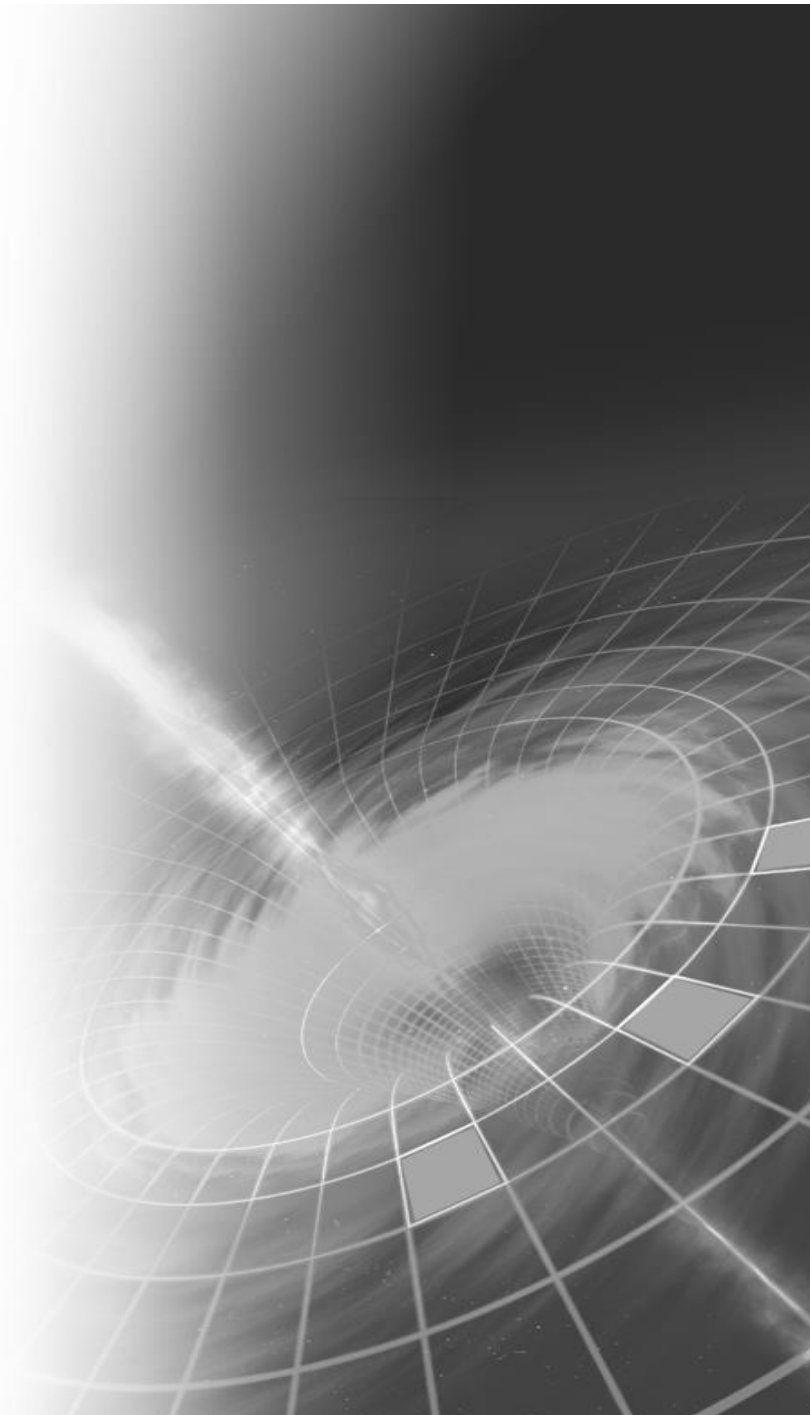
Bachelor-Programm

Compilerbau

im SoSe 2014

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

fischer@informatik.hu-berlin.de



Literatur (Compilerbau)

- Aho, Sethi, Ullman:
  „Compilers: Principles, Techniques und Tools“,
  Addison-Wesley Publishing Company, 1987 und neuere Auflagen
  bekannt als „Drachenbuch“

- Wilhelm, Maurer:
  „Übersetzerbau: Theorie, Konstruktion, Generierung“, Springer Verlag, 2. Auflage

- Fischer, LeBlanc:
  „Crafting a Compiler with C“,
  The Benjamin Cummings Publishing Company, 1991

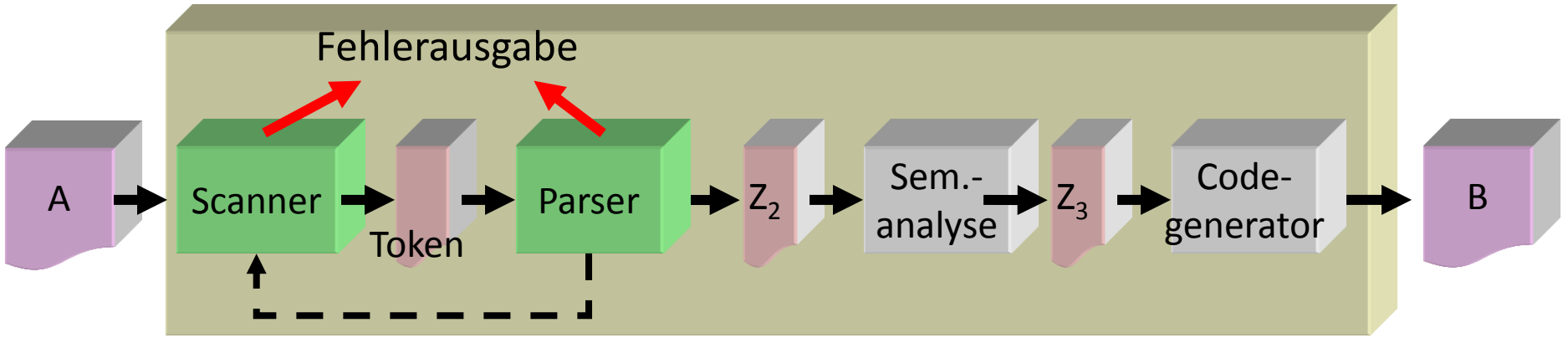
- Holub:
  „Compiler Design in C“,
  Prentice Hall, 1990

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**

Der Scanner



Aufgaben eines Scanners

- Überführung wohldefinierter Zeichenketten des Quellcodes (A) in eine Folge von Token nach Aufforderung durch den Parser
- Ignorierung von Zwischenzeichen: Blank, Tab, Kommentare, CRLF,
- Erkennung lexikalischer Fehler: illegale Zeichen, nicht beendete Kommentare, …
- Positionsermittlung von Symbolen im Quellprogramm: für Fehlerausschriften

Symbol-Klassifikation

- Scanner muss in der Lage sein,
  sehr viele verschiedene Symbole zu erkennen
  ➔ zweckmäßig: Einteilung in endlich viele Klassen

Beispiele: Token-Klasse

Menge aller
- Identifier
- Integer-Konstanten
- Schlüsselworte
- Zeichenketten
- …

- Zeichenfolgen, die als Token erkannt werden:
  als Worte über einem Alphabet Σ:
  xyz12, 125, begin, "abc", <=

- Darstellung von Symbolen: als Paar (Tokenklasse, Tokenwert)
  Beispiele: (ID, "xyz12"),
  (KEYW, "begin"),
  (REL, "LE")

Einrichtung eines Scanners

- Beschreibung zulässiger Token (ad hoc oder formal)

- Scanner-Implementierung in C: Zuordnung eines Codierungswertes je Token-Klasse

```
#define ID= …
#define NUM= …
#define KEYW= …
#define STRING= …
```

- Wert des Token

z.B. Zahlenwerte

unendlich viele Token einer Klasse

z.B. gleichrangige arithmetische Operatoren

endlich viele Token einer Klasse

Symbol-Erkennungsmuster (1)

- Zwischenzeichen (white space)

  <ws> → ' '
  | '\t'
  | <ws> ' '
  | <ws> '\t'

  und weitere

- Schlüsselworte und Operatoren

  als Muster zu spezifizieren: 'do' , 'end'

- Kommentare:

  '/*' und '*/' (C)
  '//' (C++)

Symbol-Erkennungsmuster (2)

- Bezeichner/Identifikatoren

  Zeichen des Alphabets, gefolgt von alphanumerischen Zeichen

- Zahlen
  - ganze Zahlen (Integer):

    Ziffern 0-9, beliebig häufig hintereinander

  - Dezimalzahlen:

    Integer '.' Integer

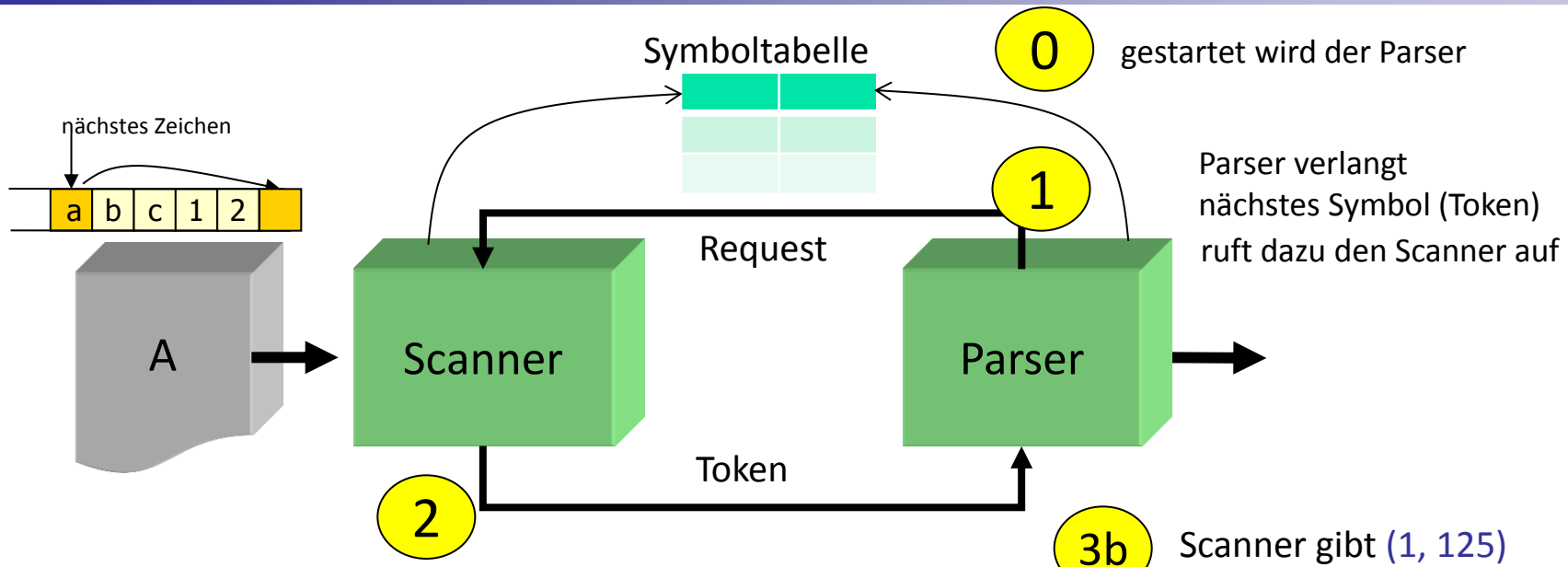
  - Real-Zahlen:

    (Integer- oder Dezimalzahlen) 'E' ('+' oder '-'),
    gefolgt von beliebigen Ziffern

  - komplexe Zahlen: '(' Real ',' Real ')'

wegen Klammerpaar jedoch
problematisches Muster für reguläre Sprache

Verschränkte Arbeitsweise von Scanner und Parser



Ann.:

- Codierung der Tokenklasse id sei als 1 festgelegt
- abc12 sei der 125. gefundene Bezeichner (→ 125. Eintrag in der Symboltabelle)

implementierungsabhängig !

Blackbox-Verhalten eines Scanners

Scanner erkennt Token als atomare Einheiten der Syntax

Beispiel (Scanner-Eingabe-Quelltext):

```
int match0 (char *s)  /*is a zero*/  {
                if ( !strcmp (s, "0.0") )
                return 1;
    }
```

Hinweis:
Programm enthält semantischen Fehler. Welchen?

zufälliger **return**-Wert im **else**-Fall

Aus dem Unix-Manual:

strcmp()

The strcmp() function compares two strings byte-by-byte, according to the ordering of your machine's character set. The function returns an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 respectively.
The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared.

Blackbox-Verhalten eines Scanners

1. Token

```
int match0 (char *s) /*is a zero*/ {
        if (!strcmp (s, "0.0"))
        return 1;
  }
```

Leerzeichen dienen als Trenner zwischen den Token und werden überlesen

nicht immer sind Trenner notwendig!

wird vom Scanner überführt in:

```
INT ID(match0) LPAREN CHAR STAR ID(s) RPAREN
LBRACE IF LPAREN BANG ID(strcmp) LPAREN ID(s) COMMA
STRING(0.0) RPAREN RPAREN RETURN
NUM(1) SEMI RBRACE EOF
```

Blackbox-Verhalten eines Scanners

2. Token

```
int match0 (char *s) /*is a zero*/ {
        if (!strcmp (s, "0.0"))
        return 1;
 }
```

einige Token können Attribute aufweisen

ID führt z.B. den jeweiligen Symboltabellenindex mit

wird überführt in:

```
INT ID(match0) LPAREN CHAR STAR ID(s) RPAREN
LBRACE IF LPAREN BANG ID(strcmp) LPAREN ID(s) COMMA
STRING(0.0) RPAREN RPAREN RETURN
NUM(1) SEMI RBRACE EOF
```

Blackbox-Verhalten eines Scanners

8. Token

```
int match0 (char *s) /*is a zero*/ {
        if (!strcmp (s, "0.0"))
        return 1;
 }
```

Kommentare werden erkannt
und ignoriert
(zuweilen aber doch berücksichtigt)

wird überführt in:

```
INT ID(match0) LPAREN CHAR STAR ID(s) RPAREN
LBRACE IF LPAREN BANG ID(strcmp) LPAREN ID(s) COMMA
STRING(0.0) RPAREN RPAREN RETURN
NUM(1) SEMI RBRACE EOF
```

Blackbox-Verhalten eines Scanners

Betriebssystemkonvention
(Unix- Vorbild moderner BS)

Letztes Token

```
int match0 (char *s) /*is a zero*/ {
        if (!strcmp (s, "0.0"))
        return 1;
  }□
```

nicht existentes Zeichen

EOF als
Ende der Eingabe

wird überführt in:

Wert, der nicht als Zeichen (char) dargestellt werden kann: -1

```
INT ID(match0) LPAREN CHAR STAR ID(s) RPAREN
LBRACE IF LPAREN BANG ID(strcmp) LPAREN ID(s) COMMA
STRING(0.0) RPAREN RPAREN RETURN
NUM(1) SEMI RBRACE EOF
```

Bau eines Ad-hoc-Scanners

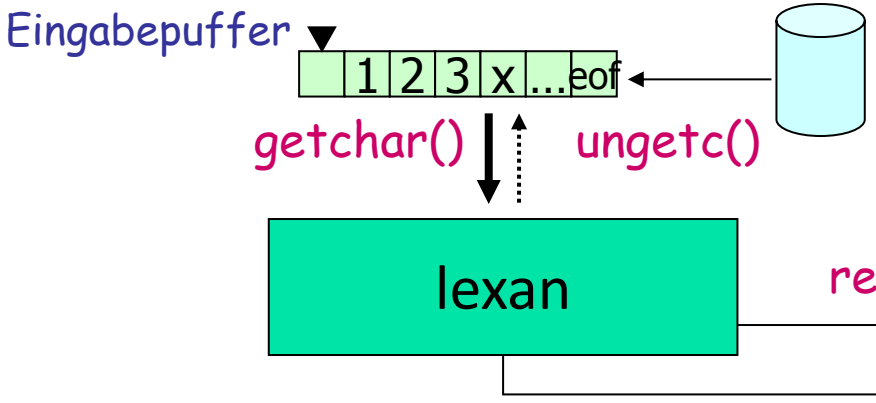
wegen EOF-Problematik
liefert getchar() int-Wert

realisiert als C-Funktion (lexan)

- erkennt (zunächst nur) Integer-Zahlenwerte als Token mit numerischem Wert: NUM

  → müsste später zum kompletten Scanner ausgebaut werden

- überliest Trennzeichen, zählt Eingabezeilen
- stellt unpassendes Zeichen zurück, es könnte der Anfang eines nächsten Token darstellen



- versuchen Teilzeichenketten mit maximaler Länge als gültige Token zu identifizieren
- müssen immer ein Zeichen mehr lesen (das nicht zum Token gehört)

NUM als codierter Tokentyp

Wert der Ziffernfolge als Attribut von NUM

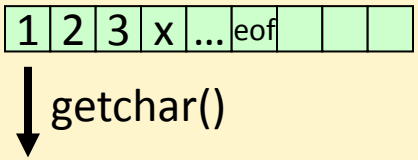
Auszug ascii-Tabelle

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

ascii + eof
möge Alphabet
unserer Beispielsprache sein

numerischer Wert einer Ziffer c:
Wert(c)= c - 48

Beispieleingabe:



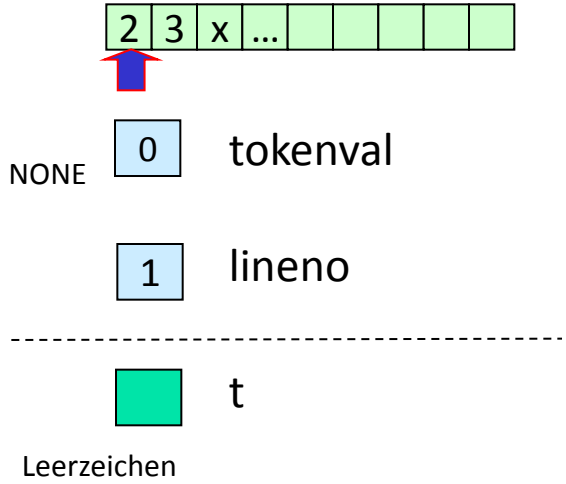
numerischer Wert
der Ziffernfolge 1,2,3 = 123

```c
#include <stdio.h>
#include <ctype.h>
#define NUM = 256      // Codierung des Tokens für Integer-Werte
#define NONE= 0        // Codierung eines unbelegten Token-Attributwertes

int lineno= 1;         // aktuelle Zeilennummer bei der Quelltextanalyse
int tokenval= NONE;    // aktueller Token-Attributwert

int lexan()            // Rückgabewert: NUM oder nichtinterpretierbares Z.
{
    int t;             // aktuelles Eingabezeichen
    while (1) {
        t= getchar();
        if (t== ' ' || t== '\t')
            ; // Leerzeichen u. Tabulator ignorieren
        else if (t== '\n')
            lineno++;
        else if (isdigit(t)) {
            tokenval= t - '0';  //numerischer Zeichenwert
            t= getchar();
            while (isdigit(t)) {
                tokenval= tokenval*10 + t – '0';
                t= getchar();
            }
            ungetc (t, stdin); // Zeichen gehört nicht zur Zahl
            return NUM;
        }
        else {
            tokenval= NONE;
            return t;
        }
    }
}
```

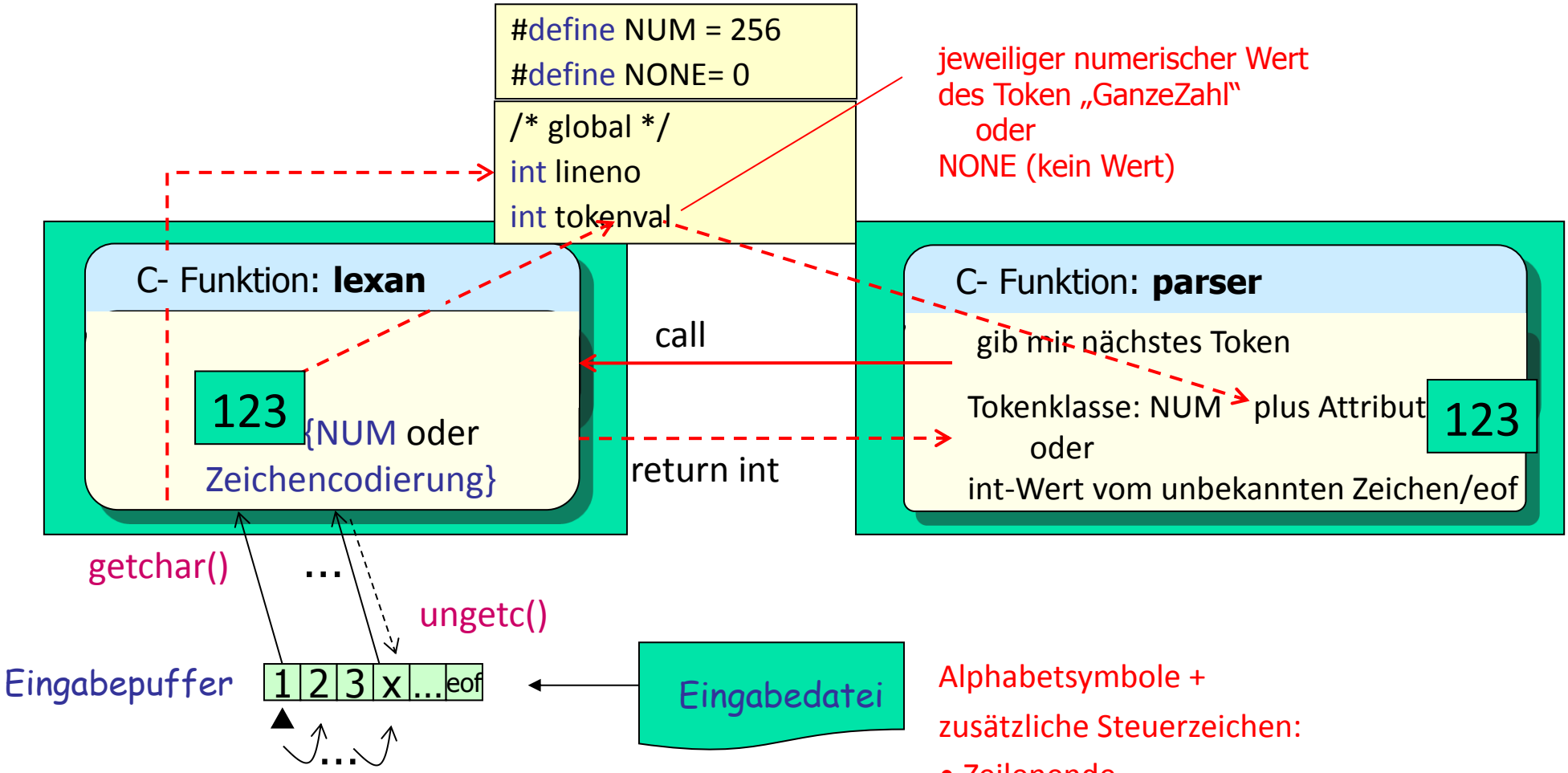
C++ Programm: **lexan**

zusätzliche Steuerzeichen:

- Zeilenende
- File-Ende



Kopplung von Scanner und Parser: Token



Zwischenfazit

- Ad-hoc-Lösungen für Scanner sind durchaus denkbar
  - letztes Beispiel: Erkennung ganzer Zahlen

unser Ziel ist aber …

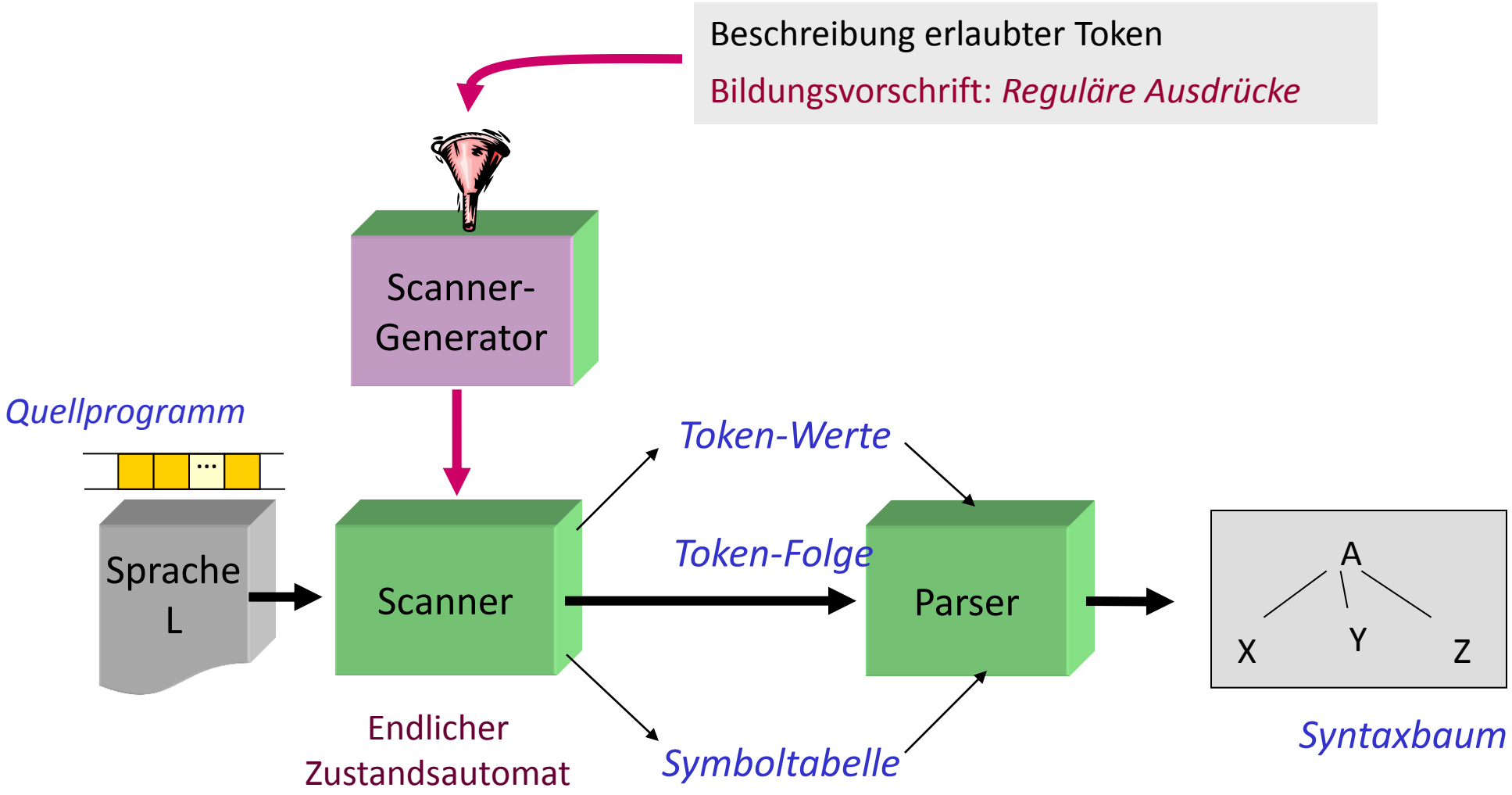
(1) strukturierte Lösung

- Spezifikation lexikalischer Token (als Muster von Zeichenfolgen) in einer formalen Notation: Reguläre Ausdrücke

- Implementation des Scanners: Zustandsautomaten und Reguläre Ausdrücke
- mathematische Verbindung der Kalküle
- Beispiel eines einfachen tabellengesteuerten Scanners als strukturierte Lösung

(2) Technologie zur automatischen Generierung von Scannern

Unser Ziel: ein generischer Scanner



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**

Theoretische Basis der lexikalischen Analyse

Wichtige (aus der Theorie bekannte) Zusammenhänge:

Token – Reguläre Sprachen – Reguläre Grammatiken - Automaten

für sämtliche Eingangssprachen gilt:

- die von einem Scanner erkannten lexikalischen Einheiten (Token) bilden eine nichtleere (sogenannte) Reguläre Sprache .
- Reguläre Sprachen können durch Reguläre Ausdrücke beschrieben werden .
- Reguläre Sprachen können durch endliche Zustandsautomaten erkannt werden .

Reguläre Ausdrücke - Erkennungsmuster

... werden als Ausdrücke einer Regulären Sprache über einem Alphabet Σ entweder

durch

- a) Reguläre Grammatiken (Diskussion später) oder durch
- b) Reguläre Ausdrücke

definiert.

damit ist **nicht** die Quellsprache selbst gemeint!

vielmehr
die Sprache zur Charakterisierung der Token einer beliebigen Quellsprache

Ziel

Charakterisierung einer (möglicherweise unendlichen) Sprache,

die ein endliches Alphabet Σ besitzt (z.B. Σ = acii)

per **Klassifikation** gültiger Zeichenfolgen,
wobei
ein Regulärer Ausdruck eine Menge gültiger Zeichenfolgen
in Form eines Musters repräsentiert

Reguläre Ausdrücke

Basisregeln: induktive Definition eines Regulären Ausdrucks (RA)

1. Epsilon: ε (das leere Wort) ist ein RA der Sprache L, der die Menge $\{\varepsilon\}$ spezifiziert
2. Symbol: Ist $a \in \Sigma$,
   dann ist a ein RA der Sprache L, der die Menge {a} spezifiziert ($L(a) = \{a\}$)

Induktionsregeln: seien r und s RA, die L(r) und L(s) spezifizieren, dann ist …

1. Alternative: $(r \mid s)$ ist ein RA, der die Sprache $L(r) \cup L(s)$ spezifiziert.
2. Verkettung: $(r \cdot s)$ ein RA, der die Sprache $L(r) \circ L(s)$ spezifiziert.
3. Wiederholung: $(r)^*$ ist ein RA, der die Sprache $L(r)^*$ spezifiziert.
4. Klammerpaare: (r) ist ein RA, der die Sprache L(r) spezifiziert.

Vorrangeln: Reguläre Ausdrücke

Konvention: falls Präzedenzen für die Operatoren vorliegen,
können die Ausdrucksklammern weggelassen werden

Vorrangregeln (übliche Reihenfolge)

1. Wiederholung (Kleen'scher Hüllenoperator), linksassoziativ
2. Verkettung, linksassoziativ
3. Alternative, linksassoziativ

Beispiel: **(a) | ((b)*(c))** und **a | b* c**

beschreiben dieselben Sprachen

Beispiele: Reguläre Ausdrücke

algebraische Eigenschaften von RAs

Beispiele: RAs für Σ ={ a, b}

RA **a*** beschreibt die Sprache {**ε, a, aa, aaa**,...}

RA **((a | b) a)** beschreibt {**aa, ba**}

RA **((a | b) ε)** beschreibt {**a, b**}

RA **((a | b) a)*** beschreibt {**ε, aa, ba, aaaa, baaa, aaba, baba, aaaaaa**, ...}

RA **(a | b) (a | b)** beschreibt {**aa, ab, ba, bb**},

d.h. **(a|b)(a|b)= aa|ab|ba|bb**

RA **(a | b)*** beschreibt die Menge aller Zeichenketten aus **a**'s und **b**'s (einschließlich **ε !**) bestehen,

d.h. **(a|b)* = (a*b*)***

Anwendungsbeispiele Regulärer Definitionen

- Bezeichner
  - letter → (**a**|**b**|**c**|…|**z**|**A**|**B**|**C**|…|**Z**)
  - digit → (**0**|**1**|**2**|…|**8**|**9**)
  - id → letter (letter|digit)*
- Zahlen
  - integer → (**+**|**-**|**ε**) (**0**| (**1**|**2**|**3**|…|**9**) digit*)
  - decimal → integer **.** (digit)*
  - real → (integer | decimal) **E** (**+**|**-**) digit*
  - complex → **(** real **,** real **)**

Token-Definition:
übliche Notation regulärer Ausdrücke in grammatikalischer Produktionsschreibweise

Token können jedoch in ihren Strukturen noch komplexer aufgebaut sein

- Erkenntnis: die meisten Token können mittels RA spezifiziert werden

Kurznotation Regulärer Ausdrücke

a	gewöhnliches Zeichen
ε	leere Zeichenkette
s | r	Alternative
sr	Schreibweise für Verkettung
s*	Wiederholung (null oder beliebig oft)
s^+	Wiederholung (einmal oder beliebig oft)
s?	höchstens einmaliges Auftreten
[a-zA-Z]	Zeichenalternative

Algebraische Eigenschaften Regulärer Ausdrücke

Axiome	Beschreibung
$r \mid s = s \mid r$	Kommutativität von Alternative
$r \mid (s \mid t) = (r \mid s) \mid t$	Assoziativität von Alternative
$(rs)\ t = r\ (st)$	Assoziativität von Verkettung
$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$	Distributivität von Verkettung und Alternative
$\varepsilon\ r = r$ $r\ \varepsilon = r$	neutrales Element der Verkettung
$r^* = (r \mid \varepsilon)^*$	Beziehung zwischen * und ε
$r^{**} = r^*$	* ist in sich abgeschlossen

Reguläre Sprachen

Satz (aus der Theorie)

Die Menge der durch reguläre Ausdrücke beschreibbaren Sprachen
ist genau die Menge der Regulären Sprachen

Einschränkung der Mächtigkeit von RAs

Reguläre Ausdrücke sind unbrauchbar
zur Beschreibung paariger oder verschachtelter Konstrukte !

(werden später mächtigere Beschreibungsmittel, als die der Regulären Sprachen, anwenden:
kontextfreie Grammatiken)

Erkennungsformalismus für RAs

Frage:

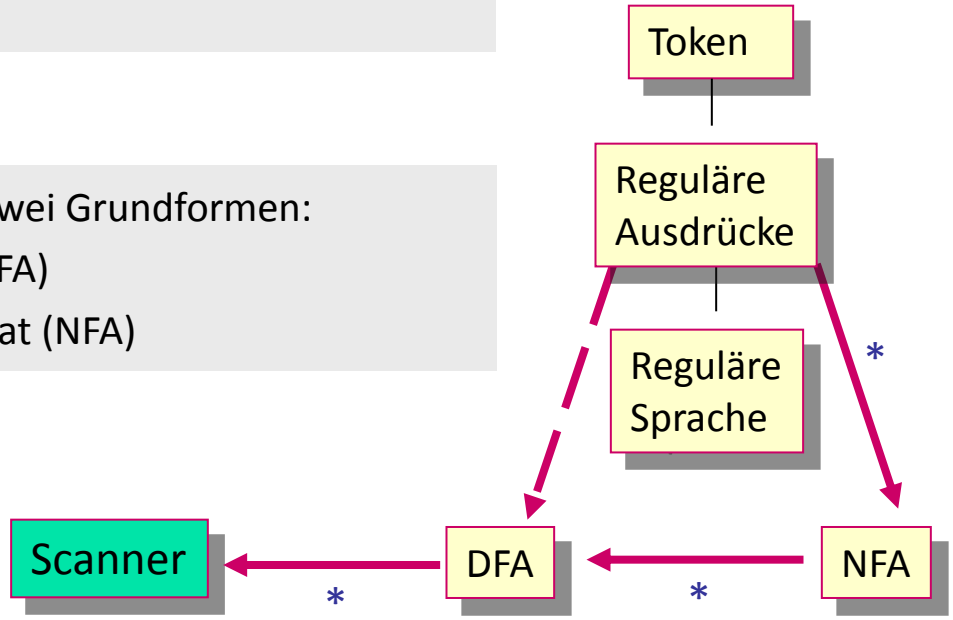
Gibt es einen Formalismus,
der eine Spracherkennung für RAs erlaubt?

Antwort: Ja

Endliche Automaten (engl.: Finite Automata) in zwei Grundformen:

(1) deterministischer endlicher Automat (DFA)

(2) nichtdeterministischer endlicher Automat (NFA)



* nach Regeln

Position

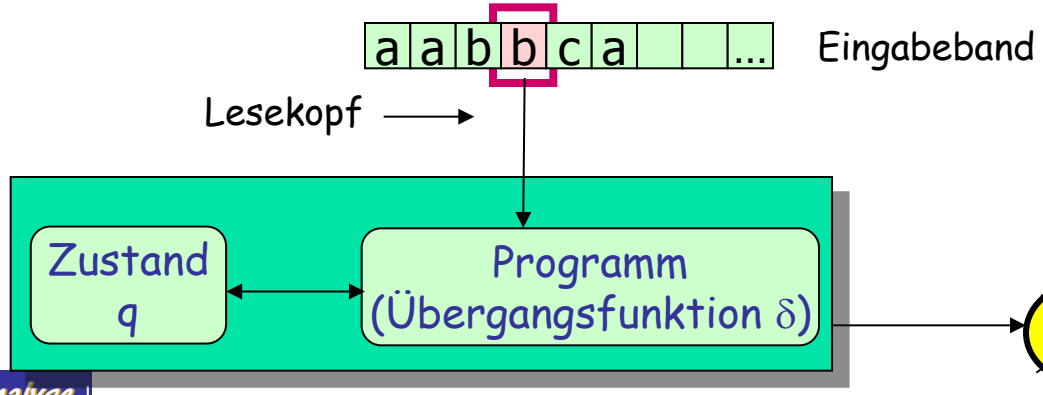
- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- **Kapitel 3**
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parsergeneratoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Ausblick: Laufzeitsystem,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

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

Was ist ein endlicher Automat ?

Merkmale (informal)

- sehr beschränkte Speicherfähigkeit:
  eine Variable (Grundzustand),
  die nur endlich verschiedene Werte (eines Aufzählungstyps) annehmen kann
- ein Lesekopf,
  mit dem das Eingabeband von links nach rechts gelesen werden kann
- programmierbare Funktionalität,
  die durch eine Zustandsübergangsfunktion beschrieben ist (prinzipieller Unterschied: DFA, NFA)
- Unterschied zwischen theoretischen Automaten und Realisierung (Erkennen des Eingabeendes)



Theoretischer Automat legt
nicht fest, wie das Ende einer Eingabe
erkannt wird
(kann auf EOF-Details verzichten)

Signalgeber bei Erreichen
eines akzeptierenden Endzustandes
(ja/nein)

Deterministischer endlicher Automat

Def: deterministischer endlicher Automat

DFA= (**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 $\Sigma \rightarrow$ **S**
  ordnet einem Zustand in Abhängigkeit des **aktuellen** Eingabesymbols genau einen Folgezustand zu
  (vollständig!!!, d.h. alle Kombinationen sind definiert)
- eine Menge von akzeptierenden Finalzuständen **F** $\subseteq$ **S**

Arbeitsweise eines DFA

DFA= (**S**, Σ, **δ**, s_0, **F**)

arbeitet schrittweise

- initial
  - Eingabewort steht auf dem Eingabeband (Lesekopf auf erstem Zeichen)
  - Signalgeber aus
  - s_0 ist aktueller Zustand

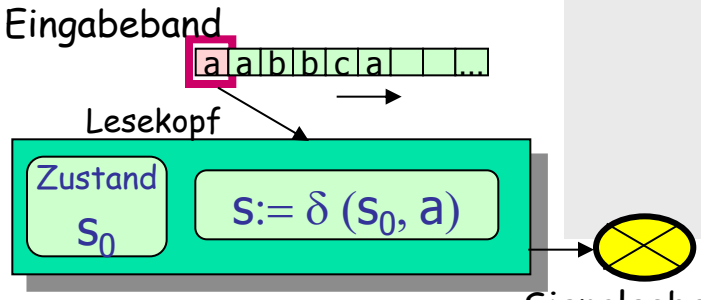
- in jedem Schritt
  1. Lesen des aktuellen Zeichens
  2. Versetzen des Lesekopfes um eine Position nach rechts
  3. Berechnung des Folgezustandes

     Fall a): neuer Zustand ist finaler Zustand
     und Eingabe komplett gelesen→ Signalgeber ein und Stopp

     Fall b): neuer Zustand ist finaler Zustand
     und Eingabe noch nicht komplett gelesen→ nächster Schritt

     Fall c): neuer Zustand ist kein finaler Zustand
     und Eingabe komplett gelesen→ Stopp (FEHLER)

     Fall d): neuer Zustand ist kein finaler Zustand
     und Eingabe noch nicht komplett gelesen→ nächster Schritt



$w \in \Sigma^*$

möglicher Trace: $<s_o,w> \rightarrow \ldots \rightarrow <s_x,\varepsilon>$

Spezifikation der Übergangsfunktion

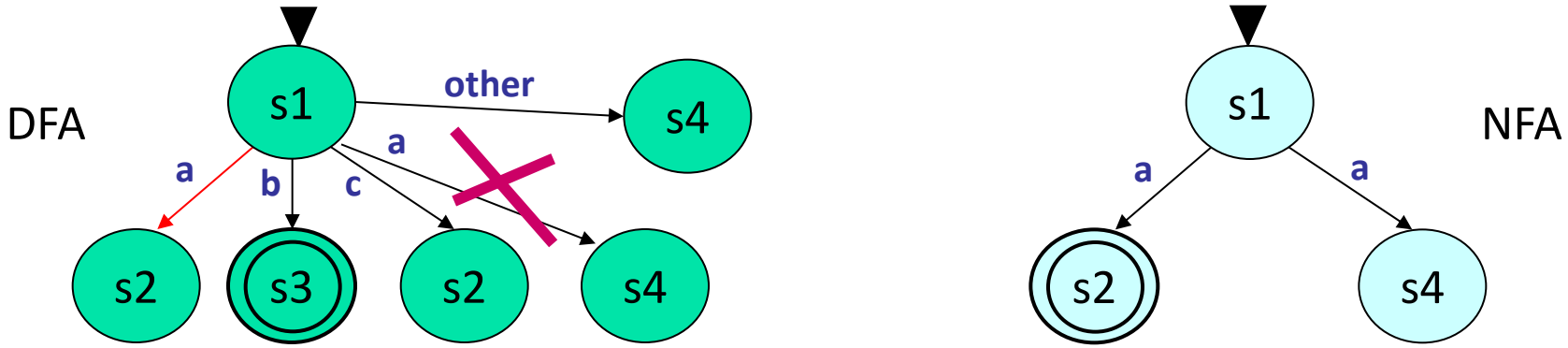
…als Zustandsgraph (**gerichteter Graph)**

- Knoten sind Zustände

  Startzustand, akzeptierende Finalzustände werden besonders markiert

- gerichtete Kanten sind Zustandsübergänge

  von s1 geht eine mit **a** gekennzeichnete Kante nach s2 , falls δ(s1, **a**)= s2 zutrifft



DFA für Dezimalzahlen

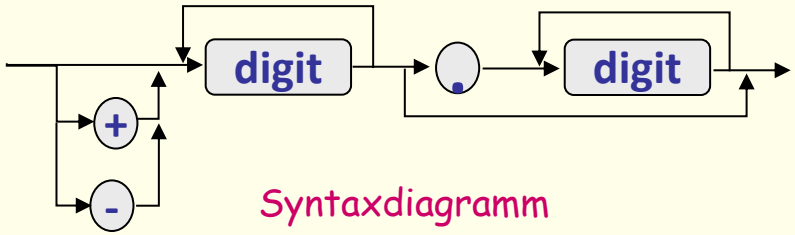
Unser Beispielautomat verklemmt sich bei Eingabe von 19..24.
Er ist nicht vollständig implementiert.
Es fehlt ein (FEHLER-)Zustand mit Übergängen, getriggert durch unzulässige Zeichen.

Beispiel: DFA mit partieller Zustandsübergangsfunktion

Σ = { +, -, ., **digit** }

Zur Vereinfachung: **digit** sei bereits Terminalsymbol

$w \in \Sigma^*$ beliebiges Eingabewort **FRAGE**: $w \in L(r)$?

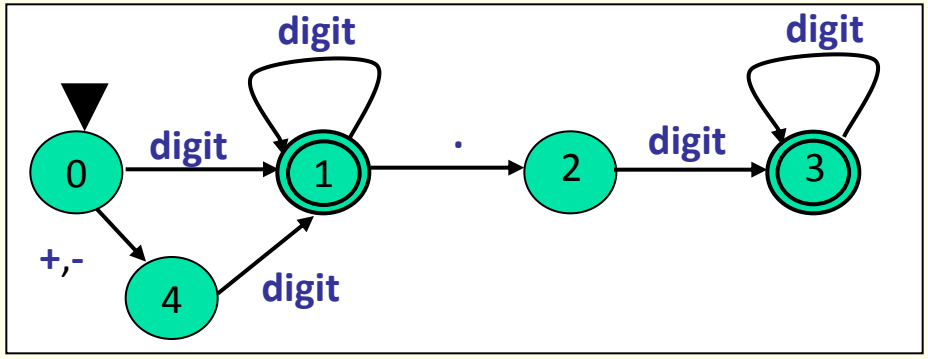


Syntaxdiagramm

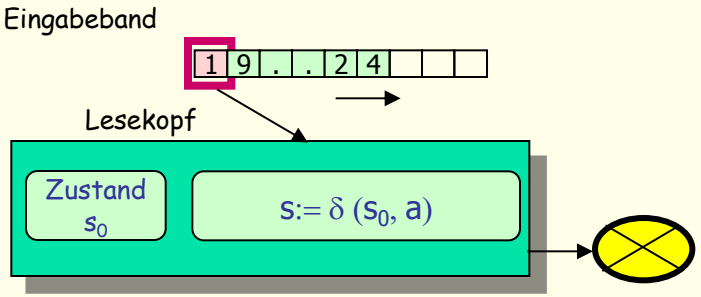
r= (+|-|ε) digit$^+$ (ε |.digit$^+$)

r= ([0-9]$^+$"."[0-9]*) | "."[0-9]$^+$)

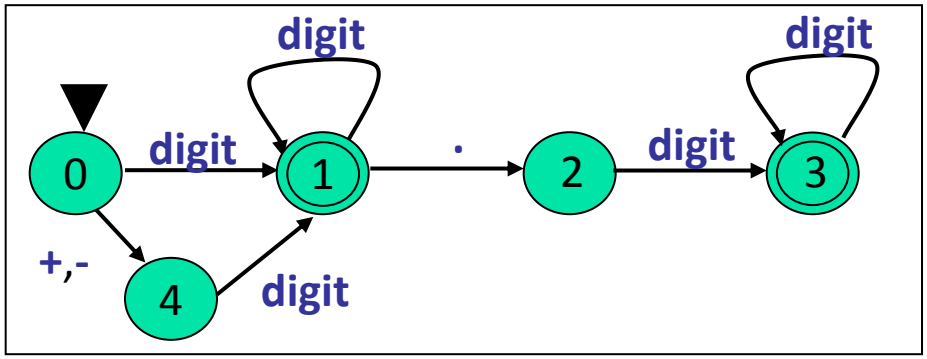
alternativer RA (erweiterte Menge von Terminalsymbolen)

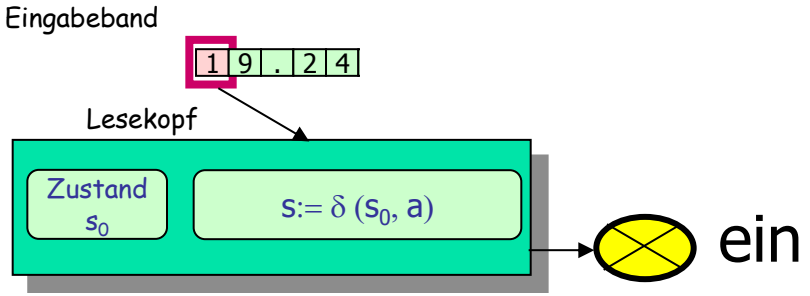


akzeptiert z.B.: w= "**19.24**"



Trace eines DFA





Trace der Verarbeitung für verschiedene Eingaben

1) $\delta(s_0,19.24) \Rightarrow \delta(s_1,9.24) \Rightarrow \delta(s_1,.24) \Rightarrow \delta(s_2,24) \Rightarrow \delta(s_3,4) \Rightarrow \delta(s_3, \varepsilon) \Rightarrow s_3$

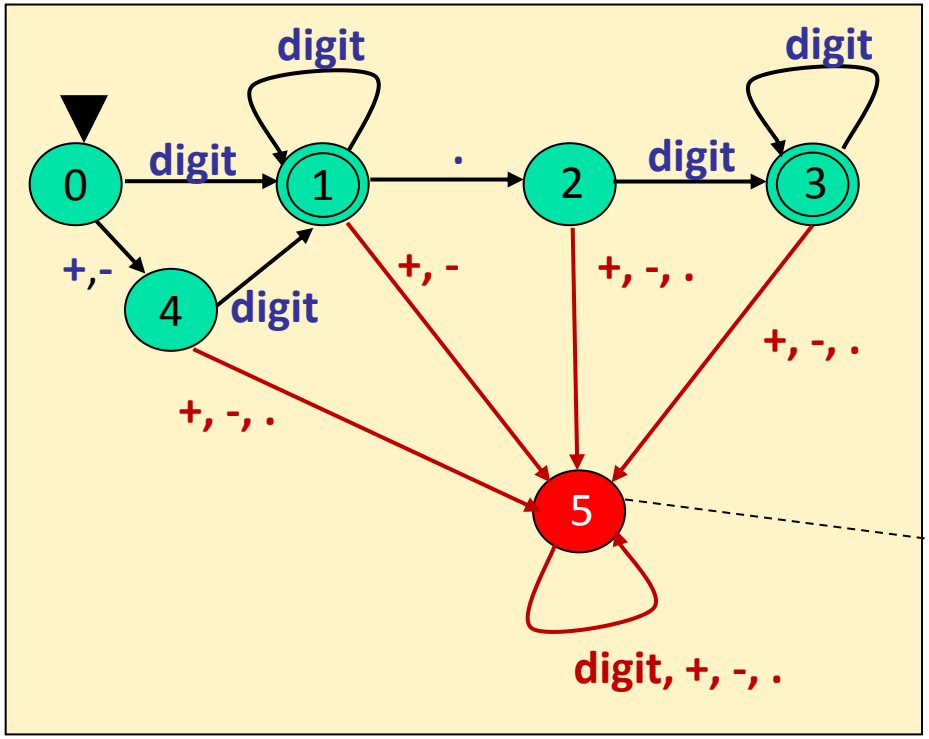
da $s_3 \in F$ ist und Eingabe komplett, wird w= "**19.24**" akzeptiert

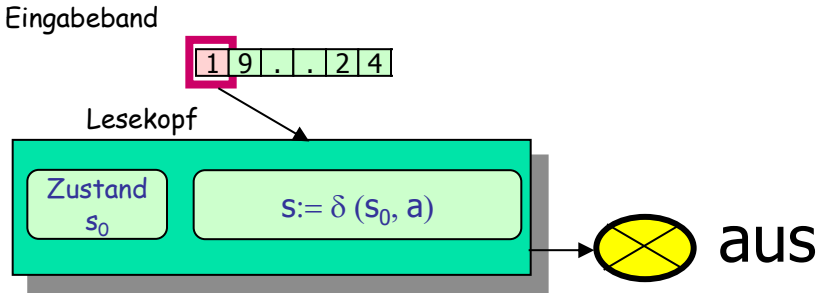
2) $\delta(s_0,19..24) \Rightarrow \delta(s_1,9..24) \Rightarrow \delta(s_1,..24) \Rightarrow \delta(s_2,.24)$

Problem $\delta(s_2,.24)$ ist nicht definiert ➔ unvollständige Automatendefinition

Forderung: falls Eingabe komplett gelesen, darf w= "**19..24**" nicht akzeptiert werden

Komplettierter DFA



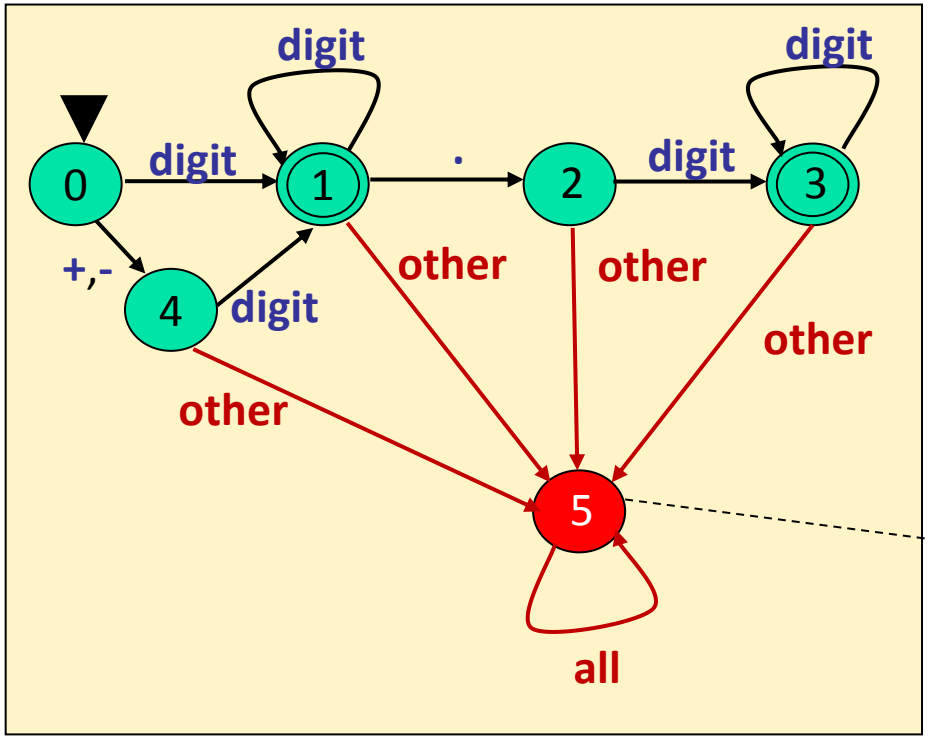


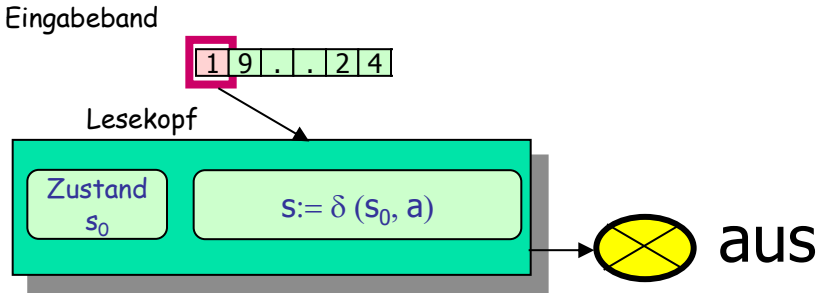
zusätzlicher Fehlerzustand

Σ = { +, -, ., **digit**, ' ' }

Frage: Kurznotation?

Komplettierter DFA (in Kurznotation)





zusätzlicher Fehlerzustand

Σ = { +, -, ., digit, ' ' }

Frage: Wie ist der DFA zu erweitern, so dass er Leerzeichenfolgen (Länge >=1) als Trenner zwischen je zwei Dezimalzahlen in einer Eingabe akzeptiert?
Der Automat soll auf diese Weise Folgen von Dezimalzahlen verarbeiten können.

Aufgabe eines DFA: Worterkennung

Satz (aus der Theorie): Worterkennung durch endlichen Automaten (DFA)

Eine Eingabezeichenkette x wird als Wort einer **regulären Sprache** von einem endlichen Automaten erkannt,

wenn ein Pfad in seinem Transitionsgraphen existiert, der

- im Startzustand s_0 beginnt und
- in einem seiner akzeptierenden Zustände so endet, dass

  die Sequenz der durchlaufenden Kanten genau x ergibt

Gilt bereits für Automaten, die nicht mehr als ja (accept) oder nein sagen können
(das sind die, welche in der Theorie betrachtet werden)

spannende Frage:
Kann man für jeden bel. RA
einen DFA als Wort-Akzeptor finden ?

Aufgabe eines DFA: Worterkennung

Satz (aus der Theorie): Zusammenhang zwischen RA und DFA

Jede Sprache, die durch einen RA definiert wird,
wird auch durch einen deterministischen endlichen Automaten definiert,
der diesen RA erkennt (akzeptiert).

Weitere spannende Fragen:

Kann man den RA-Akzeptor auch konstruieren ?

Wenn ja,
Kann man den RA-Akzeptor auch effizient konstruieren ?

Kann man,
einfache DFAs als RA-Akzeptoren zu Akzeptoren komplexerer RAs zusammensetzen ?

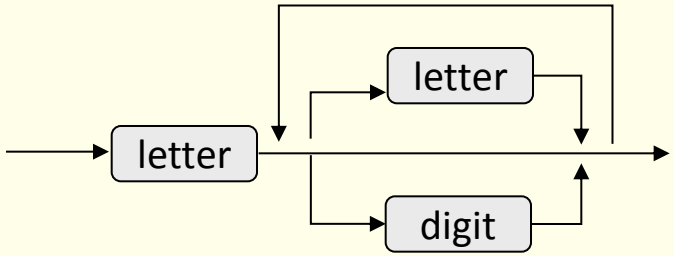
Erkennung von Identifier-Token

Beispiel: (Erkennungs-) Automaten für vereinfachte Identifier

Σ ={ a, ..., z, 0, ...,9}

$w \in \Sigma^*$ beliebiges Eingabewort

FRAGE: $w \in L(r)$?

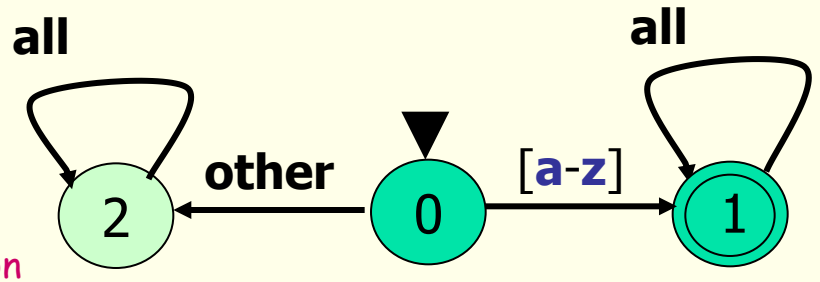


r= [a-z] ([a-z] | [0-9])*

Syntaxdiagramm

Automat akzeptiert z.B.:
w= "**abc3**"

aber nicht
w= "**007agent**"

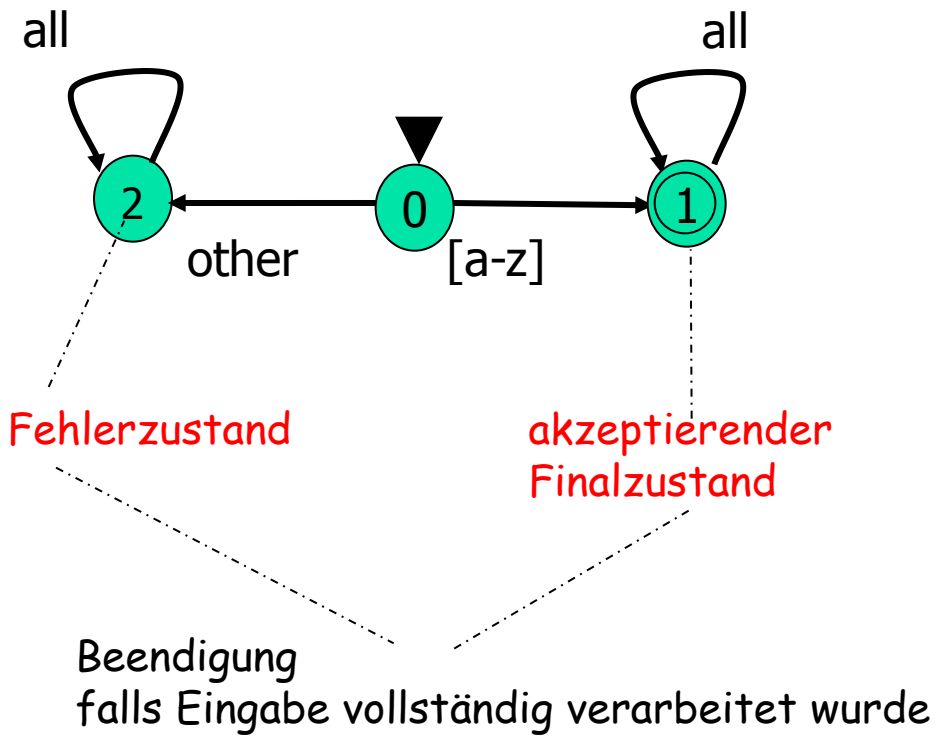


liest Eingabe komplett

totale Zustandsübergangsfunktion

Zustandsübergangstabelle eines DFA

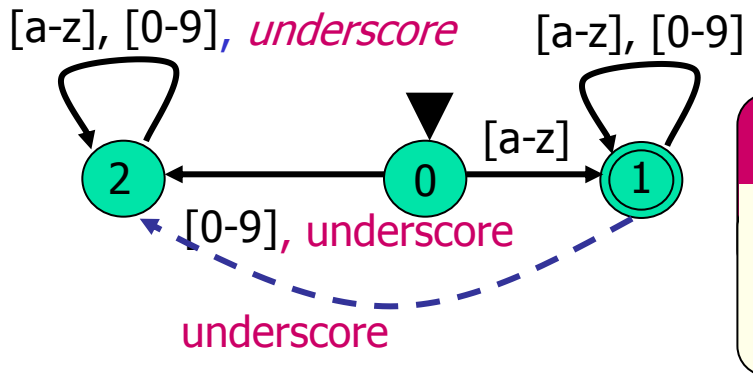
Ausgangs zustand	Eingabezeichen		Akzeptanz
	a-z	0-9	
0 (start)	**1**	**2**	nein
1 (accept)	**1**	**1**	ja
2 (error)	**2**	**2**	nein



Übergang: partielle → totale Übergangsfunktion

Beispiel: Zunahme des Underscore-Zeichens

Σ= {**a**, ...,**z** , **0**,..., **9**, **_** }, w= "**abc3_x**"



nötige Erweiterung:
bisheriger DFA ist unterspezifiziert,
(DFA muss Eingabe immer komplett lesen können)

Satz: Vervollständigung partieller Übergangsfunktionen

Jede partielle Übergangsfunktion eines DFA A
kann in eine totale Übergangsfunktion eines Automaten B
mit L(A) = L(B) überführt werden.

DFA befindet sich nach vollständigem Lesen im **nicht-akzeptierenden** Zustand s_2
→ **abc3 x** ist damit eine ZK, die **nicht** dem vorgeg. RA entspricht

vereinfachende Konvention:
Vervollständigung der δ-Funktion kann **nachträglich** immer vorgenommen werden
(Fehler-Zustände werden deshalb häufig zunächst vernachlässigt)

Theoretischer Automat – Praktischer Automat

Theoretischer Automat

- kümmert sich nicht um seine Realisierung (z.B. Erkennung des Ende einer Eingabe)
- kennt a priori die Länge des zu analysierenden Wortes
- jeder Zustandsübergang konsumiert ein Zeichen

Praktischer Automat (für lexikalische Analyse)

- kennt nicht a priori die Länge der Eingabe

  fordert statt dessen ein besonderes Eingabezeichen zur Ende-Erkennung: EOF
  (kein Element des Alphabets)

- liefert kein reines (0,1)-Resultat,
  sondern kann „akzeptierende“ Aktionen ausführen (z.B. Token mit Attributen liefern)

- kann Teilstrings akzeptieren,
  d.h. akzeptiert die jeweils längstmögliche Zeichenfolge für das Token

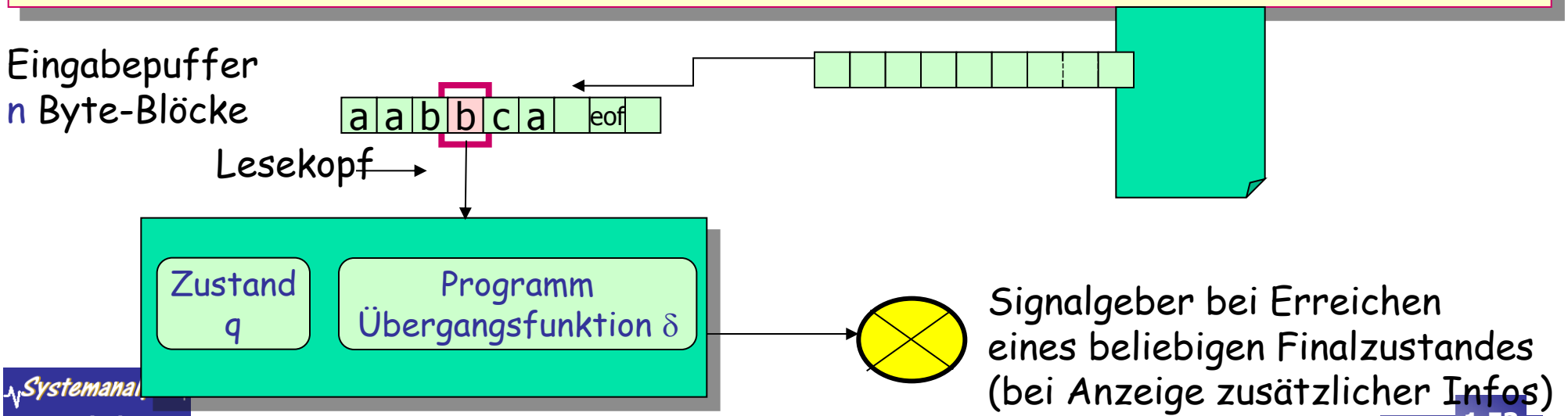
- bei Eingabe einer ZK als Folge von Token muss u.U. der Lesekopf um eine Position zurückgesetzt werden können,

  falls es zur nächstfolgenden Token-Bildung beitragen kann
  (Konvention: Zustand wird besonders zu markieren sein, z.B. mit *)

Praxisrelevanter DFA

DFA-Erweiterungsmerkmale

1. kehrt in einen Finalzustand zurück **ohne** Eingabe komplett gelesen zu haben und stoppt
2. liefert beim Stopp ein Rückgabesignal mit optionalen zusätzlichen Informationen: Attribute, Aktionen
3. erkennt
   - Token entsprechend vorgeg. RA-Muster (bei Abdeckung sämtlicher erlaubter Alphabet-Kombinationen)
   - EOF
   - undefinierte Zeichen (außerhalb des Alphabetes)
4. bei wiederholtem Start wird letzte Position des Lesekopfes übernommen (Bearbeitung von Token-Folgen)



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	
Fehlererkennung	→ DFA_6	ERROR

Komposition als komplexer Automat

DFA soll ZK einlesen und erkennen,
ob es sich um eines der obigen Tokenklassen handelt

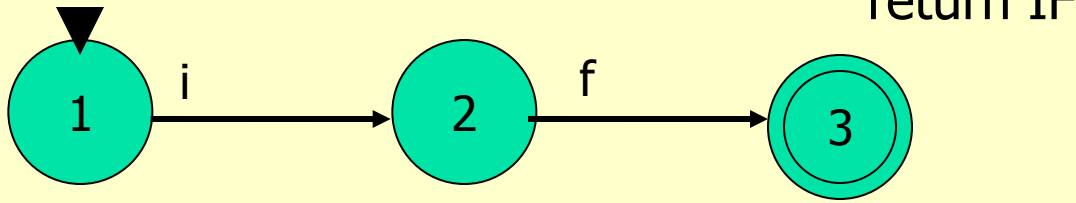
Kombination gelingt aber nur mit Intuition -

also: noch kein strukturierter Ansatz

Erkennung lexikalischer Token: IF, ID

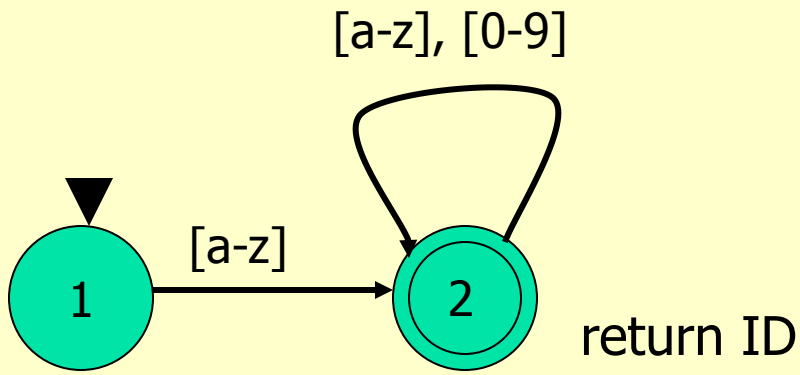
DFA_1

RA: **i f**



DFA_2

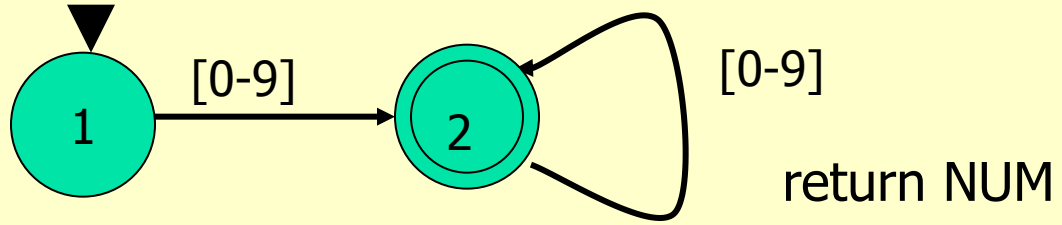
RA: [**a-z**] ([**a-z**] | [**0-9**])*



Erkennung lexikalischer Token: NUM, REAL

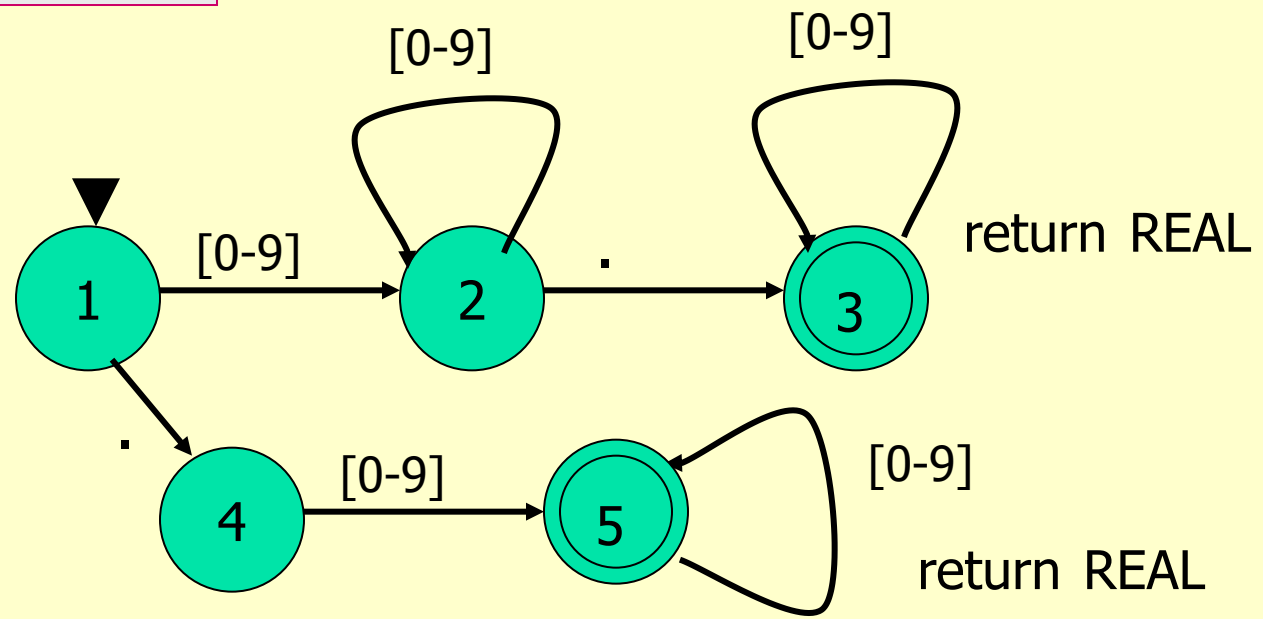
DFA_3

$[0\text{-}9]^+$



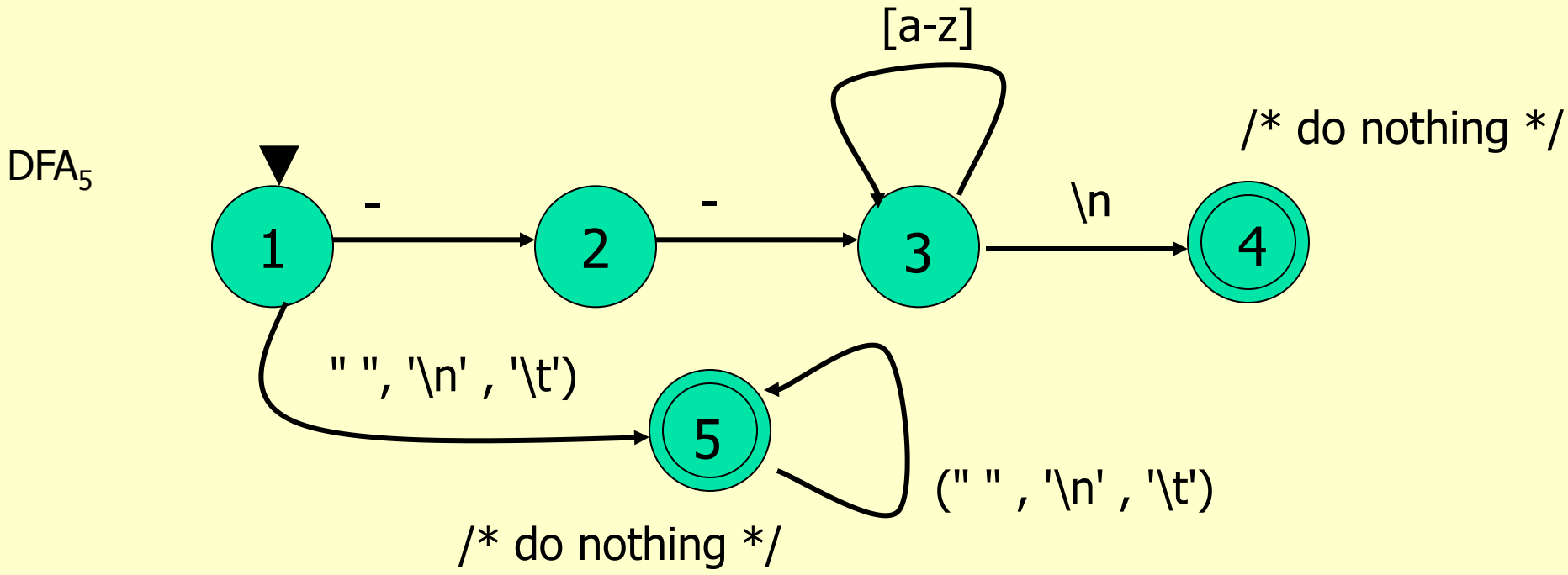
$([0\text{-}9]^+"."[0\text{-}9]^*) \mid "."[0\text{-}9]^+)$

DFA_4



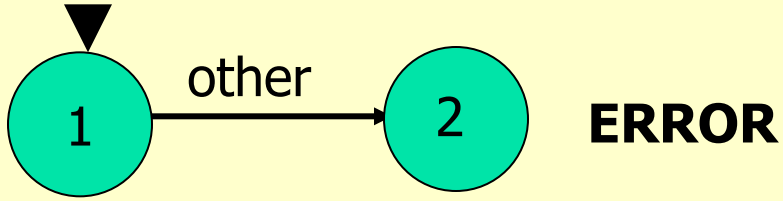
Ignorierung von White Spaces

("--"[a-z]*'\n') | (" " | '\n' | '\t')+



Erkennung lexikalischer Token: ERROR

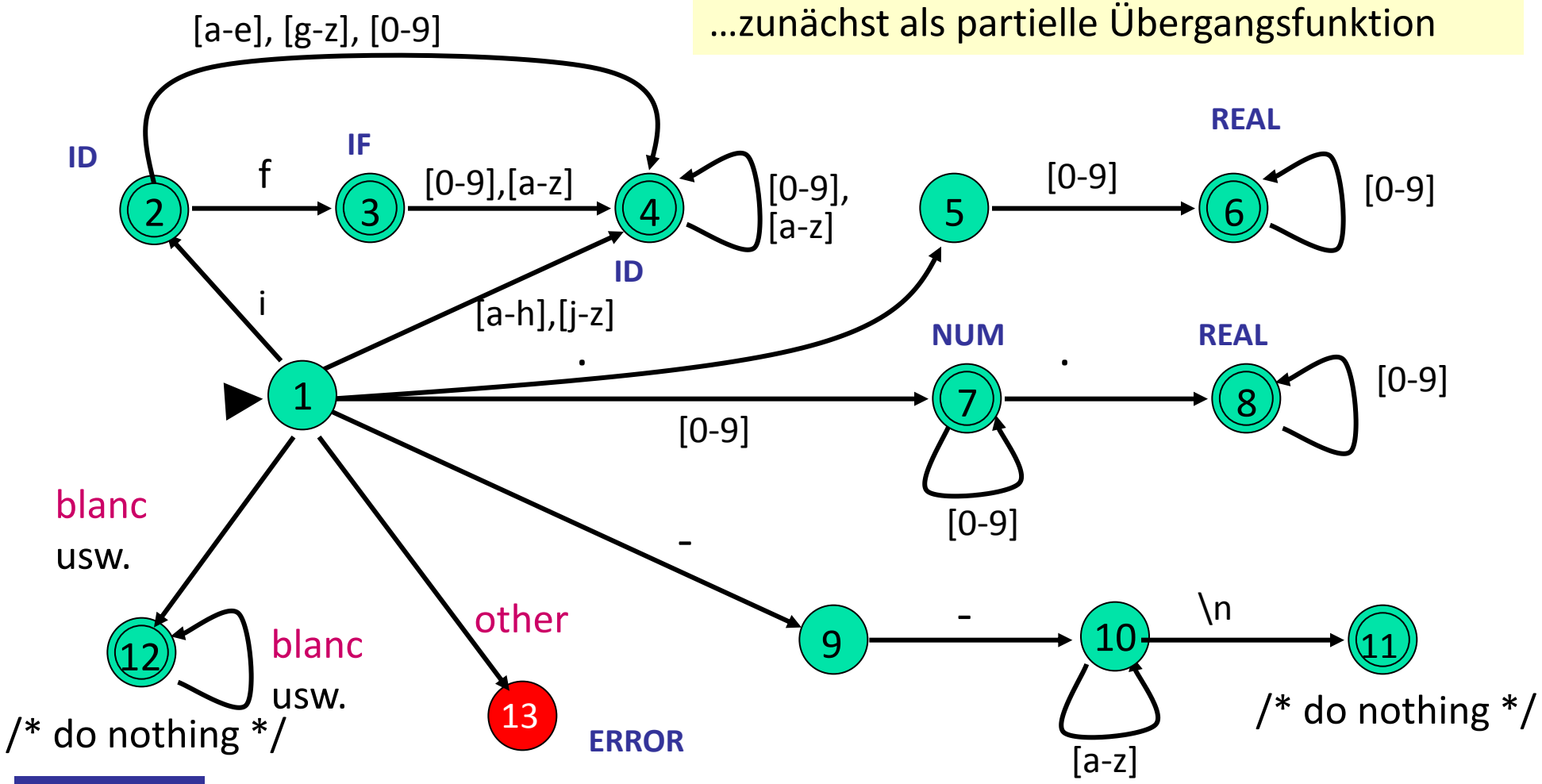
DFA_6



other: Zusammenfassung aller Übergänge aus Zustand s_1, die
den „Rest“ von Zeichen aus dem Eingabealphabet konsumieren

Kombination von Einzelautomaten (ad-hoc)

…zunächst als partielle Übergangsfunktion



Vervollständigung partieller Übergangsfunktionen
(ausschließliche Akzeptanz kompletter Eingabeworte)

Vervollständigung erfolgt in Abhängigkeit davon, ob

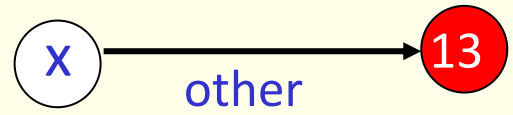
- Automat nur komplette Eingaben oder auch
- Anfangsteilworte akzeptieren soll

Beispiel: Vervollständigung der Übergangsfunktion eines kompositorischen DFAs

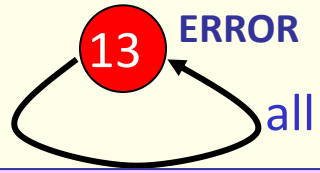
Ann.: Automat akzeptiert Eingabe komplett
nur Zustand 1 ist bislang für **alle** Eingaben vorbereitet

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

jetzt ist δ komplett definiert,
jede Eingabe wird vom DFA bis zum Ende gelesen.

all= alle mögliche Eingabezeichen

Typische praktische Analyseprobleme

Fragen

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

Wir benötigen zusätzliche Regeln (die auch beim realen Scanner-Bau benutzt werden)

Regeln:

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

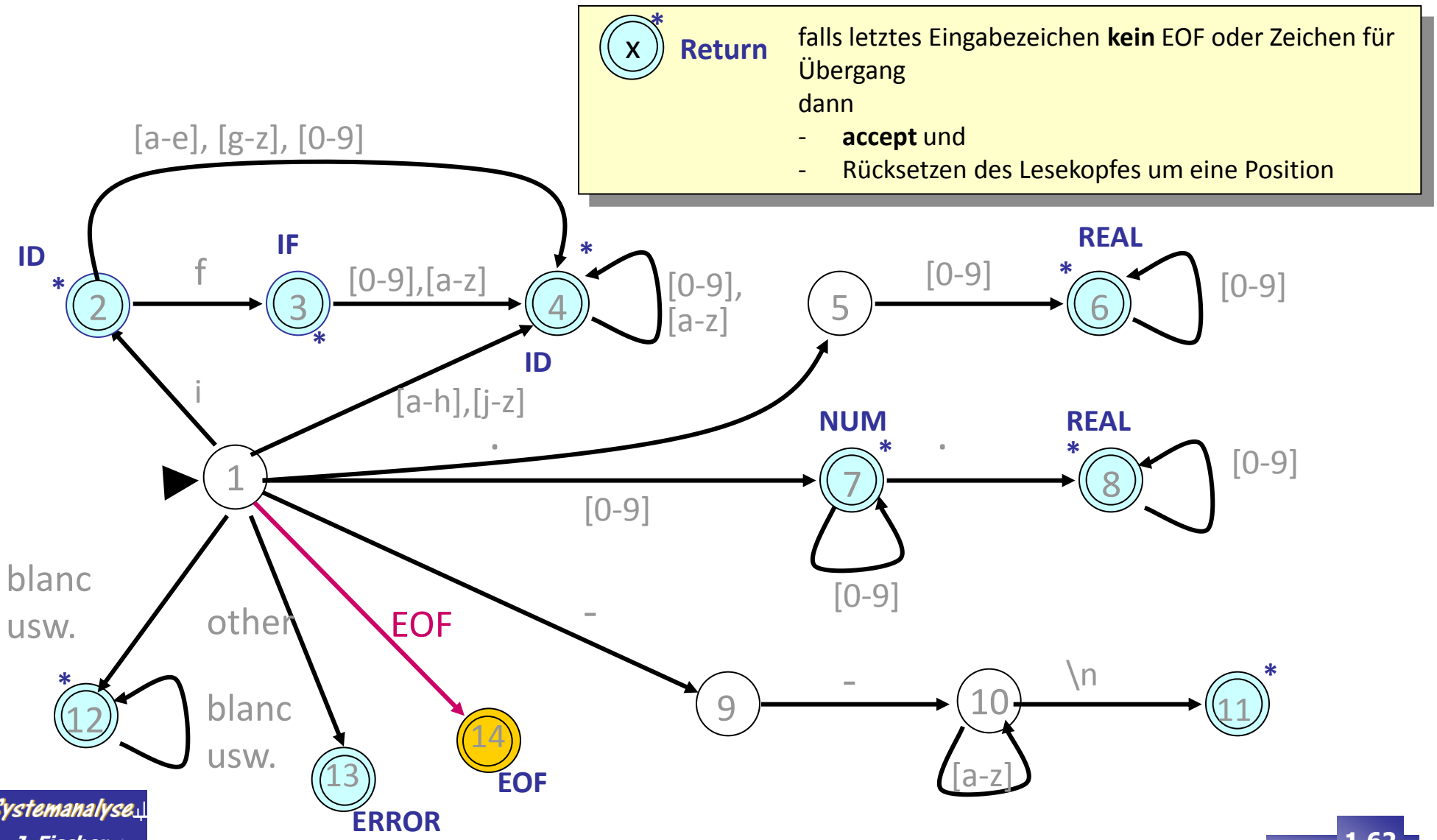
(2) die Bestimmung der Token-Klasse erfolgt entsprechend einer **priorisierten** Liste
(d.h. Reihenfolge der RA-Akzeptanz ist signifikant)

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

(Akzeptanz maximal möglicher Anfangsteile von Eingabeworten)



Vervollständigung partieller Übergangsfunktionen

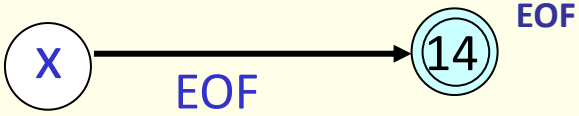
(Akzeptanz maximal möglicher Anfangsteile von Eingabeworten)

Beispiel: Vervollständigung der Übergangsfunktion eines kompositorischen DFAs

Ann.: Automat liest Eingabe entspr. maximaler Token-Länge

Ergänzungsschritte:

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)