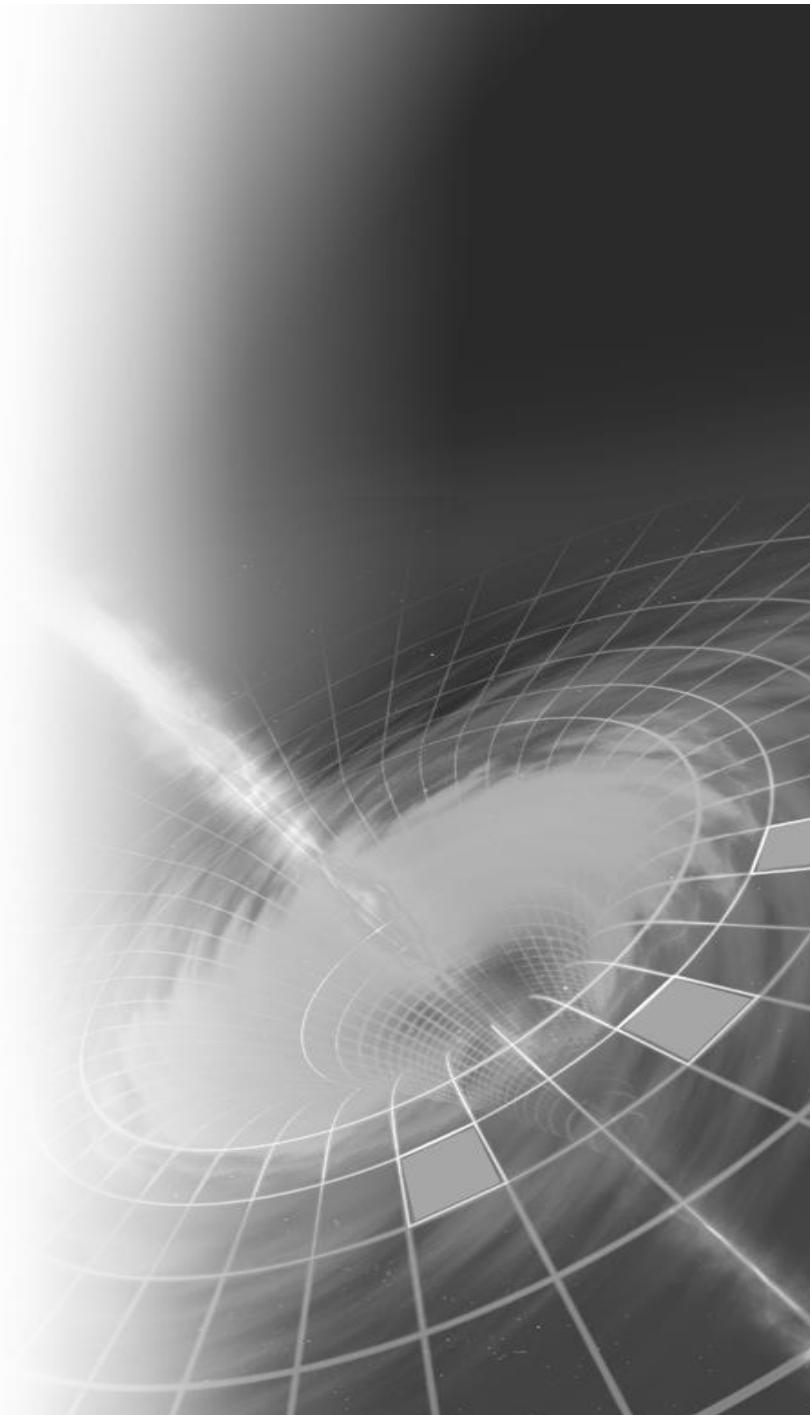
Bachelor-Programm

Compilerbau

im SoSe 2014

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dr. Andreas Kunert
Dipl.-Inf. Ingmar Eveslage

fischer@informatik.hu-berlin.de



Position

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- **Kapitel 3**
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parsergeneratoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Ausblick: Laufzeitsystem,
  Codegenerierung

- Aufgaben eines Scanners als Compiler-Komponente
- Reguläre Ausdrücke
- **Endliche Automaten zur Erkennung regulärer Ausdrücke**

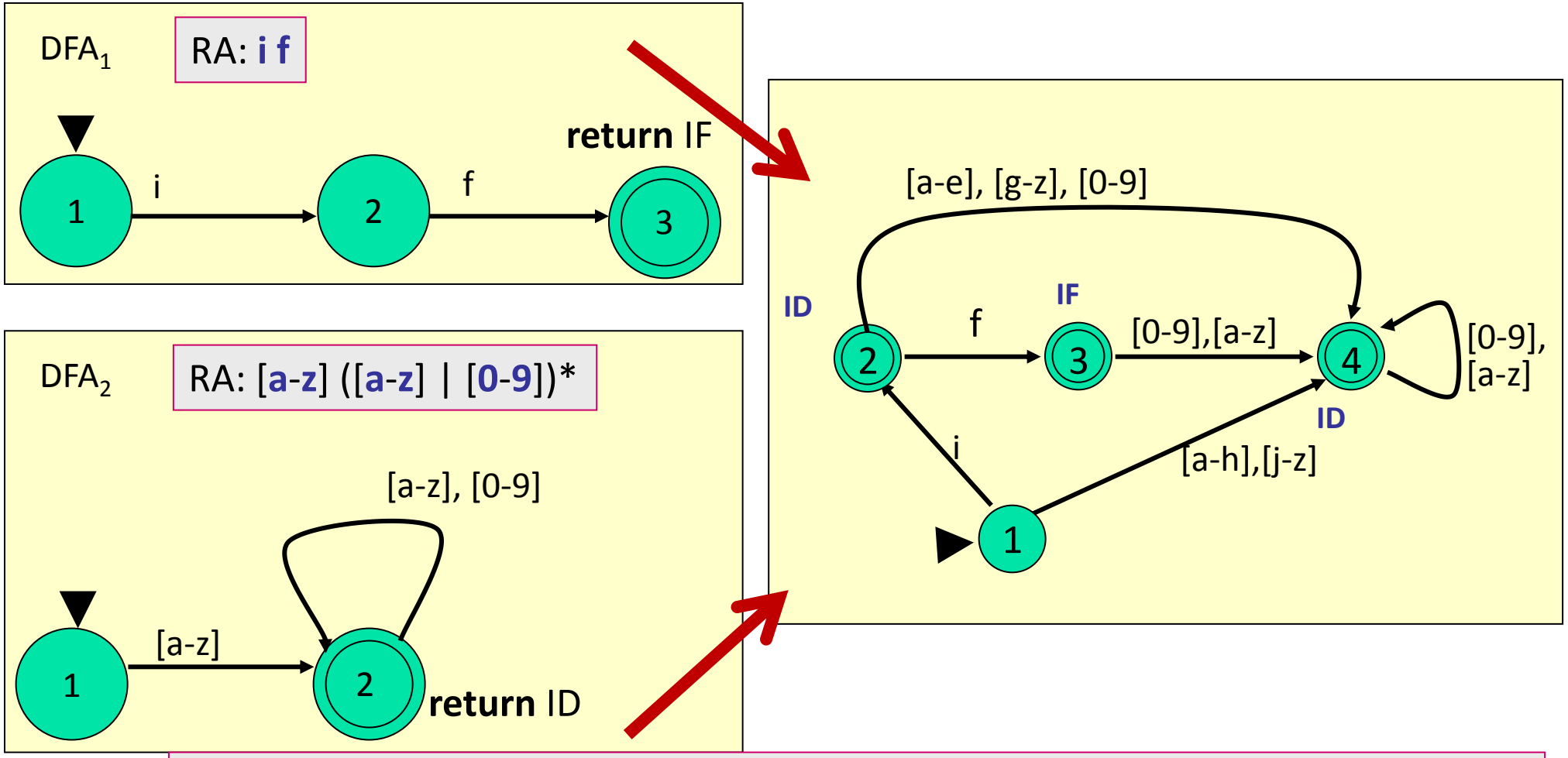
Kompositorischer DFA: Beispiel

		Token-Klassen
RA: Schlüsselwort (if)	→ DFA_1	IF
RA: Bezeichner	→ DFA_2	ID
RA: positive ganze Zahl	→ DFA_3	NUM
RA: positive reele Zahl	→ DFA_4	REAL
RA: White Space	→ DFA_5	
RA : Zeilenkommentar	→ DFA_6	COMMENT
Fehlererkennung	→ DFA_7	

Komposition als komplexer Automat

DFA soll ZK einlesen und erkennen,
ob es sich um eines der obigen Token-Klassen handelt

Erkennung lexikalischer Token: IF, ID

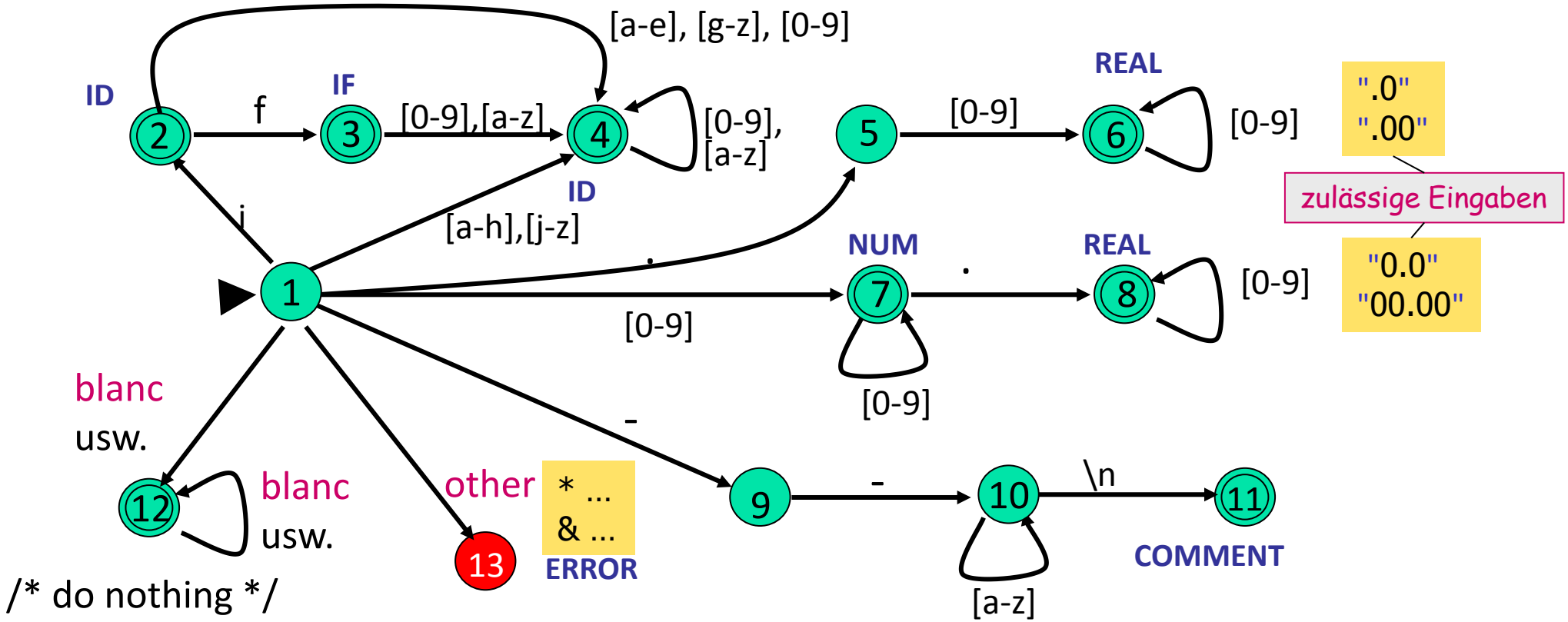


Kombination gelingt aber nur mit Intuition -

also: noch kein strukturierter Ansatz

Kombination von Einzelautomaten (Ad-hoc-Lösung)

… ergibt den folgenden Automaten



Problem: Automat ist noch **unvollständig**.

a) nur Zustand **1** ist bislang für alle möglichen Eingaben vorbereitet

b) returniert nur bei kompletter Eingabe

Zusammenhang von Scanner und DFA

- Scanner benutzt einen tabellengesteuerten DFA

- Scanner benötigt zur Erkennung eines RA ein **Abschlusszeichen**,
  das **nicht** zum RA des DFA-Eingabewortes gehört

- das Abschlusszeichen könnte aber,
  wenn es **weder** Trennzeichen
  noch Kommentarbestandteil **oder** EOF ist,
  das erste Zeichen des Folge-Tokens sein

  und muss dann in den Eingabepuffer zurückgeschrieben werden,
  wie bereits beim Ad-hoc-Scanner (C-Programm)

Vervollständigung partieller Übergangsfunktionen (1)

… erfolgt in Abhängigkeit davon, ob

a) Automat nur komplette Eingaben oder auch

b) Anfangsteilworte akzeptieren

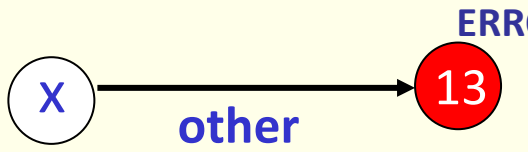
soll

Fall a) Eingabewort ist komplett zu lesen

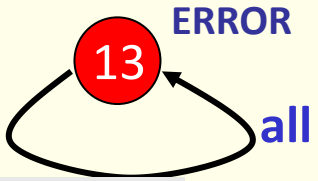
Beispiel: DFA als Resultat der bisherigen Komposition

Ergänzungsschritte:

1. jeder Zustand x, außer 1 (Start) und 13(Fehler) erhält Transition(en) für **other**-Eingaben nach Zustand 13:



2. Zustand 13 erhält Transition für **all**-Eingaben nach Zustand 13:



other= alle Eingabezeichen, die in x noch nicht behandelt wurden

all= alle mögliche Eingabezeichen

Typische praktische Analyseprobleme

Fall b) Eingabewort ist eine Token-Folge. Das jeweils nächstes Anfangsteilwort ist zu akzeptieren.

Fragen

- ist "if8" ein Bezeichner oder zwei Token if und 8?
- beginnt "if 89" mit einem Schlüsselwort oder einem Bezeichner?

Regeln (eingesetzt auch beim realen Scanner-Bau):

(1) das jeweils nächste Token wird stets als **maximal** mögliche gültige Zeichenkette bestimmt

(2) die Bestimmung der Token-Klasse erfolgt entsprechend einer **priorisierten** Liste
(d.h. Reihenfolge der RA-Akzeptanz ist signifikant)

Antworten

- "if8" wird als **Bezeichner** erkannt: nach 1.Regel
- "if 89" wird als Folge von **Schlüsselwort** und **Zahl** erkannt: nach 2.Regel

Vervollständigung partieller Übergangsfunktionen (2)

Fall b) Eingabewort ist eine Token-Folge. Das jeweils nächstes Anfangsteilwort ist zu akzeptieren.

Vorbereitungsschritt: Markierung von Endzuständen mit folg. Semantik

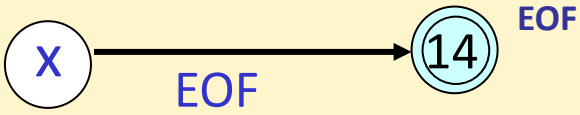
Return <Tokenklasse>

falls letztes Eingabezeichen
- **kein** Zeichen für definierten Übergang

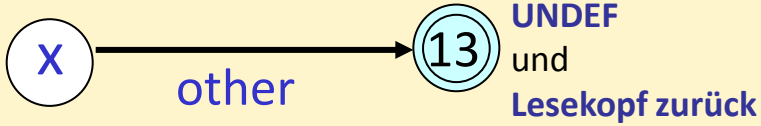
dann
- **accept** und
- Rücksetzen des Lesekopfes um eine Position

Vervollständigungsschritt

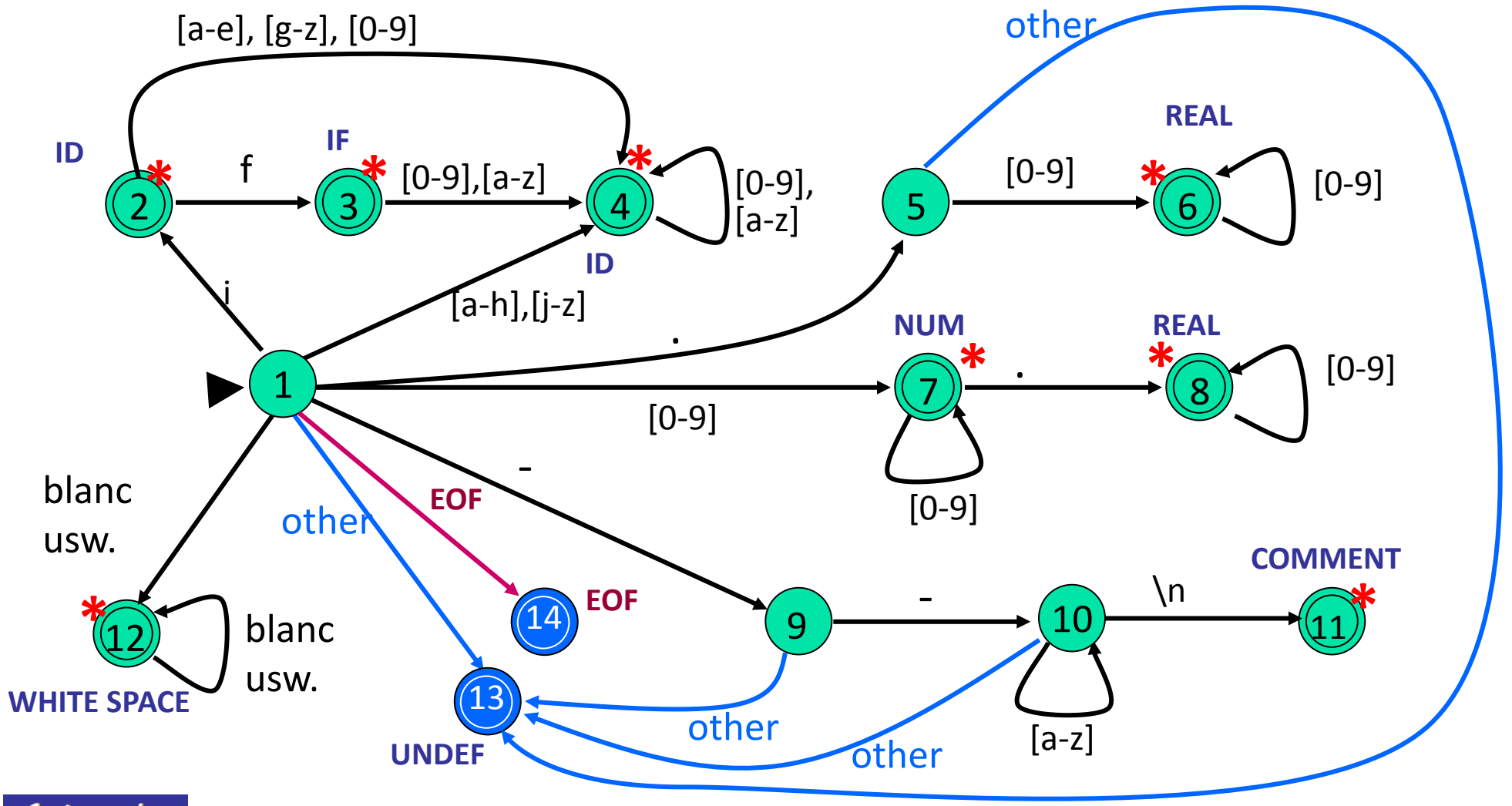
1. ERROR-Zustand 13 gibt **UNDEF** zurück (Parser entscheidet über Fehler)
2. neuer Zustand 14 mit **EOF**-Rückgabe aus jedem Zustand erreichbar



3. jeder Zustand x, außer 1 (Start) und Finalzustände erhält zusätzl. Transition(en)



Finales Ergebnis der Vervollständigung



Position

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- **Kapitel 3**
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parsergeneratoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Ausblick: Laufzeitsystem,
  Codegenerierung

- Aufgaben eines Scanners als Compiler-Komponente
- Reguläre Ausdrücke
- Endliche Automaten zur Erkennung regulärer Ausdrücke
- **Tabellengesteuerter Scanner und Symboltabelle**

Methodisches Ansatz zur Scanner-Code-Generierung

Frage

können evtl. Code-Komponenten des Scanners erzeugt werden,
die unabhängig von der zu erkennenden Sprache sind?

Antwort ja

Konstruktion der Erkennungsfunktion (Übergangsfunktion) des DFA

spezieller Scanner erfordert dann „nur“ noch
spezielle Scanner-Tabellen !!!

Programmsteuerung benötigt lediglich zwei Eingabetabellen:

(1) char_class: Zeichen → Zeichenklasse

(2) next_state: Zustand × Zeichenklasse → Zustand

am Beispiel einer **Bezeichnererkennung** (bei Rücksprache mit der Symboltabelle)

Symboltabelle: Motivation

Tabelle wird benötigt

- zur Verwaltung von Informationen über verschiedene Konstrukte der Quellsprache
  - reservierte Bezeichner, Bezeichner des Programms, Konstanten, …
- zur Unterstützung der lexikalischen Analyse
  - Erzeugung und Ablage des Token-Attributes eines Bezeichners in der Symboltabelle
- für spätere Compilerphasen (zugreifbar für Parser und Semantikanalysator)
  - Beherrschung unterschiedlicher Verwendungen von Bezeichnern: z.B. als Namen einer
    - Funktion,
    - Variablen,
    - Sprungmarke,
    - Speicheradresse, …

… und steht global bereit

Symboltabelle: Operationales Interface

Abstrakte Datenstruktur: Symboltabelle

Operationen zur Speicherung und Aufsuchen von Bezeichner-Attributen

```
int insert (char* s, int t)
    /* gibt den Index eines neuen Eintrages für die
       Zeichenkette s und den Tokentyp t zurück  */

int lookup (char* s)
    /* gibt den Index des gesuchten Eintrages für die Zeichenfolge s zurück,
       oder 0, falls s nicht gefunden wurde */
```

Symboltabelle: Behandlung reservierter Bezeichner

z.B. **begin**, **mod**, ...

```
insert ("begin", BEGIN);
insert ("mod", MOD);
...
```

} Initialisierung der Symboltabelle

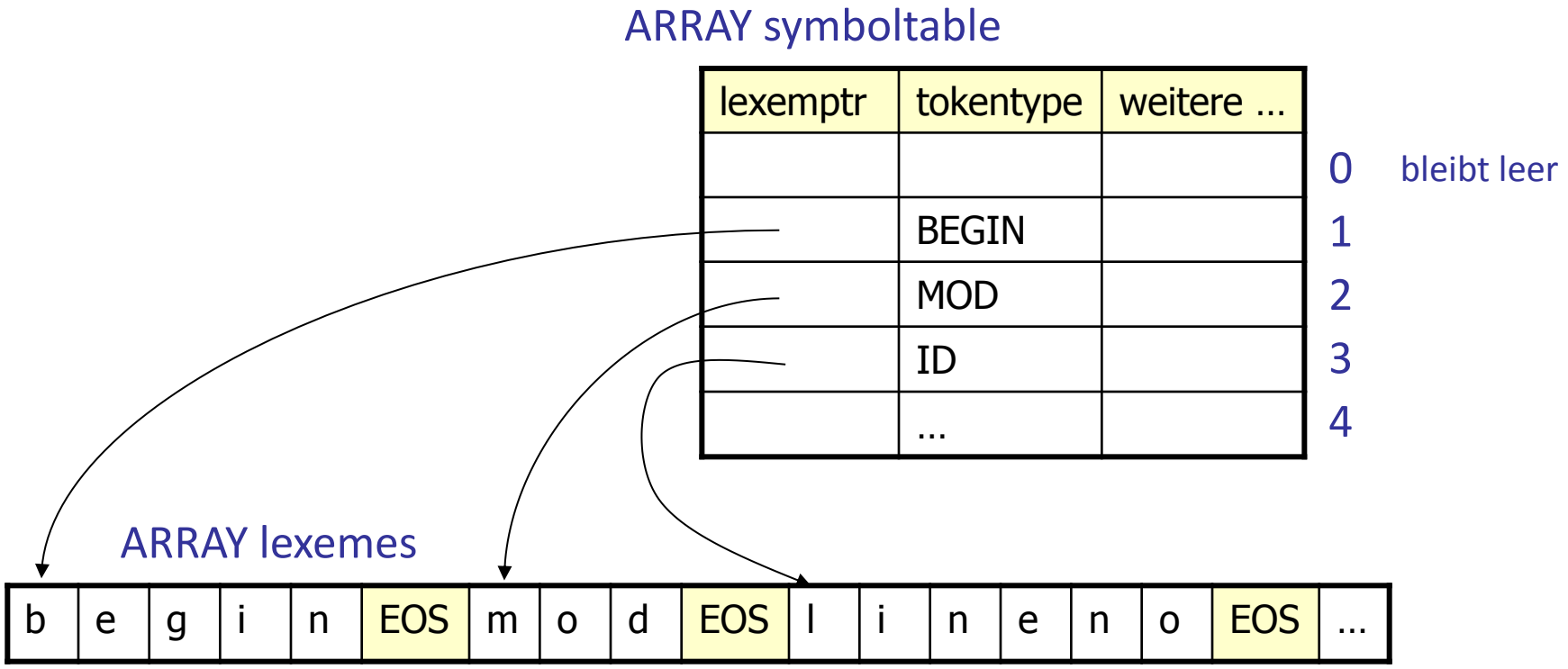
mit jedem nachfolgenden Aufruf von

lookup("mod")

wird als vereinbarter zugeordneter Tokentyp MOD geliefert,
so dass mod nicht mehr als Bezeichner verwendet werden kann

Symboltabelle: mögliche Implementierung

- **Bezeichnerwerte** (Zeichenketten auch als Lexeme bezeichnet) sollten dynamisch ausgefasste Speicherbereiche zugewiesen werden



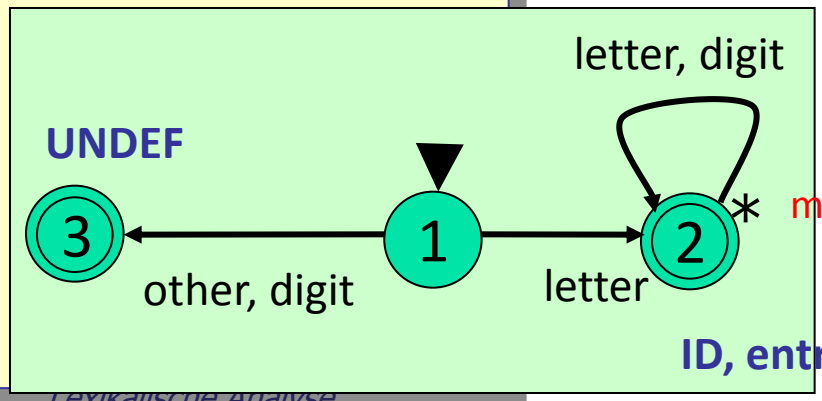
Tabellengesteuerter Scanner zur Bezeichner-erkennung bei Symboltabellenansteuerung

```
current= next_char();
state= 1;            /* code for state 1 */
done= false;
token_type= NONE;
token_value= NONE ;
lexbuf := "";        /* empty string, array[0..100] of char; /*

while (not done) {
      class= char_class [current] ;

      state= next_state [state, class] ;
      switch (state)  {
            case 2:   /* building an id */
                        lexbuf= concatenate (lexbuf, current);
                        current= next_char();
                        break;
            case -:   /* accept state */
                        token_type= ID;
                        index= lookup (lexbuf);
                        if (index=0) then index= insert (lexbuf, id);
                                    else ...;
                        token_value= index;
                        get_back(current);
                        done= true;
                        break;
            case 3:   /* error */
                        token_type= UNDEF;
                        done= true;
                        break;
            }
}
```

Sprachabhängige Steuerungstabellen des DFA

	a-z	0-9	other
class	letter	digit	other

	Eingabezeichenklasse			Aktionen
	letter	digit	other	
1	2	3	3	-
2	2	2	-	return: ID, entry
3	-	-	-	return: UNDEF



Zwischenfazit

noch offen: Suche nach einem regelbasierten Ansatz zur Konstruktion des DFA's

- Lexikalische Einheiten der Quellsprache → Token
  (strukturelle Beschreibung als regulärer Ausdruck)

- Regulärer Ausdruck → endlicher Zustandsautomat mit
  partieller Zustandsübergangsfunktion
  → Komplettierung zum Automaten mit totaler
  Zustandsübergangsfunktion
  (Diskussionen: Endebehandlung, maximale Token-Erkennung,
  Fehlerbehandlung)

- noch manuell umgesetzte Kombination von endlichen Zustandsautomaten (DFAs)

- Tabellengesteuerter Scanner (mit Interface zur Symboltabelle)

- Ableitung sprachabhängiger Tabellen aus dem Automaten zur RA-Erkennung
  1. Zeichenklassen
  2. Zustandsübergangstabellen

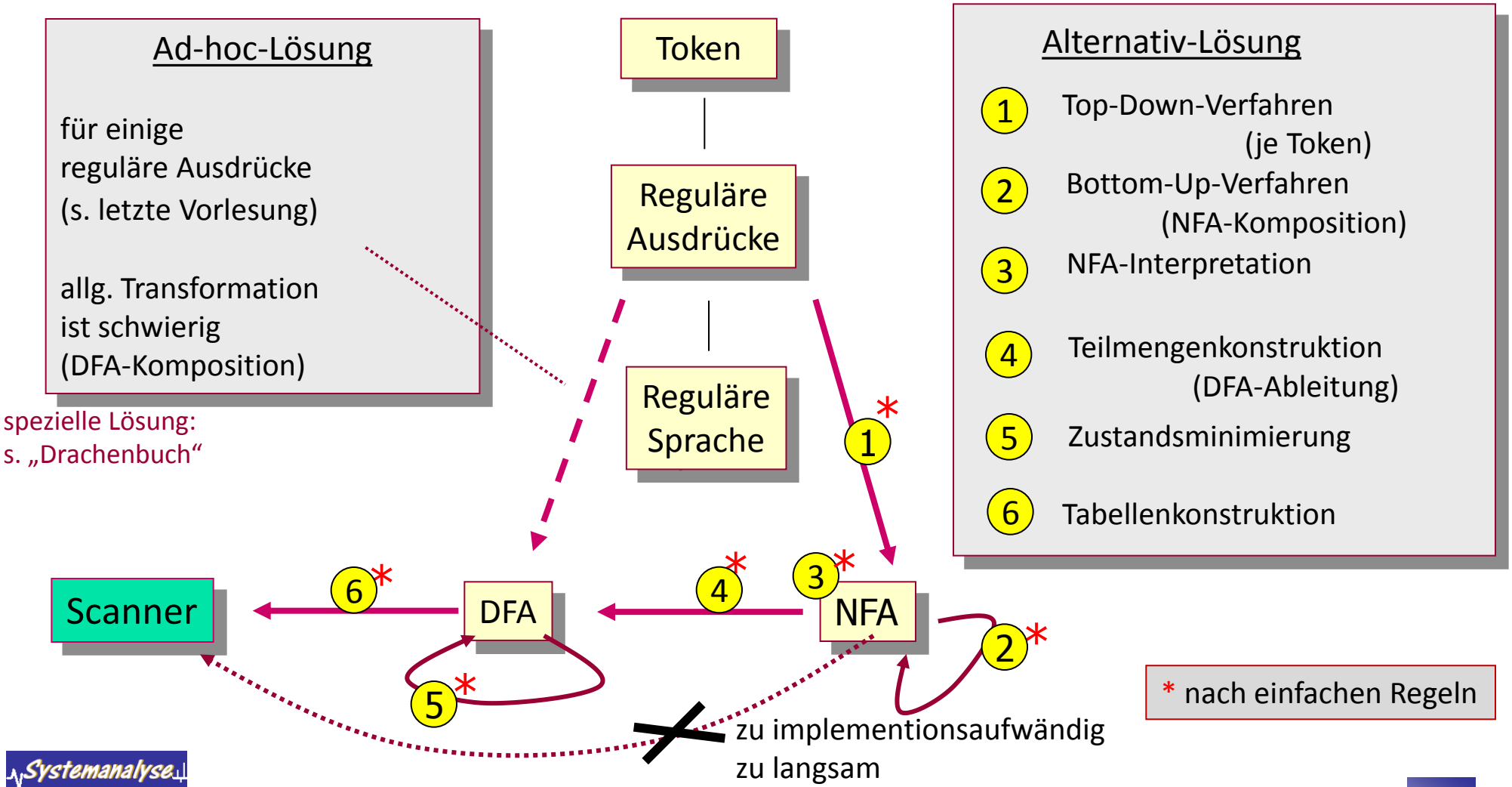
  } Tabellen steuern den Scanner-Automaten

Position

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- **Kapitel 3**
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parsergeneratoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Ausblick: Laufzeitsystem,
  Codegenerierung

- Aufgaben eines Scanners als Compiler-Komponente
- Reguläre Ausdrücke
- Endliche Automaten zur Erkennung regulärer Ausdrücke
- Tabellengesteuerter Scanner und Symboltabelle
- **Konzepte für einen generischen Scanner**
- Scanner-Architektur
- Scanner-Architektur

Konzepte für einen generischen Scanner



Nichtdeterministischer endlicher Automat

Def: nichtdeterministischer endlicher Automat

NFA= (**S**, Σ, δ, s_0, **F**)

- endliche Menge **S** von Zuständen: **S** = $\{s_0,\ldots,s_n\}$
- ein ausgezeichneter Startzustand $s_0 \in$ **S**
- Menge von Eingabesymbolen: Alphabet Σ mit $\Sigma \cap$ **S** = $\emptyset$
- Zustandsübergangsfunktion (Transition) δ **: S** x $(\Sigma \cup \{\varepsilon\}) \rightarrow \wp$**(S)**,
  die einem Zustand Folgezustände spontan oder in Abhängigkeit des aktuellen Eingabesymbols zuordnet
  - (Zustand, Symbol)- Paare werden in Zustandsmengen überführt
  - Übergänge der Art: $(s_i, \varepsilon) \rightarrow \wp$ (**S**) sind **spontane** Übergänge, d.h. Übergänge ohne Konsumtion von Eingabezeichen
- eine Menge von akzeptierenden Finalzuständen **F** $\subseteq$ **S**

$\wp(X)$ ist die Potenzmenge von X

Zwischenfazit: Vergleich von DFA und NFA

Gemeinsamkeiten

- endliche Automaten (egal DFA oder NFA) erkennen,
  ob eine Eingabezeichenkette ein Wort der Sprache ist
- sie erkennen insbesondere Sprachen, die sich mit regulären Ausdrücken beschreiben lassen

Unterschiede

- ein DFA ist ein spezieller NFA
- Zeit- und Platzbedarf
  DFA schneller mit mehr Zuständen als NFA,
  NFA langsamer mit weniger Zuständen als DFA

NFA- Scanner

Aufgaben eines Scanners als NFA

a) Zerlegung eines Eingabewortes in Teilworte (Token) der Sprache

b) jedes erkannte Teilwort bringt ihn
von seinem Anfangszustand in einen der möglichen Endzustände

c) es besteht erneut das Problem, das Ende eines Teilwortes zu erkennen
(max. Ausprägung eines Teilwortes)

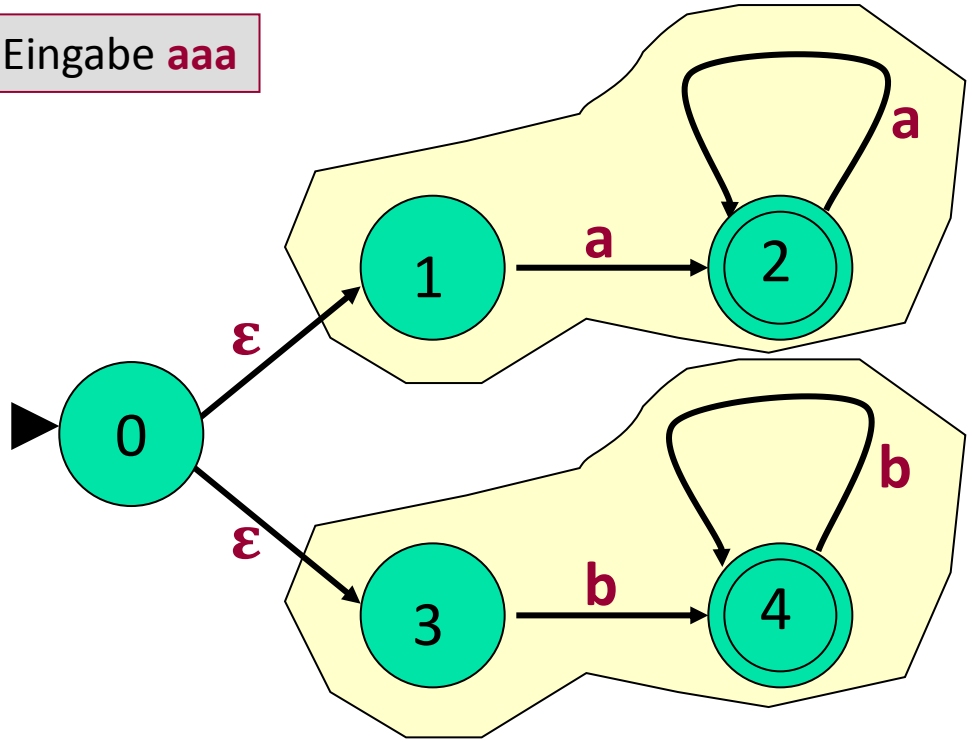
d) EOF-Erkennung kann wie gehabt erfolgen

e) übliche Startkonvention
   - der NFA wird in seinem Anfangszustand gestartet
   - der Lesekopf befindet sich dabei auf dem ersten noch nicht konsumierten Zeichen

1.NFA-Beispiel: aa*|bb*

Fazit: NFA zeigt Verhalten entspr. Trace-1

Eingabe **aaa**



Trace-1

$0 \xrightarrow{\varepsilon} 1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{a} 2$

ε's „verschwinden" im Trace

aaa wird akzeptiert

Trace-2

$0 \xrightarrow{\varepsilon} 3$ würde blockieren und nicht zum Ziel führen

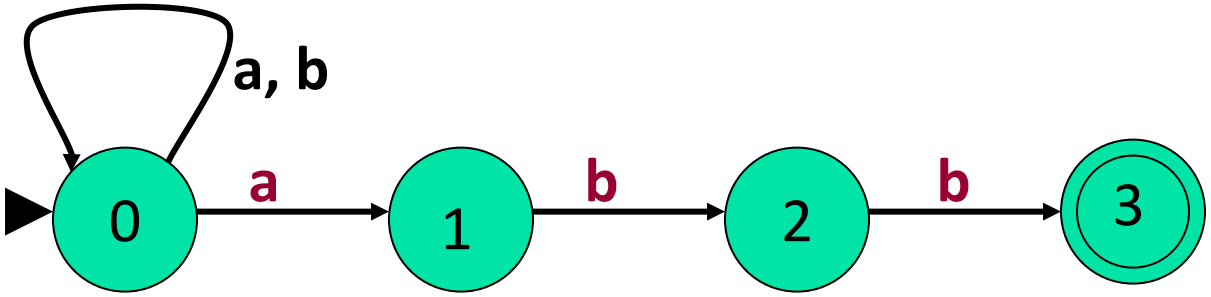
Arbeitsweise eines NFA

- NFA (Theorie) wählt immer die richtigen Ableitungsalternativen - und zwar die, die zum Ziel führen
- NFA (Praxis) arbeitet mit Backtracking-Strategie:
  alle Varianten werden ausgeführt, bewertet und danach wird die Auswahl getroffen

NFA-Implementierungsproblem

- Welcher NFA erkennt den RA (a|b)*abb ?

aabb



0 → 0 → 1 → 2 → 3
a a b b

0 → 0 → 0 → 0 → 0
a a b b

i.allg. viele Ausführungen möglich,
aber nur einer führt zur Akzeptanz

→ NFA-Implementierungsproblem

NFA-Beispiel: Erkennung von Integer- und Real-Konstanten

[0-9]$^+$ | [0-9]$^+$. [0-9]$^+$ (ε | E [0-9] [0-9])

Beispiele akzeptierter Wörter

0123

123.456

123.4E56

führende Nullen sind erlaubt

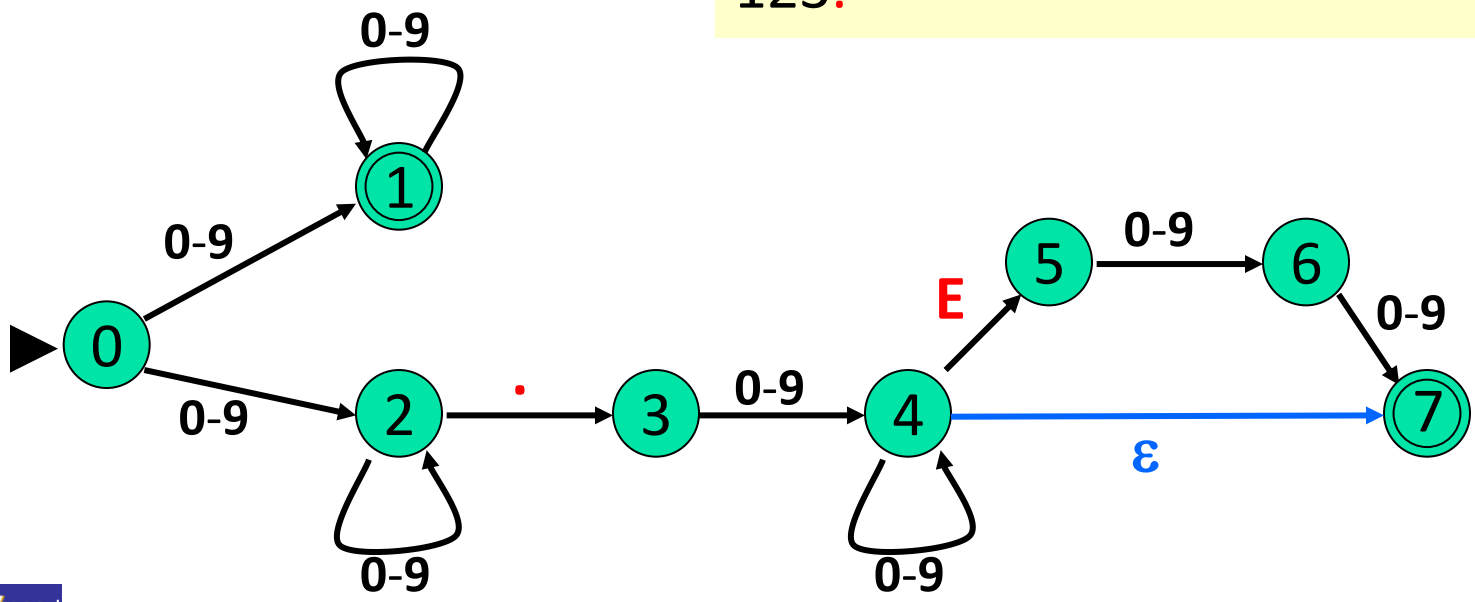
Beispiele nicht akzeptierter Wörter

.0123

123E12

E12

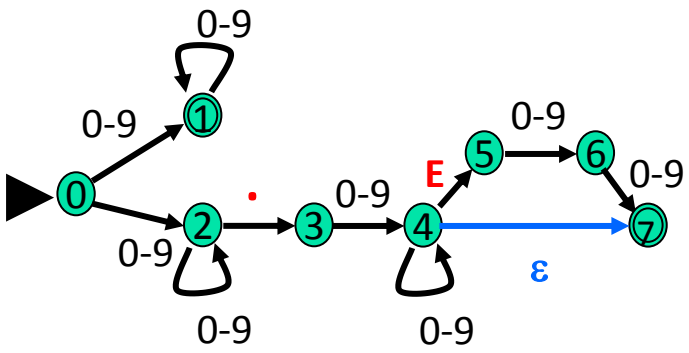
123.



NFA: Zustandsübergangstabelle

Σ= {**0**, **1**,..., **9**, **.**, **E**}

s_0= 0
F = {1, 7}

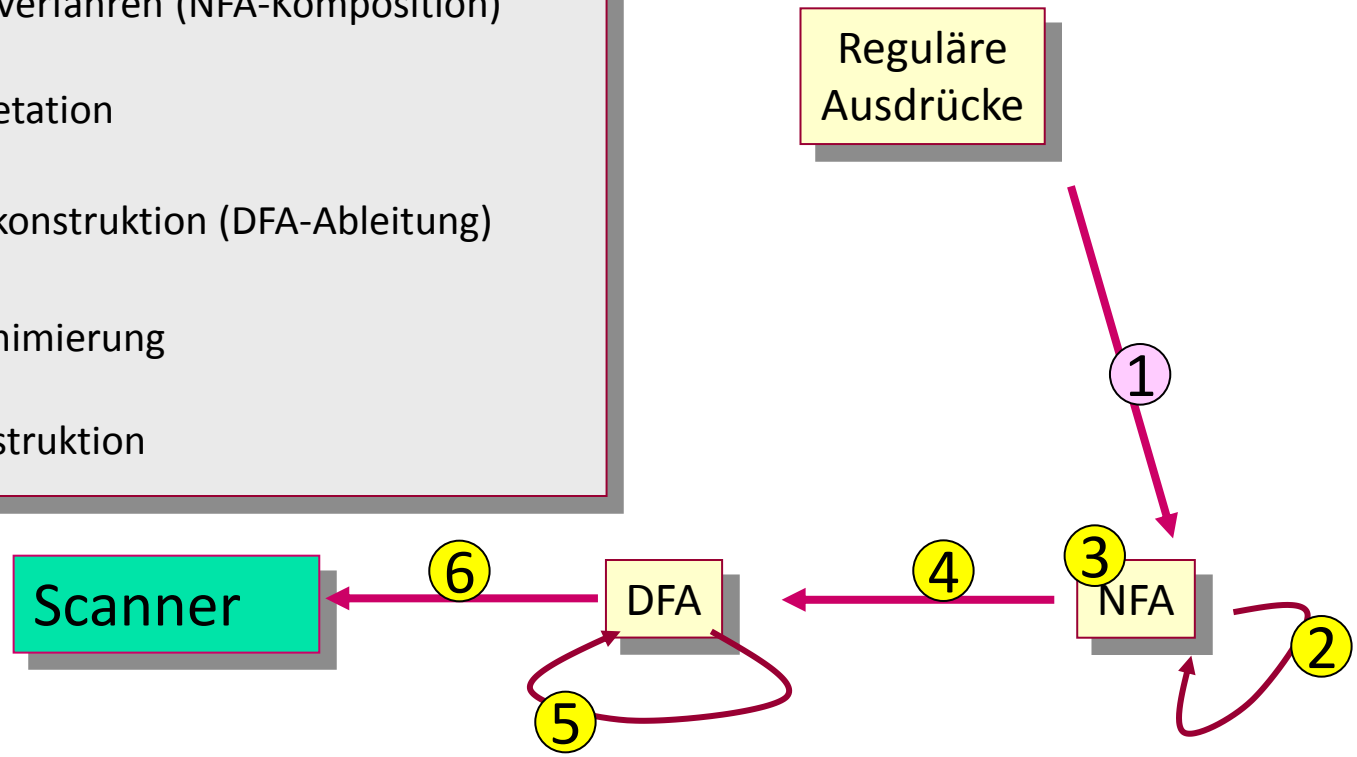


	Eingabezeichen			
	{0,1,...,9}	**.**	**E**	**ε**
0	{1, 2}	Ø	Ø	Ø
1	{1}	Ø	Ø	Ø
2	{2}	{3}	Ø	Ø
3	{4}	Ø	Ø	Ø
4	{4}	Ø	{5}	{7}
5	{6}	Ø	Ø	Ø
6	{7}	Ø	Ø	Ø
7	Ø	Ø	Ø	Ø

Problem 1: vom RA zum NFA

vom RA zum Scanner

1. Top-Down-Verfahren (je Token)
2. Bottom-Up-Verfahren (NFA-Komposition)
3. NFA-Interpretation
4. Teilmengenkonstruktion (DFA-Ableitung)
5. Zustandsminimierung
6. Tabellenkonstruktion



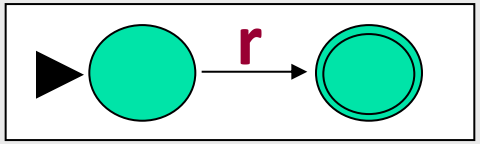
Zusammenhang von RA und NFA

Satz: akzeptierte Sprache eines endlichen Automaten (DFA, NFA)

Zu jedem regulären Ausdruck **r** über einem Alphabet **Σ** gibt es einen NFA,
der die von **r** beschriebene Menge von Worten (reguläre Sprache) akzeptiert

Top-Down-Konstruktionsalgorithmus (allgemeines Schema)

a) Startkonfiguration

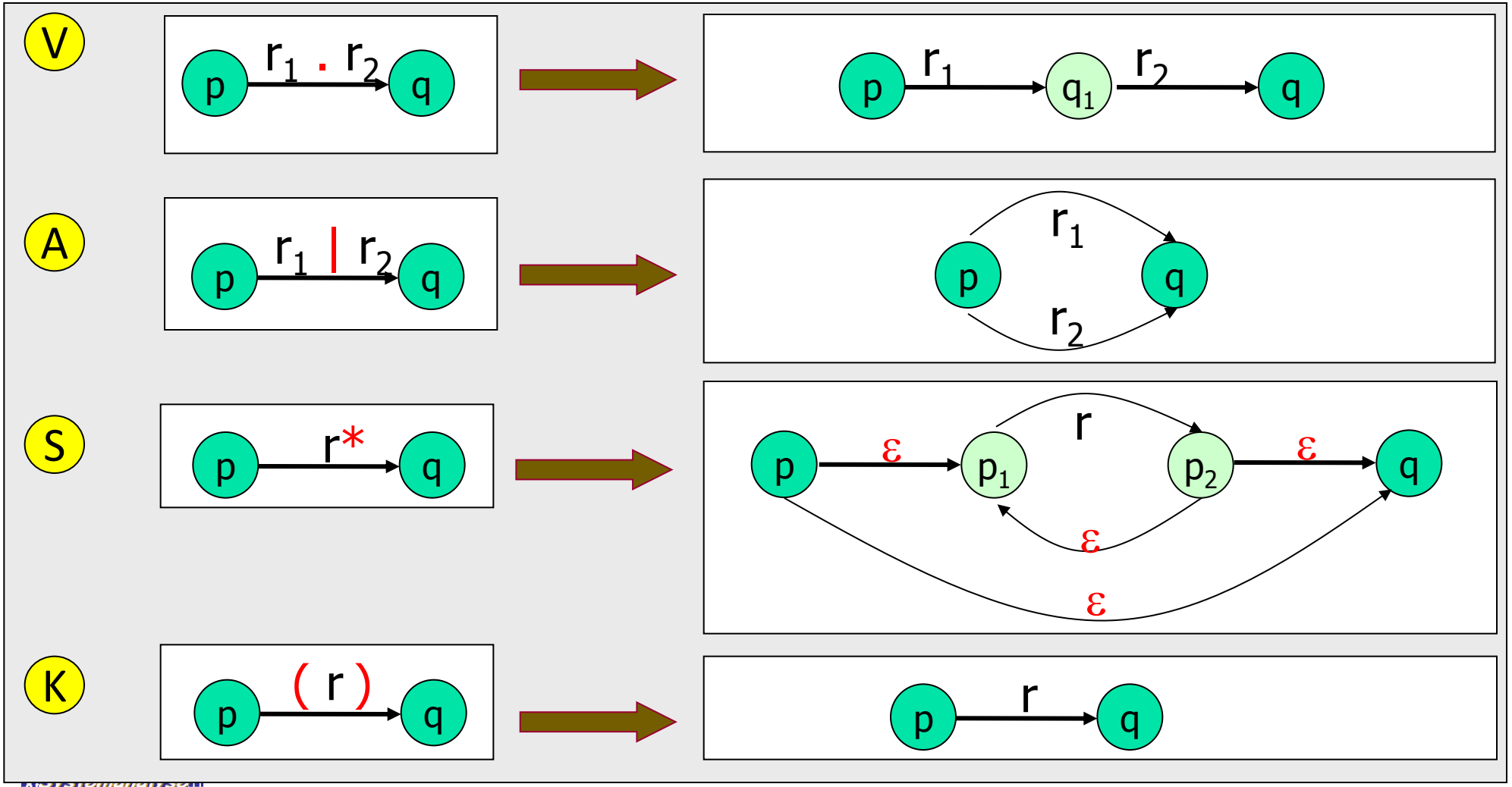


b) syntaktische Zerlegung von **r**, Verfeinerung des Zustandsübergangsdiagramms
Regeln

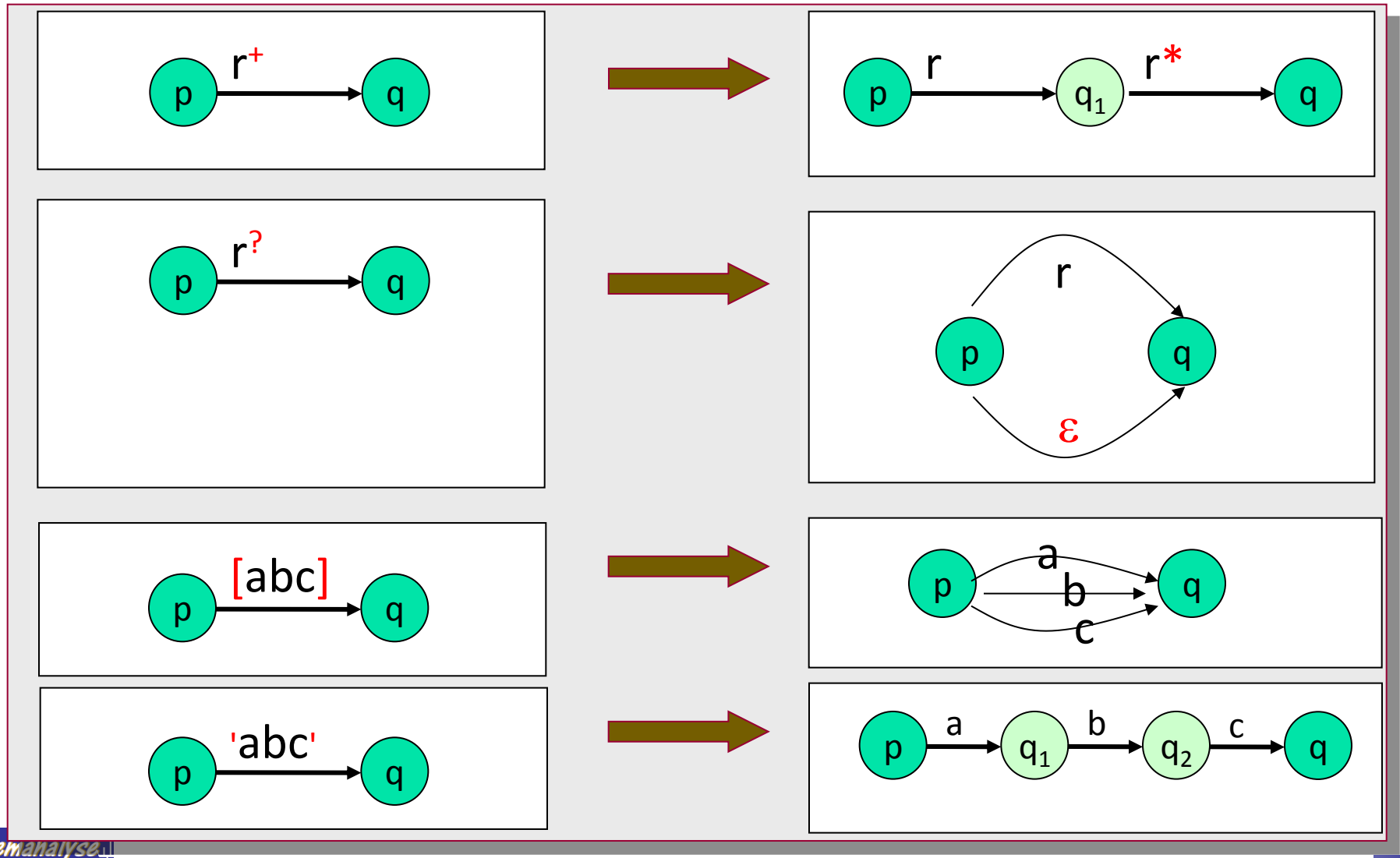
c) Fortsetzung der Zerlegung bis nur noch Kanten übrig sind,
die mit Zeichen aus **Σ** oder **ε** markiert sind — Abbruchkriterium

Regeln: RA → NFA (1/2)

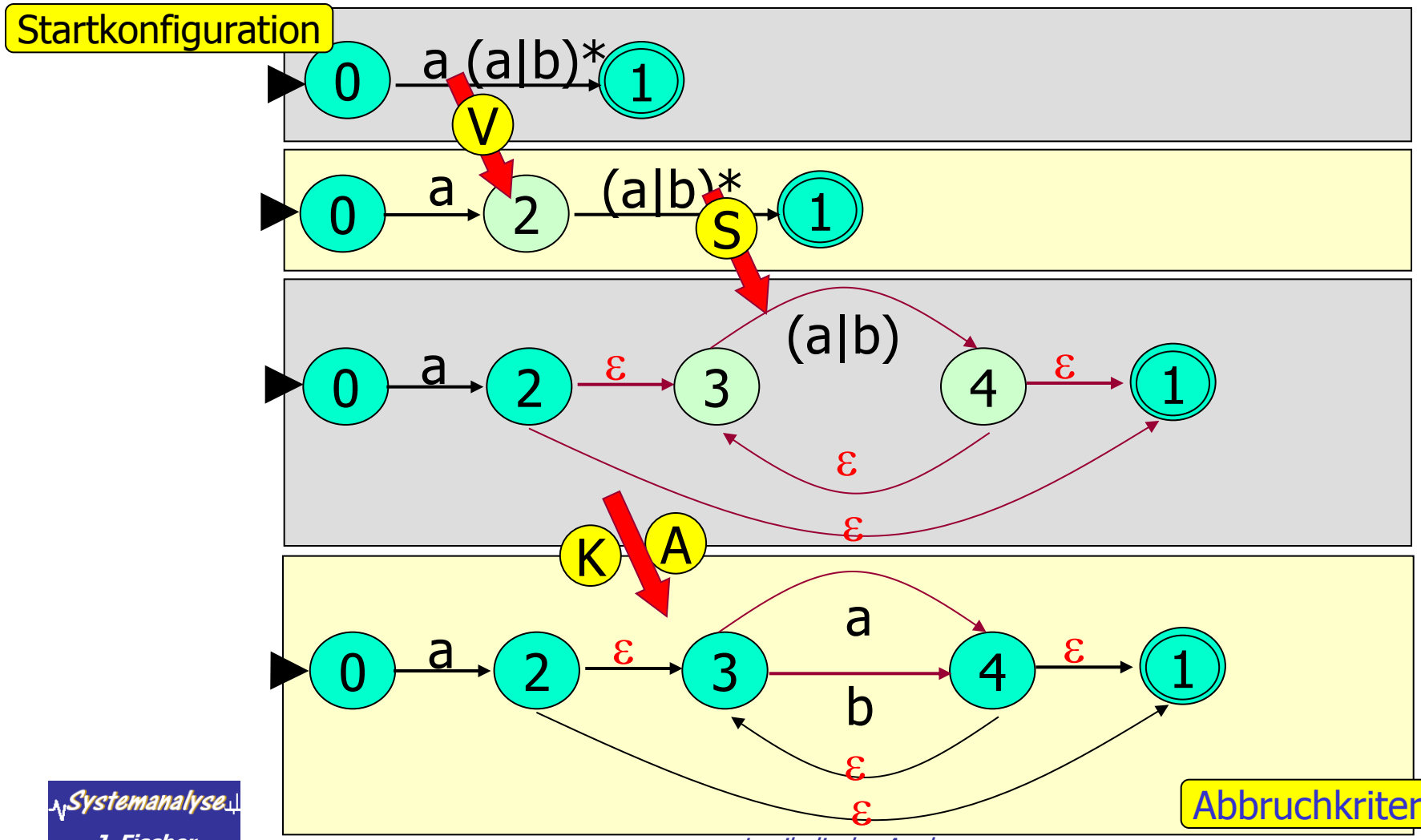
Regel-Kürzel



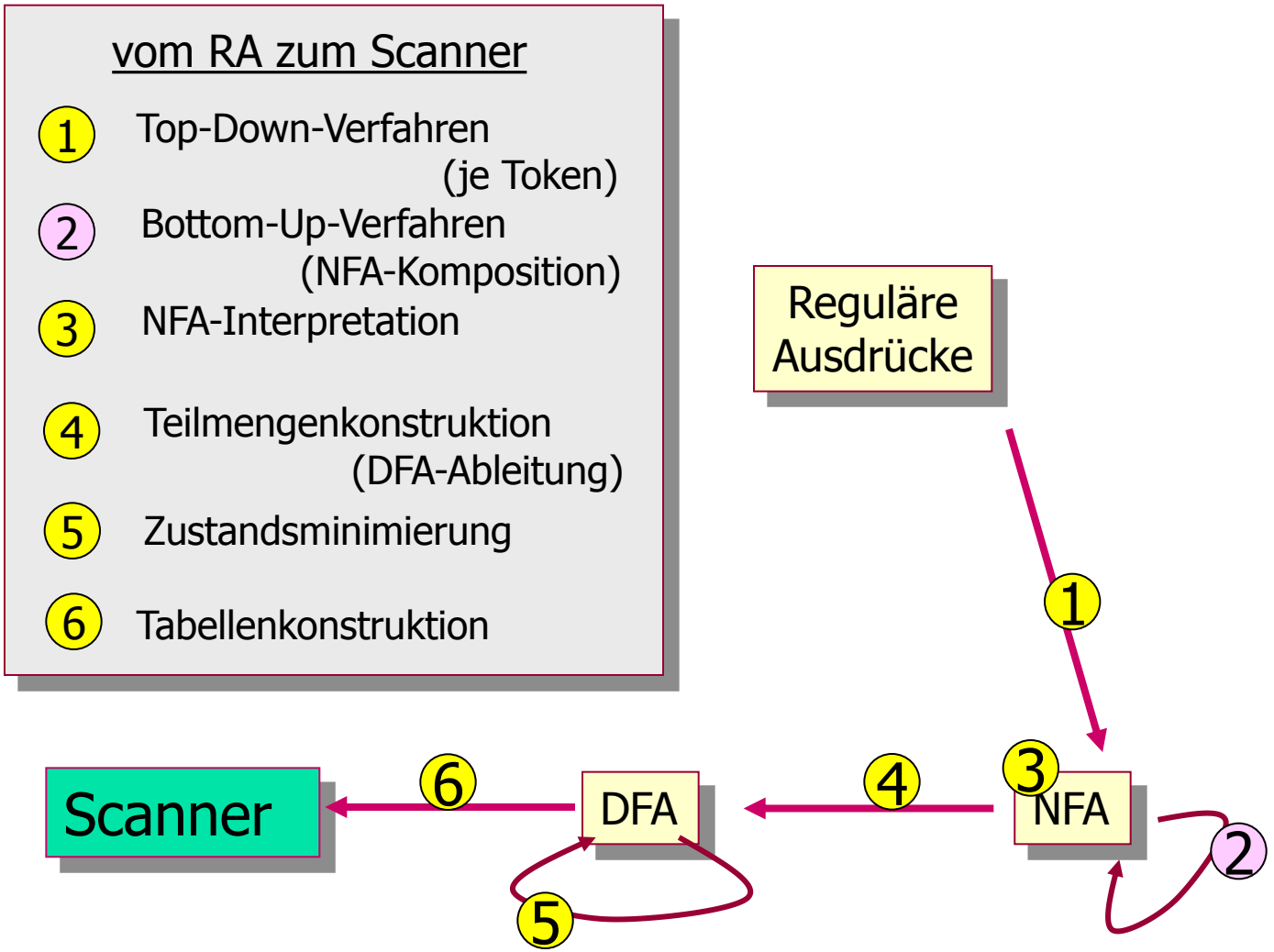
Regeln: RA → NFA (2/2)



Beispiel der Top-Down-Generierung: a(a|b)*



Problem 2: von einfachen NFAs zum komplexen NFA

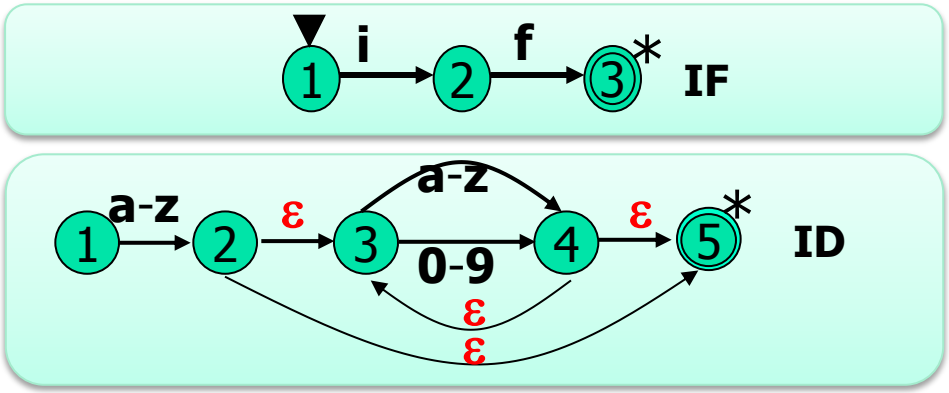


Komposition von NFAs

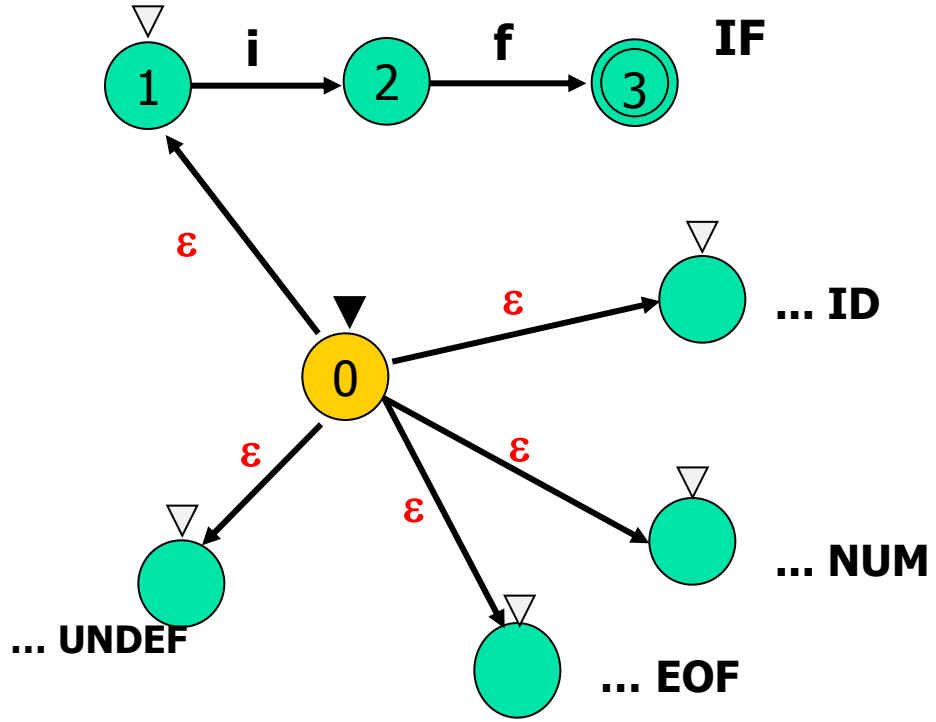
Einfacher Bottom-Up-Konstruktionsalgorithmus:

Einzelautomaten für elementare RAs werden aus einem neuen Startzustand beginnend mit ε-Übergängen in die Startzuständen der Einzelautomaten kombiniert (bei Umbenennung der Zustände)

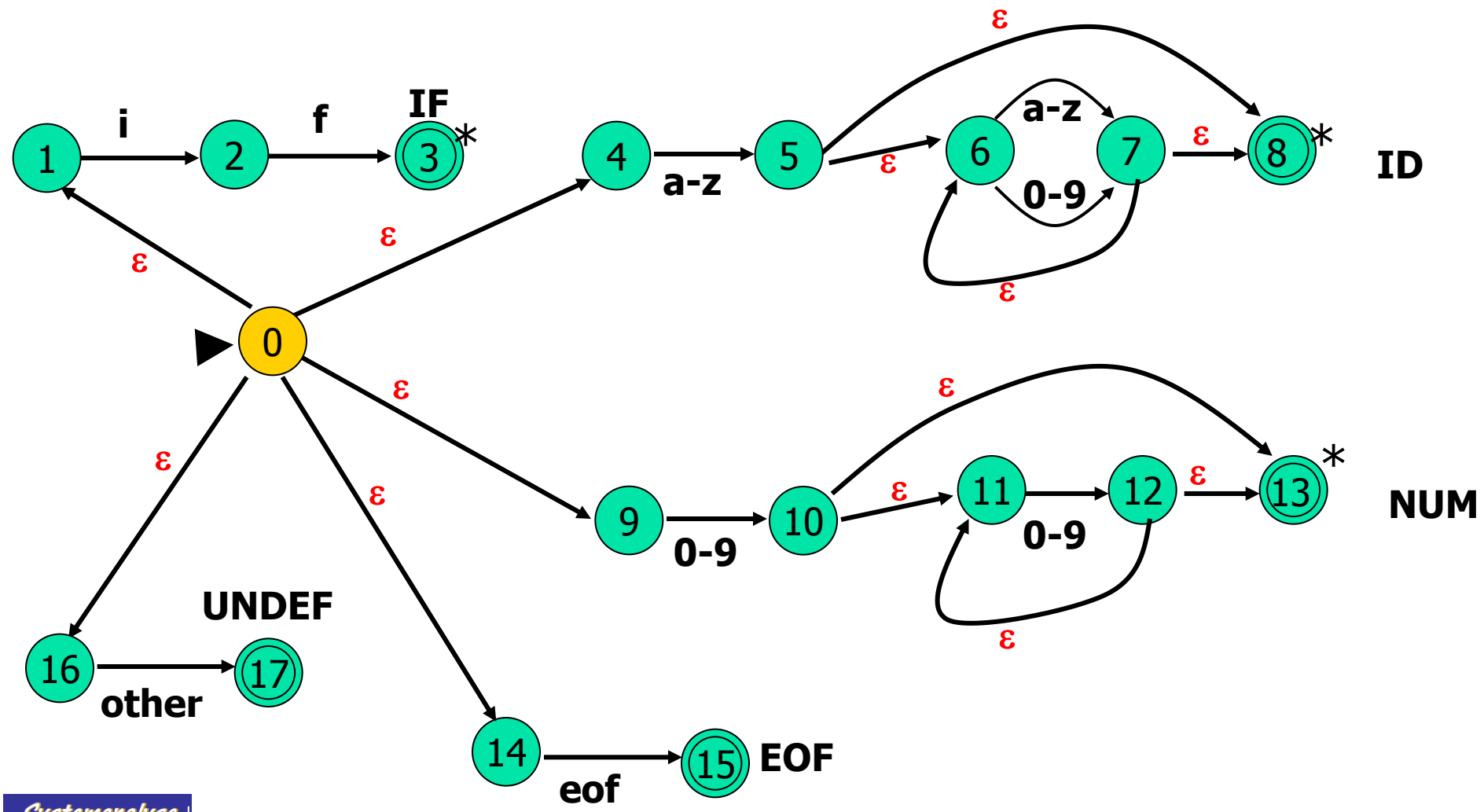
Beispiel (einfache NFAs):



NUM, EOF, UNDEF



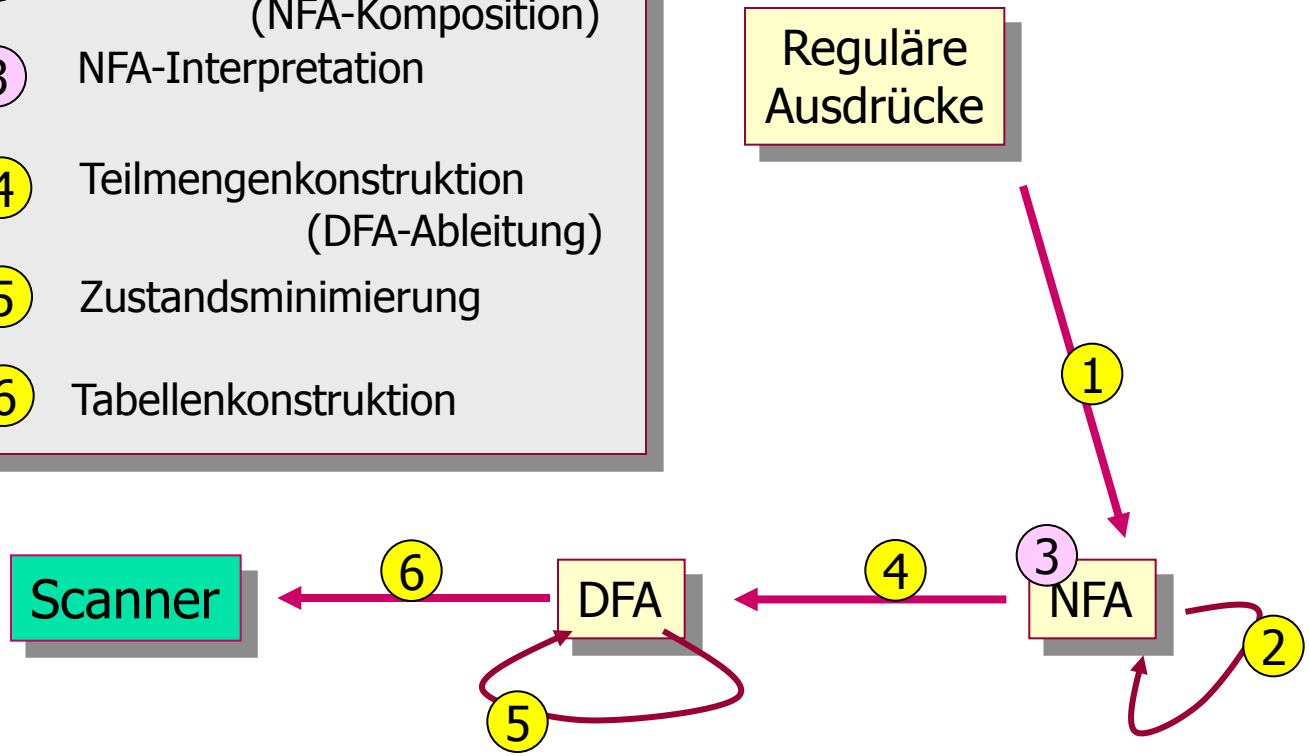
NFA-Konstruktion am Beispiel



Problem 3: NFA-Interpretation

vom RA zum Scanner

1. Top-Down-Verfahren (je Token)
2. Bottom-Up-Verfahren (NFA-Komposition)
3. NFA-Interpretation
4. Teilmengenkonstruktion (DFA-Ableitung)
5. Zustandsminimierung
6. Tabellenkonstruktion



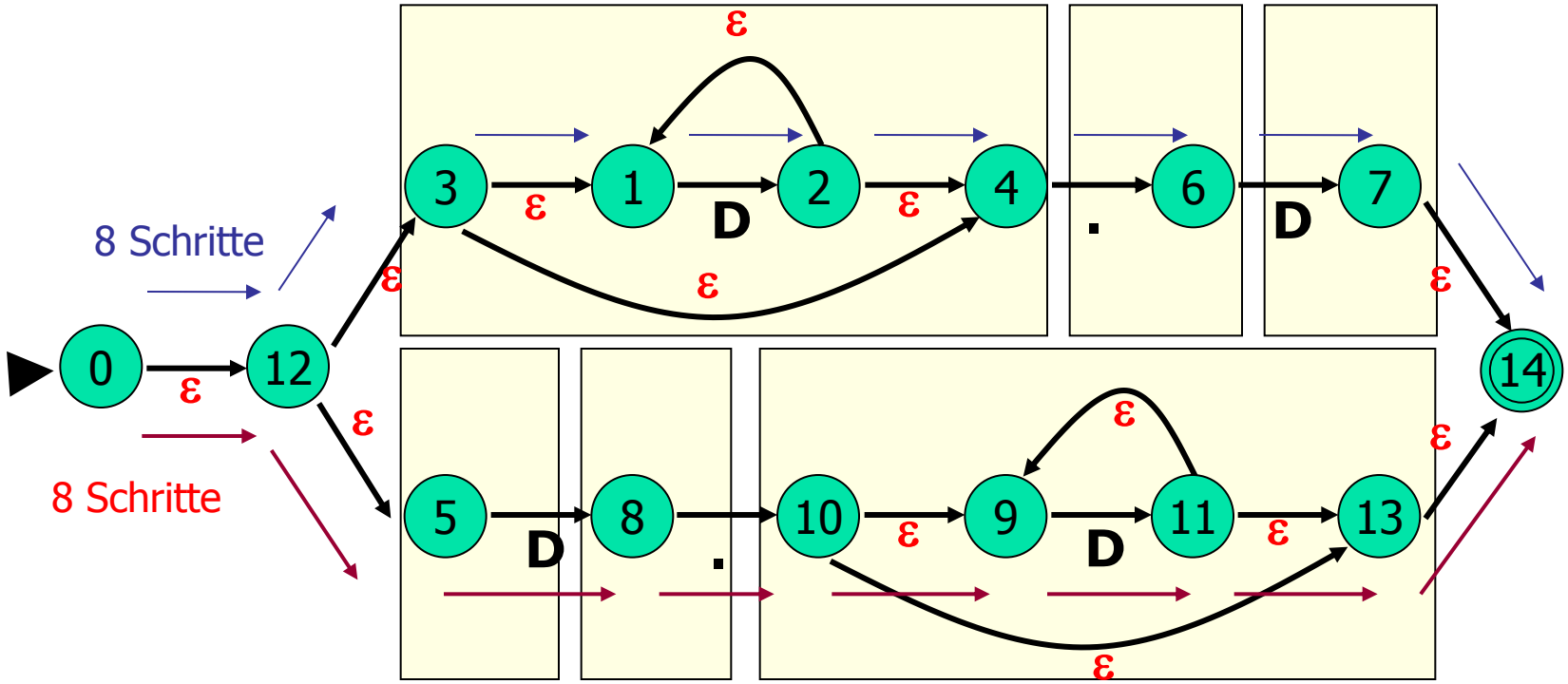
Interpretation eines NFA

Frage: Wie arbeitet der NFA generell ?

Eingabezeichenkette: 1.2

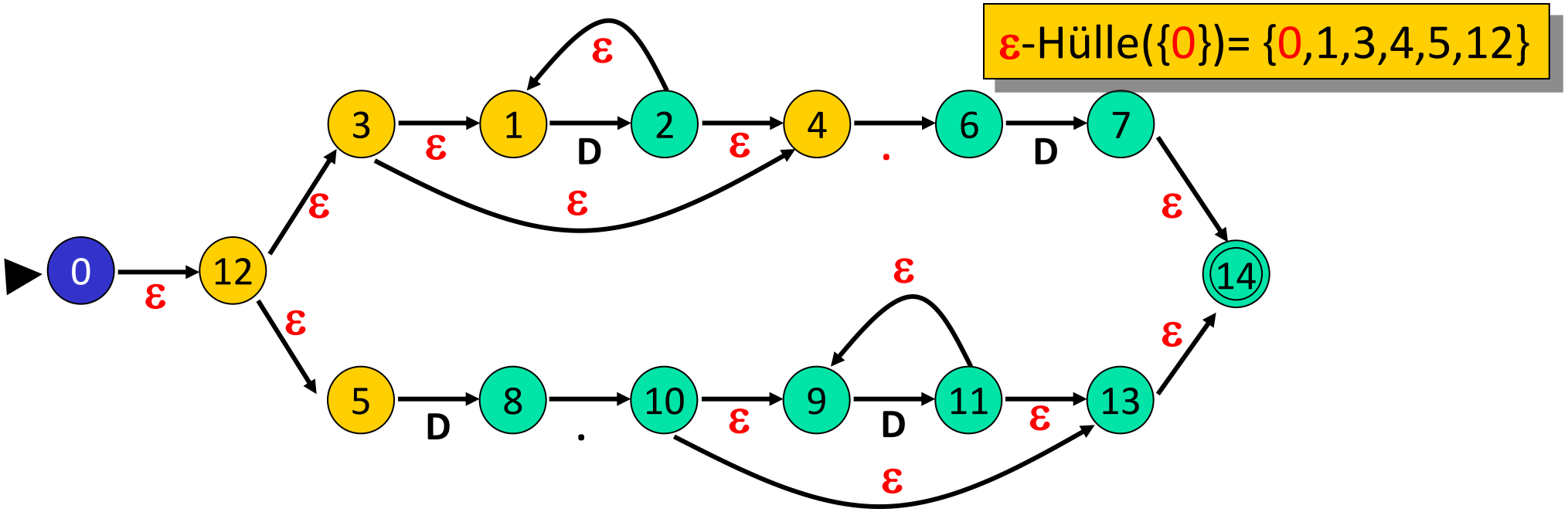
Beispiel Zahlendarstellung
RA= (**D***.**D**|**D.D***)

D - digit



Interpretation eines NFA: (D.D|D.D*)*

Erkenntnis: falls NFA im Zustand 0, ist er gleichzeitig in Zuständen 1,3,4,5,12



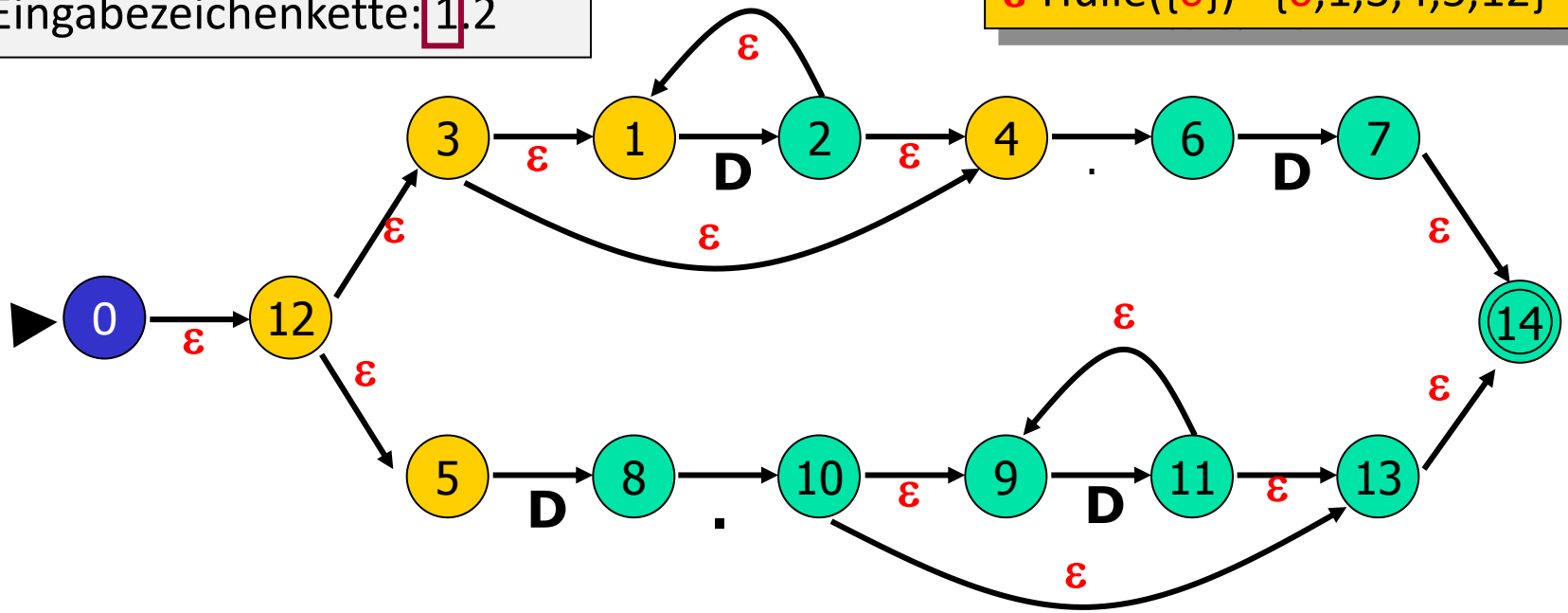
Def: **ε-Hülle** eines NFA-Zustandes x

… ist die Menge der Zustände, die von diesem Zustand x ausschließlich mittels ε-Übergängen erreicht werden kann, zzgl. x selbst

Interpretation eines NFA: (D*.D|D.D*)

Eingabezeichenkette: 1.2

ε-Hülle({0})= {0,1,3,4,5,12}



nextstate(ε-Hülle({0}), •)=

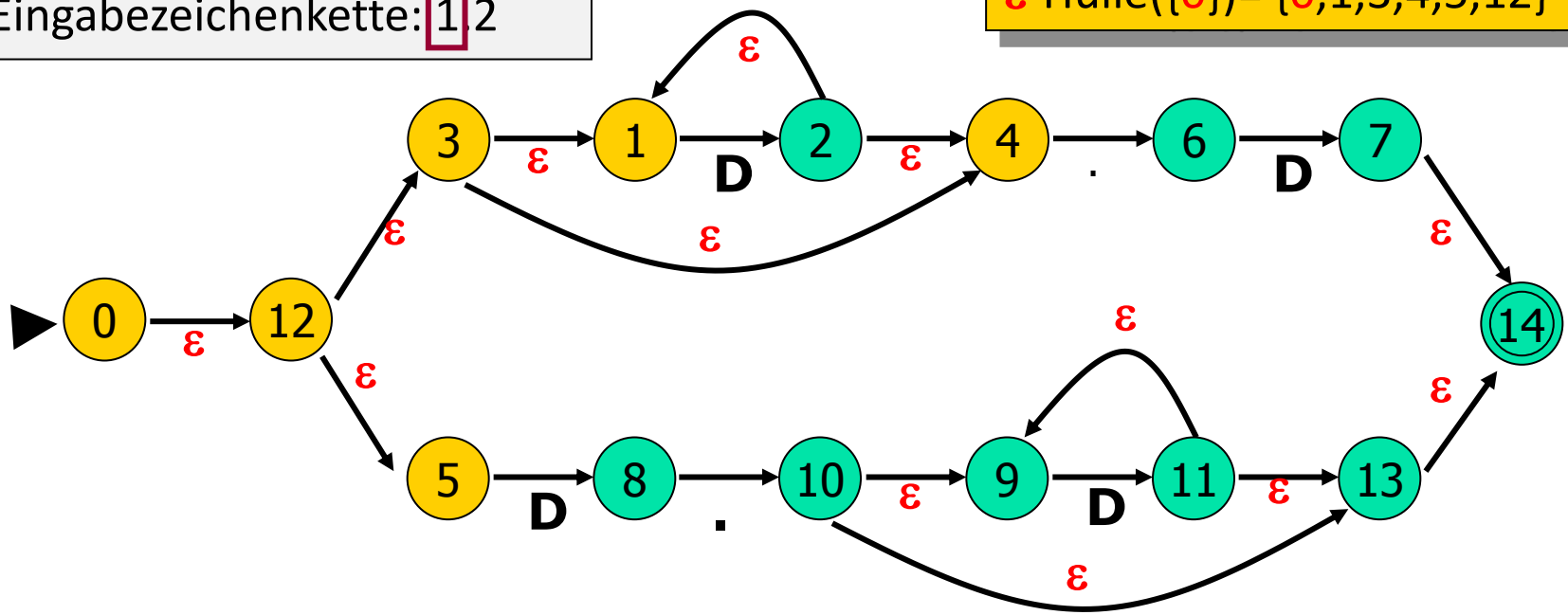
aktuelle Eingabezeichen

Zustandsübergangsgraph/
Tabelle

Interpretation eines NFA: (D*.D|D.D*)

Eingabezeichenkette: 1.2

ε-Hülle({0})= {0,1,3,4,5,12}

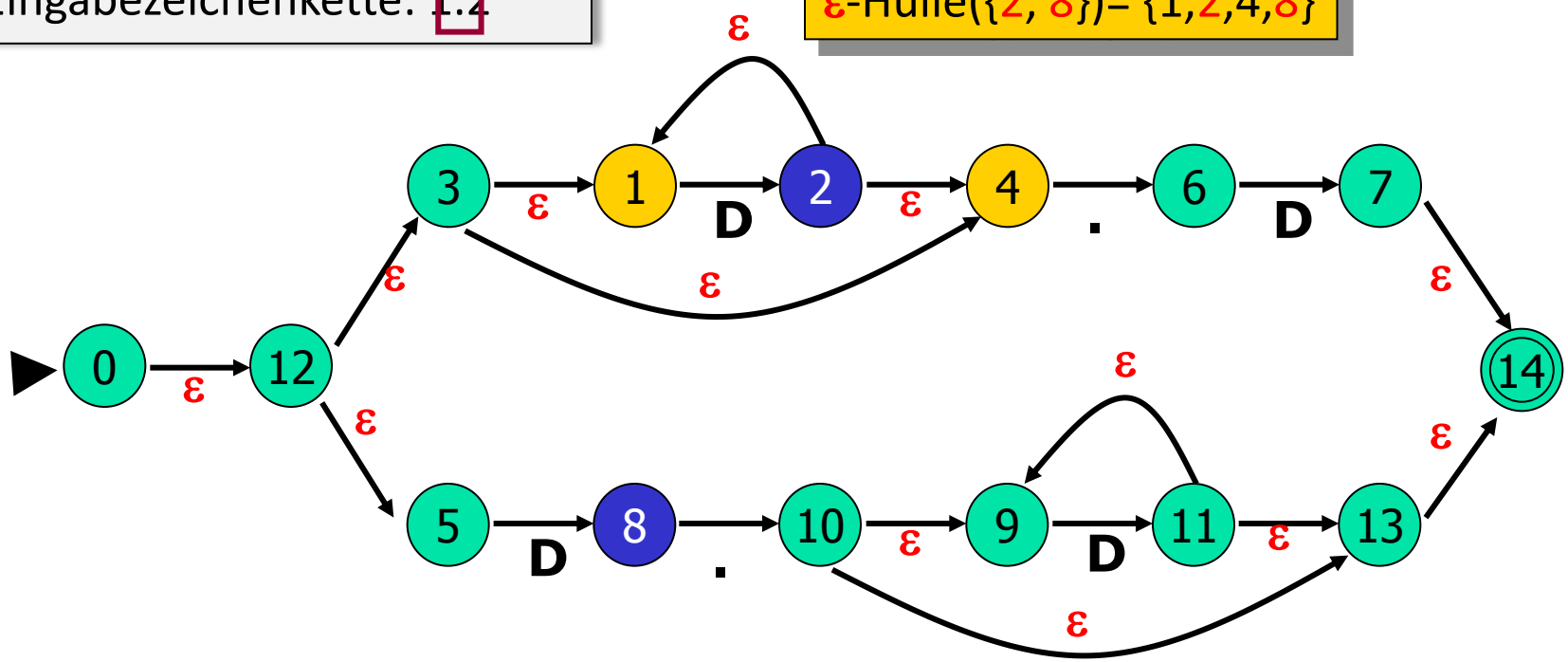


$$\text{nextstate}(\varepsilon\text{-Hülle}(\{0\}), D) = \left\{ \begin{array}{l} \text{nextstate}(0, D) = - \\ \text{nextstate}(1, D) = 2 \\ \text{nextstate}(3, D) = - \\ \text{nextstate}(4, D) = - \\ \text{nextstate}(5, D) = 8 \\ \text{nextstate}(12, D) = - \end{array} \right\} = \{2,8\}$$

Interpretation eines NFA: (D*.D|D.D*)

Eingabezeichenkette: 1.2

ε-Hülle({2, 8})= {1,2,4,8}

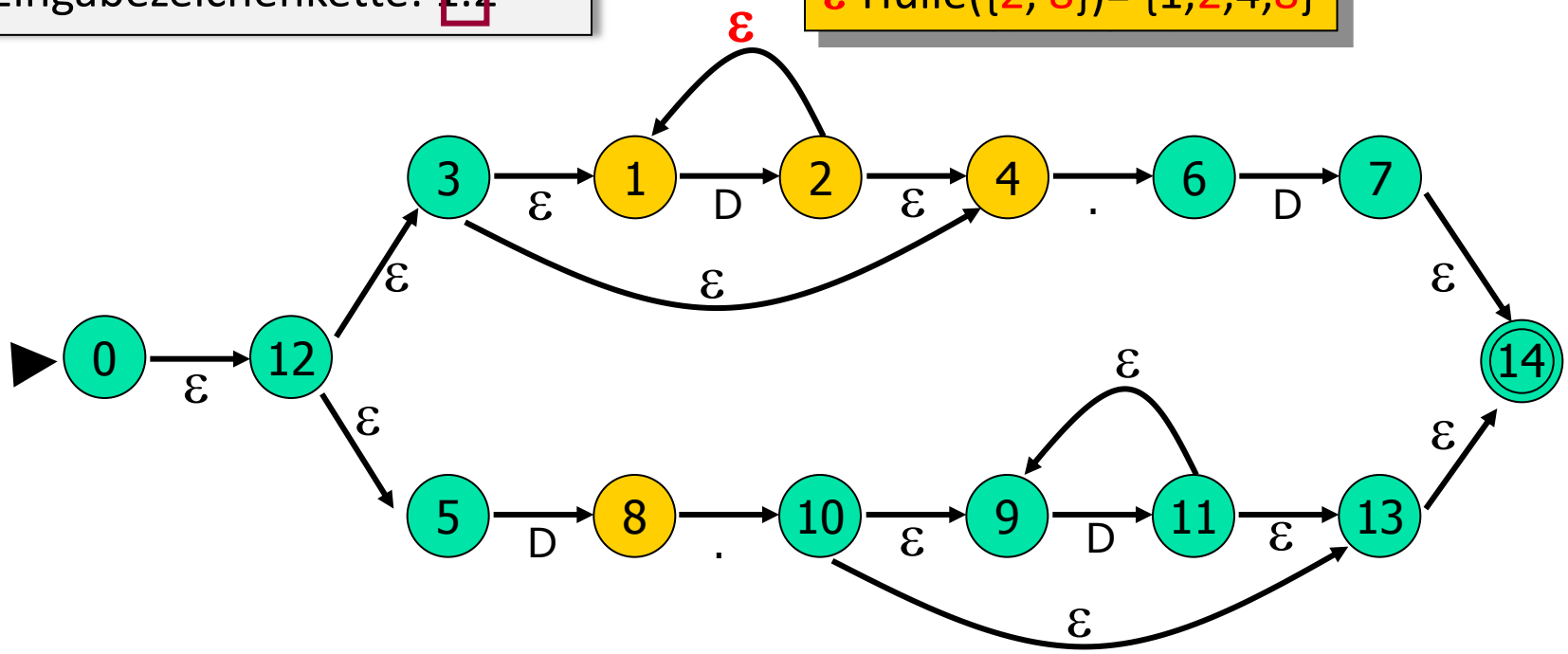


nextstate(ε-Hülle({2,8}), '.')=

Interpretation eines NFA: (D*.D|D.D*)

Eingabezeichenkette: 1.2

ε-Hülle({2, 8})= {1,2,4,8}

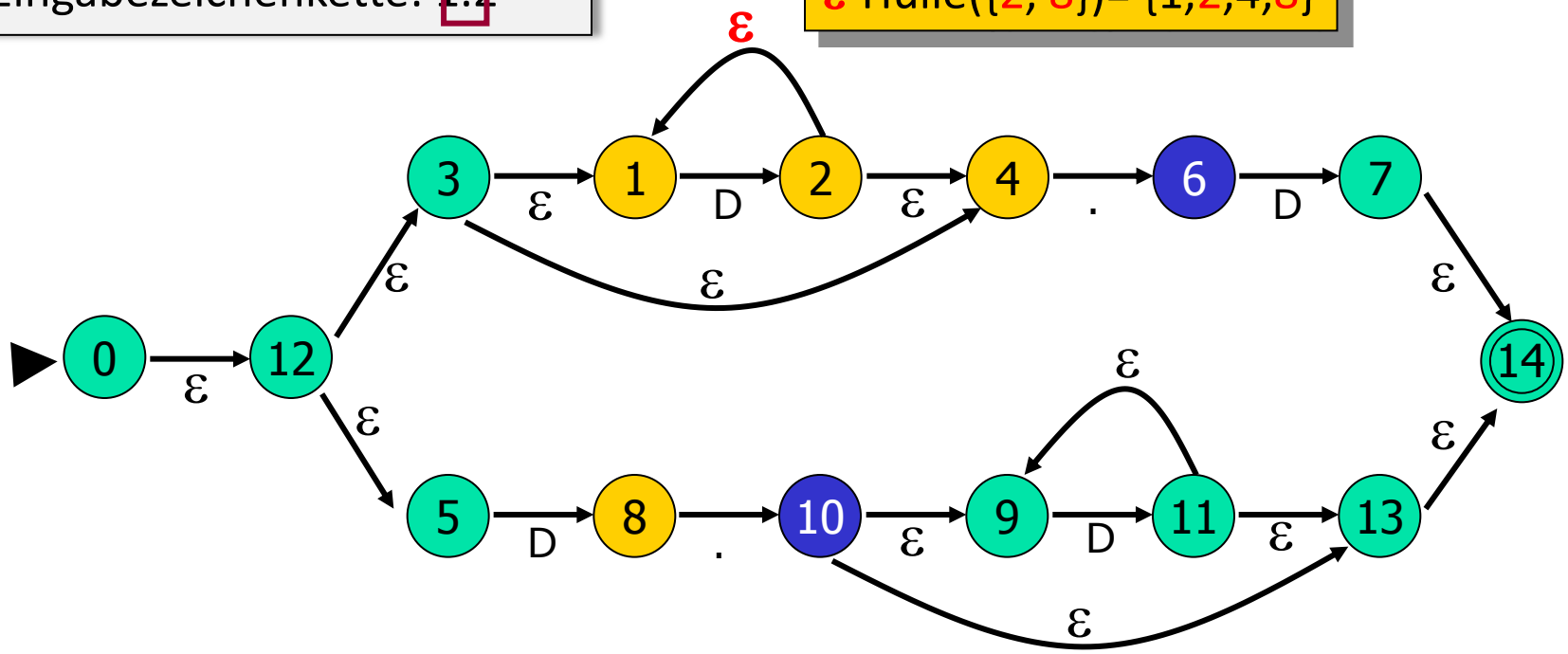


nextstate(ε-Hülle({2,8}), '.')= { nextstate(1, '.') = -, nextstate(2, '.') = -, nextstate(4, '.') = 6, nextstate(8, '.') = 10 } = {6,10}

Interpretation eines NFA: (D*.D|D.D*)

Eingabezeichenkette: 1.2

ε-Hülle({2, 8})= {1,2,4,8}

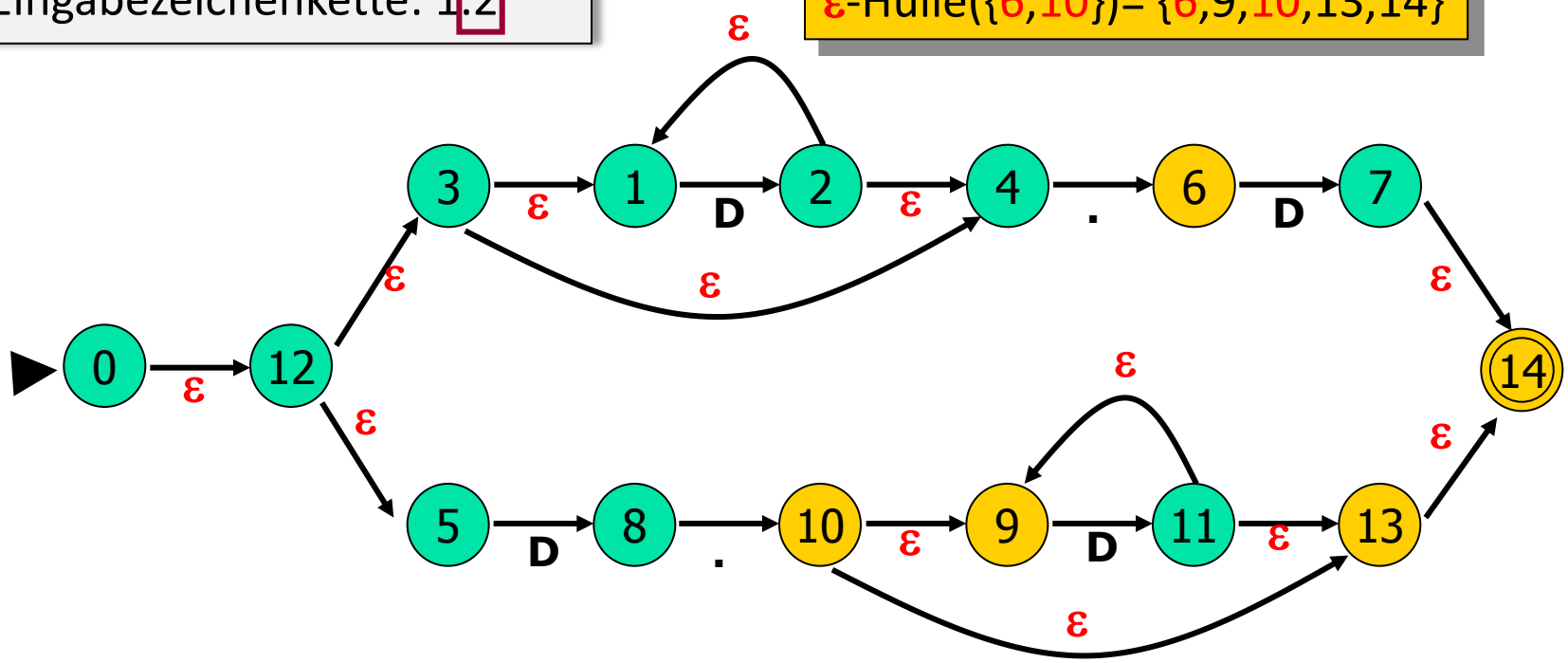


nextstate(ε-Hülle({2,8}), '.')= { nextstate(1, '.') = -, nextstate(2, '.') = -, nextstate(4, '.') = 6, nextstate(8, '.') = 10 } = {6,10}

Interpretation eines NFA: (D*.D|D.D*)

Eingabezeichenkette: 1.2

ε-Hülle({6,10})= {6,9,10,13,14}

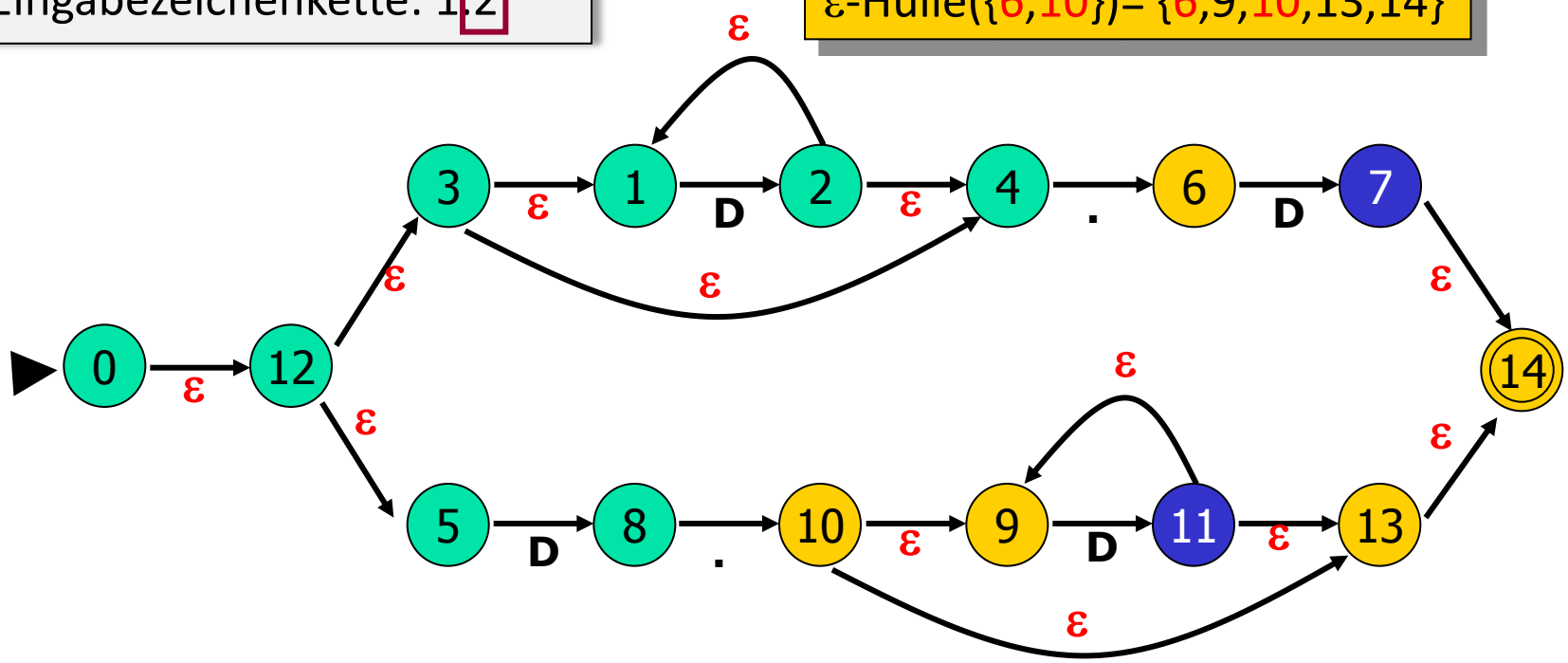


nextstate(ε-Hülle({6,9,10,13,14}), D)=

Interpretation eines NFA: (D*.D|D.D*)

Eingabezeichenkette: 1.2

ε-Hülle({6,10})= {6,9,10,13,14}

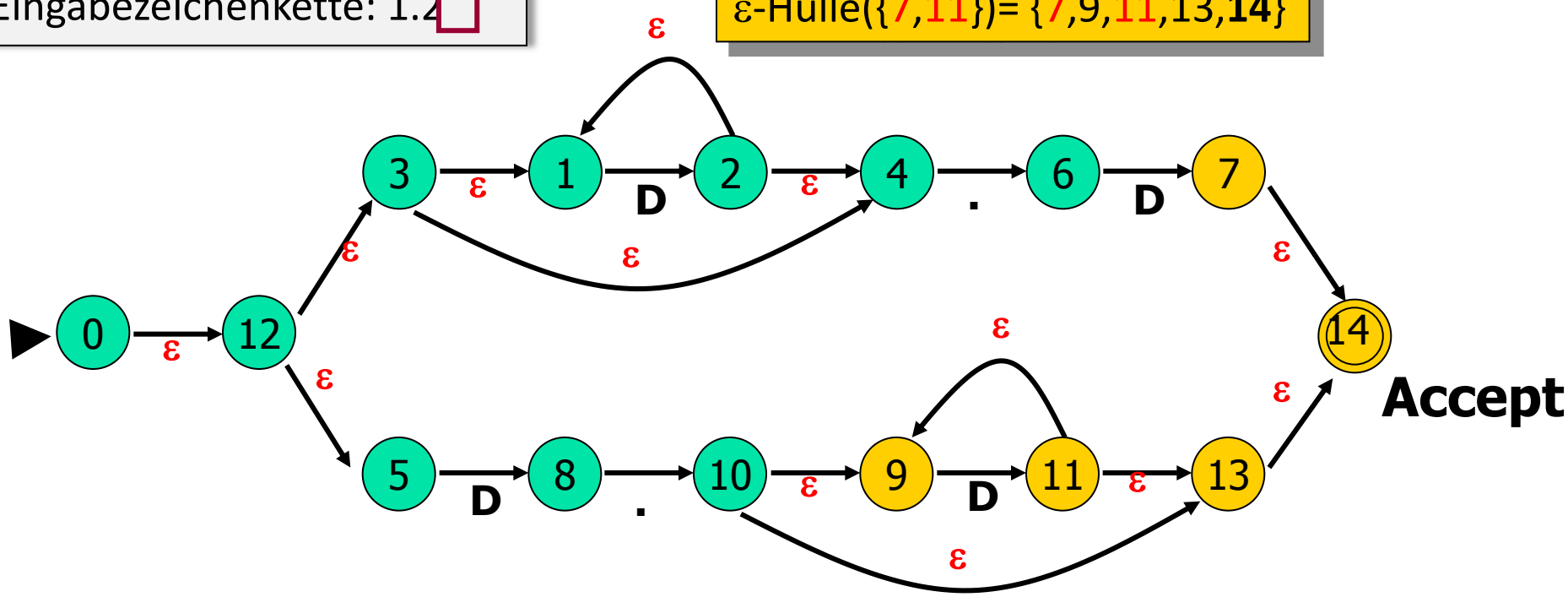


nextstate(ε-Hülle({6,9,10,13,14}), D)= { nextstate(6, D) = 7; nextstate(9, D) = 11; nextstate(10, D) = -; nextstate(13, D) = -; nextstate(14, D) = - } = {7,11}

Interpretation eines NFA: (D*.D|D.D*)

Eingabezeichenkette: 1.2

ε-Hülle({7,11})= {7,9,11,13,**14**}



ε-Hülle({11})= {9,11,13,**14**}

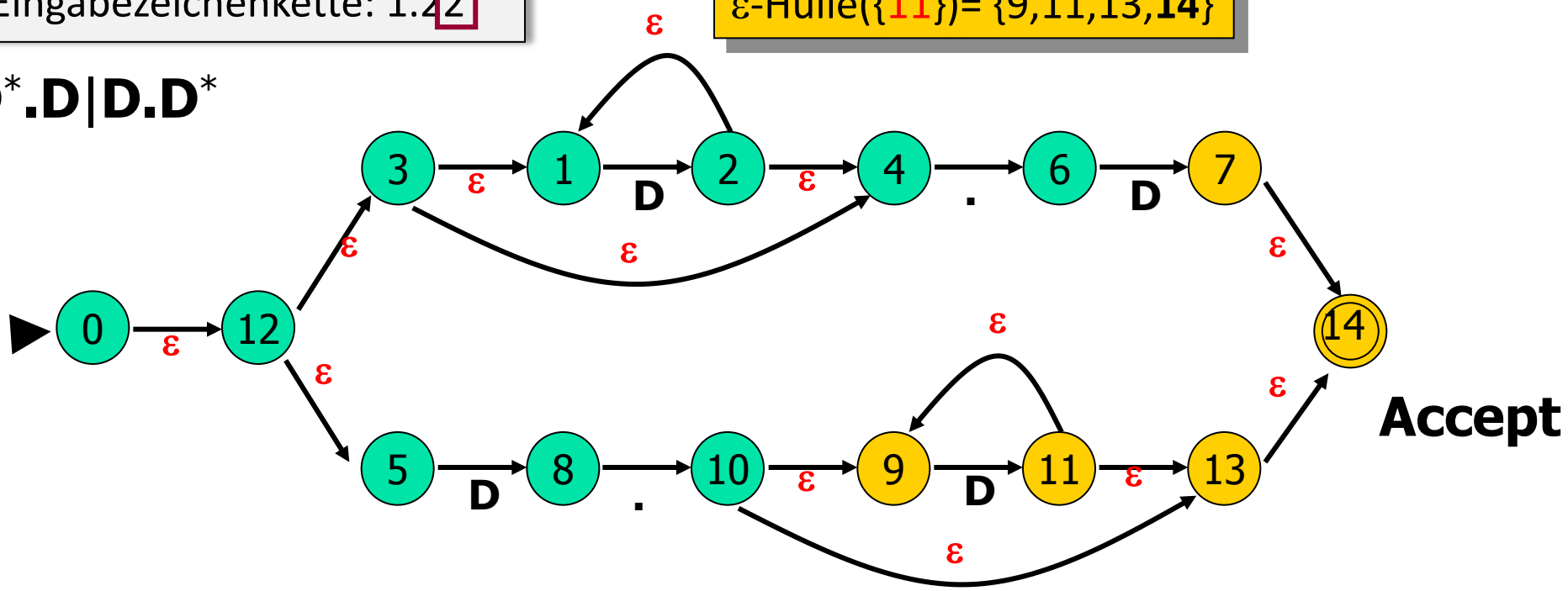
nextstate(ε-Hülle({7,9,11,13,14}), D)= { nextstate(7, D) = - ; nextstate(9, D) = 11 ; nextstate(11, D) = - ; nextstate(13, D) = - ; nextstate(14, D) = - } = {11}

Interpretation eines NFA: (D*.D|D.D*)

Annahme: weiteres Digit

Eingabezeichenkette: 1.22

ε-Hülle({11})= {9,11,13,**14**}



nextstate(ε-Hülle({9,11,13,14}), D)= { nextstate(9, D) = 11; nextstate(11, D) = -; nextstate(13, D) = -; nextstate(14, D) = - } = {11}

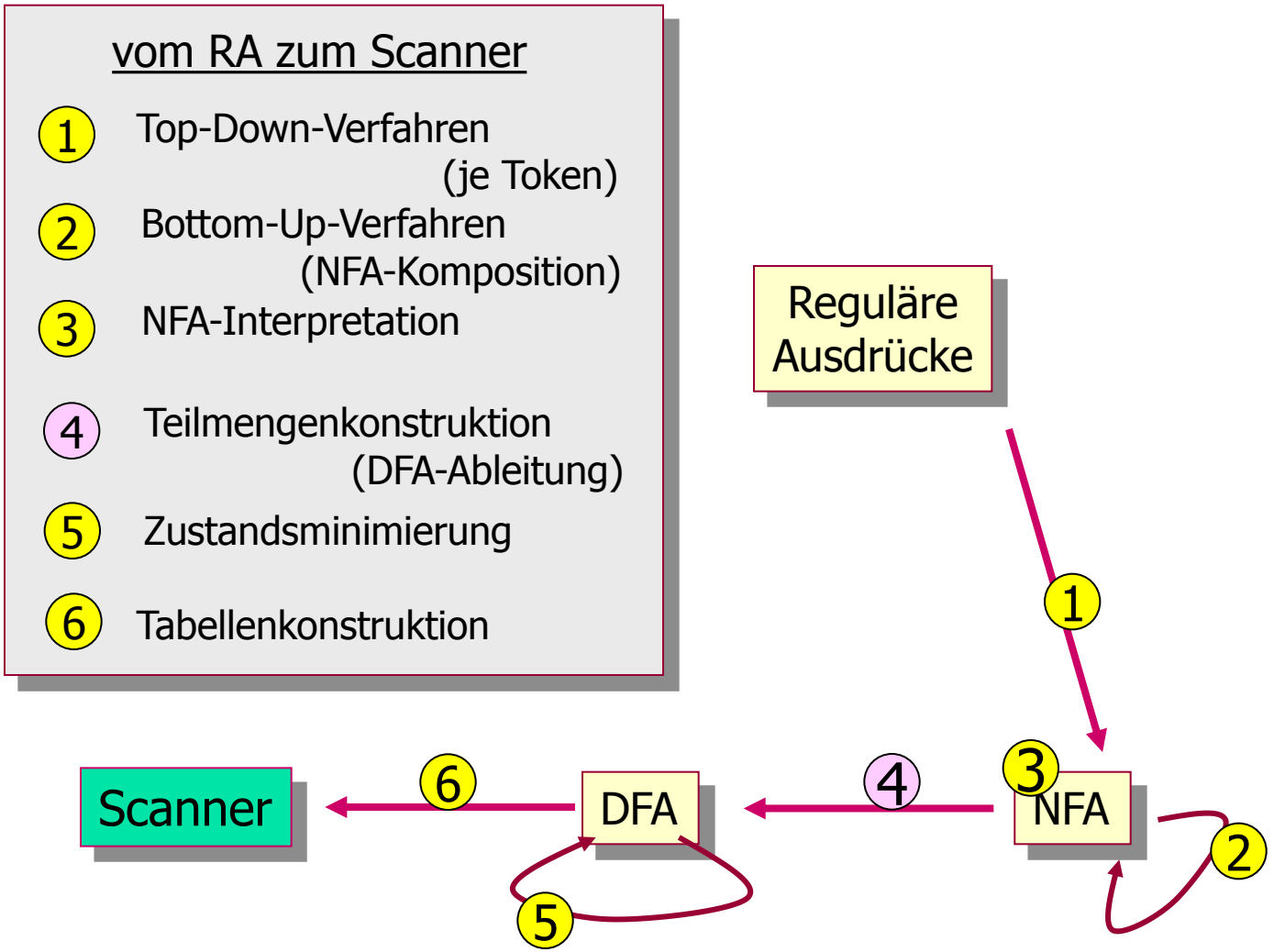
Allgemeines NFA-Implementierungsproblem

- **einfacher Fall:**
  Ein Scanner meldet die Erkennung eines Tokens,
  wenn er in einem **Endzustand** ist und für das nächste Eingabezeichen keinen Übergang mehr hat.

- **schwieriger Fall:**
  Hat er aus dem aktuellen Zustand **keinen** Übergang,
  ohne dass der Zustand ein Endzustand ist,
  so muss er die letzten Übergänge **rückgängig machen** bis zu dem letzten durchlaufenden Endzustand.

  Gibt es für das aktuelle Token noch keinen solchen Übergang
  (also **zurückgesetzt** bis zum **Startzustand**), so liegt ein Fehler vor.

Problem 4: vom NFA zum DFA



Eigenschaften von Endlichen Automaten

Satz: Äquivalenz von DFA und NFA

Für jeden NFA gibt es einen DFA,

der die gleiche Sprache erkennt und umgekehrt.

NFA-DFA: Überführungsalgorithmus von DFA und NFA

Basis: DFAs sind spezielle NFAs

jeder NFA kann in einen DFA konvertiert werden,
indem die gleichzeitig erreichbaren Zustände des NFA im DFA »simuliert« werden:

- jeder Zustand im abgeleiteten DFA
  korrespondiert mit einer Menge von Zuständen im NFA
- Überführung: NFA → DFA
  erfolgt mit dem Verfahren der sog. Teilmengenkonstruktion

Teilmengenkonstruktion (Prinzip)

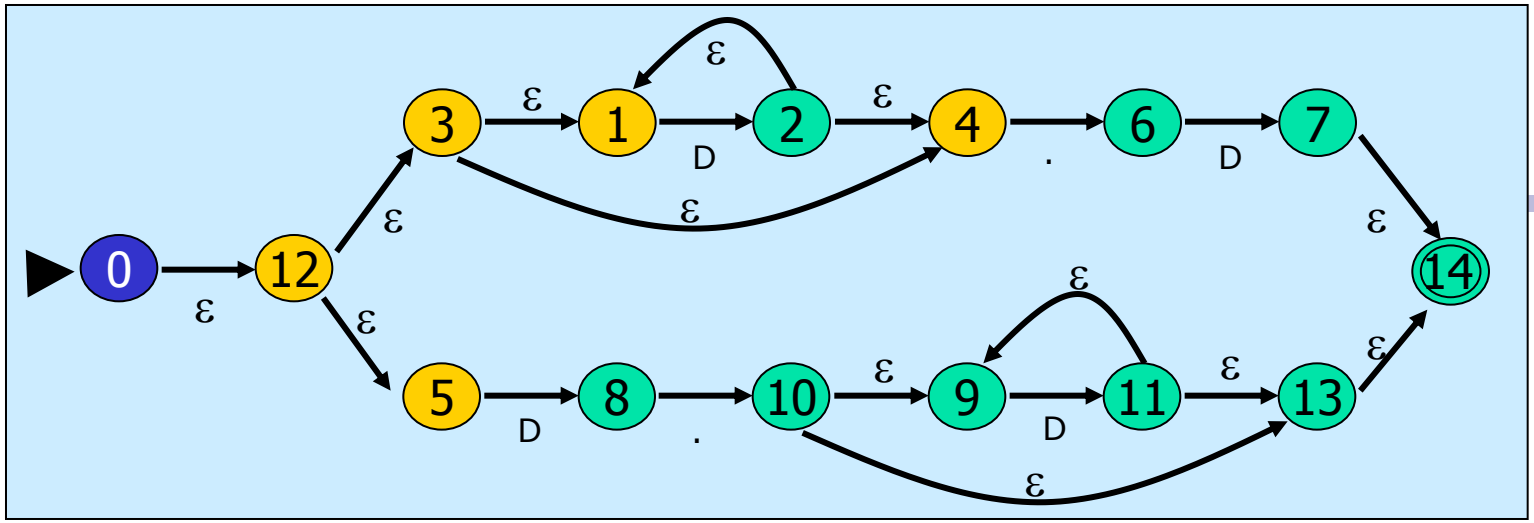
Der DFA merkt sich in seinen Zuständen alle möglichen Zustände,
in denen sich der NFA nach Lesen der einzelnen Zeichen befinden kann

d.h.:

nach Lesen von $a_1a_2…a_n$ befindet sich der DFA in einem Zustand,
der die Teilmengen aller Zustände repräsentiert,
die vom Startzustand des NFA entlang eines mit $a_1a_2…a_n$ markierten Pfades erreicht werden können

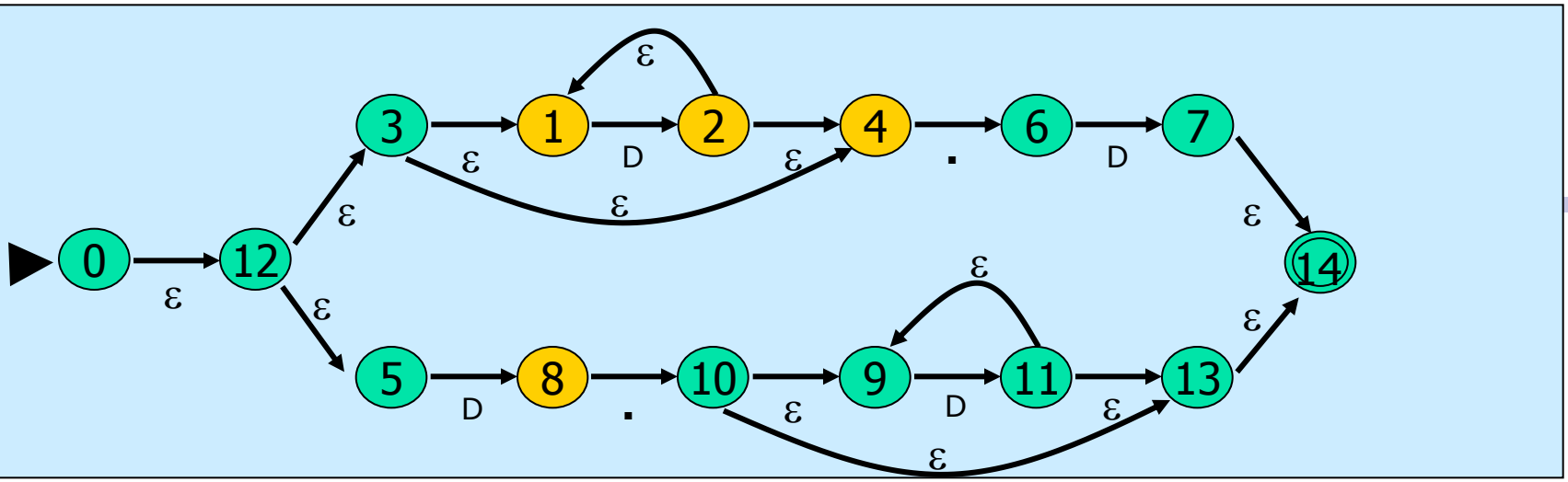
Achtung: Die Anzahl der Zustände eines DFA kann exponentiell größer sein
als die des entsprechenden NFA
(Worst Case tritt aber in der Praxis nicht wirklich ein)

➔ **Zustandsminimierung** stellt dennoch eine wichtige nachgeordnete Aufgabe dar (lex/flex)

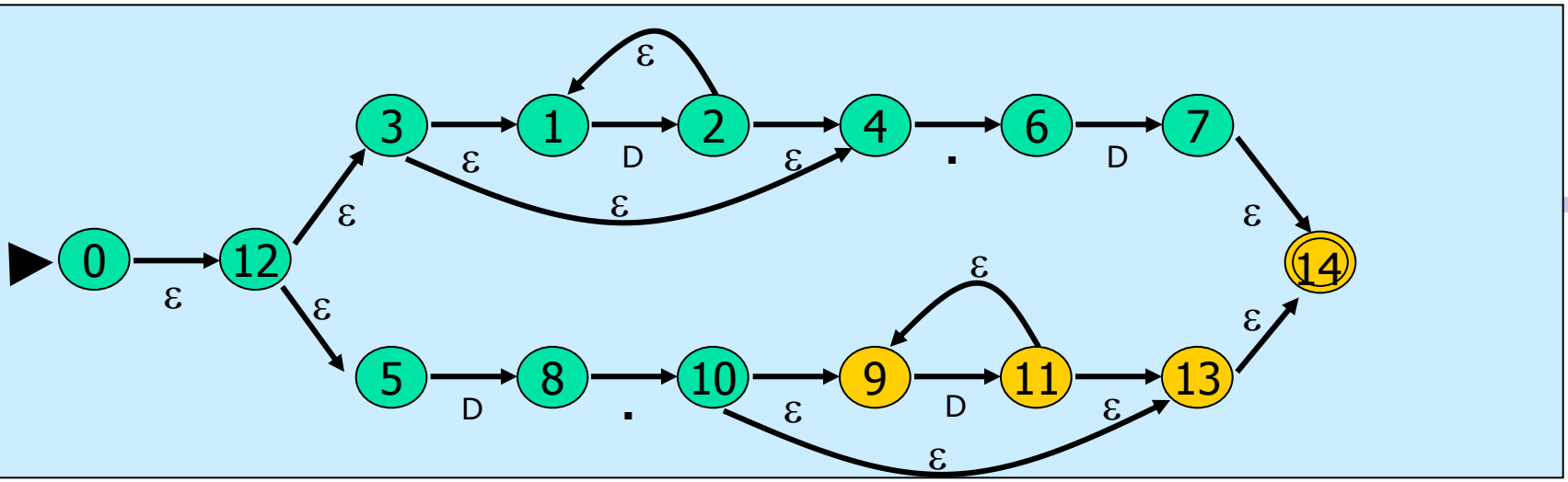


RA: (D*.D|D.D*)

DFA-Zustand	NFA-Zustand	ε-Hülle	Nextstate(**D**)		Nextstate(**.**)		Akzeptanz
			NFA	DFA	NFA	DFA	
0	{0}	{0,1,3,4,5,12}	{2,8}	**1**	{6}	**2**	0

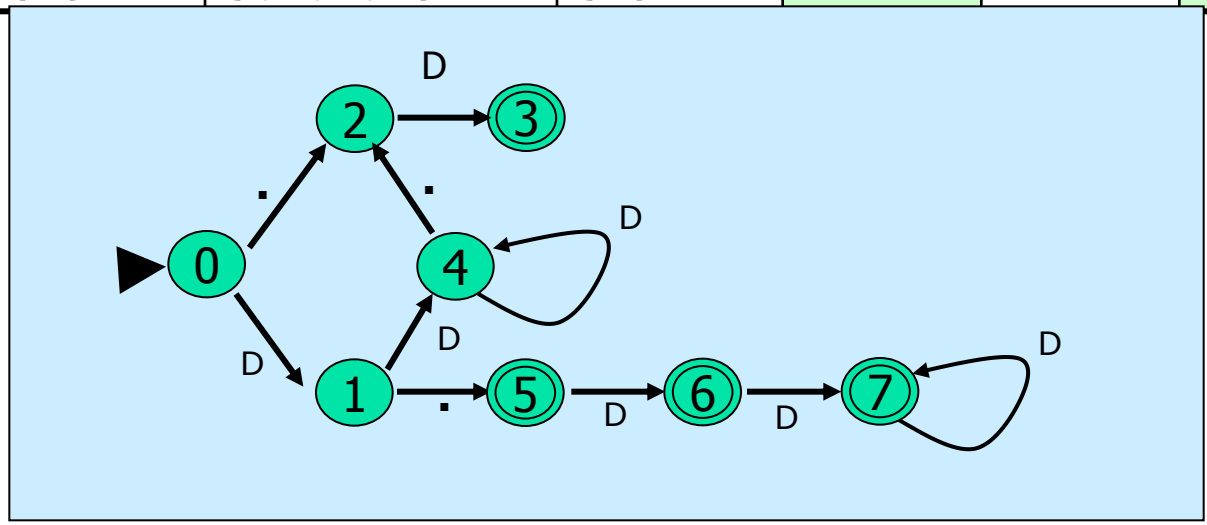


DFA-Zustand	NFA-Zustand	ε-Hülle	Nextstate(**D**)		Nextstate(**.**)		Akzeptanz
			NFA	DFA	NFA	DFA	
0	{0}	{0,1,3,4,5,12}	{2,8}	1	{6}	2	0
1	{2, 8}	{1,2,4,8}	{2}	4	{6,10}	5	0



DFA-Zustand	NFA-Zustand	ε-Hülle	Nextstate(**D**)		Nextstate(**.**)		Akzeptanz
			NFA	DFA	NFA	DFA	
0	{0}	{0,1,3,4,5,12}	{2,8}	1	{6}	2	0
1	{2, 8}	{1,2,4,8}	{2}	4	{6,10}	5	0
2	{6}	{6}	{7}	3	ø	-	0
3	{7}	{7,**14**}	ø	-	ø	-	**1**
4	{2}	{1,2,4}	{2}	4	{6}	2	0
5	{6, 10}	{6,9,10,13,**14**}	{7,11}	6	ø	-	**1**
6	{7, 11}	{7,9,11,13,**14**}	{11}	7	ø	-	**1**
7	{11}	{9,11,13,**14**}	{11}	7	ø	-	**1**

DFA-Zustand	NFA-Zustand	ε-Hülle	Nextstate(**D**)		Nextstate(.)		Akzeptanz
			NFA	DFA	NFA	DFA	
0	{0}	{0,1,3,4,5,12}	{2,8}	1	{6}	2	0
1	{2, 8}	{1,2,4,8}	{2}	4	{6,10}	5	0
2	{6}	{6}	{7}	3	ø	-	0
3	{7}	{7,13,14}	ø	-	ø	-	1
4	{2}	{1,2,4}	{2}	4	{6}	2	0
5	{6, 10}	{6,9,10,13,14}	{7,11}	6	ø	-	1
6	{7, 11}	{7,9,11,13,14}	{11}	7	ø	-	1
7	{11}	{9,11,13,14}	{11}	7	ø	-	1



Ergebnis:
konstruierter DFA

Zusammenfassung der Zustände 6 und 7 möglich

NFA → DFA: Teilmengenkonstruktion (Formale Beschreibung einzelner Operationen)

Eingabe: ein NFA **N**

Ausgabe: ein DFA **D** mit L(**D**) = L(**N**) (**N** und **D** akzeptieren dieselben Sprachen)

Methode: Erstelle für **D** eine Übergangstabelle **Tran^D** , so dass **D** alle möglichen Übergänge (Bewegungen) simuliert, die **N** für eine gegebene Eingabezeichenkette machen kann.

Sei **s** ein Zustand in **N (s ∈ S^N)** und **T** eine Menge von Zuständen in **N** (**T** ⊆ **S^N**), dann benutze dabei die folgenden Operationen:

Operation	Definition
ε-Hülle (s)	Menge **aller** N-Zustände, die von **einem** N-Zustand s **allein** mit ε-Übergängen erreicht werden können
ε-Hülle (T)	Menge **aller** N-Zustände, die von **allen** N-Zuständen s einer Zustandsmenge T (T ⊆ S^N) **allein** mit ε-Übergängen erreicht werden können
nextstate (T, a)	Menge **aller** N-Zustände, die von **allen** N-Zuständen s (s ∈ T) für **ein Eingabesymbol** a erreicht werden können

NFA → DFA: Teilmengenkonstruktion (formaler Algorithmus)

in Pseudocode: sukzessiver Aufbau der Zustandsmenge
bei Festlegung der Zustandsübergangsfunktion

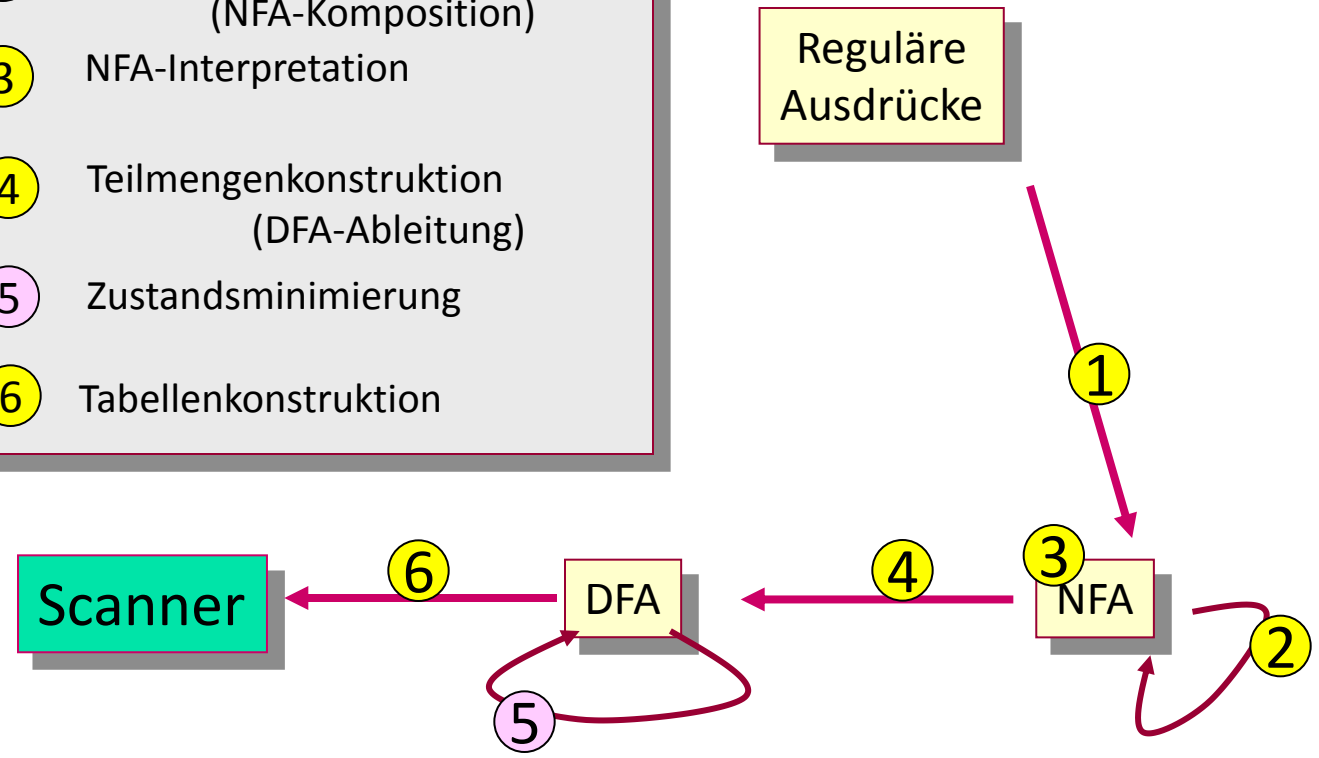
```
s0^D := ε-Hülle(s0^N) //d.h.: es gibt einen unmarkierten Zustand in S^D

while es existiert ein noch unmarkierter Zustand z in S^D do
      markiere z;
      for jedes Eingabesymbol a do
            U := ε-Hülle (nextstate (z, a));
            if U nicht in S^D then
                  füge U als unmarkierten Zustand der Menge S^D hinzu
            endif

            Tran_D [z, a] := U;
      endfor
endwhile
```

Problem 5: Zustandsminimierung eines DFAs

vom RA zum Scanner

1. Top-Down-Verfahren (je Token)
2. Bottom-Up-Verfahren (NFA-Komposition)
3. NFA-Interpretation
4. Teilmengenkonstruktion (DFA-Ableitung)
5. Zustandsminimierung
6. Tabellenkonstruktion

führe (falls vorhanden) kompatible Zustände zusammen



Minimierung der Zustandsmenge eines DFA

Satz: Äquivalenz von DFA und NFA

Für jeden RA gibt es einen kleinsten DFA
(DFA mit minimaler Zustandsanzahl),
der den RA erkennt und bis auf Umbenennungen der Zustände eindeutig ist.

Verfahren

1. Konstruktion entsprechend der Beweisidee
   (s. Theoretische Informatik)

2. Partitionierungsalgorithmus
   (Verwendung beim praktischen Scanner-Bau)

Minimalistischer DFA: Partitionierungsmethode

Prinzip: Partitionierungsmethode

Grundidee

- Bestimmung **aller** Gruppen von Zuständen, die durch eine beliebige Eingabe a getrennt sind:

  zwei Zustände s1 und s2 sind dann getrennt, wenn es eine Eingabe a gibt, so dass nextstate(s1,a) und nextstate(s2,a) verschiedenen Gruppen angehören

- eine Gruppe von Zuständen, die nicht weiter zerlegt werden kann, wird zu einem einzigen Zustand verschmolzen (Zustandsreduktion)

Start

- am Anfang besteht die Partitionierung aus zwei Gruppen: **akzeptierende** und **nicht-akzeptierende** Zustände

Ziel

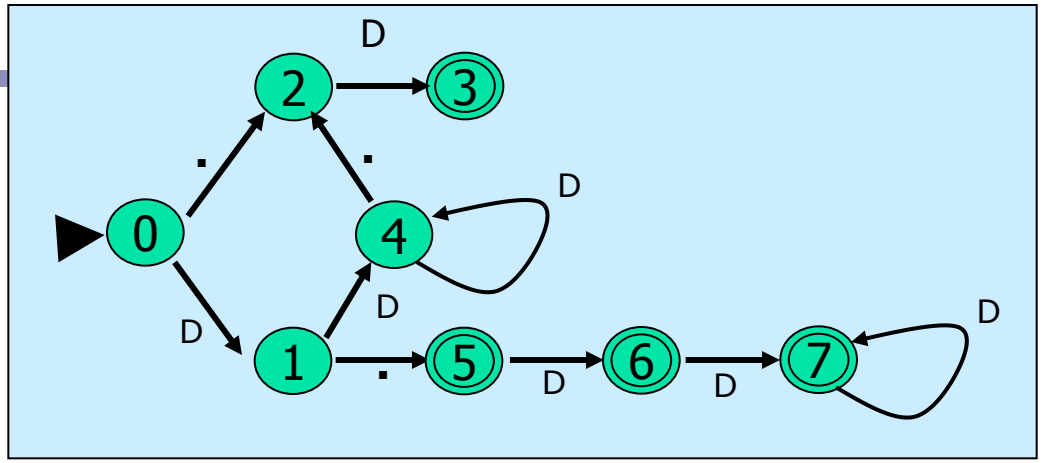
- schrittweise Durchführung einer Maximierung der Partitionierung, und zwar durch systematische Untersuchung aller Eingaben für die Zustände der bisherigen Gruppe

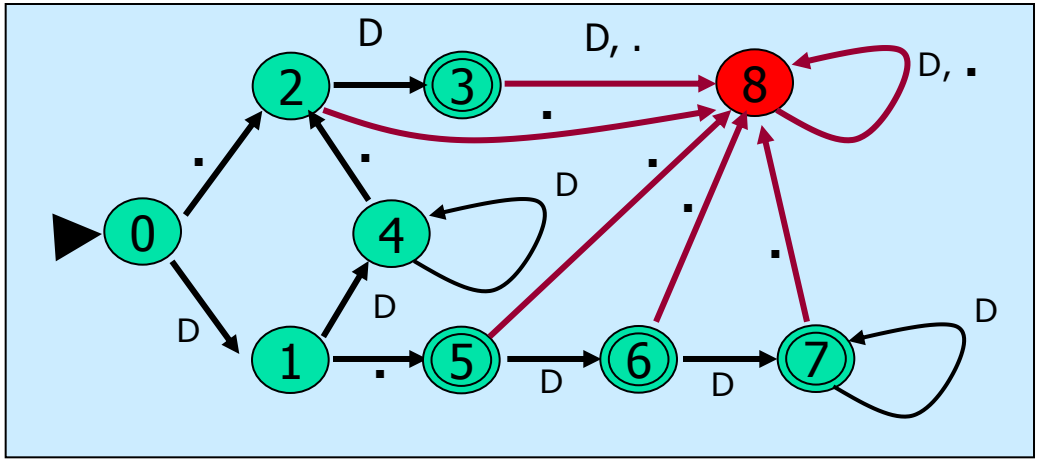
- **Abbruch** wenn keine Gruppe weiter partitioniert werden kann

Minimalistischer DFA: Zustandspaare (1)

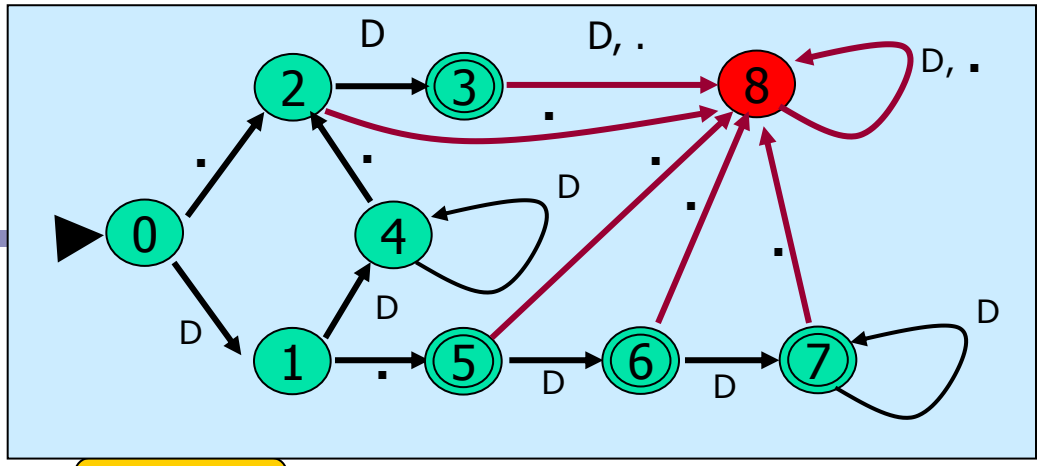
Eingabe: DFA

Schritt 0: Komplettiere um Fehlerzustand





Minimalistischer DFA: Zustandspaare (2)



Schritt 1:

Tabelle aller Zustandspaare {z, z‘} mit z ≠ z‘

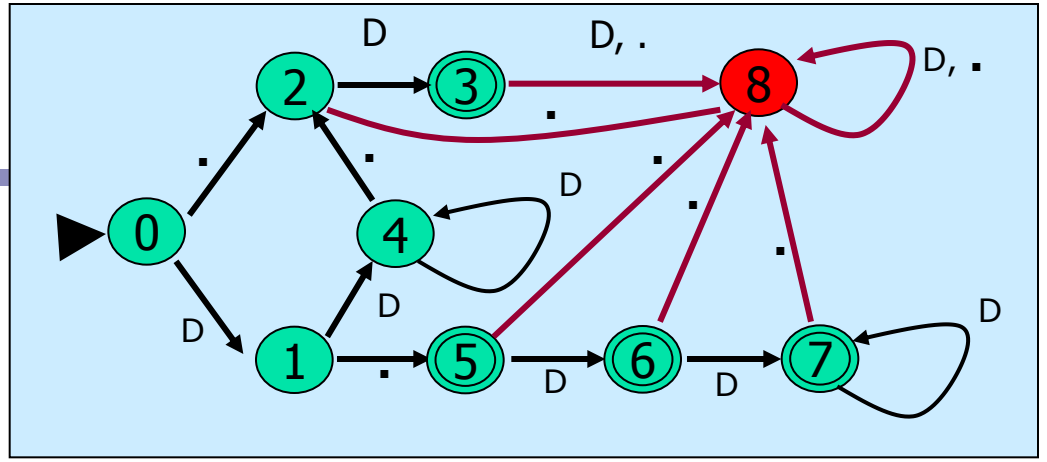
	0	1	2	3	4	5	6	7
1								
2								
3								
4								
5								
6								
7								
8								

Schritt 2:

markiere alle {z, z‘} mit z ϵ F und z‘ $\notin$ F und umgekehrt

	0	1	2	3	4	5	6	7
1								
2								
3	*	*	*					
4				*				
5	*	*	*		*			
6	*	*	*		*			
7	*	*	*		*			
8				*		*	*	*

Minimalistischer DFA: Zustandspaare (3)



	0	1	2	3	4	5	6	7
1								
2								
3	*	*	*					
4				*				
5	*	*	*		*			
6	*	*	*		*			
7	*	*	*		*			
8				*		*	*	*

Schritt 3:

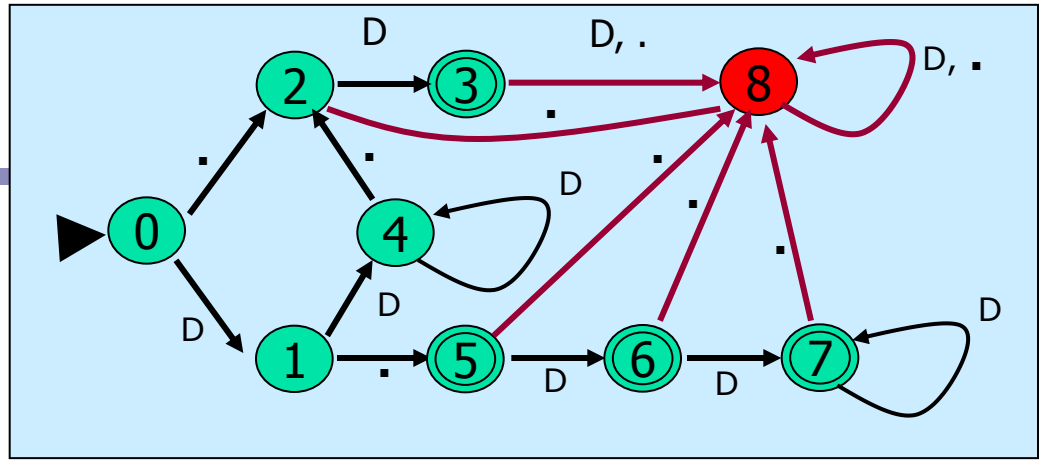
teste für jedes noch unmarkierte {z, z'} und jedes a ϵ S, **ob** { δ(z,a), δ(z',a) } bereits markiert ist
wenn ja: markiere auch {z, z'}

Schritt 3.1:

{ δ(1, .), δ(0, .) } = { 5, 2 } ➔ **ja** ➔ {1, 0}
{ δ(1, D), δ(0, D) } = { 4, 1 }

	0	1	2	3	4	5	6	7
1	*							
2								
3	*	*	*					
4				*				
5	*	*	*		*			
6	*	*	*		*			
7	*	*	*		*			
8				*		*	*	*

Minimalistischer DFA: Zustandspaare (4)



1	*							
2								
3	*	*	*					
4				*				
5	*	*	*		*			
6	*	*	*		*			
7	*	*	*		*			
8				*		*	*	*
	0	1	2	3	4	5	6	7

Schritt 3.2:

$\{ \delta(2, .), \delta(0, .) \} = \{ 8, 2 \}$

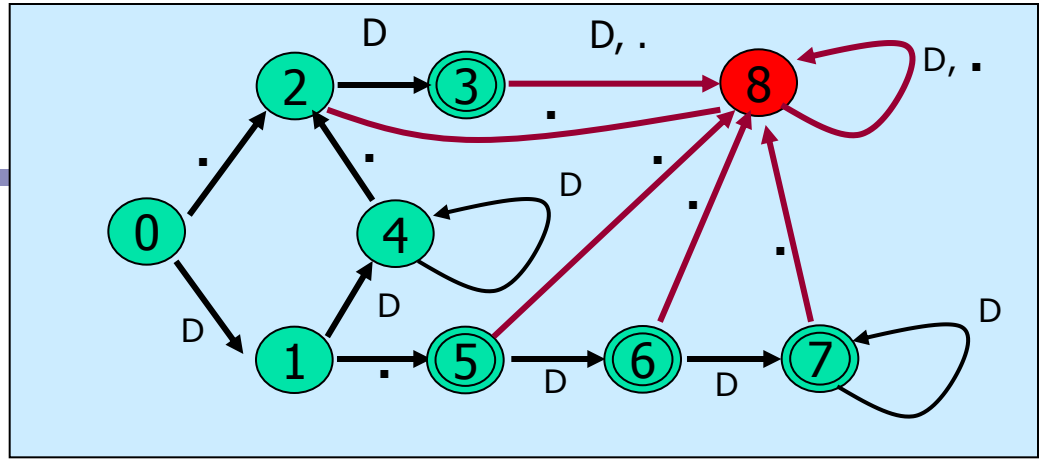
$\{ \delta(2, D), \delta(0, D) \} = \{ 3, 1 \}$ ➔ **ja** ➔ {2, 0}

$\{ \delta(2, .), \delta(1, .) \} = \{ 8, 5 \}$ ➔ **ja** ➔ {2, 1}

$\{ \delta(2, D), \delta(1, D) \} = \{ 3, 4 \}$ ➔ **ja** ➔ {2, 1}

1	*							
2	*	*						
3	*	*	*					
4				*				
5	*	*	*		*			
6	*	*	*		*			
7	*	*	*		*			
8				*		*	*	*
	0	1	2	3	4	5	6	7

Minimalistischer DFA: Zustandspaare (4)



1	*							
2	*	*						
3	*	*	*					
4				*				
5	*	*	*		*			
6	*	*	*		*			
7	*	*	*		*			
8				*		*	*	*
	0	1	2	3	4	5	6	7

Schritt 3.3:

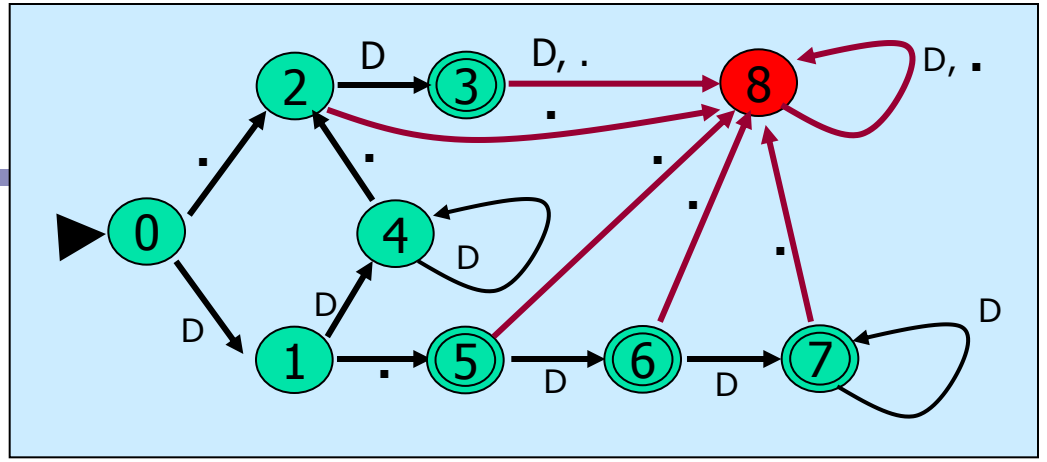
$\{ \delta(3, .), \delta(0, .) \} = \{ 8, 2 \}$
$\{ \delta(3, D), \delta(0, D) \} = \{ 8, 1 \}$

$\{ \delta(3, .), \delta(1, .) \} = \{ 8, 5 \}$ ➔ **ja** ➔ {3, 1}
$\{ \delta(3, D), \delta(1, D) \} = \{ 8, 4 \}$

$\{ \delta(3, .), \delta(2, .) \} = \{ 8, 8 \}$
$\{ \delta(3, D), \delta(2, D) \} = \{ 8, 3 \}$ ➔ **ja** ➔ {3, 2}

1	*							
2	*	*						
3	*	*	*					
4				*				
5	*	*	*		*			
6	*	*	*		*			
7	*	*	*		*			
8				*		*	*	*
	0	1	2	3	4	5	6	7

Minimalistischer DFA: Zustandspaare (5)



Schritt 3.4:

{ δ(4, .), δ(0, .) } = { 2, 2 }
{ δ(4, D), δ(0, D) } = { 4, 1 }

{ δ(4, .), δ(1, .) } = { 2, 5 } ➔ **ja** ➔ {4, 1} ➔ {4, 0}
{ δ(4, D), δ(1, D) } = { 4, 4 }

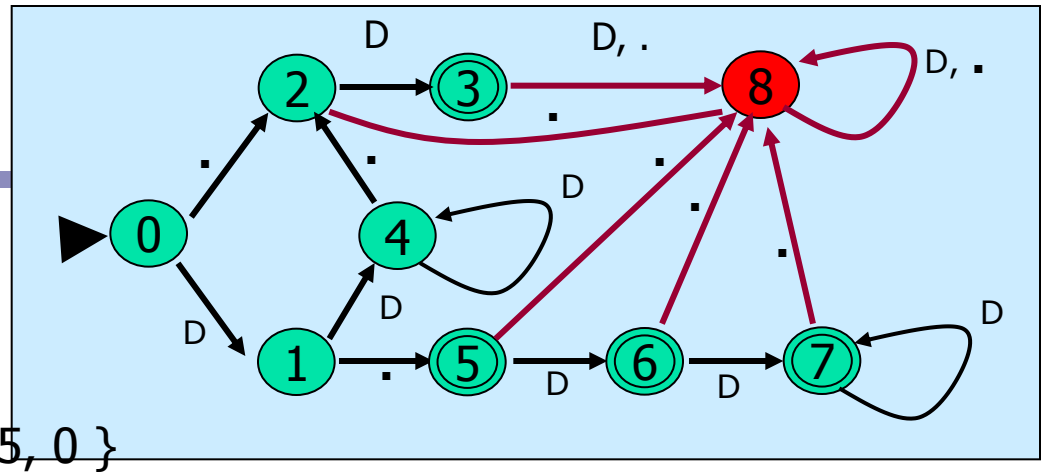
{ δ(4, .), δ(2, .) } = { 2, 8 }

{ δ(4, D), δ(2, D) } = { 4, 3 } ➔ **ja** ➔ {4, 2}

{ δ(4, .), δ(3, .) } = { 2, 8 }
{ δ(4, D), δ(3, D) } = { 4, 8 }

	0	1	2	3	4	5	6	7
1	*							
2	*	*						
3	*	*	*					
4	*	*	*	*				
5	*	*	*		*			
6	*	*	*		*			
7	*	*	*		*			
8				*		*	*	*

Minimalistischer DFA: Zustandspaare (6)



Schritt 3.5:

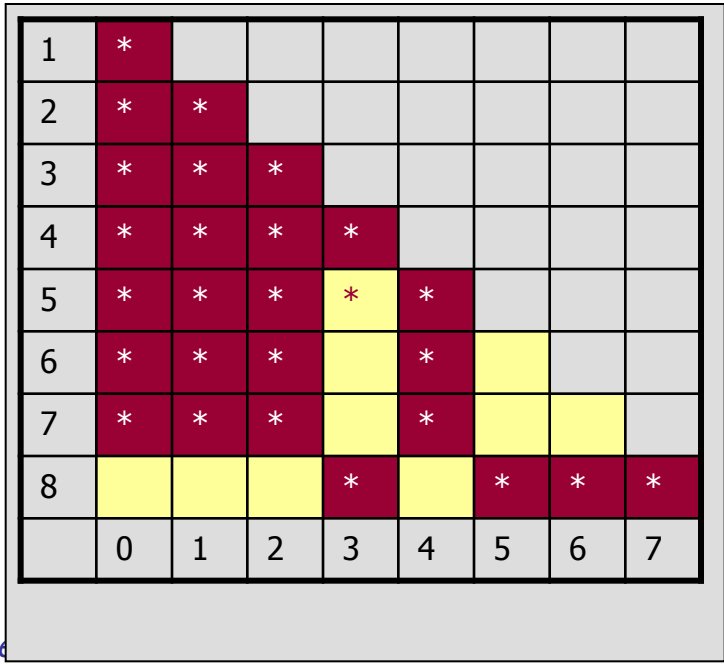
{ δ(5, .), δ(0, .) } = { 8, 2 }
{ δ(5, D), δ(0, D) } = { 6, 1 } ➔ ja ➔ { 5, 0 }

{ δ(5, .), δ(1, .) } = { 8, 5 } ➔ ja ➔ { 5, 1 }
{ δ(5, D), δ(1, D) } = { 6, 4 } ➔ ja ➔ { 5, 1 }

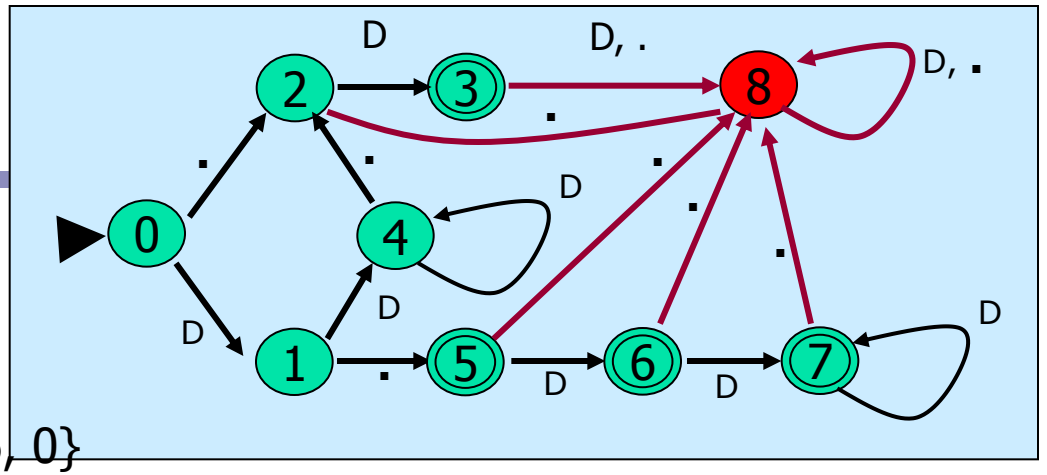
{ δ(5, .), δ(2, .) } = { 8, 8 }
{ δ(5, D), δ(2, D) } = { 6, 3 }

{ δ(5, .), δ(3, .) } = { 8, 8 }
{ δ(5, D), δ(3, D) } = { 6, 8 } ➔ ja ➔ {5, 3}

{ δ(5, .), δ(4, .) } = { 8, 2 }
{ δ(5, D), δ(4, D) } = { 6, 4 } ➔ ja ➔ { 5, 4 }



Minimalistischer DFA: Zustandspaare (7)



Schritt 3.6:

{ δ(6, .), δ(0, .) } = { 8, 2 }
{ δ(6, D), δ(0, D) } = { 7, 1 } ➔ ja ➔ {6, 0}

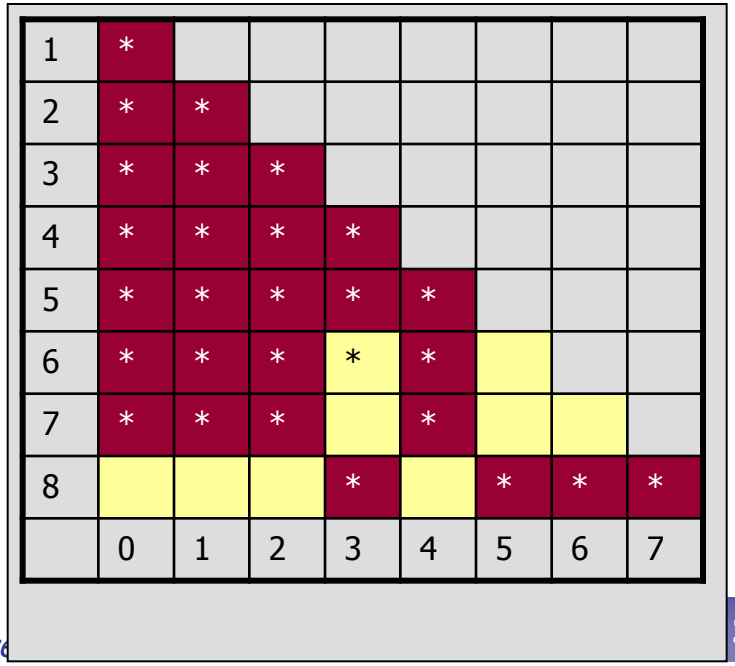
{ δ(6, .), δ(1, .) } = { 8, 5 } ➔ ja ➔ {6, 1}
{ δ(6, D), δ(1, D) } = { 7, 4 } ➔ ja ➔ {6, 1}

{ δ(6, .), δ(2, .) } = { 8, 8 }
{ δ(6, D), δ(2, D) } = { 7, 3 }

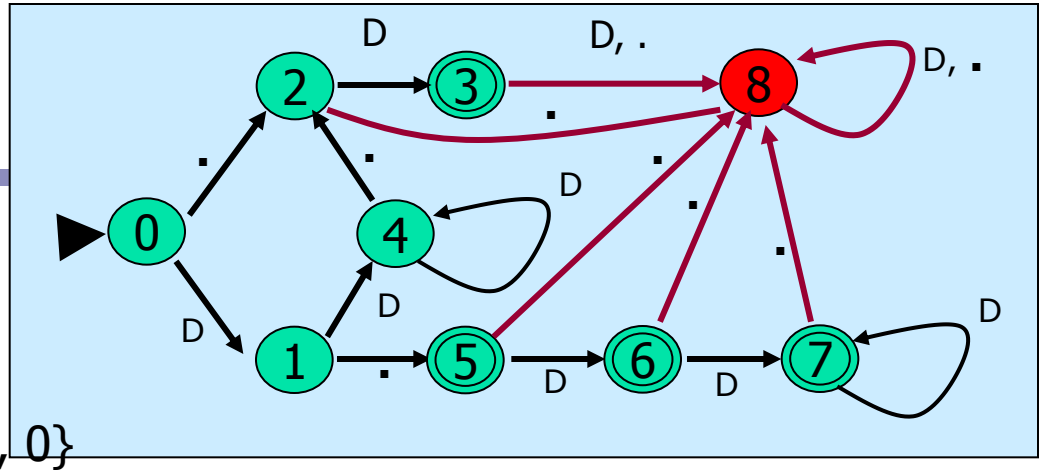
{ δ(6, .), δ(3, .) } = { 8, 8 }
{ δ(6, D), δ(3, D) } = { 7, 8 } ➔ ja ➔ {6, 3}

{ δ(6, .), δ(4, .) } = { 8, 2 }
{ δ(6, D), δ(4, D) } = { 7, 4 } ➔ ja ➔ {6, 4}

{ δ(6, .), δ(5, .) } = { 8, 8}
{ δ(6, D), δ(5, D) } = { 7, 6 }



Minimalistischer DFA: Zustandspaare (8)



Schritt 3.7:

{ δ(7, .), δ(0, .) } = { 8, 2 }
{ δ(7, D), δ(0, D) } = { 7, 1 } ➔ ja ➔ {7, 0}

{ δ(7, .), δ(1, .) } = { 8, 5 } ➔ ja ➔ {7, 1}
{ δ(7, D), δ(1, D) } = { 7, 4 } ➔ ja ➔ {7, 1}

{ δ(7, .), δ(2, .) } = { 8, 8 }
{ δ(7, D), δ(2, D) } = { 7, 3 }

{ δ(7, .), δ(3, .) } = { 8, 8 }
{ δ(7, D), δ(3, D) } = { 7, 8 } ➔ ja ➔ {7, 3}

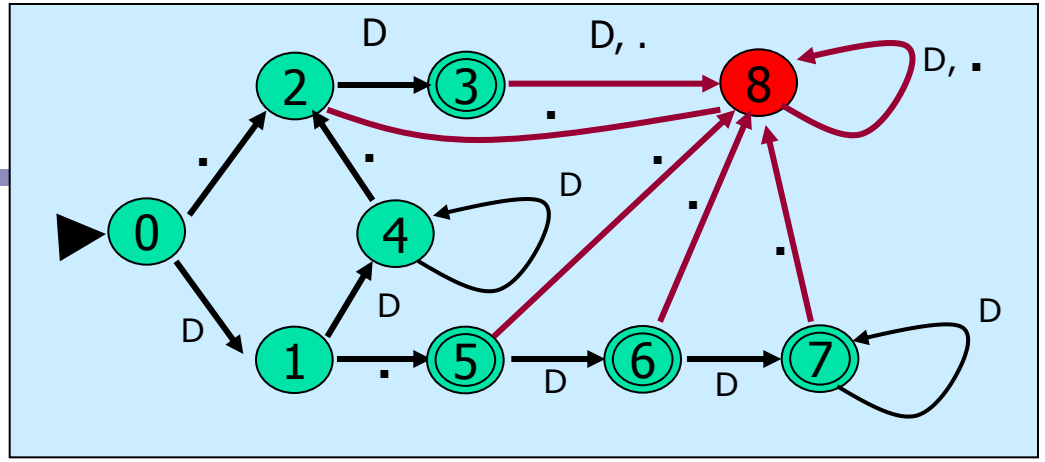
{ δ(7, .), δ(4, .) } = { 8, 2 }
{ δ(7, D), δ(4, D) } = { 7, 4 } ➔ ja ➔ {7, 4}

{ δ(7, .), δ(5, .) } = { 8, 8 }
{ δ(7, D), δ(5, D) } = { 7, 6 }

{ δ(7, .), δ(6, .) } = { 8, 8}
{ δ(7, D), δ(6, D) } = { 7, 7 }

	0	1	2	3	4	5	6	7
1	*							
2	*	*						
3	*	*	*					
4	*	*	*	*				
5	*	*	*	*	*			
6	*	*	*	*	*			
7	*	*	*	*	*			
8				*		*	*	*

Minimalistischer DFA: Zustandspaare (9)



Schritt 3.8:

{ δ(8, .), δ(0, .) } = { 8, 2 }
{ δ(8, D), δ(0, D) } = { 8, 1 }

{ δ(8, .), δ(1, .) } = { 8, 5 } ➔ ja ➔ {8, 1} ➔ {8, 0}
{ δ(8, D), δ(1, D) } = { 8, 4 }

{ δ(8, .), δ(2, .) } = { 8, 8 }
{ δ(8, D), δ(2, D) } = { 8, 3 } ➔ ja ➔ {8, 2}

{ δ(8, .), δ(3, .) } = { 8, 8 }
{ δ(8, D), δ(3, D) } = { 8, 8 }

{ δ(8, .), δ(4, .) } = { 8, 2 } ➔ ja ➔ {8, 4}
{ δ(8, D), δ(4, D) } = { 8, 4 }

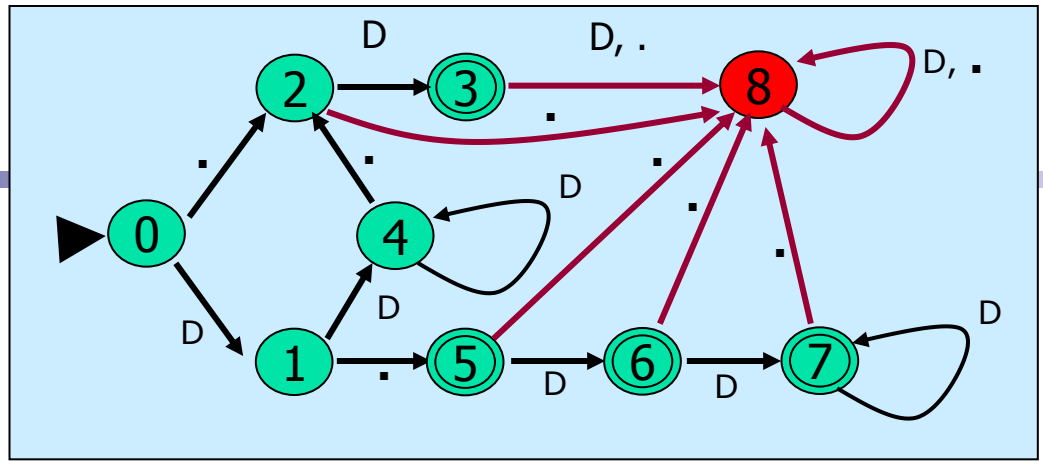
{ δ(8, .), δ(5, .) } = { 8, 8 }
{ δ(8, D), δ(5, D) } = { 8, 6 } ➔ ja ➔ {8, 5}

{ δ(8, .), δ(6, .) } = { 8, 8}
{ δ(8, D), δ(6, D) } = { 8, 7 }

{ δ(8, .), δ(7, .) } = { 8, 8}
{ δ(8, D), δ(7, D) } = { 8, 7 }

	0	1	2	3	4	5	6	7
1	*							
2	*	*						
3	*	*	*					
4	*	*	*	*				
5	*	*	*	*	*			
6	*	*	*	*	*			
7	*	*	*	*	*			
8	*	*	*	*	*	*	*	*

Minimalistischer DFA: Zustandspaare (10)



Schritt 4:

Wiederholung von Schritt 3,
solange bis Tabelle unverändert

1	*							
2	*	*						
3	*	*	*					
4	*	*	*	*				
5	*	*	*	*	*			
6	*	*	*	*	*			
7	*	*	*	*	*			
8	*	*	*	*	*	*	*	*
	0	1	2	3	4	5	6	7

Schritt 5:

7 und 6,
6 und 5
7 und 5 sind zu verschmelzen