Systemanalyse
INSTITUT FÜR INFORMATIK
Humboldt-Universität zu Berlin

Praktische Informatik 3

Teil II - Compilerbau

Vorlesung des Grundstudiums
Prof. Dr. Joachim Fischer
Institut für Informatik, Humboldt-Universität zu Berlin
WS 2002 / 03

Systemanalyse
INSTITUT FÜR INFORMATIK
Humboldt-Universität zu Berlin

Teil II: Compilerbau

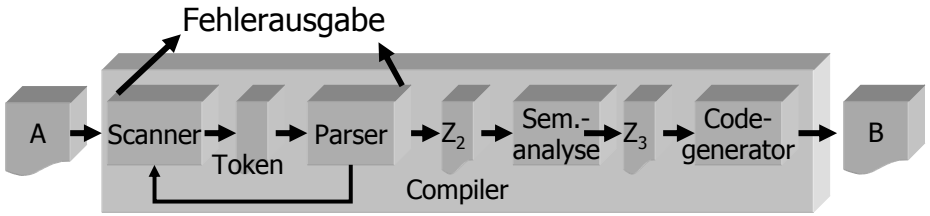
2. Lexikalische Analyse (Forts.)

Vorlesung des Grundstudiums
Prof. Dr. Joachim Fischer
Institut für Informatik, Humboldt-Universität zu Berlin
WS 2002 / 03

Überblick

- Reguläre Ausdrücke
- Endliche Automaten
  - deterministisch, nicht-deterministisch
  - Überführung
- Erzeugen und Erkennen regulärer Ausdrücke
- Architektur eines Scanners
- Praktische Aspekte eines Scanners (Lex-Werkzeug)

Der Scanner (Wdh.)



Aufgaben eines Scanners

- Überführung von Zeichenketten des Quellcodes in **Token**
- Entfernung von Zwischenzeichen: Blank, Tab, Kommentare, ...
- Erkennung lexikalischer Fehler: illegale Zeichen in Token, ...
- Positionsermittlung von Symbolen im Quellprogramm: für Fehlerausschriften

Kurznotation Regulärer Ausdrücke (Wdh.)

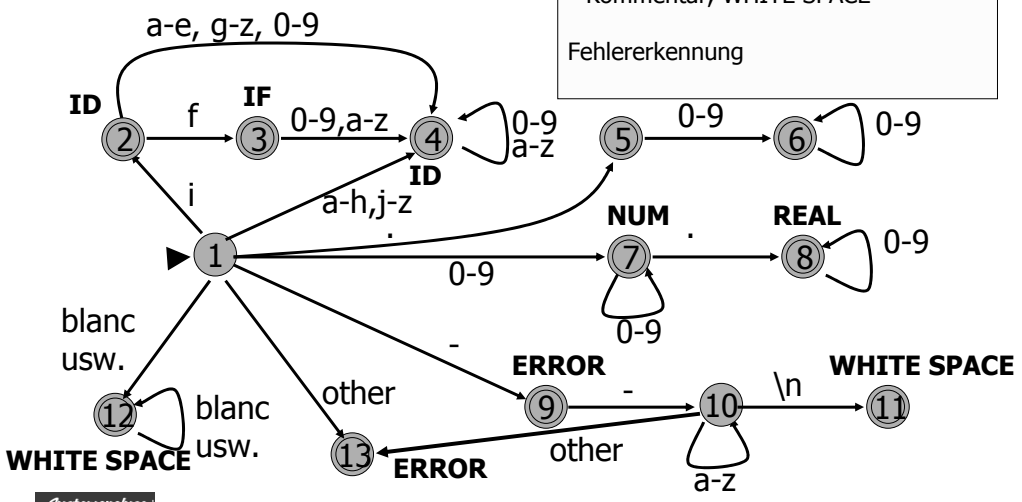
a	gewöhnliches Zeichen
ε	leere Zeichenkette
s | r	Alternative
s · r	Verkettung
sr	andere Schreibweise für Verkettung
s*	Wiederholung (null oder beliebig oft)
s^+	Wiederholung (einmal oder beliebig oft)
s?	optionale Wiederholung (null oder einmal)
[a-zA-Z]	Zeichenalternative
.	jedes beliebe Zeichen (außer newline)
"a.+*"	Literal

Kombination von Einzela

Erkennung von RAs zur Token-Bildung:
- IF, ID, NUM, REAL,

Erkennung von RAs zur Elimination von:
- Kommentar, WHITE SPACE

Fehlererkennung



Arbeitsweise des Automaten

- liegt in einem beliebigen Zustand (Startzustand, Zwischenzustand, Terminalzustand) ein akzeptables Eingabezeichen vor, wird
  - das Zeichen akzeptiert (d.h. konsumiert) und
  - der Folgezustand i.Abh. des akzeptierten Zeichens angenommen
- liegt ein nicht akzeptables Zeichen vor, sind folgende alternative Verhaltensformen des Automaten (im Unterschied zur theoret. Definition) möglich:
  - aktueller Zustand= Startzustand oder beliebiger Zwischenzustand
    a) es gibt einen expliziten Übergang für "other" mit Fehlerbehandlungsprozedur
    b) es gibt einen impliziten Übergang, der für den "Rest" in einen Fehlerzustand führt
  - aktueller Zustand= ERROR-Endzustand
    unvollständige Token-Bildung, das Zeichen wird nicht konsumiert
    explizite Fehlerbehandlungsprozedur
  - aktueller Zustand= Endzustand (kein ERROR)
    Token- bzw. Eliminationstext-Bildung ist abgeschlossen, Ausgabe des Textes
    das aktuelle Zeichen wird nicht konsumiert, der Automat stoppt

Vermeidung von Analyseproblemen (Wdh.)

wichtige Regeln beim Scanner-Bau

1. nächstes Token wird stets als maximale mögliche gültige Zeichenkette bestimmt
2. Die Bestimmung der Token-Klasse erfolgt entsprechend einer priorisierten Liste (d.h. Reihenfolge der RA ist signifikant)

Auswirkungen

- "if8" wird als Bezeichner erkannt: nach der 1.Regel
- "if 89" wird als Schlüsselwort erkannt: nach der 2.Regel

Tabelle für einen Scanner

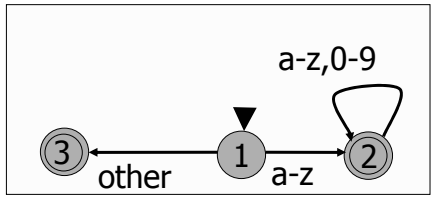
- zwei Tabellen steuern den Scanner
- Zeichenklasse (char_class):

Pseudoklasse (kontextabhängig)

	a-z	**0-9**	**other**
value	letter	digit	other

- Zustandsübergangstabelle (next_state):

	1	**2**	**3**
letter	2	2	-
digit	3	2	-
other	3	-	-



Vorteil: schneller Zugriff
Nachteil: Speicherplatz

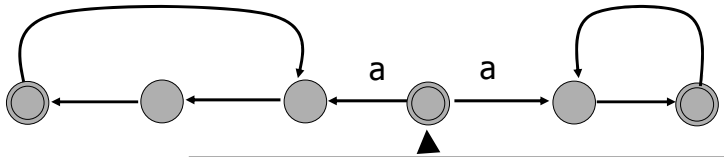
- neue Sprache bedeutet neue Tabellen

Pseudocode für einen Scanner

```
char ← next_char();
state ← 1;            /* code for state 1 */
done ← false;
token_value ← " "   /* empty string */
while (not done) {
      class ← char_class[char] ;
      state ← next_state[class, state] ;
      switch (state)  {
            case 2:    /* building an id */
                        token_value ← concatenate(token_value, char);
                        char  ← next_char();
                        break;
            case 3:   /* error */
                        token_type ← error;
                        done ← true;
                        break;
            case -:    /* accept state */
                        token_type ← identifier;
                        done ← true;
                        break;
            }
}
```


Nichtdeterminierter Endlicher Zustandsautomat

- Nondeterministic Finite Automaton (NFA)



entweder in den linken oder rechten Teilgraphen

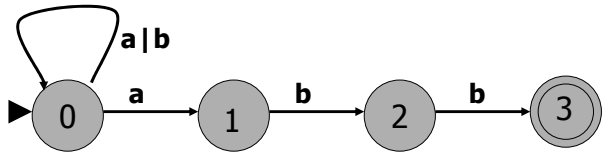
Bemerkung

- es ist einfacher NFAs zur Erkennung von RAs zu konstruieren als DFAs
- beide Automatentypen erkennen Sprachen, die sich mit regulären Ausdrücken beschreiben lassen
- Automaten lassen sich ineinander überführen
- es gibt Unterschiede beider Automatentypen für Ressourcenbedarf (Zeit, Speicher)

Übergänge in einem NFA

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

"aabb"



- Zustand 0 hat mehrere Übergänge für a
  ⇒ nicht-deterministischer endlicher Automat (NFA)

$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$ ... $0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$

viele Übergange möglich, aber nur einer führt zur Akzeptanz

Definition eines NFA

Ein nicht-deterministischer endlicher Automat
NFA=(**S**, Σ, **T**, s_0, **F**):

- Menge **S** von Zuständen: **S** = $\{s_0,\ldots,s_n\}$
- Menge von Eingabesymbolen: Alphabet Σ
- Zustandsübergangsfunktion (Transition) **T**,
  die einem Zustand Folgezustände in Abh. des Eingabesymbols zuordnet
  (Zustand,Symbol)-Paare werden in Zustandsmengen überführt)
- ein ausgezeichneter Startzustand s_0
- eine Menge von akzeptierenden (End-) Zuständen **F**

Ein NFA akzeptiert eine Zeichenkette x, falls

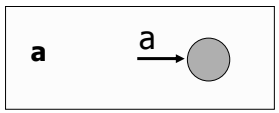
- ein Pfad durch den Transitionsgraph existiert, der in s_0 beginnt und in einem Endzustand so endet, dass die Sequenz der durchlaufenden Kanten genau x ergibt

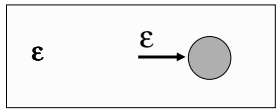
Spezialfall eines NFA: DFA

- Ein deterministischer endlicher Automat (DFA) ist ein Spezialfall eines NFAs:
  - es gibt keine ε-Transitionen (spontane Übergange ohne Zeichen)
  - für jeden Zustand s und Eingabesymbol a existiert maximal eine Kante a, die s verlässt, d.h. für jedes (Zustand,Symbol)-Paar gibt es maximal einen neuen Zustand
- Ein DFA akzeptiert die Zeichenkette x, falls
  ein eindeutiger Pfad durch den Transitionsgraph existiert, der in s_0 beginnt und in einem Endzustand so endet, dass die Sequenz der durchlaufenden Kanten genau x ergeben

Abbildung von RAs zu NFAs

Wie konstruiert man einen NFA?
• man baut Einzelautomaten und
• fügt diese nach Vorschrift zusammen (beim DFA: nur ad-hoc)





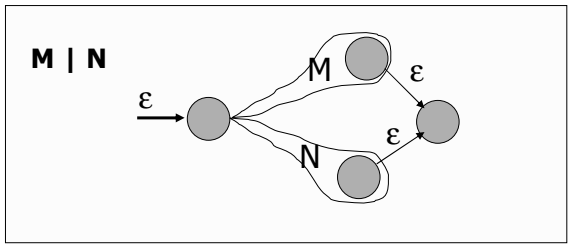
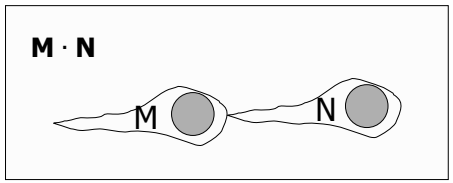


Abbildung von RAs zu NFAs (2)



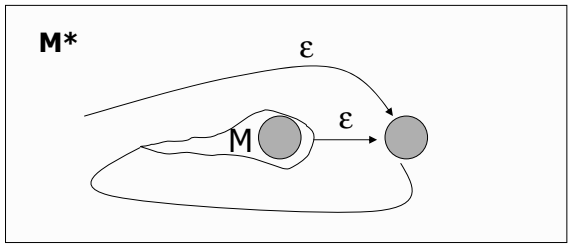
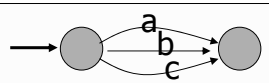


Abbildung von RAs zu NFAs (3)

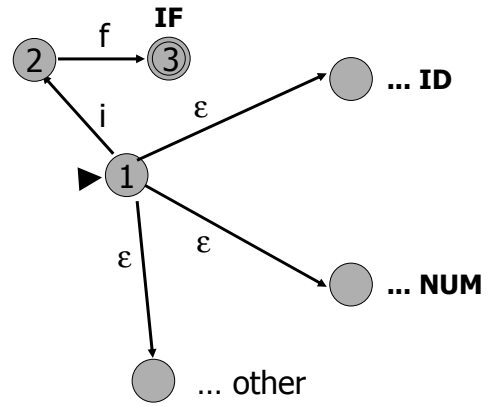
M+ konstruiert als M · M*

M? konstruiert als M | ε

[abc] 

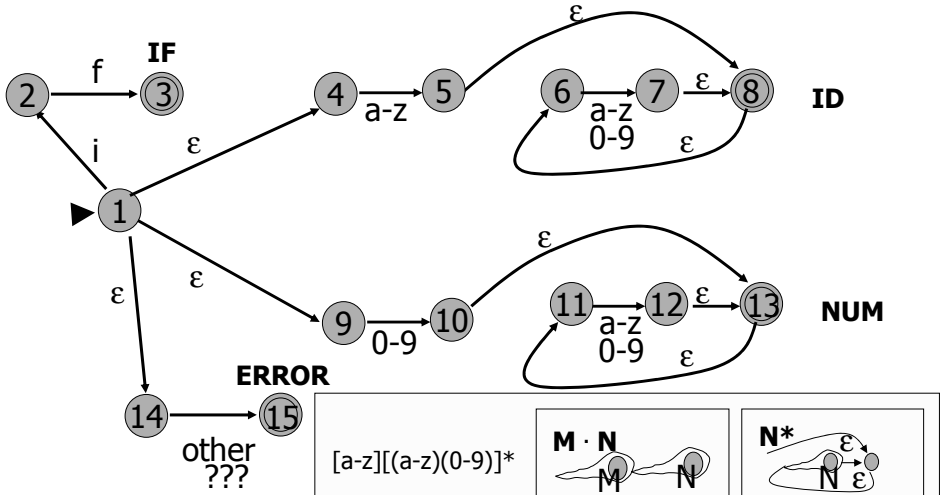
'abc' konstruiert als a·b·c

Konstruktion eines NFAs (für nur vier RAs)



if 

Konstruktion eines NFAs (für nur vier RAs)



Implementierung eines NFAs

- es ist einfacher, aus einem RA einen NFA zu konstruieren als einen DFA
- aber: Implementation eines NFA ist i.Allg. nicht effizient möglich
  - z.B. Einsatz von Backtracking-Verfahren
    - problematisch sind alternative Übergänge aus einem Zustand mit spontanen (ε-Transitionen) und Eingabe-getriebenen Übergängen
- besser: Umwandlung eines konstruierten NFA in einen DFA

Eigenschaften von FAs

DFAs und NFAs sind äquivalent (Theoretische Informatik)

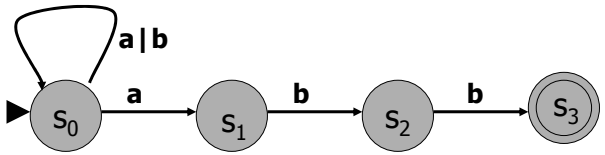
- D.h. für jeden NFA gibt es einen DFA, der die gleiche Sprache erkennt und umgekehrt.

Warum?

- DFAs sind eine Untermenge von NFAs
- jeder NFA kann in einen DFA konvertiert werden, indem die gleichzeitig erreichten Zustände des NFA im DFA »simuliert« werden:
  - jeder Zustand im abgeleiteten DFA korrespondiert mit einer Menge an Zuständen im NFA

Überführung: NFA ⇒ DFA

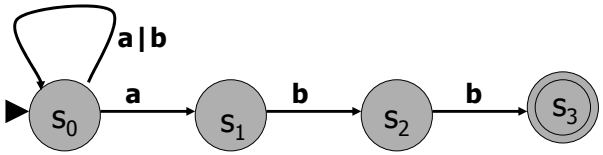
- Verfahren der Teilmengenkonstruktion
  am Beispiel: (a|b)*abb



	a	b

Überführung NFA ⇒ DFA

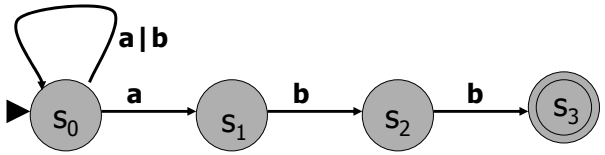
- Verfahren der Teilmengenkonstruktion



	a	b
$\{s_0\}$	$\{s_0,s_1\}$	$\{s_0\}$

Überführung NFA ⇒ DFA

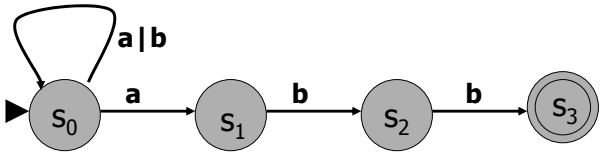
- Verfahren der Teilmengenkonstruktion



	a	b
$\{s_0\}$	$\{s_0,s_1\}$	$\{s_0\}$
$\{s_0,s_1\}$	$\{s_0,s_1\}$	$\{s_0,s_2\}$

Überführung NFA ⇒ DFA

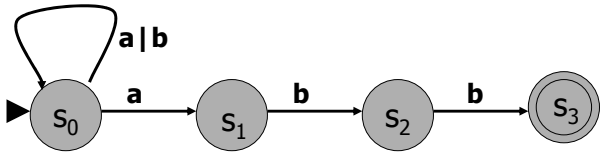
- Verfahren der Teilmengenkonstruktion



	a	b
$\{s_0\}$	$\{s_0,s_1\}$	$\{s_0\}$
$\{s_0,s_1\}$	$\{s_0,s_1\}$	$\{s_0,s_2\}$
$\{s_0,s_2\}$	$\{s_0,s_1\}$	$\{s_0,s_3\}$

Überführung NFA ⇒ DFA

- Verfahren der Teilmengenkonstruktion

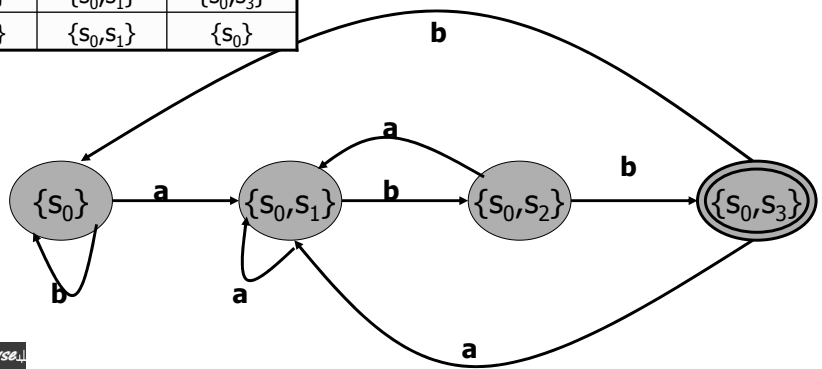


	a	b
$\{s_0\}$	$\{s_0,s_1\}$	$\{s_0\}$
$\{s_0,s_1\}$	$\{s_0,s_1\}$	$\{s_0,s_2\}$
$\{s_0,s_2\}$	$\{s_0,s_1\}$	$\{s_0,s_3\}$
$\{s_0,s_3\}$	$\{s_0,s_1\}$	$\{s_0\}$

Überführung NFA ⇒ DFA (2)

	a	b
$\{s_0\}$	$\{s_0,s_1\}$	$\{s_0\}$
$\{s_0,s_1\}$	$\{s_0,s_1\}$	$\{s_0,s_2\}$
$\{s_0,s_2\}$	$\{s_0,s_1\}$	$\{s_0,s_3\}$
$\{s_0,s_3\}$	$\{s_0,s_1\}$	$\{s_0\}$

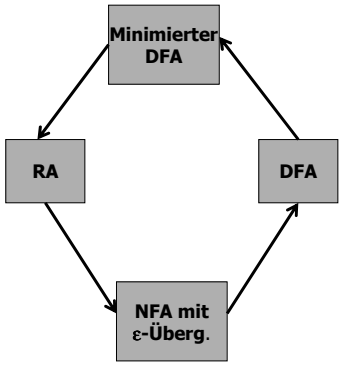
Jeder DFA-Zustand,
der mindestens einen Endzustand
des NFAs enthält,
wird zu einem Endzustand des DFAs



Konstruktion eines DFA von RE

- RA → NFA mit ε-Übergängen
  - Erzeuge NFA für jeden Teilausdruck
  - Verbinde mit Übergängen
- NFA mit ε-Übergängen → DFA
  - Durch »Teilmengen«-Konstruktion
- DFA → minimierten DFA
  - Führe »kompatible« Zustände zusammen
- DFA → RA
  - Konstruiere $R^k_{ij} = R^{k-1}_{ik} (R^{k-1}_{kk})^* R^{k-1}_{kj} \cup R^{k-1}_{ij}$

Details s. Theoretische Informatik



NFA → DFA: Teilmengenkonstruktion

Eingabe: NFA **N**

Ausgabe: Ein DFA **D** mit Zuständen $\mathbf{D_Z}$ und Transitionen $\mathbf{D_T}$ mit L(**D**) = L(**N**)

Methode: Sei s ein Zustand in N und T eine Menge von Zuständen in N, dann benutze die folgenden Operationen:

Operation	**Definition**
ε-Hülle(s)	Menge aller NFA-Zustände, die vom NFA-Zustand s mit ε-Übergängen allein erreicht werden können
ε-Hülle(T)	Menge aller NFA-Zustände, die von einem NFA-Zustand s aus T allein mit ε-Übergängen erreicht werden können
move(T,a)	Menge aller NFA-Zustände, zu denen ein Übergang für das Eingabesymbol a von einem NFA-Zustand s aus T führt

NFA → DFA: Teilmengenkonstruktion (2)

Idee des Verfahrens:

- jeder Zustand des DFA entspricht einer Menge von Zuständen des NFA
- DFA merkt sich in seinen Zuständen alle möglichen Zustände, in denen sich der NFA nach Lesen von Eingabezeichen befinden kann,

  d.h., der DFA befindet sich nach Eingabe von $a_1\, a_2 \ldots a_n$ in einem Zustand, der die Teilmenge T aller Zustände vom NFA bestimmt, die vom Startzustand des NFA entlang eines mit $a_1\, a_2 \ldots a_n$ markierten Pfades erreicht werden kann

Bemerkung

- Die Anzahl von DFA-Zuständen kann exponentiell größer sein, als die des NFA (Fall tritt aber in der Praxis kaum auf)

NFA → DFA: Teilmengenkonstr. (2)

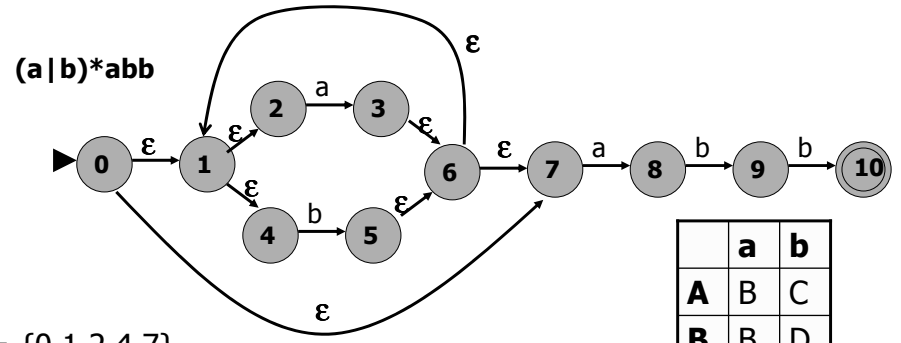
- Algorithmus:

  Anfangsschritt: ε-Hülle(s_0) bildet den Startzustand von **D**

  Füge Zustände aus T = ε-Hülle(s_0) als unmarkierte Zustände $\mathbf{D_Z}$ hinzu
  while es existieren noch unmarkierte Zustände T in $\mathbf{D_Z}$ **do**
  markiere T;
  for jedes Eingabesymbol a **do**
  U = ε-Hülle(move(T,a));
  if U nicht in $\mathbf{D_Z}$ **then** füge U als unmarkiert $\mathbf{D_Z}$ hinzu **endif**
  $\mathbf{D_T}$ [T,a] = U
  endfor
  endwhile

  - Ein Zustand von **D** ist ein Endzustand, falls er mindestens einen Endzustand aus N enthält

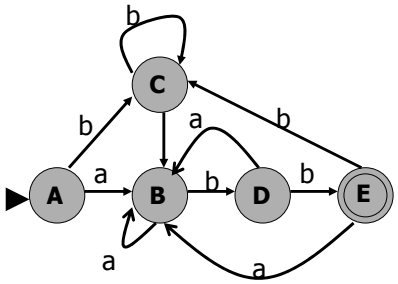
NFA → DFA: weiteres Beispiel



A= {0,1,2,4,7}
B= {1,2,3,4,6,7,8}
C= {1,2,4,5,6,7}
D= {1,2,4,5,6,7,9}
E= {1,2,4,5,6,7,10}

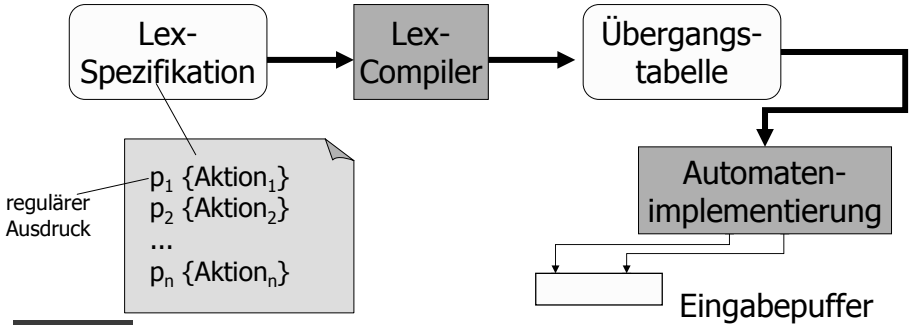
	a	**b**
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

NFA → DFA: weiteres Beispiel (2)



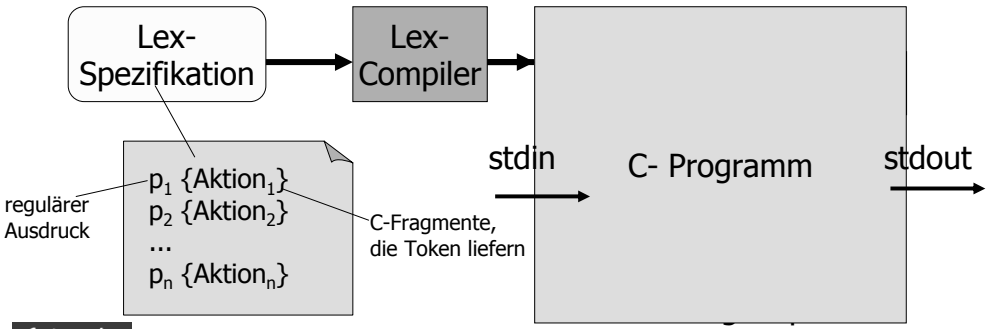
Entwurf eines Scanner-Generators (Lex)

- Konstruktion eines Automaten ist eine automatisierbare Aufgabe



Entwurf eines Scanner-Generators (Lex)

- Konstruktion eines Automaten ist eine automatisierbare Aufgabe



Lex-Spezifikation (Überblick)

```
%{
/* C Declarations: */
#include "tokens.h"  /*definitions of IF, ID, NUM, ...*/
#include "errormsg.h"
union {int ival; string sval; double fval;} yylval;
int charPos=1;
#define ADJ (EM_tokPos=charPos, charPos+=yyleng)
%}
```

1.Teil Deklarationen

```
/* Lex-Definitions: */
digits [0-9]+          /* non-empty sequence of digits */
%%
```

2. Teil
- Abkürzungen regulärer Ausdrücke
- Zustands-deklarationen

Lex-Spezifikation (Überblick-2)

```
/* Regular Expressions and Actions: */
if                                  {ADJ; return IF; }
[a-z][a-z0-9]*                      {ADJ; yylval.sval= String(yytext);
                                      return ID; }
{digits}                            {ADJ; yylval.ival= atoi(yyte
                                      return NUM; }
({digits}"."[0-9]*)|([0-9]*)"."{digits}  {ADJ; yylval.fval= atof(yytext);
                                      return REAL; }
("--"[a-z]*"\n")|(" "|"\n"|"\t")+   {ADJ;}
.                                   {ADJ; EM_error("illegal character");}
```

3.Teil
RAs und
Aktionen

- jede Aktion liefert return-Wert vom Tokentyp (Typ int)
- yytext ist Zeiger zum String (NULL-terminiert), der dem RA entspricht,
- Länge von yytext* ist yyleng
- charPos (Position des Token vom File-Beginn in Anzahl von char)
- EM_tokPos (aus errormsg.h: zählt Aufrufe von ADJ als Info für den Parser
- yyval enthält semantische Werte eines Token

Lex-Spezifikation (Überblick- 3)

```
/* Regular Expressions and Actions: */
if                                  {ADJ; return IF; }
[a-z][a-z0-9]*                      {ADJ; yylval.sval= String(yytext);
                                      return ID; }
{digits}                            {ADJ; yylval.ival= atoi(yyte
                                      return NUM; }
({digits}"."[0-9]*)|([0-9]*)"."{digits}  {ADJ; yylval.fval= atof(yytext);
                                      return REAL; }
("--"[a-z]*"\n")|(" "|"\n"|"\t")+   {ADJ;}
.                                   {ADJ; EM_error("illegal character");}
```

3.Teil
RAs und
Aktionen

- jede Aktion liefert return-Wert vom Tokentyp (Typ int)
- yytext ist Zeiger zum String (NULL-terminiert), der dem RA entspricht,
- Länge von yytext* ist yyleng
- charPos (Position des Token vom File-Beginn in Anzahl von char)
- EM_tokPos (aus errormsg.h: zählt Aufrufe von ADJ als Info für den Parser
- yyval enthält semantische Werte eines Token

Vordefinierte Bezeichner von Lex

int yylex(void)	Ruf des Lexers, liefert Token zurück
char *yytext	Zeiger zum String, das dem RA entspricht
yyleng	Länge des Strings, das dem RA entspricht
int yywrap(void)	Anwendung auf mehrere Files (1 falls fertig, 0 sonst)
FILE *yyout	Ausgabefile (default: stdout)
FILE *yyin	Inputfile (default: stdin)
INITIAL	initiale Startbedingung
BEGIN condition	Einschalten der Startbedingung
ECHO	Ausgabe des akzeptierten Strings

Lex-Beispiel: Zeilennummerierung

```
%{
/* C Declarations: */
int yylineno= 0;
%}
/* no Definitions */
%%
/* Regular Expressions and Actions: */
^.*\n      { printf("%4d\t%s", ++yylineno, yytext); }
%%
int main (int argc, char *argv[]) {
    yyin= fopen(argv[1], "r");
    yylex();
    fclose(yyin);
    return 0;
}
```

- Jeder Input wird akzeptiert
- Einfügen von Zeilennummern in den Text

Lex-Beispiel: Zählen von Zeichen, ...

```
%{
/* C Declarations: */
int nchar, nword, nline;;
%}
%%
\n              {nline++; nchar++; }
[^ \t\n]+       {nword++; nchar +=yyleng; } /* alle Zeichenketten, die kein Leerzeichen,
                                              Tabulator, Newline enthalten */
.               {nchar++; }
%%
int main (void) {
    yylex();
    printf("%d\t%d\t%d\n", n
    return 0;
}
```

- Jeder Input wird akzeptiert
- Einfügen von Zeilennummern in den Text

Möglicher Mix von Zuständen und RAs in Lex

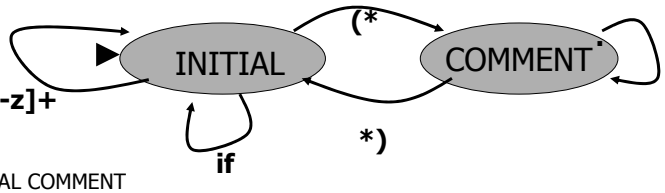
- man kann eine Menge von Startzuständen deklarieren,
- jeder RA kann mit Menge von Startzuständen präfigiert sein, für die er gültig ist
- im Aktionsteil kann der Startzustand explizit geändert werden

damit lassen sich Übergänge mit RA markieren (u. nicht nur mit Symbolen)

Effekt: abstraktere Automaten mit Wechsel von einem zum anderen

"Hierarchische" Automaten

- eigentlich Automaten auf gleicher Ebene




```
...
/* 2.Teil */
%Start INITIAL COMMENT
%%
 <INITIAL>if              {ADJ; return IF;}
 <INITIAL>[a-z]+          {ADJ; yyval.sval= String(yytext); return ID;}
 <INITIAL>"(*"            {ADJ; BEGIN COMMENT;}     /* Automatenumschaltung */
 <INITIAL>.               {ADJ; EM_error("illegal character");}
 <COMMENT>"*)"            {ADJ; BEGIN INITIAL;}     /* Automatenrückschaltung */
 <COMMENT>.               {ADJ; BEGIN COMMENT;} /* Automatenumschaltung */
```


Fragen