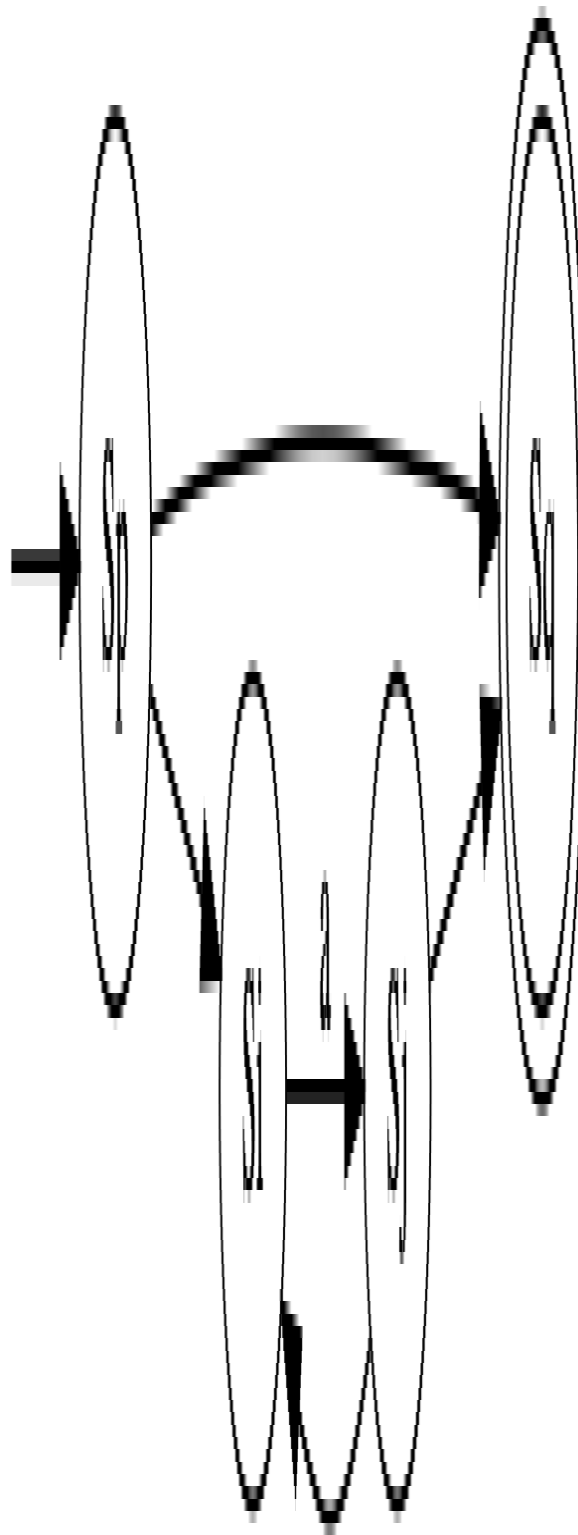
NEA für b



NEA für *ab*

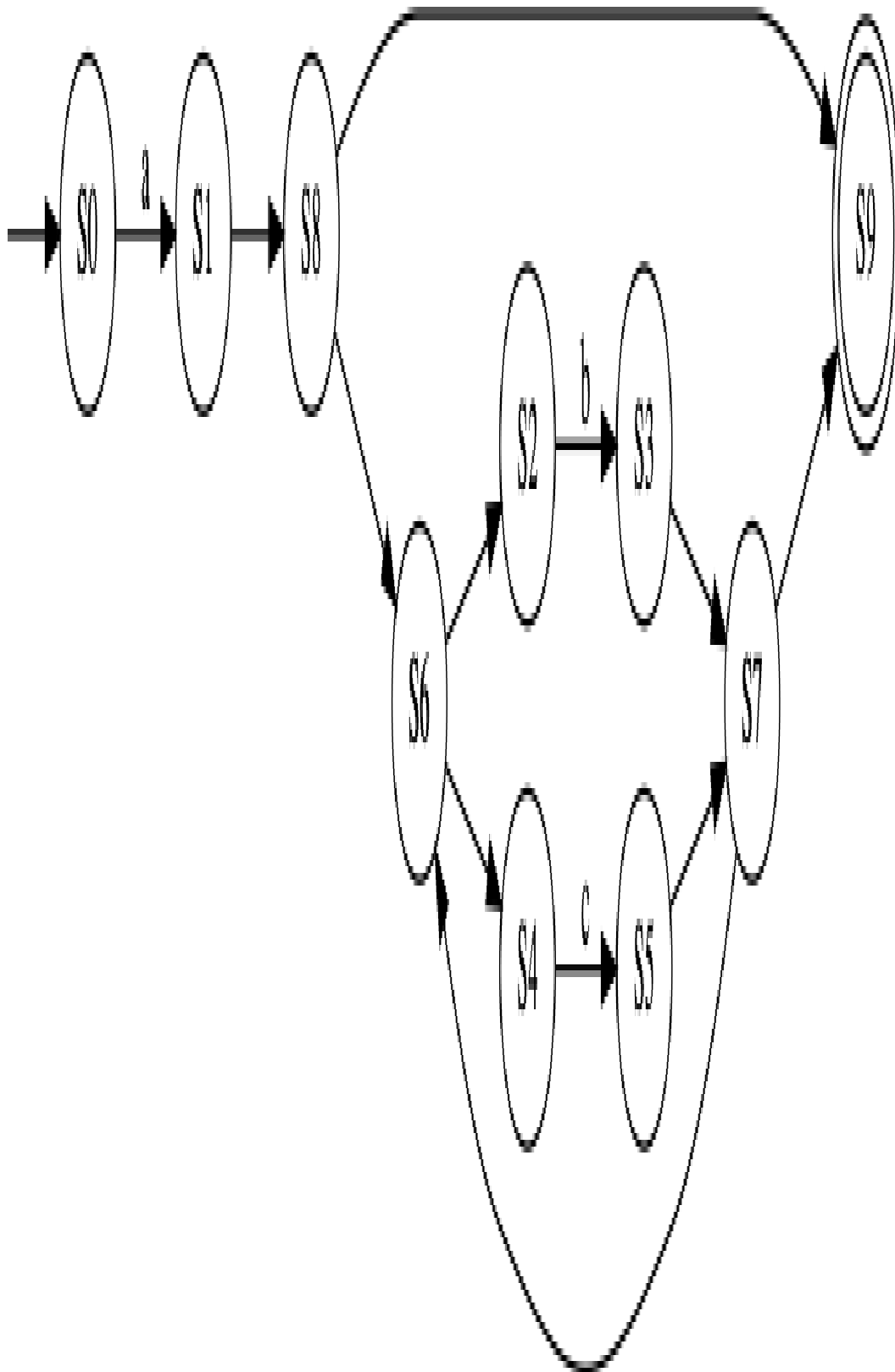
RE nach NEA: Thompson's Construction (2)





NEA für a^*

Anwendung von Thompson's Construction



Haskell-Datentyp für Reguläre Ausdrücke

- Datentyp für reguläre Ausdrücke

```
data RE = PrimitiveRE Char
       | ConcatenatedRE RE RE
       | AlternativeRE RE RE
       | ClosureRE RE
```

- Beispiele

```
- a: PrimitiveRE 'a'
- a*: ClosureRE (PrimitiveRE 'a')
- ab:
  ConcatenatedRE (PrimitiveRE 'a') (PrimitiveRE 'b')
- a|b:
  AlternativeRE (PrimitiveRE 'a') (PrimitiveRE 'b')
```

Beispiel: RE $(aa^*a|(c|d)^*b)^*e$ in Haskell

```
ConcatenatedRE
  (ClosureRE
    (AlternativeRE
      (ConcatenatedRE
        (PrimitiveRE 'a')
        (ConcatenatedRE
          (ClosureRE
            (PrimitiveRE 'a'))
            (PrimitiveRE 'a'))))
      (ConcatenatedRE
        (ClosureRE
          (AlternativeRE
            (PrimitiveRE 'c')
            (PrimitiveRE 'd'))))
        (PrimitiveRE 'b'))))
    (PrimitiveRE 'e')
```

Haskell-Datentyp für NFAs

```
data NFASState = NFASState
  { nfaStateNumber :: Integer
  }
```

```

data NFA = NFA
  { -- nfaStates    :: Set NFAState -- wird berechnet
    -- nfaBigSigma :: [Char]       -- wird berechnet
    nfaSigma :: Set ((NFAState, Maybe Char), NFAState)
  , nfaStart  :: NFAState
  , nfaAcceptingStates :: Set NFAState
  }

```

Beispiel: NFA in Haskell für ab

NFA

```

{ nfaSigma = Set.fromList
  [ ((NFAState 0, Just 'a'), NFAState 1)
  , ((NFAState 1, Nothing), NFAState 2)
  , ((NFAState 2, Just 'b'), NFAState 3)
  ]
  , nfaStart = NFAState 0
  , nfaAcceptingStates = Set.singleton $ NFAState 3
}

```

Beispiel: NFA in Haskell für $a|b$

NFA

```

{ nfaSigma = Set.fromList
  [ ((NFAState 0, Nothing), NFAState 1)
  , ((NFAState 0, Nothing), NFAState 3)
  , ((NFAState 1, Just 'a'), NFAState 2)
  , ((NFAState 3, Just 'b'), NFAState 4)
  , ((NFAState 2, Nothing), NFAState 5)
  , ((NFAState 4, Nothing), NFAState 5)
  ]
  , nfaStart = NFAState 0
  , nfaAcceptingStates = Set.singleton $ NFAState 5
}

```

Thompson's Construction in Haskell

[ThompsonsConstruction.hs @ GitHub](#)

NEA nach DEA: Die Teilmengenkonstruktion

die Teilmengenkonstruktion nimmt einen NEA

- $(N, \Sigma, \sigma_N, n_0, N_A)$

und produziert einen DEA

- $(D, \Sigma, \sigma_D, d_0, D_A)$

Algorithmus zur Teilmengenkonstruktion

```

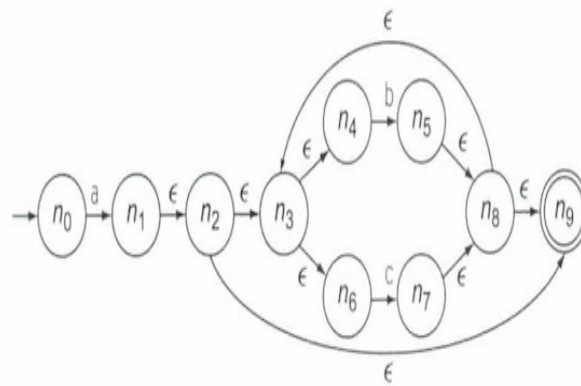
q_0 = epsilonClosure({n_0});
Q = q_0;
Worklist = {q_0};
while (Worklist != {}) do
    remove q from Worklist;
    for each character c elem Sigma do
        t = epsilonClosure(Delta(q,c));
        T[q,c] = t;
        if t not elem Q then
            add t to Q and to Worklist;
    end;
end;

```

Von Q nach D

- jedes $q_i \in Q$ benötigt einen Zustand $d_i \in D$
- wenn q_i einen akzeptierenden Zustand im NEA enthält, dann ist d_i ein Endzustand des DEA
- σ_D kann direkt aus T konstruiert werden durch die Abbildung von q_i nach d_i
- der Zustand der aus q_0 konstruiert werden kann, ist d_0

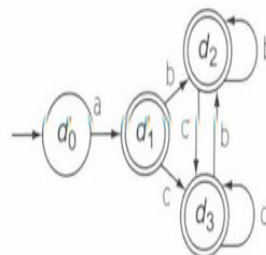
Beispiel



(a) NFA for "a(b | c)*" (With States Renumbered)

Set Name	DFA States	NFA States	ϵ -closure(Delta(q,*))		
			a	b	c
q_0	d_0	n_0	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	- none -	- none -
q_1	d_1	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	- none -	$\{n_5, n_8, n_9\}$	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$
q_2	d_2	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	- none -	q_2	q_3
q_3	d_3	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$	- none -	q_2	q_3

(b) Iterations of the Subset Construction



(a) Resulting DFA

Haskell-Datentyp für DFAs

```
data DFAState = DFAState Integer
```

```
data DFA = DFA
  { -- dfaStates    :: Set DFAState -- wird berechnet
    -- dfaBigSigma :: [Char]      -- wird berechnet
    dfaSigma  :: Set ((DFAState, Char), DFAState)
  , dfaStart  :: DFAState
  , dfaAcceptingStates :: Set DFAState
  }
```

- Wesentlicher Unterschied zum NFA ist die σ -Funktion, die hier keinen ϵ -Übergang zulässt
 - Char statt Maybe Char

Teilmengenkonstruktion in Haskell

[SubsetConstruction.hs @ GitHub](#)

Fixpunkt-Berechnungen

- die Teilmengenkonstruktion ist ein Beispiel einer Berechnung eines Fixpunkts
- diese ist eine Berechnungsart die an vielen Stellen in der Informatik genutzt wird
- eine monotone Funktion wird wiederholt auf ihr Ergebnis angewendet
- die Berechnung terminiert wenn sie einen Zustand erreicht bei dem eine weitere Iteration das selbe Ergebnis liefert
- das ist ein Fixpunkt
- im Compilerbau sind auch häufig Fixpunkt-Berechnung zu finden

Erzeugen eines minimal DFA aus einem beliebigen DFA: Hopcroft's Algorithmus

- der mit der Teilmengenkonstruktion hergeleitete DEA kann eine sehr große Anzahl von Zuständen haben
 - damit benötigt ein Scanner viel Speicher
- Ziel: äquivalente Zustände finden

- Hopcroft's Algorithmus konstruiert eine Partition $P = \{p_1, p_2, \dots, p_m\}$ der DEA-Zustände
- gruppiert die Zustände bzgl. des Verhaltens
 - wenn $d_i \xrightarrow{c} d_x, d_j \xrightarrow{c} d_y$ und $d_i, d_j \in p_s$, dann müssen d_x und d_y in der selben Teilmenge p_t sein
 - d.h. wir splitten bei Zeichen die von einem Zustand in p_s bleiben und beim anderen nicht (nicht kann auch sein, dass es keine Transition für diesen Buchstaben gibt)
- jede Teilmenge $p_s \in P$ muss maximal groß sein

Hopcroft's Algorithmus

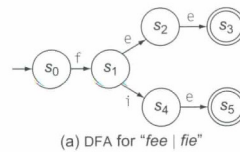
```

T = { D_A, { D - D_A } };
P = { };
while (P != T) do
  P = T;
  T = { };
  for each set p in P do
    T = T `union` Split(p);
  end;
end;

Split(S) {
  for each c in Sigma do
    if c splits S into s1 and s2
      then return {s1,s2};
  end;
  return S;
}

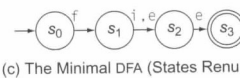
```

Beispiel



Step	Current Partition	Examines		
		Set	Char	Action
0	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	—	—	—
1	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	$\{s_3, s_5\}$	all	none
2	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	$\{s_0, s_1, s_2, s_4\}$	e	split $\{s_2, s_4\}$
3	$\{\{s_3, s_5\}, \{s_0, s_1\}, \{s_2, s_4\}\}$	$\{s_0, s_1\}$	f	split $\{s_1\}$
4	$\{\{s_3, s_5\}, \{s_0\}, \{s_1\}, \{s_2, s_4\}\}$	all	all	none

(b) Critical Steps in Minimizing the DFA



Aus Cooper & Torczon, Engineering a Compiler

Hopcroft's Algorithmus in Haskell

[Hopcroft.hs @ GitHub](#)

Vom DEA zum Recognizer

- aus dem minimalen DEA kann der Code für den Recognizer hergeleitet werden
- der Recognizer muss als Ergebnis liefern
 - die erkannte Zeichenkette
 - die syntaktische Kategorie
- um Wortgrenzen zu erkennen, können wir Trennzeichen, z.B. Leerzeichen, zwischen die Wörter schreiben
- das bedeutet aber, wir müssten $2 + 5$ statt $2+5$ schreiben

Eine andere Art zu erkennen

- der Recognizer muss das längste Wort finden, dass zu einem der regulären Ausdrücke passt
- er muss solange weiter machen bis er einen Zustand s erreicht von dem es keinen Übergang mit dem folgenden Zeichen gibt

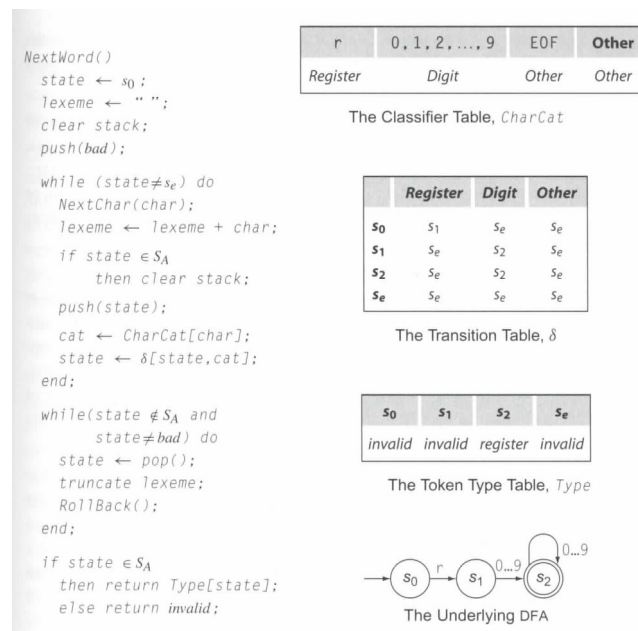
- wenn s ein Endzustand ist, gibt der Scanner das Wort und die syntaktische Kategorie zurück
- sonst muss er den letzten Endzustand finden (backtracking)
- wenn es keinen gibt \Rightarrow Fehlermeldung
- es kann im ursprünglichen NEA mehrere Zustände geben, die passen
 - z.B. ist `new` ein Schlüsselwort aber auch ein Bezeichner
- der Scanner muss entscheiden können welche Kategorie er vorzieht

Implementierung von Scannern

Table-Driven Scanner

- nutzt das Gerüst eines Scanners zur Steuerung und
- eine Menge von generierten Tabellen die das sprachspezifische Wissen enthalten
- der Compilerbauer muss eine Menge von lexikalischen Mustern (REs) zur Verfügung stellen
- der Scanner-Generator erzeugt die Tabellen

Beispiel



Aus Cooper & Torczon, Engineering a Compiler

Exzessiven Rollback vermeiden

- gegeben sei der RE $ab|(ab)^*c$
 - für $ababababc$ gibt der Scanner die gesamte Zeichenkette als einzelnes Wort zurück
 - für $abababab$ muss der Scanner alle Zeichen lesen bevor er entscheiden kann, dass der längste Präfix ab ist
 - als nächstes liest er $ababab$ und erkennt ab
 - ...
- ⇒ im schlechtesten Fall: quadratische Laufzeit
- der *Maximal Munch Scanner* (*munch* heisst mampfen) vermeidet so ein Verhalten durch drei Eigenschaften
 1. ein globaler Zähler für die Position im Eingabe-Zeichenstrom
 2. ein Bit-Array um sich Übergänge in "Sackgassen" zu merken
 3. eine Initialisierungsroutine die vor jedem neuen Wort aufgerufen wird
 - er merkt sich spezifische Paare (Zustand, Position im Eingabestrom) die nicht zu einem akzeptierenden Zustand führen können

Direct-Coded Scanners

- Um die Performanz eines Table-Driven Scanners zu verbessern, müssen wir die Kosten reduzieren vom
 - Lesen des nächsten Zeichens
 - Berechnen des nächsten Zustandübergangs
- Direct-Coded Scanners reduzieren die Kosten der Berechnung des nächsten Zustandübergangs durch
 - ersetzen der expliziten Repräsentation durch eine implizite
 - und dadurch Vereinfachung des zweistufigen Tabellenzugriffs

Overhead des Tabellen-Lookups

- der Table-Driven Scanner macht zwei Tabellen-Lookups, einer in `CharCat` und einer in σ
- um das i . Element von `CharCat` zu bekommen, muss die Adresse $@\text{CharCat}_0 + i \times w$ berechnet werden
 - $@\text{CharCat}_0$ ist eine Konstante die die Startadresse von `CharCat` im Speicher bezeichnet

- w ist die Anzahl Bytes von jedem Element in CharCat
- für $\sigma(state, cat)$ ist es $@\sigma_0 + (state \times \text{numberofcolumnsin } \sigma + cat) \times w$

Ersatz für die while-Schleife des Table-Driven Scanners

- ein Direct-Coded Scanner hat für jeden Zustand ein eigenes spezialisiertes Codefragment
- er übergibt die Kontrolle direkt von Zustands-Codefragment zu Zustands-Codefragment
- der Scanner-Generator kann diesen Code direkt erzeugen
- der Code widerspricht einigen Grundsätzen der strukturierten Programmierung
- aber nachdem der Code generiert wird, besteht keine Notwendigkeit ihn zu lesen oder gar zu debuggen

Beispiel

erkennt $r[0\dots9]^+$

```

sinit: lexeme ← "";
       clear stack;
       push(bad);
       goto s0;

s0: NextChar(char);
    lexeme ← lexeme + char;
    if state ∈ S_A
        then clear stack;
    push(state);
    if (char = 'r')
        then goto s1;
    else goto sout;

s1: NextChar(char);
    lexeme ← lexeme + char;
    if state ∈ S_A
        then clear stack;
    push(state);
    if ('0' ≤ char ≤ '9')
        then goto s2;
    else goto sout;

s2: NextChar(char);
    lexeme ← lexeme + char;
    if state ∈ S_A
        then clear stack;
    push(state);
    if ('0' ≤ char ≤ '9')
        then goto s2;
    else goto sout;

sout: while (state ∉ S_A and
           state ≠ bad) do
        state ← pop();
        truncate lexeme;
        RollBack();
    end;
    if state ∈ S_A
        then return Type[state];
    else return invalid;

```

Aus Cooper & Torczon, Engineering a Compiler

Hand-coded Scanner

- generierte Scanner benötigen eine kurze, konstante Zeitspanne pro Zeichen
- viele Compiler (kommerzielle und Open Source) benutzen handgeschriebene Scanner

- z.B. wurde *flex* entwickelt um das *gcc* Projekt zu unterstützen
- aber *gcc 4.0* nutzt handgeschriebene Scanner in mehreren Frontends
- handgeschriebene Scanner können den Overhead der Schnittstellen zwischen Scanner und dem Rest des Systems reduzieren
- eine umsichtige Implementierung kann die Mechanismen verbessern, die
 - Zeichen lesen und
 - Zeichen manipulieren

außerdem die Operationen

- die benötigt werden um eine Kopie des aktuellen Lexem als Output zu erzeugen