Lexikalische Analyse

- **aus Einführung:**

  Scanner

  – Programmteil, der lexikalische Analyse durchführt.

  – Unterprogramm des Parsers (Syntaxanalyse)

  – übergibt Parser Token (Bezeichner der Lexemklassen):
    `IDENT, NUMBER, ...`

Bsp.:

```
bez1 = 354*bez2;
-> IDE"="NUM"*"IDE";"
```

  – für Parser Token ausreichend

  – bei Codeerzeugung müssen aber Bezeichnernamen und Zahlwerte bekannt sein

-> da Scanner einziger Teil des Compilers, der Quellprogramm durchläuft, liefert er weitere Informationen als Attribute

Token & Attribute

- **Aufgaben des Scanners:**
  - Zerlegung des Quellprogramms in Folge von Paaren aus
    - Token   Bezeichnung der Lexemklassen
    - Attribute Lexemwerte
      - für Bezeichner: Name
      - für Zahlen: Werte
      - nur, wenn nötig

Bsp: `bez1 = 354*bez2;`

```
->    (IDENT,bez1) ("=", ) (NUMBER,354) ("*", )
      (IDENT,bez2) (";", )
```

- **Verwendung der Scannerausgabe:**
  - Token: Parser
  - Attribute: Codeerzeugung

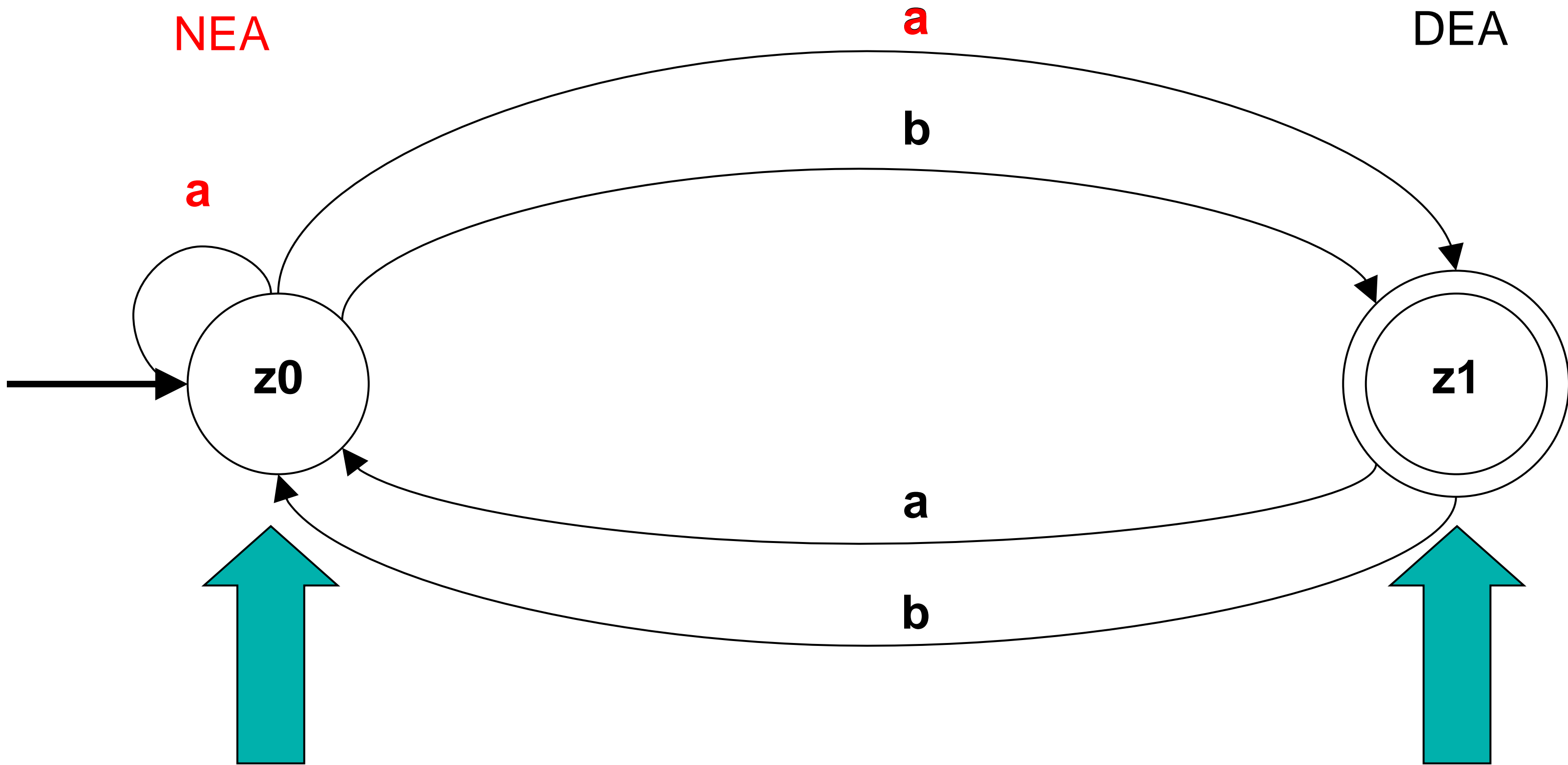
Analyseverfahren

- **bisher:**
  Beschreibung der Lexemstrukturen durch reguläre Ausdrücke
- **RA**
  - Def.: ε, `a` **sind RAs für jedes Eingabezeichen** `a`
    Seien a, b reguläre Ausdrücke. Dann auch
    a|b : a oder b
    ab : a konkateniert mit b
    a* : ε oder a kein mal, einmal oder beliebig oft mit sich selbst konkateniert (Kleene Stern)
  - Bsp.: Siehe Beschreibung von Bezeichnern (Erweiterungen)
  - deklarative Beschreibung
  - können in Prologprogrammierung verwendet werden
  - nicht in imperativen oder objektorientierten Sprachen

Scannerprogrammierung in C oder Java?

- **Implementierung durch DEA (det. endl. Automaten)**

Endliche Automaten (Wdh.)



RAs für DEA & NEA?

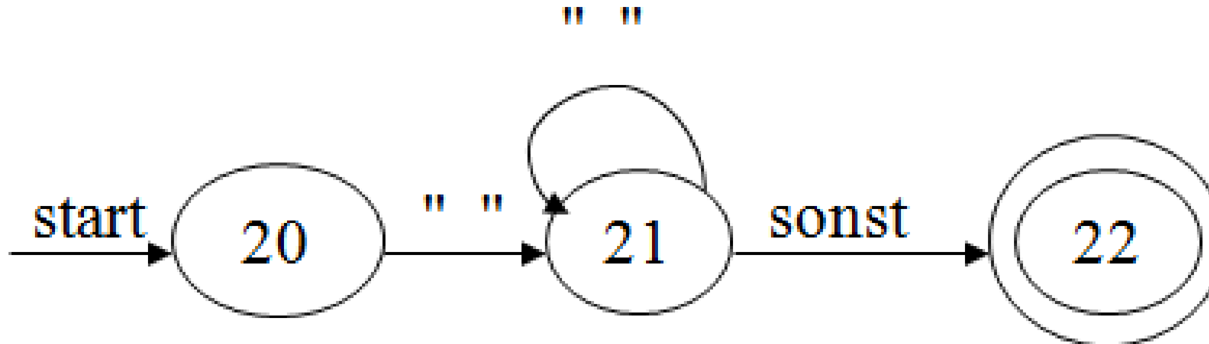
FH AACHEN UNIVERSITY OF APPLIED SCIENCES

Händische Scannerkonstruktion (EAs)

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

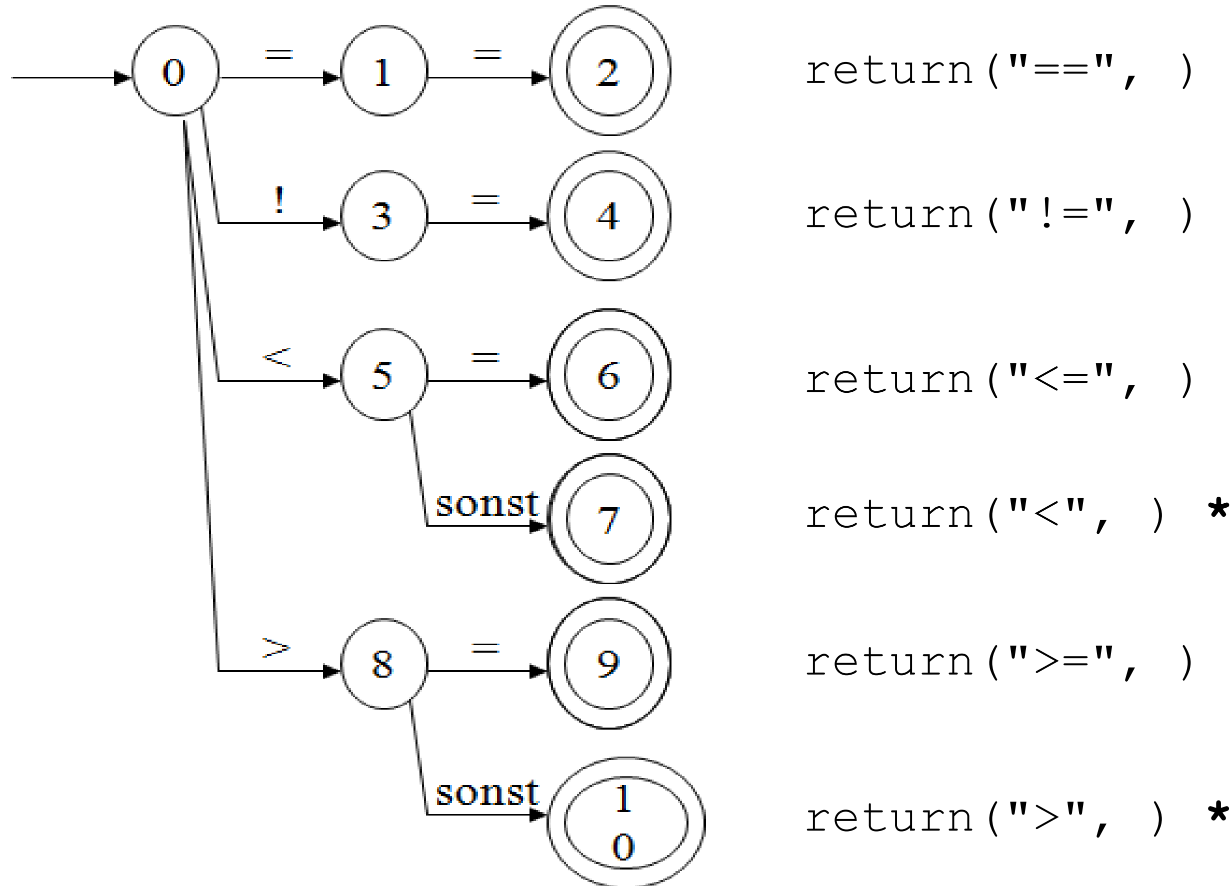
- pro Lexemklasse ein oder mehrere erkennende EAs
- Transformation in Programmteile
- Sukzessives Durchlaufen mit **look-ahead** auf der Eingabe (Lexemtrennung!), da sonst kein Zurücksetzen möglich

- **Bsp.: EA für Blank**



  – Hier: keine Rückgabe eines Tokens an den Parser (Überlesen)
  – Scanner muss bis zum nächsten Token weiterarbeiten
  – Implement. des Scanners durch Umsetzung der EAs in PSn

EA für CompOPs



- sonst: Übergang mit jedem anderen Zeichen
- *: Rücksetzen des look-ahead Zeigers um 1 Zeichen

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

automatische Scannerkonstruktion

- Grundlage: **RAs -> DEAs**
- **in javacc:**

Konfigurationsdatei

`Test.jj` —javacc→ `...,` `TestTokenManager.java`

Implementierung des EAs

wird automatisch übersetzt bei Übersetzung von `Test.java`

- Spezifikation der RAs in Konfigurationsdatei -> Einführung
- **unter Unix:**
  Scanner- und Parser-Erzeuger getrennt in **lex, yacc**
  -> Konfigurationsdatei von lex enthält nur RAs

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

DEA-Methode

- **Ziel**:
  - automatische Konstruktion RA -> DEA
  - Implementierung von DEA in C oder Java
- **Umweg:** RA -> NEA -> DEA automatisch konstruierbar
- **1. RA -> NEA**

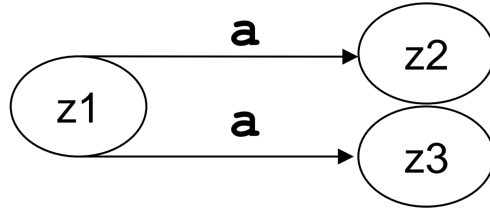
  Konstruktion von **Kleene** induktiv über Aufbau der RAs
  siehe Theoretische Informatik oder Literatur
- **2. NEA -> DEA**

  Potenzmengenkonstruktion PMK
- **Problem**

  DEA kann exponentiell viele Zustände haben
  auch die hier vorgestellte verbesserte Version

Wdh.: PMK

- **Idee:**
  - NEA



$$(z1, a, z2), (z1, a, z3) \in \text{dRel}$$

  - DEA

$$\text{dFun}(\{z1\}, a) = \{z2, z3\}$$

- **Formalisierung:**
  - Geg.: NEA N = (States,Σ,dRel,z0,E)
  - Konstruiere DEA D = (DStates,Σ,dFun,{z0},ED) mit

DStates = P(States)

$$\text{dFun}(\text{T}, \text{a}) = \bigcup_{\text{z} \in \text{T}} \{\text{z}' \in \text{States} \mid (z, a, z') \in d\,\text{Re}\,l\}$$

$$\text{ED} = \{T \in \text{P(States)} \mid \exists \text{e} \in T : \text{e} \in E\}$$

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

PMK-Algorithmus

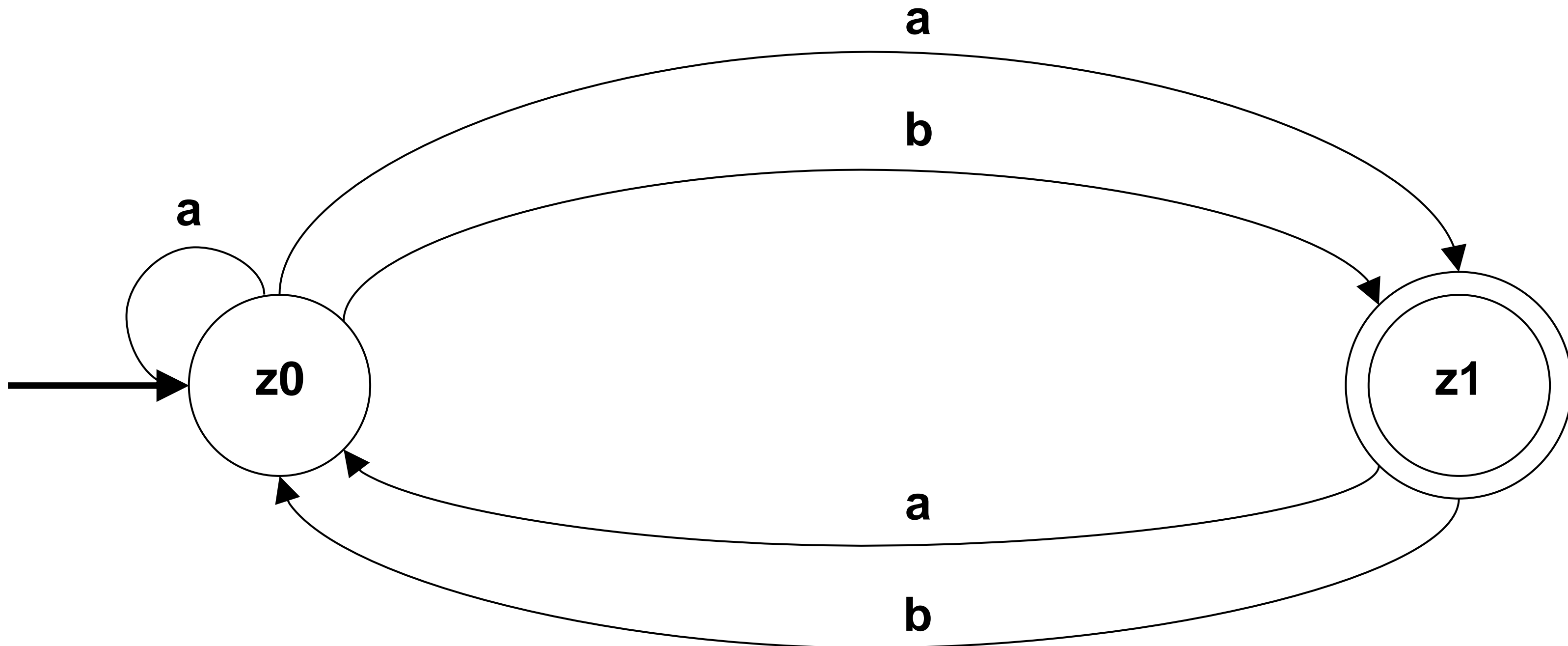
- Füge nur erreichbare Zustände zu DStates ($\subseteq$ P(States)) so in TIWS gemacht:

```
DStates = {{z0}[ohne Marke]};
while T[ohne Marke] ∈ DStates
{
     markiere T in DStates;
     for each a ∈ Σ
     {
          U = {};     //U ist Folgezustand von T mit a
          for each z ∈ T  U ∪= {z' | (z,a,z') ∈ dRel};
          if (U ∉ DStates) DStates ∪= {U[ohne Marke]};
               dFun(T,a) = U;
     }
}
```

- In ED alle Mengen aus DStates, die Endzustand aus E enthalten

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

PMK-Beispiel

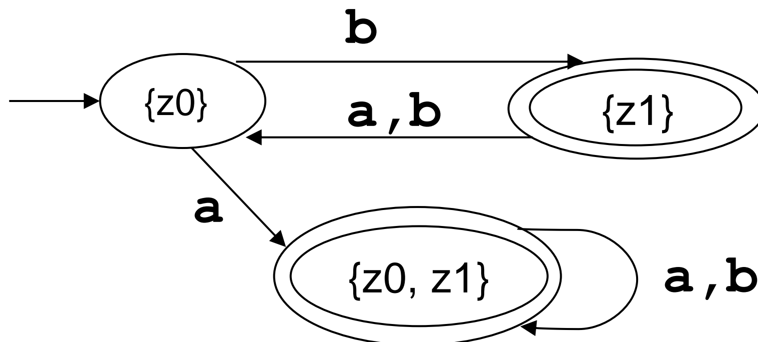


Formalisierung des NEA?

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

PMK-Beispiel

- geg.: NEA ({z0,z1},{a,b},dRel,z0,{z1})
  mit dRel = {(z0,a,z0),(z0,a,z1),(z0,b,z1),(z1,a,z0),(z1,b,z0)}
- berechne: DEA (DStates,{a,b},dFun,{z0},ED) Marke ~
  - DStates = {{z0}} -> dFun({z0},a) = {z0,z1}
    dFun({z0},b) = {z1}
  - DStates = {{z0}~,{z0,z1},{z1}} -> dFun({z1},a) = {z0}
    dFun({z1},b) = {z0}
  - DStates = {{z0}~,{z0,z1},{z1}~} -> dFun({z0,z1},a) = {z0,z1}
    dFun({z0,z1},b) = {z0,z1}
  - **DStates = {{z0}~,{z0,z1}~,{z1}~} ED = {{z1},{z0,z1}}**



w = aaa in L
w = bb nicht in L

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

NEA-Methode

- **Problem bei DEA-Methode:**
  u.U. exponentiell viele Zustände (Beispiel?)
- **Idee bei NEA-Methode:**
  Simulation der PKM zur Laufzeit, d.h. eines Laufes durch Potenzmengenautomat bzgl. einer Eingabe
- **NEA-Methode:**
  2 Stacks zum Speichern einer Zustandsmenge und der jeweiligen Folgezustände
- **Eingaben:**
  - NEA N = (States, Σ, dRel, z0, E)
  - Wort w
- **Ausgaben:**
  - Ja, falls w in zu erkennender Sprache
  - Nein, sonst

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

NEA-Methoden-Algorithmus

```
{
      T = {z0};
      a = nextchar(w);

      while (a != eof)
      {
            U = {};     //U ist Folgezustand von T mit a
            for each z∈T  U ∪= {z' | (z,a,z')∈ dRel};
            T = U;
            a = nextchar(w);
      }

      if (T∩E != {}) return "JA";
      else           return "Nein";
}
```

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

NEA-Methoden-Beispiel

- geg.: NEA von Folie 4 ({z0,z1},{a,b},dRel,z0,{z1})
  mit dRel = {(z0,a,z0),(z0,a,z1),(z0,b,z1),(z1,a,z0),(z1,b,z0)}
- Lauf für w = aaa:
  - T = {z0} nextchar(w) = a
  - T = {z0,z1} nextchar(w) = a
  - T = {z0,z1} nextchar(w) = a
  - T = {z0,z1} nextchar(w) = eof

$T \cap E = \{z1\}$

-> "Ja"

- Lauf für w = bb:
  - T = {z0} nextchar(w) = b
  - T = {z1} nextchar(w) = b
  - T = {z0} nextchar(w) = eof

$T \cap E = \emptyset$

-> "Nein"

Komplexitätsvergleich

- **|States| : Anzahl der Zustände des NEA**
- **|w| : Länge von w**

	Platz	Zeit
DEA-Methode	$O(2^{\vert States \vert})$	O(|w|)
NEA-Methode	O(|States|)	O(|States|*|w|)

- **zur Zeit-Komplexität des NEA-Verfahrens**
  - while-Schleife wird |w| mal durchlaufen
  - for-Schleife wird |T| (<=|States|) mal durchlaufen
  - -> |w| * |T| Durchläufe <= O(|w| * |States|)

Allgemeines Problem: Lexemende?

- `b, be, bez, bez1` alles Bezeichner
- Konvention: **"Principle of the Longest Match"**
  d.h. Zeichenreihe bestimmt ein Lexem, wenn keine Verlängerung zu einer Lexemklasse gehört
- manchmal inkorrekt
- Bsp.: (Fortran)

  `363, 363.E1, .EQ.` alles Lexeme

  Bei `363.EQ.363` Principle of the Longest Match

  ^ lexikalischer Fehler
- Lösung: Look-ahead von 3 Zeichen (mindert Effizienz)
- in javacc Standard-Look-ahead von 1

Zusammenfassung

FH AACHEN UNIVERSITY OF APPLIED SCIENCES

- Scanner zerlegt Quellprogramm in Token und Attribute
- Struktur der Lexeme durch reg. Ausdrücke beschreibbar
- Angabe der Struktur in javacc durch reguläre Ausdrücke
- Analyse durch endliche Automaten, hierzu:
  - **Kleene:** Transformation der regulären Ausdrücke in NEA
  - **PMK:** Transformation NEA -> DEA
    DEA-Methode -> hoher Platzbedarf
  - **Laufzeit-PMK:** Transformation NEA -> DEA zur Laufzeit
    NEA-Methode -> höherer Laufzeitbedarf