# Lexikalische Analyse

**Eingabe:** ***Programm als Zeichenfolge***

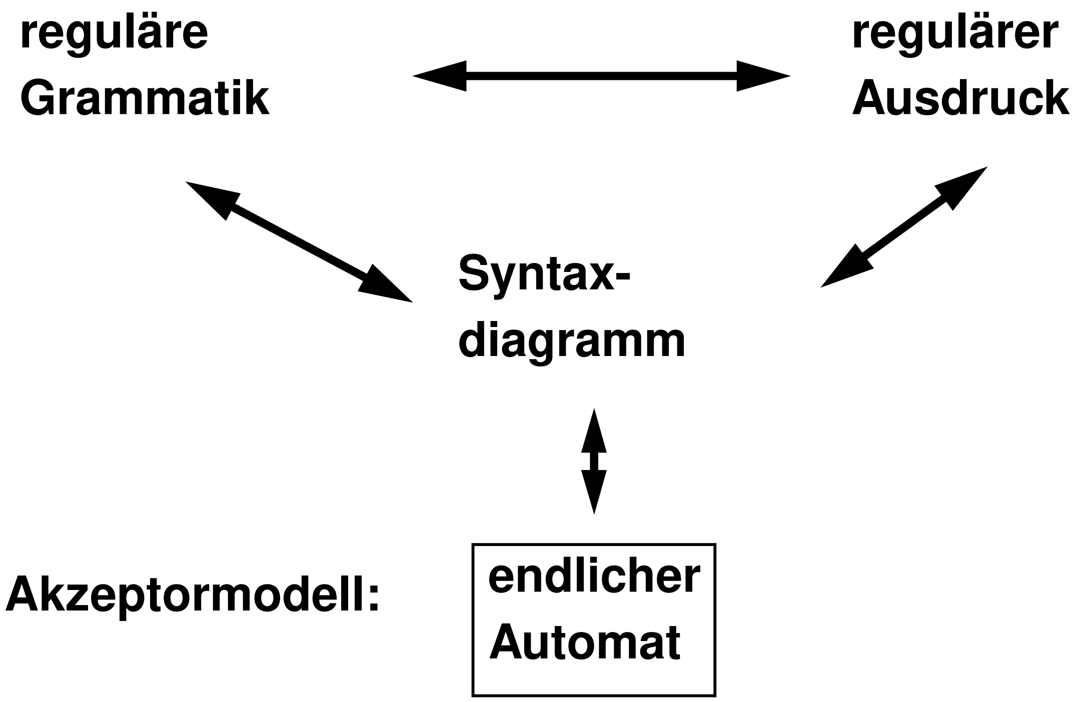
| **Aufgaben:** | **Übersetzermodul:** |
|---|---|
| | Eingabeverarbeitung |
| Erkennen und klassifizieren der Symbole | Scanner (zentrale Phase, endlicher Automat) |
| Ausblenden bedeutungsloser Zeichen | |
| Kodieren der Symbole | Bezeichnermodul |
| Speichern von Symbolinformation | Literalmodule |
| | Zeichenspeicher |
| Konversion | |

**Ausgabe:** ***Programm als kodierte Symbolfolge***

**Symbolrepräsentation:**

| **Syntax-Code** | **Attribut** | **Quellposition** |
|---|---|---|
| Terminal der konkreten Syntax | Wert oder Referenz auf Datenmodul | für Fehlermeldungen in späteren Übersetzerphasen |

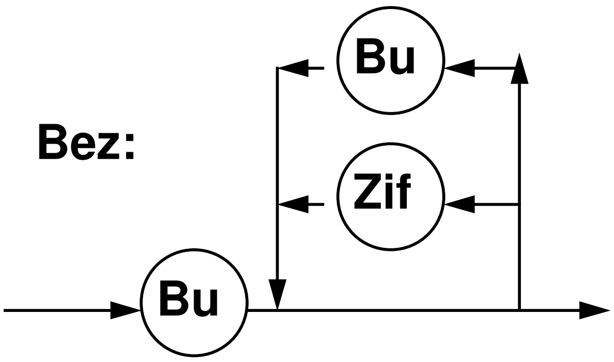
# Spezifikation von Grundsymbolen

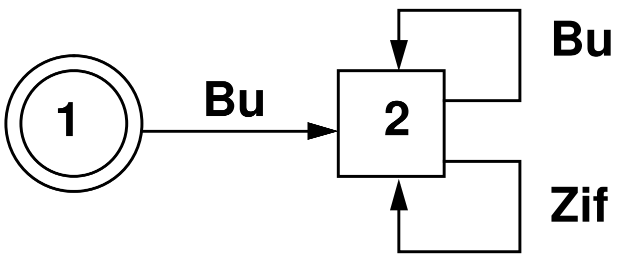



**Beispiel Bezeichner:**

```
Bez ::= Bu X
X   ::= Bu X
X   ::= Zif X
X   ::=
```

Bez = Bu (Bu | Zif)*



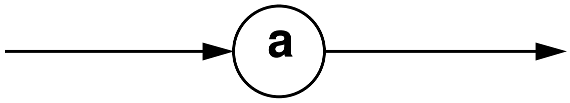
# Reguläre Ausdrücke in Syntaxdiagramme

## Transformationsregeln:

**regulärer Ausdruck A** — **Syntaxdiagramm für A**

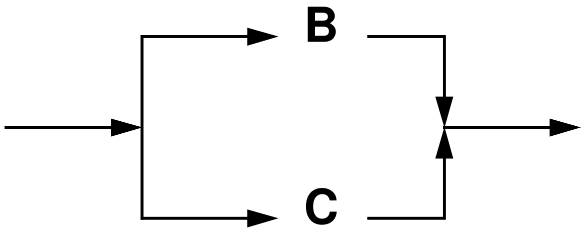
***leer***

**a**

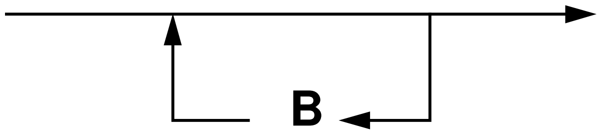



**B C**

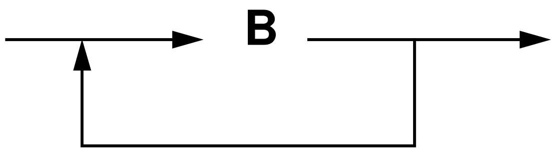

**B | C**




**B***




**B+**



# Automat aus Syntaxdiagramm konstruieren

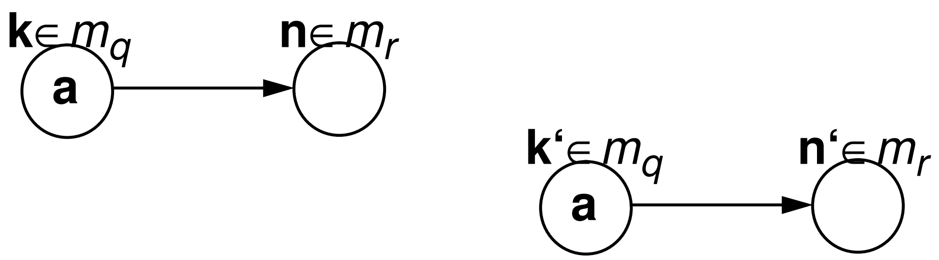
| **Syntaxdiagramm** | **Automat** |
|---|---|
| Knoten, Kanten | Übergänge, Zustände |
| **Knotenmenge** $m_q$ | **Zustand** $q$ |
| unter gleichem Zeichen *a* verbundene Knotenmengen $m_q$ und $m_r$ | Übergänge $q$ ---> $r$ mit Zeichen *a* |

**Konstruktion:**

1. **Knoten numerieren**
   Diagrammausgang = *0*

2. **Anfangsmenge** $m_1$ — **Anfangszustand** *1*
   enthält alle vom Anfang erreichbaren Knoten

3. **konstruiere neue Knotenmengen** und **Übergänge**
   für ein Zeichen *a* und eine Menge $m_q$ mit Knoten *k*
   bilde Knotenmenge $m_r$ mit allen Knoten *n*, wobei gilt




4. **Schritt 3 wiederholen** bis keine neuen Knotenmengen

5. Alle Zustände *q* sind **Endzustand** gdw. *0* in $m_q$ ist.

# Teilautomaten zusammensetzen

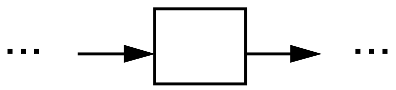
- **Anfangszustände der Teilautomaten identifizieren**

  ggf. auch Übergänge unter gleichem Zeichen identifizieren
- **unterschiedliche Endzustände beibehalten**

  klassifizieren die Symbole
- **Teilautomaten für Kommentare und bedeutungslose Zeichen zufügen**

**Übergänge aus Endzuständen:**

wann hält der Automat?



**Regel des längsten Musters:**

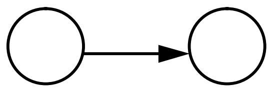
- **Automat hält in beliebigem Zustand, falls kein Übergang mit nächstem Zeichen möglich**
- **setzt zurück auf zuletzt durchlaufenen Endzustand**
- **Fehler, falls kein Endzustand durchlaufen**

Konsequenzen für die Trennung aufeinanderfolgender Symbole!

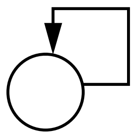
An der Sprachdefinition und der Syntax prüfen!

# Scanner: Implementierungsaspekte

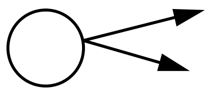
- **Laufzeit proportional zur Zahl der Zeichen des Programms**
- **Operationen pro Zeichen schnell - sonst dominiert der Scanner die Übersetzungszeit**
- **Tabellen-gesteuerte Automaten zu langsam**
  Schleife, 2-fache Array-Indizierung, Verzweigung
- **direkt programmierter Automat schneller**
  Übergänge in Programmstrukturen umsetzen:



**Sequenz**



**Schleife**



**Verzweigung o. Sprung**

- **schnelle Schleife für bedeutungslose Zwischenräume**
- **Klassen gleichartiger Zeichen: Bitmuster** oder **Indizierung -** statt teurer Mengenoperationen
- **Zeichen nicht aus Eingabepuffer kopieren, sondern**
  Zeiger in Eingabepuffer

# Bezeichner- und Literalmodule

- **einheitliche Schnittstelle:**

  Eingabeparameter: Zeiger auf Symboltext

  Ausgabeparameter: Attribut, Syntax-Code

- **Bezeichnermodul kodiert Bezeichner bijektiv, erkennt Wortsymbole**

  Implementierung: Hash-Vektor, erweiterbare Tabelle,

  Kollisionslisten

- **Literalmodule**

  (Gleitpunktzahlen, ganze Zahlen, Zeichenreihen)

  **Repräsentation zur Speicherung:**

  Quellsymbol,

  Wert konvertiert in Übersetzerdatum

  Wert konvertiert in Zieldarstellung

  **Vorsicht:**

  Überlauf bei Konversion abfangen!

  Übersetzerdatum evtl. verschieden von Zieldarstellung!

- **Zeichenspeicher**

  speichert Zeichenfolgen ohne Längenbegrenzung

  wird von den übrigen Modulen benutzt

# Scanner-Generatoren

**generieren die zentrale Funktion der lexikalischen Analyse**

| | |
|---|---|
| **GLA** | University of Colorado, Boulder; Komponente von Eli |
| **Lex** | Unix Standardwerkzeug |
| **Flex** | Nachfolgesystem zu Lex |
| **Rex** | GMD Karlsruhe |

**Symbolspezifikation: reguläre Ausdrücke**

| | |
|---|---|
| **GLA** | + programmierte Teilautomaten |
| | + Muster für vordefinierte Symbole |
| **Lex, Flex, Rex** | + programmierbare Zustandsübergänge |

**Schnittstellen:**

| | |
|---|---|
| **GLA** | wie in diesem Kapitel beschrieben, |
| | kooperiert mit anderen Eli-Komponenten |
| **Lex, Flex, Rex** | Aktionen zu reg. Ausdrücken für Symbole |
| | (beliebige Anweisungen der Impl.sprache), |
| | Schnittstelle zu Parser-Generator Yacc |

**Implementierung:**

| | |
|---|---|
| **GLA** | direkt programmierter Automat in C |
| **Lex, Flex, Rex** | Tabellen-gesteuerter Automat in C |
| **Rex** | auch in Modula-2 |
| **Flex, Rex** | schneller, kleiner als bei Lex |