Teil VII: Modelle, Algorithmen, Eigenschaften

1. Motivation
2. Grammatiken und Formale Sprachen
3. Formale Sprachen und Endliche Automaten
4. Suchen in Zeichenketten

© K. Murmann, H. Neumann, Fakultät für Informatik, Universität Ulm, 2000

Motivation

Suche nach Zeichenkombinationen

Beispiel

„Richard Feynman war sehr amerikanisch. Wie das große Land selbst, an dessen Ostküste er 1918 geboren und an dessen Ostküste er 1988 starb, so steckt ch voller Gegensätze. Während sich die USA zum Beispiel als Land von Mickey Mouse und Mondfahrt charakterisieren lassen, kann man Feynman als genialen Physiker und großen Kindskopf zugleich beschreiben. So wie sich in Amerika sowohl die größte Zahl der Analphabeten in der westlichen Welt als auch die meisten Nobelpreisträger in den Naturwissenschaften finden lassen, so stellt man in Feynman die höchste Originalität in der Physik und die platteste Banalität in Kunst und Philosophie fest. ...“

(aus E.P. Fischer. Einstein & Co. Piper, München, 1995)

Aufgaben und Fragen:

- Wir haben uns vertippt und **Feynmann** statt **Feynman** geschrieben. Wir wollen die Fehler korrigieren !
  Lösungsidee : Überprüfung des Textes von Anfang bis Ende auf Zeichenfolgen, die mit **Feynmann** übereinstimmen

- Die Jahreszahlen **19xx** müssen auf Richtigkeit überprüft werden !
  Lösungsidee : Überprüfung des Textes auf 4-ziffrige Zeichenfolgen mit **19** am Anfang sowie **2** weiteren beliebigen Ziffern

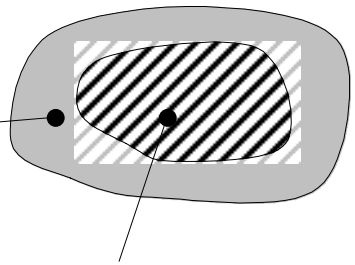
Arithmetische Ausdrücke

Aufgabe: Für geklammerte arithmetische Ausdrücke muss entschieden werden, ob sie korrekt gebildet sind !

Es sei:

- S ist eine **Zeichenmenge** (**Alphabet**), $S = \{(,),+,-,*,/,a\}$
  (a ist Platzhalter für beliebige Konstanten oder Variablen)
- eine **(formale) Sprache L** (über S) ist eine Teilmenge aus der Menge der **Konkatenationen** der Zeichen aus S (bezeichnet als) S^*

  hier: $EXPR \subseteq S^*$



Einige arithmetische Ausdrücke:

$$(a-a)*a+a/(a+a)-a \in EXPR$$

$$(((a))) \in EXPR$$

$$((a+)-a(\notin EXPR$$

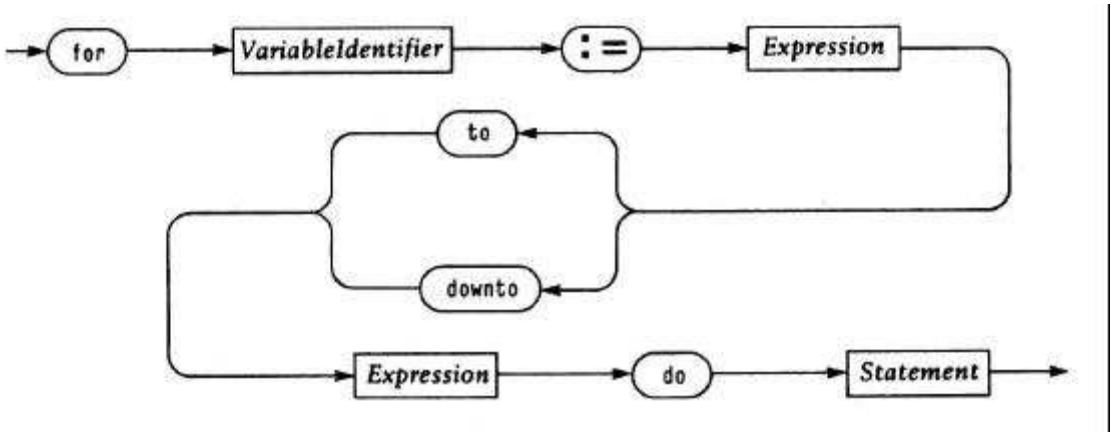
$$a*/a \notin EXPR$$

Problem: Sprachen sind allgemein unendliche Objekte (i.S. beliebig langer und strukturierter Zeichenfolgen), die selbst endlich beschrieben werden müssen !

➔ **Grammatiken** und **Automaten**

Interpretation / Entwurf syntaktischer Beschreibungen

geg.: 1. **Syntaxdiagramm** für Wiederholungsschleifen in der Programmiersprache **Pascal**



2. **EBNF**-Syntax für die Spezifikation von Zahlen in **Pascal**

Definition:

```
UnsignedNumber   = UnsignedInteger | UnsignedReal .
UnsignedInteger  = DigitSequence .
UnsignedReal     = UnsignedInteger "." DigitSequence
                     ["e" ScaleFactor ] |
                     UnsignedInteger "e" ScaleFactor .
ScaleFactor      = [ Sign ] UnsignedInteger .
Sign             = "+" | "-" .
DigitSequence    = Digit { Digit } .
```

Beispiele:

Examples of unsigned integers:

1 100 00100

Examples of unsigned reals:

0.1 0.1e0 87.35e+8 1E2

Einige Fragen und Anwendungen:

1. Dem Buch über die (Programmier-) Sprache „**Y**“ liegt eine **Syntaxbeschreibung** bei. Wie können wir diese lesen und verstehen ?

2. Sie haben für eine **interaktive Software eine Benutzerschnittstelle** entwickelt und müssen nun für die Benutzer die Formate der erlaubten Kommandos mit ihren Parametern spezifizieren. Wie kann man dies sauber (am besten) beschreiben ?

3. Sie wollen eine **Editierhilfe für ein Softwarewerkzeug** entwickeln und nehmen dabei Anleihen aus dem bekannten *vi*-Editor (so soll z.B. in einer Eingabe ein Textstück gesucht werden können mit **:/<string>**/). Wie lassen sich derartige Kommandos beschreiben und anschliessend realisieren ?

4. Wir haben eine **eigene (Programmier-) Sprache** entwickelt. Wie kann man einen Compiler für diese Sprache definieren, mit dem Programme in meiner neuen Sprache automatisch in ausführbaren Code übersetzt werden ?

Grammatiken und Formale Sprachen

Grammatik und Sprache

Def.:

Grammatik

$$G = (S_N, S_T, P, w_s)$$

S_N : Menge der **Nichtterminal-Zeichen** (**Variablen**)

S_T : Menge der **Terminal-Zeichen**

$P = \{p_i\}$, $|P| = n < \infty$: **Produktions- / Ableitungs- / Ersetzungsregeln**

$$P \subseteq (S_N \cup S_T)^{+} \times (S_N \cup S_T)^{*}$$

es können nur Zeichen echt **ersetzt** werden !

w_s : **Startvariable,** $w_s \in S_N \quad (|w_s| = 1)$

nur ein Zeichen !

Def.:

Sprache

$$L(G) = \{w \in S_T^* \mid S \xrightarrow{*}_G w\}$$

mit $\xrightarrow{*}_G$: endliche Folgen von Ableitungen (= (Einzel-) Ersetzungen) $\rightarrow_G$

Achtung: Die Bezeichnung hat nichts mit den **Variablen** (im Gegensatz zu **Konstanten**) eines **Programms** zu tun !

Erläuterungen:

1. Die **Variablen** legen die Menge der **nicht-terminalen Symbole** fest

2. Die **Terminalsymbole** bilden die Menge der „eigentlichen“ Symbole;
   jede Zeichenfolge einer mittels der Produktionen abgeleiteten Konstruktionen enthält nur Terminalsymbole !

3. Die **Regeln** der schrittweisen **Zeichenersetzungen** werden durch die **Produktionen** (Ableitungen) festgelegt;
   sie haben allgemein die Form:

   linke Seite ⟶ rechte Seite

   die erste Ersetzung beginnt stets bei der **Startvariablen** !

Ableitung einer Zeichenfolge (Wortproblem)

Problem :Es liegt eine Zeichenfolge w vor !

Frage : **Gehört w zur Sprache, d.h. ist** $w \in R(S)$ **?**

⇒ Dieses Entscheidungsproblem ist **nicht** für beliebige Grammatiken lösbar, jedoch für solche mit bestimmten Ersetzungsregeln !

Strategien :

1. **Top-Down**
   → ausgehend von der Startvariable wird solange abgeleitet, bis die gegebene Folge produziert wird !

2. **Bottom-Up**
   → ausgehend von der gegebenen Folge werden die Regeln „rückwärts“ angewandt, bis die Startvariable erreicht ist !

Bsp. :

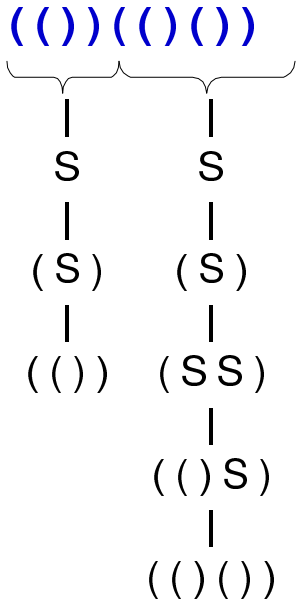
G = ({ S },
{ (,) },
{S $\xrightarrow{1}$ (), S $\xrightarrow{2}$ (S) , S $\xrightarrow{3}$ SS },
S)

„Ausgeglichene Klammern“

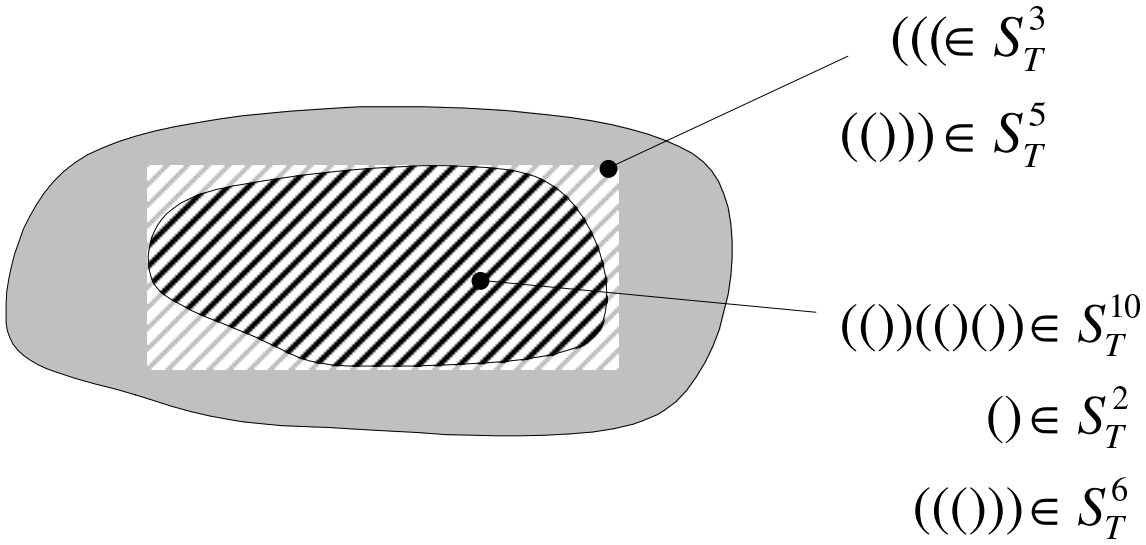
Frage : Gehört **(( )) (( ) ( ))** zur Sprache ?

Top-Down Ableitung :

S $\xrightarrow{3}$ **S S**
$\xrightarrow{2}$ **(S)** S
$\xrightarrow{1}$ (**( )**) S
$\xrightarrow{2}$ (( )) **(S)**
$\xrightarrow{3}$ (( )) (**S S**)
$\xrightarrow{1}$ (( )) (**( )** S)
$\xrightarrow{1}$ (( )) (( ) **( )**)



Aus der Menge aller durch Konkatenation aus Klammern über $S_T = \{(,)\}$ gebildeter Zeichenketten ($S_T^* = \{(,)\}^*$) werden **nur die ausgeglichenen (balancierten) Konstruktionen** generiert !



Konkrete Anwendungen :

- Prüfung der Gültigkeit geschachtelter Ausdrücke (**BEGIN** ≡ **(** , **END** ≡ **)**)
- Editor mit automatischer Prüfung ausgewogener Ausdrücke während der Eingabe **(... (** (**...)**

Chomsky-Hierarchie

Def.:

Typ 0 – Grammatik

- jede Grammatik ist zunächst automatisch Typ 0 !
- Produktionen $(w_1 \rightarrow w_2 \in P)$ keine Einschränkungen
  (→ allgemeine Phrasenstruktur-Grammatiken)

Typ 1 – Grammatik

- kontextsensitiv
- Produktionen $(w_1 \rightarrow w_2 \in P)$ mit $|w_1| \leq |w_2|$

Kontextsensitivität (Bsp.) :

Produktionsregeln der Art

$$\begin{array}{lcl} \alpha A \beta & \rightarrow & \alpha \gamma \beta \\ \delta A \eta & \rightarrow & \delta \phi \eta \end{array} \quad \text{mit} \quad |w_1| = |w_2|$$

Def.:

Typ 2 – Grammatik

- kontextfrei
- Produktionen $(w_1 \to w_2 \in P)$ mit $|w_1| = 1, w_1 \in S_N$

  d.h Produktionen sind Einzelzeichenersetzungen

Typ 3 – Grammatik

- regulär
- Produktionen $(w_1 \to w_2 \in P)$ mit $|w_1| = 1, w_1 \in S_N$

  und zusätzlich : $w_2 \in S_T \cup S_T S_N$

  d.h. die Ableitungen sind alle von der Form

  - $\rho : B \to \gamma$ einzelne Terminalzeichen, oder
  - $\rho : B \to \gamma A$ ein Terminalzeichen, gefolgt von einer Variablen (**Links**ableitung)

  (→ bei **Rechts**ableitungen ist die 2. Ersetzungsregel stets $\rho : B \to A\gamma$

Eine **Sprache** $L \subseteq S_T^*$ heisst vom Typ 0, 1, 2, 3, falls es eine Typ 0, 1, 2, 3 – Grammatik G mit L(G) = L gibt !

Syntaxbäume (Ableitungsbäume)

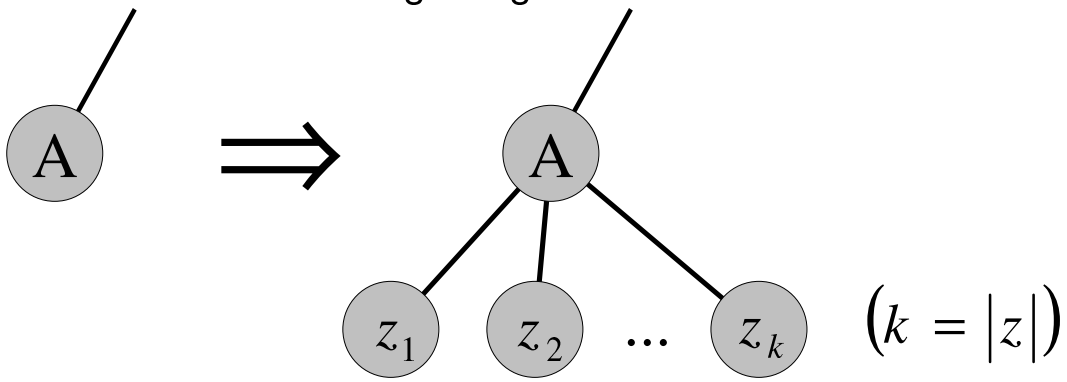
- **Ableitungen** eines Wortes w aus einer **Typ 2** (oder **Typ 3**) Grammatik G kann man einen **Ableitungsbaum** zuordnen !

geg.: $w \in L(G)$,

$$S = w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n = w$$

ist eine Ableitung des Wortes w

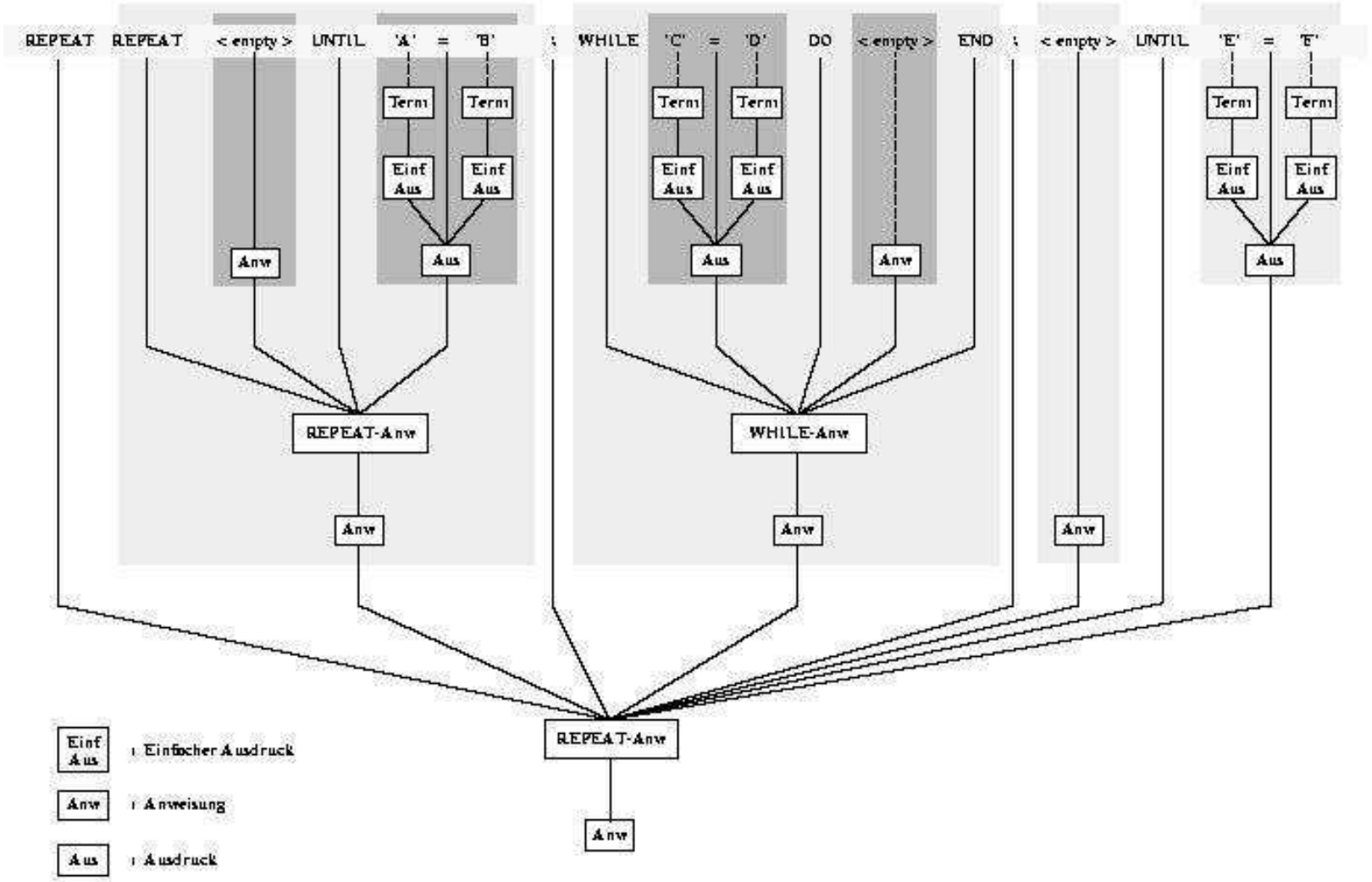
- i = 0 Startvariable S ist die Wurzel des Ableitungsbaums
- i = 1, 2, ..., n für j-ten Ableitungsschritt (Übergang $w_{j-1} \Rightarrow w_j$) mit Ersetzung $(A \rightarrow z) \in P$ werden im Baum am Knoten A |z| Söhne angehängt und mit den Zeichen von z beschriftet



Bsp.: Ableitung einer MODULA-2 Programmsequenz
(zur Anschauung ...)

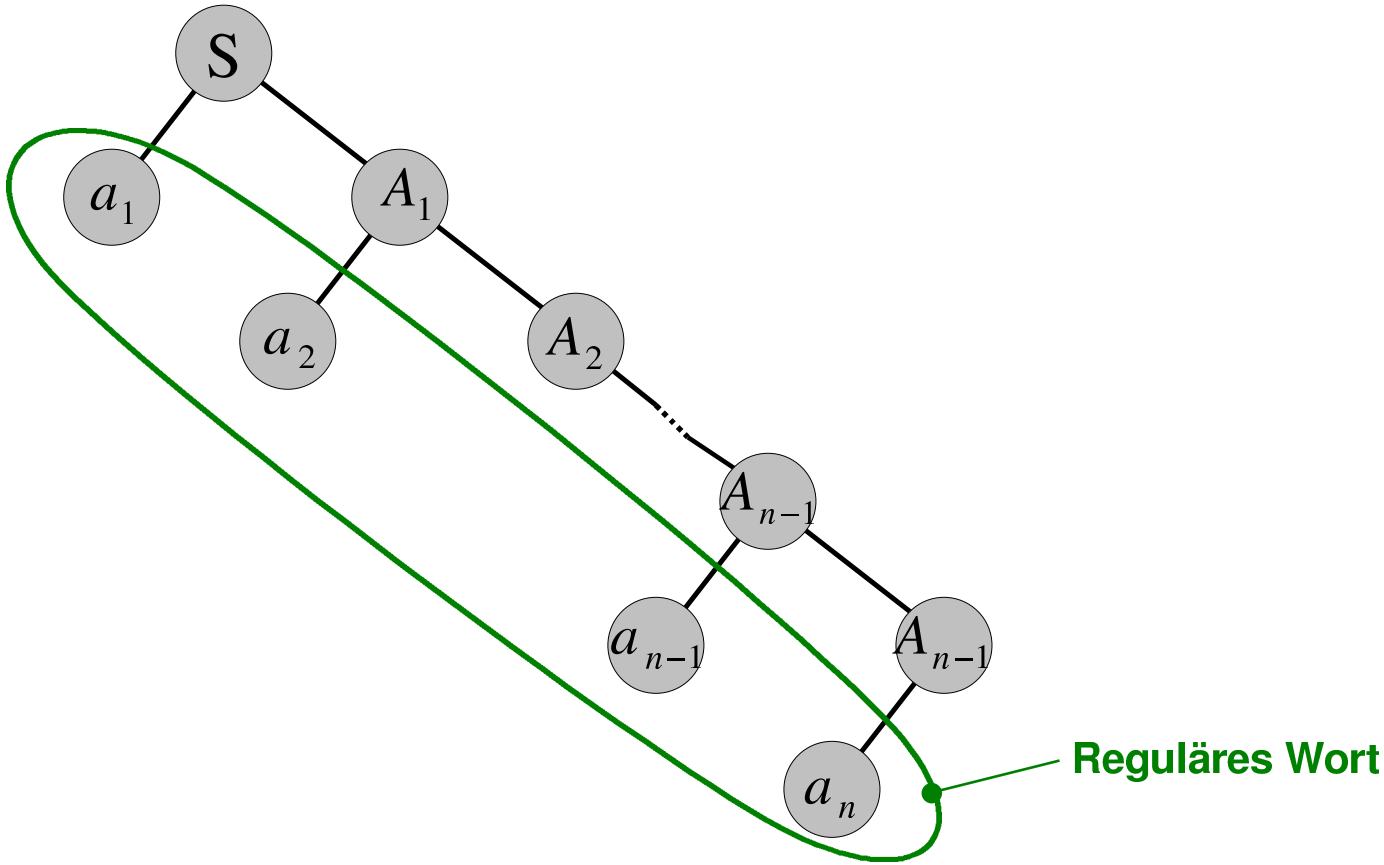
```
REPEAT
    REPEAT
    UNTIL A = B;
    WHILE C = D DO
    END;
UNTIL E = F
```

Achtung
Endlosschleife



- Ableitungsbäume bei **regulären Grammatiken** sind stets „entartet“, d.h. es werden lineare Ketten mit geneigten Bäumen erzeugt !

  hier: **Links**ableitung



Backus-Naur-Form

Backus-Naur-Form (BNF)

→ Formalismus zur Darstellung einer **kontextfreien Grammatik** (Typ 2)

Die Darstellung von Regeln mit derselben linken Seite wird zusammengefasst

$$\left.\begin{array}{l} A \rightarrow \beta_1 \\ A \rightarrow \beta_2 \\ \quad\vdots \\ A \rightarrow \beta_n \end{array}\right\} \text{ wird zu einer Meta-Regel } \qquad A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

mithilfe des Meta-Symbols **|** (Alternative, Auswahl)

(Hinweis: Backus / Naur verwenden statt → das Zeichen ::=)

Erweiterte BNF (EBNF)

Einführung meta-sprachlicher Nicht-Terminalsymbole, insbesondere **Klammerkonstrukte** :

$\rightarrow$	Ableitungssymbol (Original ::=)
.	Markierung des Endes einer Ableitungsregel
|	Alternative (Auswahl)
[…]	Optionale Ausdrücke : 0- oder 1-mal auftretend
{ … }	Optionale Ausdrücke : 0-, 1- oder mehrmals auftretend
(…)	Präzedenzregelnde (Strukturierungs-) Klammern

Erweiterungen durch die **E**BNF !

Beispiel: Syntax für ganze Dezimalzahlen (evtl. mit Vorzeichen)

Herkömmliche Notation :

⟨GanzeZahl⟩ → ⟨Zahl⟩.
⟨GanzeZahl⟩ → ⟨Vorzeichen⟩⟨Zahl⟩.
⟨Zahl⟩ → ⟨Ziffer⟩.
⟨Zahl⟩ → ⟨Ziffer⟩⟨Zahl⟩.
⟨Vorzeichen⟩ → +.
⟨Vorzeichen⟩ → −.
⟨Ziffer⟩ → 0.
⟨Ziffer⟩ → 1.
⟨Ziffer⟩ → 2.
⟨Ziffer⟩ → 3.
⟨Ziffer⟩ → 4.
⟨Ziffer⟩ → 5.
⟨Ziffer⟩ → 6.
⟨Ziffer⟩ → 7.
⟨Ziffer⟩ → 8.
⟨Ziffer⟩ → 9.

BNF :

⟨GanzeZahl⟩ → ⟨Zahl⟩ | ⟨Vorzeichen⟩⟨Zahl⟩.
⟨Zahl⟩ → ⟨Ziffer⟩ | ⟨Ziffer⟩⟨Zahl⟩.
⟨Vorzeichen⟩ → + | −.
⟨Ziffer⟩ → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

EBNF :

⟨GanzeZahl⟩ → [+ | −]⟨Ziffer⟩{⟨Ziffer⟩}.
⟨Ziffer⟩ → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

Syntaxdiagramme

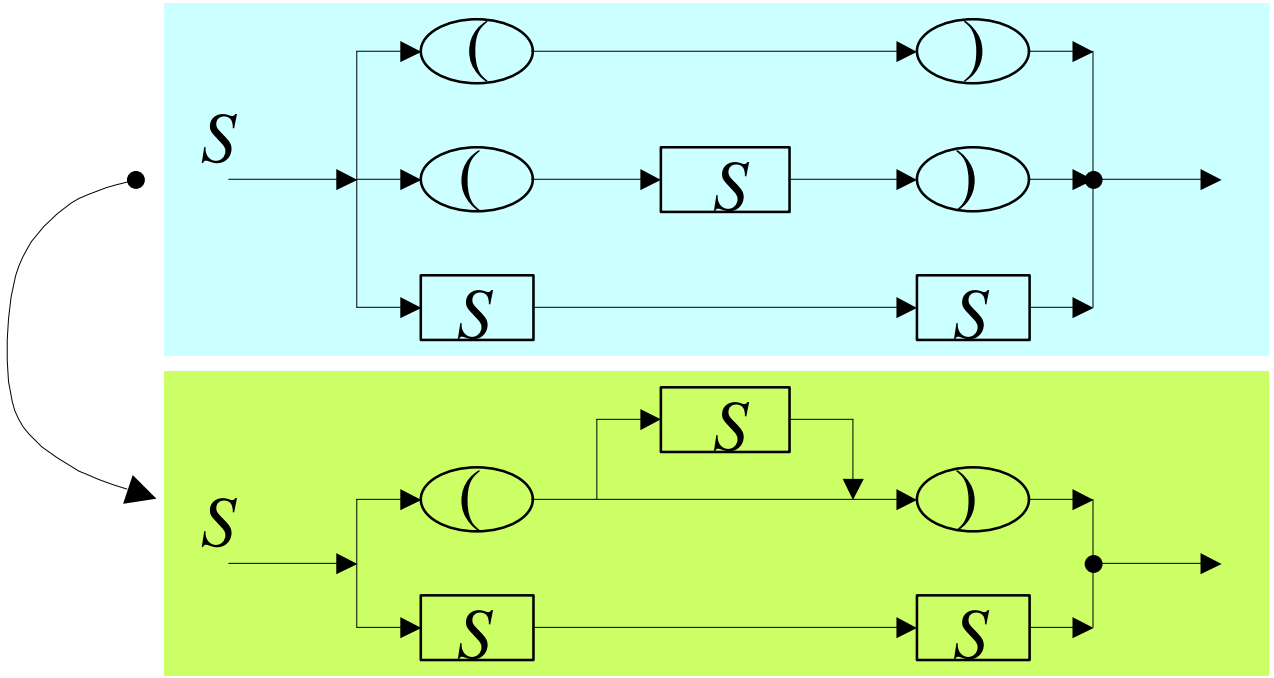
Umsetzung : Produktionen → Syntaxdiagramm

Bsp.: **„Ausgeglichene Klammern“**

1. **Produktionen und EBNF**

$$P \to \{S \to (), S \to (S), S \to SS\} \quad \Longleftrightarrow \quad P \to (\,[\langle S \rangle]\,) \mid \langle S \rangle \langle S \rangle$$

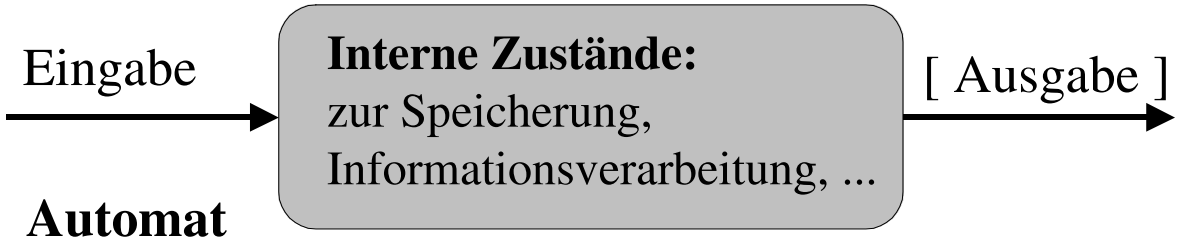
2. **Diagramme**



Formale Sprachen und Endliche Automaten

Endliche Automaten

Allgemeines Modell eines Automaten



Automat
(informationsverarbeitende Maschine)

Def.:

Deterministischer endlicher Automat M (DEA)
(„discrete finite automaton“, DFA) **ohne Ausgabe**

$$M = (Z, S, \delta, z_0, T)$$

Z : Menge der **Zustände** , $|Z| = n + 1 < \infty$

S : **Eingabe-Alphabet** , $|S| = m < \infty$

$\delta : Z \times S \to Z$: **Zustandsübergangs-Funktion**

$z_0 \in Z$: **Anfangs- / Startzustand**

$T \subseteq Z$: Menge der **End- / Terminalzustände**

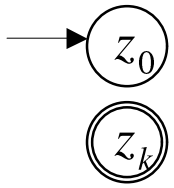
Zustandstafel und Zustandsübergangsfunktion (Tabellenform) :

Zustand \ Eingabe	s_1	…	s_k	…	s_m
z_0					
⋮					
z_i			z_j		
⋮					
z_n					

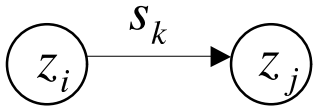
➔ Automat sei im Zustand z_i und erhält die Eingabe s_k,
dann geht der Automat in den Zustand z_j über !

Graphische Darstellung : **Zustandsdiagramm / -graph**

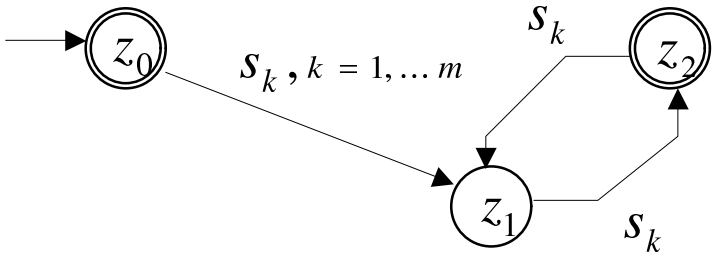
- **Knoten (= Zustände)**
  - **Anfangs- / Startzustand** (offener Eingang)
  - **End- / Terminalzustand** (Doppelkreis)



- **Kanten**
  - **Zustandsübergang** $\delta(z_i, s_k) = z_j$



Einfaches <u>Beispiel</u>:



Für welche Eingaben gelangt der Automat in einen Endzustand ?

Akzeptor einer Sprache

Ein endlicher Automat M beschreibt (erkennt, **akzeptiert**) eine Sprache $L \subseteq S^*$

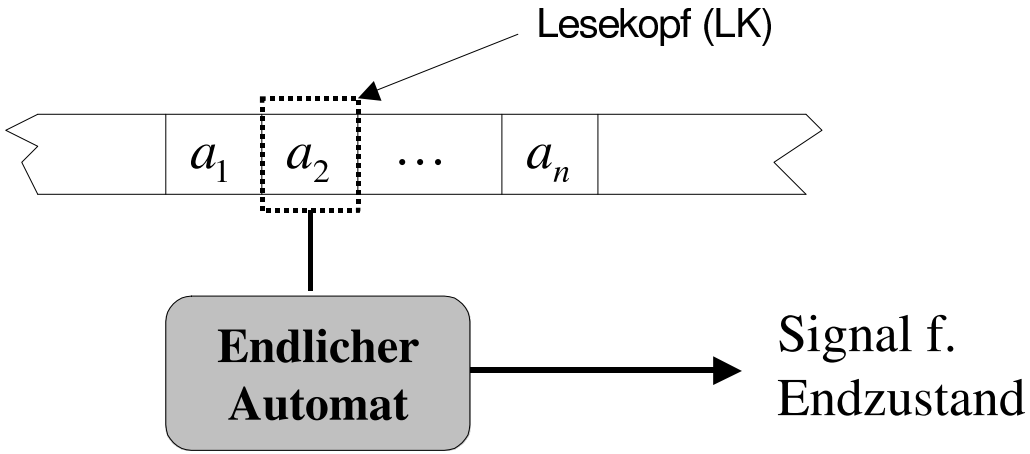
- ein Wort $a_1 a_2 \ldots a_n$ wird elementweise gelesen
- der Automat durchläuft schrittweise eine Folge von Zuständen $z_1, z_2, \ldots, z_n$ mit

  z_0 : Startzustand

  $\delta(z_{i-1}, a_i) = z_i$, für i = 1, .., n

- es muss abschliessend gelten

  $z_n \in T$



Def.:

Erweiterung der Zustandsübergangsfunktion von Einzelzeichen zu Wörtern

geg.: $M = (Z, S, \delta, z_0, T)$

Induktive Festlegung einer neuen Funktion

$$\hat{\delta} : Z \times S^* \rightarrow Z$$

mit $\hat{\delta}(z, \lambda) = z$

$\hat{\delta}(z, ax) = \hat{\delta}(\delta(z, a), x)$ **wobei** $z \in Z, x \in S^*, a \in S$

Es gilt : $\hat{\delta}(z, a) = \delta(z, a), a \in S$

$$\hat{\delta}(z, a_1 a_2 \ldots a_n) = \delta(\ldots \delta(\delta(z, a_1), a_2) \ldots, a_n)$$

→ vgl. Ableitungsregeln von **regulären Sprachen** !

Die von M akzeptierte Sprache ist

$$T(M) = \left\{ x \in S^* \middle| \hat{\delta}(z_0, x) \in T \right\}$$

Satz:

Jede durch endliche Automaten erkennbare Sprache ist regulär (also Typ 3)

(ohne Beweis !)

Bsp.: **geg.:** $M = (S, Z, \delta, z_0, T)$

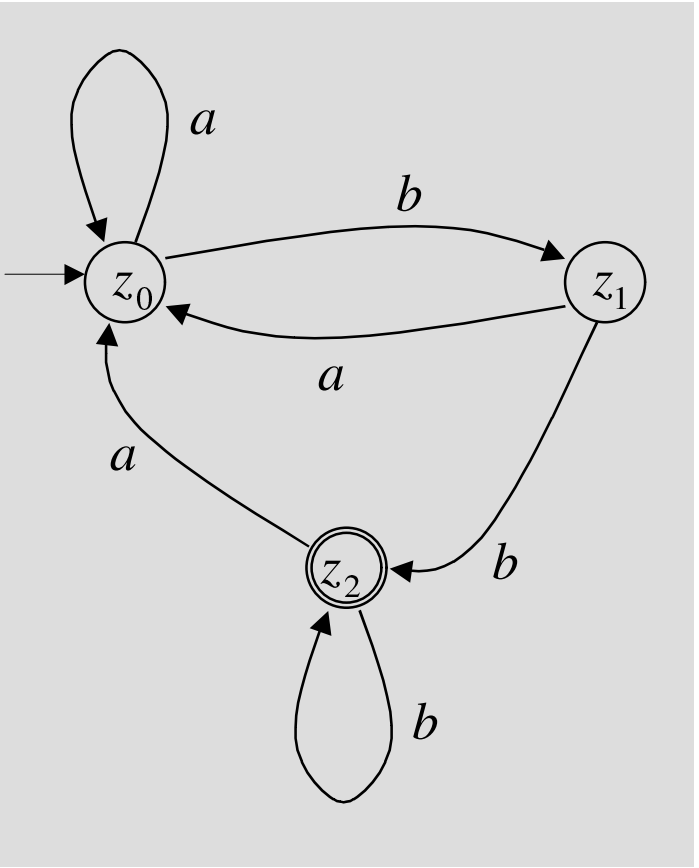
$S = \{a, b\}$

$Z = \{z_0, z_1, z_2\}$

$T = \{z_2\}$

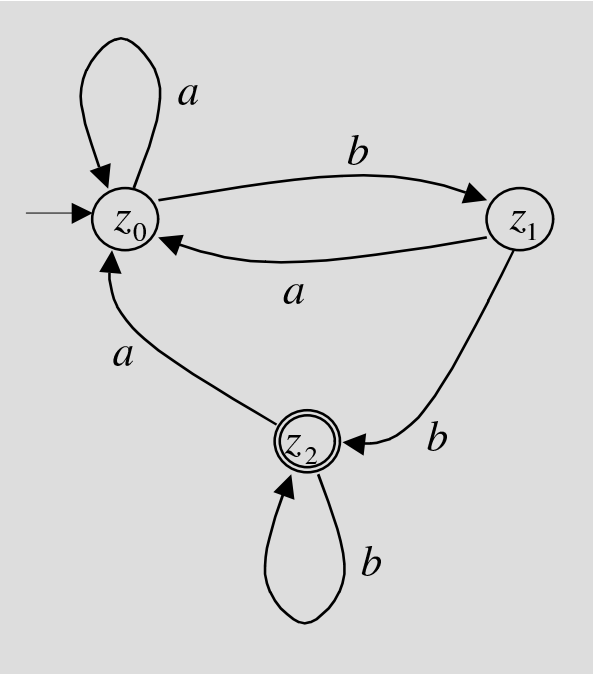
δ :

	a	b
z_0	z_0	z_1
z_1	z_0	z_2
z_2	z_0	z_2



Bsp. (cont'd)

Eingaben	End-Zustand
aab	$\hat{\delta}(z_0, aab) = z_1$
$baba$	$\hat{\delta}(z_0, baba) = z_0$
abb	$\hat{\delta}(z_0, abb) = z_2$
$abbabb$	$\hat{\delta}(z_0, abbabb) = z_2$
bba	$\hat{\delta}(z_0, bba) = z_0$



Frage : **Was leistet der Automat ?**

$\Rightarrow$ **Akzeptor für** **... bb !**

Simulation

Algorithmus eines deterministischen endlichen Automaten (DEA):

BEGIN
  Bringe EA in Zustand z_0;
  Setze LK über linkes Zeichen des Eingabewortes (falls vorhanden);
  WHILE Zeichen unter LK vorhanden **DO**
    Gehe in Folgezustand gemäss $\delta : Z \times S \rightarrow Z$;
    Bewege LK um ein Feld nach rechts
  END (* WHILE*)
END.

Reguläre Ausdrücke

Def.:

Reguläre Ausdrücke (induktiv)

- die leere Menge $\{\}$ ist ein regulärer Ausdruck
- das leere Wort λ ist ein regulärer Ausdruck
- jedes Terminalsymbol $a \in S$ ist ein regulärer Ausdruck
- geg. : reguläre Ausdrücke α, β
  dann sind $\alpha\beta$, $(\alpha \mid \beta)$, $(\alpha)^*$ reguläre Ausdrücke

  Auswahl, Folge

➔ Meta-Symbole: ( ) , * , |

Zuordnung einer **regulären Sprache L(γ)** (γ ist ein regulärer Ausdruck)

- $= \{\}$ $\Rightarrow$ $L(\gamma) = \{\}$
- $\gamma = \lambda$ $\Rightarrow$ $L(\gamma) = \{\lambda\}$
- $= a$ $\Rightarrow$ $L(\gamma) = \{a\}$
- $\gamma = \alpha\beta$ $\Rightarrow$ $L(\gamma) = L(\alpha)L(\beta)$
- $= (a \mid b)$ $\Rightarrow$ $L(\gamma) = L(\alpha) \cup L(\beta)$
- $\gamma = (\alpha)$ $\Rightarrow$ $L(\gamma) = L(\alpha)^*$

Bsp.: **geg.:** **regulärer Ausdruck** $(0 \mid (0 \mid 1)^* 00)$

In der Sprache enthalten sind alle Formeln mit

- x = **0** oder
- x endet mit **...00**

Grammatik :

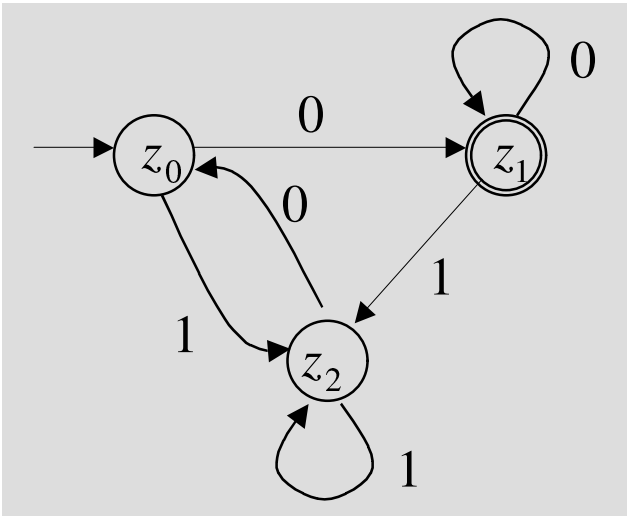
{ 0, 1 },

{A → 0 | B, B → 0C, C → D0, D → , D → D0 },

$$M = (\{z_0, z_1, z_2\}, \{0,1\}, \delta, z_0, \{z_1\})$$

δ :

	0	
z_0	z_1	z_2
z_1	z_1	z_2
z_2	z_0	z_2



Konkrete Anwendungen :

- Zur Erinnerung (**Teil IV / 3 –** Modularisierung und Programmentwicklung) : Telegramme

Eingabe-Format als regulärer Ausdruck :

```
( zzzzzz (b)+ ((c)*(b))* <STOP> (b)+ <STOP> (b)+ )+
```

mit z ∈ {0, ..., 9}, b : Leerzeichen, c : Druckzeichen,
(.)+ : Metazeichen zur 1- oder mehrmaligen Wiederholung,
(.)* : Metazeichen zur 0- oder mehrmaligen Wiederholung

Beispiel :

```
635444 WEIHNACHTEN IST NAHE STOP STOP 634560 IM
WINTERSEMESTER GIBT ES EINE ERHOLUNGSPAUSE
IN DER MITTE DER VORLESUNGSZEIT STOP UND BALD
IST ES SOWEIT STOP STOP
```

- Reguläre Ausdrücke und UNIX-Kommandos

`c`	Terminalzeichen `c`
`\c`	Escapesequenz, oder falls `c` Meta-Zeichen, `c` als Terminalsymbol
`^`	Anfang (Zeile, Wort) – Meta-Zeichen
`$`	Ende (Zeile, Wort) – Meta-Zeichen
`.`	beliebiges Einzel-Zeichen – Meta-Zeichen
`[abc...]`	ein Terminalzeichen aus der Aufzählung zwischen `[]`
`[^abc...]`	alle Zeichen als Terminalzeichen außer den Aufgezählten in `[]`
`[a-f]`	alle Zeichen zwischen `a` und `f` nach ASCII-Ordnung als Terminalzeichen
`[^a-f]`	alle Zeichen außer denen zwischen `a` und `f` als Terminalzeichen
`r1|r2`	`r1` oder `r2`
`(r1)(r2)`	`r1` gefolgt von `r2`
`(r)*`	beliebige viele `r`'s hintereinander (auch keins)
`(r)+`	beliebig viele, aber mindestens ein `r`
`(r)?`	ein oder kein `r`
`(r)`	Klammern dienen der Gruppierung – Meta-Zeichen

Metazeichen sind also: `( ) [ ] \ . ^ + * ? $`

- UNIX-Kommando **egrep** : Sucht in einer Datenmenge (Standardeingabe, Datei)

Aufruf: **egrep** options pattern {file}

pattern regulärer Ausdruck (mit Meta-Symbolen gebildet)

Bsp.1: Der Beispieltext vom Anfang (plus einiger Zeilen ...) – Datei „SuchText.txt“

```
Richard Feynman war sehr amerikanisch. Wie das grosse
Land selbst, an dessen Ostkueste er 18 geboren wurde
und an dessen Westkueste er 88 starb, so steckte auch
er voller Gegensaetze. Waehrend sich die USA zum Beispiel
als Land von Mickey Mouse und Mondfahrt charakterisieren
lassen, kann man Feynman als genialen Physiker und grossen
Kindskopf zugleich beschreiben. So wie sich in Amerika
sowohl die groesste Anzahl der Analphabeten in der
westlichen Welt als auch die meisten Nobelpreistraeger in
den Natuerwissenschaften finden lassen, so stellt man in
Feynman hoechste Originalitaet in der Physik und die
platteste Banalitaet in Kunst und Philosophie fest.
-------------
Noch'n paar Zeilen ...
Ab  Uhr gibt's Bier bis 23 Uhr -- danach Kaffee und Saft.
```

UNIX-Kommando: `egrep -n [ ] SuchText.txt > 19.doc`

```
2:Land selbst, an dessen Ostkueste er 18 geboren wurde
3:und an dessen Westkueste er 88 starb, so steckte auch
15:Ab  Uhr gibt's Bier bis 23 Uhr -- danach Kaffee und Saft.
```

UNIX-Kommando: `egrep -n [ ] SuchText.txt > 1900.doc`

```
2:Land selbst, an dessen Ostkueste er 18 geboren wurde
3:und an dessen Westkueste er 88 starb, so steckte auch
```

Bsp.2: Ein Modula-2 Programm – Datei „power.m2“

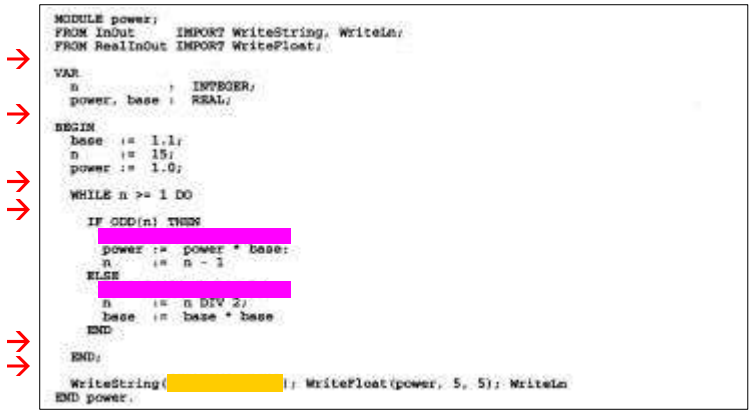
```
MODULE power;
FROM InOut      IMPORT WriteString, WriteLn;
FROM RealInOut IMPORT WriteFloat;

VAR
  n            :  INTEGER;
  power, base  :  REAL;

BEGIN
                }

  WHILE n >= 1 DO

    IF ODD(n) THEN
      (* -- n ungerade -- *)

    ELSE
      (* -- n gerade -- *)

    END

  END;

  WriteString("Ergebnis :  "); WriteFloat(power, 5, 5); WriteLn
END power.
```

Programm berechnet den Wert $1.1^{15} = 4.17725$

Aufgabe: **Alle Zuweisungen anzeigen !**

UNIX-Kommando: **`egrep -n`** [redacted] **`power.m2 > powerAssign.doc`**

```
10:  base  :=  1.1;
11:  n     :=  15;
12:  power :=  1.0;
18:      power :=  power * base;
19:      n     :=  n - 1
22:      n     :=  n DIV 2;
23:      base  :=  base * base
```



```
MODULE power;
FROM InOut     IMPORT WriteString, WriteLn;
FROM RealInOut IMPORT WriteFloat;

VAR
  n           :  INTEGER;
  power, base :  REAL;

BEGIN
  base  :=  1.1;
  n     :=  15;
  power :=  1.0;

  WHILE n >= 1 DO

    IF ODD(n) THEN
      power :=  power * base;
      n     :=  n - 1
    ELSE
      n     :=  n DIV 2;
      base  :=  base * base
    END

  END;

  WriteString(             ); WriteFloat(power, 5, 5); WriteLn
END power.
```

Aufgabe: **Alle (nicht-leeren) String-Konstanten anzeigen !**

UNIX-Kommando: `egrep -n [          ] power.m2 > powerStringConst.doc`

```
28:  WriteString("Ergebnis :  "); WriteFloat(power, 5, 5); WriteLn
```

Aufgabe: **Wieviele Leerzeilen ?**

UNIX-Kommando: `egrep -n [    ] power.m2 | wc -l > powerLeer.doc`

wc options {file}

options: clw – Zeichen, Zeilen, Worte

```
     6
```

Aufgabe: **(Reine) Kommentarzeilen ?**

UNIX-Kommando: `egrep -n [          ] power.m2 > powerComment.doc`

```
17:      (* -- n ungerade -- *)
21:      (* -- n gerade -- *)
```

Suchen in Zeichenketten

Fragestellung

Beispiel

geg.: String (Text)

„Als Strukturwissenschaften wird man nicht nur die reine und angewandte Mathematik bezeichnen, sondern auch das in seiner Gliederung noch nicht voll durchschaute Gebiet der Wissenschaften, die man mit Namen wie Systemanalyse, Informationstheorie, Kybernetik, Spieltheorie bezeichnet. Sie sind gl
Mathematik zeitlicher Vorgänge, die durch menschliche Entscheidung, durch Planung, durch Strukturen, die sich darstellen lassen, als seinen sie geplant, oder schließlich durch Zufall gesteuert werden. Sie sind also Strukturtheorien zeitlicher Veränderung. Ihr wichtigstes praktisches Hilfsmittel ist der Computer, dessen Theorie selbst eine der Strukturwissenschaften ist. ...“

(aus C.F. v. Weizsäcker. Die Einheit der Natur. dtv, München, 1974)

Pattern (Muster) : **Kybernetik**

Aufgabe :

Wir wollen ein Programm entwickeln, das in dem angegebenen **Text** ein bestimmtes **(Vergleichs-) Muster** sucht und im Falle des Vorhandenseins die **Position (Index) des ersten Zeichens des Musters** ausgibt !

Randbedingungen für die Realisierung – Spezifikation und Entwurf

a) Wie wird das Ende einer Zeichenkette („string“) gekennzeichnet ?

Realisierung : 0-Character (0C, CHR(0)) (Modula-2)

b) Wie wird die Zeichenkette repräsentiert ?

Vorschlag : Feld (ARRAY) fester maximaler Länge, A[1..maxLENGTH]

c) Was liefert die Vergleichs-Prozedur als Ergebnis ?

Vorschlag : wenn gefunden : pos ∈ {1, 2, ..., maxLENGTH}

wenn nicht gefunden : pos := UNDEFINED (≡ -1)

d) Wie soll das Ergebnis lauten, wenn der gesuchte Text mehrmals vorkommt ?

Vorschlag : Position des 1. Auftretens wird geliefert

(wenn pos < maxLENGTH – LENGTH(pattern), dann kann für den restlichen Text nochmals das Muster gesucht werden !)

Einfacher Lösungsansatz

Struktur

geg.: String (Text) **PETER BAUER HOERT BACH UND SINGT(0C)**

Muster **BACH**

- **Regulärer Ausdruck**

  → vgl. Struktur der „String-Anfragen“ in UNIX

  UNIX-Kommando: **`egrep ´BACH´ String.txt`**

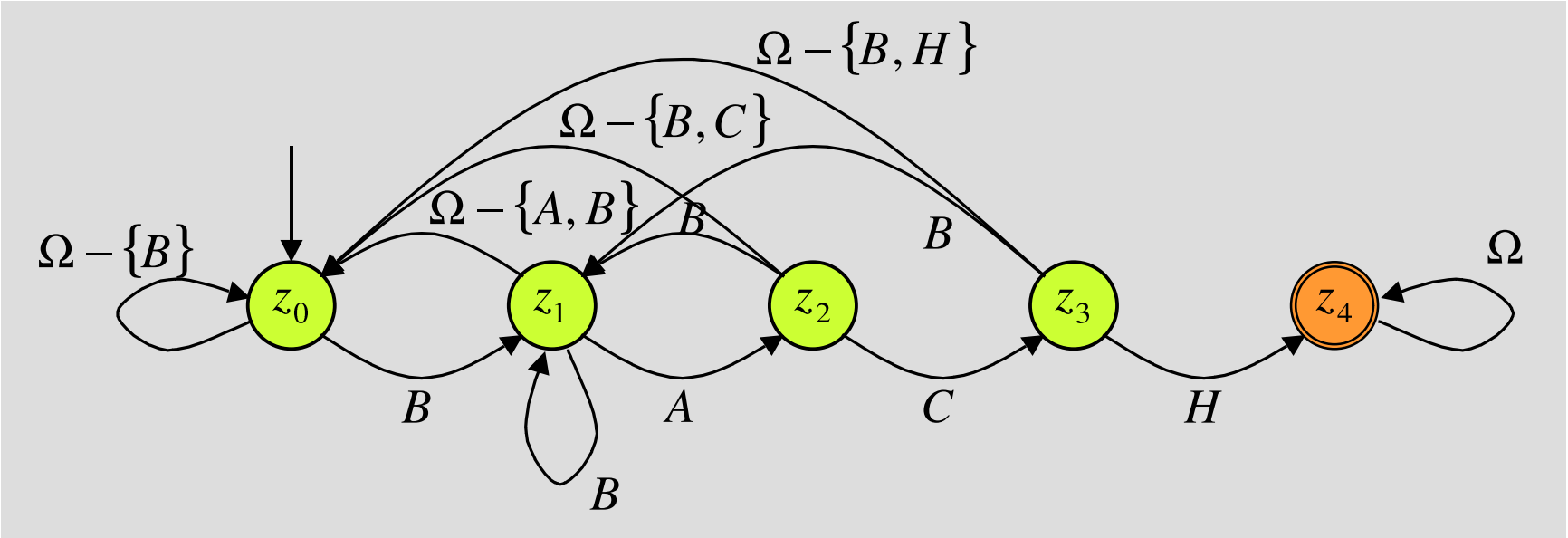
- **Automat** $M = (S, Z, \delta, z_0, T)$

  Ziel : Definition eines Akzeptors für eine Zeichenkette, die als Teilkette mindestens 1-mal die Folge „**BACH**“ enthält !

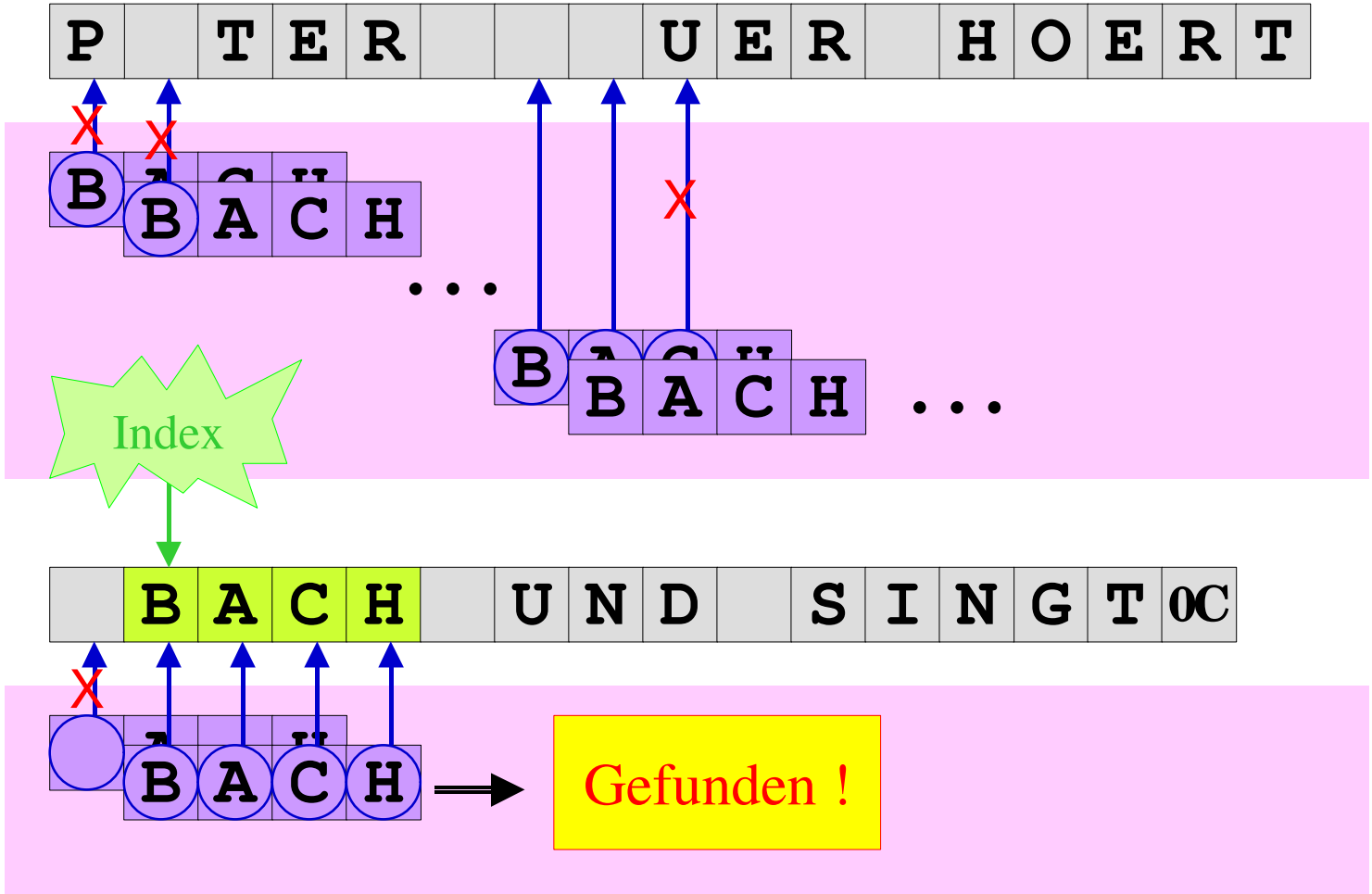
: $S = \{A, B, \ldots, Z\} = \Omega$

ände $Z = \{z_0, z_1, z_2, z_3, z_4\}$

Terminalzustand : $T = \{z_4\}$



- **Ablauf**


P T E R U E R H O E R T
B A C H
B A C H
Index
B A C H U N D S I N G T 0C
B A C H
Gefunden !

Programm Realisierung des Automaten

Besonderheiten :

neuer Datentyp FILE

Dateioperationen :

- Fopen(...)
- Fclose(...)

Eingabe :

- FreadChar(...)
- Done (~ Flag zur Rückmeldung der erfolgreichen Leseoperation)

- Ein-/Ausgabe-Ströme als Dateien

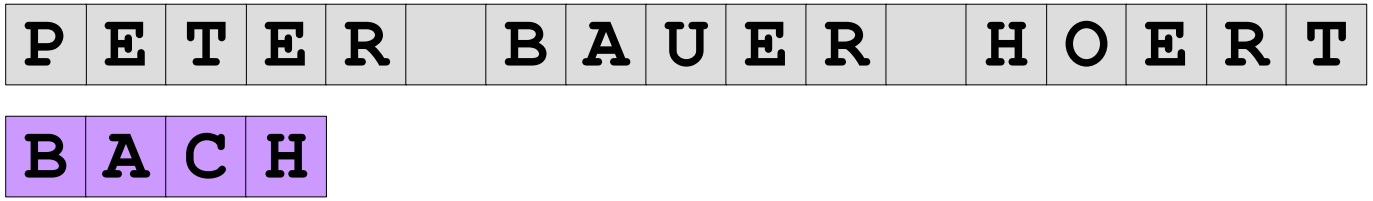
  stdin

```
MODULE SucheMuster;
   FROM ASCII    IMPORT nl;
   FROM InOut    IMPORT ReadString, WriteInt, WriteLn, WriteString, Write, Read;
   FROM FtdIO    IMPORT Done, FreadChar;
   FROM StdIO    IMPORT stdin, read, FILE, Fopen, Fclose;
   FROM Strings IMPORT StrLen;

   CONST
      maxLENGTH    =  100000;
      notInString =  -1;

   TYPE
      string =  ARRAY [1..maxLENGTH] OF CHAR;

   VAR
      filename, text, pattern :  string;
      index                   :  INTEGER;
      file                    :  FILE;
      result                  :  BOOLEAN;
      ch                      :  CHAR;

   PROCEDURE ReadLine(
         file :  FILE;
      VAR s    :  ARRAY OF CHAR) :  BOOLEAN;
   VAR
      i  :  INTEGER;
      ch :  CHAR;

   BEGIN
      i :=  0;
      FreadChar(file, ch);
      WHILE Done & (ch <> nl) & (i < INTEGER(HIGH(s))) DO
         s[i] :=  ch;
         INC(i);
         FreadChar(file, ch)
      END;

      s[i] :=  0C;
      IF Done & (ch = nl) THEN
         RETURN TRUE
      ELSE
         RETURN FALSE
      END
   END ReadLine;

   PROCEDURE ReadText(
      file  :  FILE;
      VAR s :  ARRAY OF CHAR) :  BOOLEAN;
   VAR
      i : INTEGER;
      ch: CHAR;

   BEGIN
      i := 0;
      FreadChar(file, ch);
      WHILE Done & (i < INTEGER(HIGH(s))) DO
         s[i] := ch;
         INC(i);
         FreadChar(file, ch)
      END;

      s[i] :=  0C;
      IF -Done THEN
         RETURN TRUE
      ELSE
         RETURN FALSE
      END
   END ReadText;

   PROCEDURE MatchSimple(
         s :        string;
         pattern :  ARRAY OF CHAR;
      VAR index  :  INTEGER);

   VAR
      plen, ppos, spos : INTEGER;

   BEGIN
      index :=  notInString;

      plen :=  StrLen(pattern);
      IF ( plen > 0 ) THEN
         spos :=  1;
         ppos :=  0;
         WHILE ( pattern[ppos] <> 0C ) & ( s[spos] <> 0C ) DO
            IF ( s[spos] = pattern[ppos]) THEN
               spos :=  spos + 1;
               ppos :=  ppos + 1
            ELSE
               spos :=  spos - ppos + 1;
               ppos :=  0
            END;
         END;

         IF ( pattern[ppos] = 0C ) THEN
            index :=  spos - plen
         ELSE
            index :=  notInString
         END
      END
   END MatchSimple;

BEGIN (* -- Hauptprogramm "SucheMuster" -- *)
   WriteString("Bitte den Namen der zu durchsuchenden Datei eingeben :  ");
   ReadString(filename);

   IF Fopen(file, filename, read, TRUE) THEN
      IF ReadText(file, text) THEN

         WriteString("Bitte Suchmuster eingeben :  ");
         IF ReadLine(stdin, pattern) THEN

            MatchSimple(text, pattern, index);

            IF index = notInString THEN
               WriteString("*** Muster nicht gefunden ! ***");
               WriteLn
            ELSE
               WriteString("Muster an Position ");
               WriteInt(index, 0);
               WriteString(" im Text gefunden !");
               WriteLn
            END
         ELSE
            WriteString("*** Fehler: konnte Muster nicht lesen ! ***");
            WriteLn
         END
      ELSE
         WriteString("*** Fehler: konnte Text nicht lesen ! ***");
         WriteLn
      END;

      result := Fclose(file);
   ELSE
      WriteString("*** Fehler: konnte Datei nicht oeffnen ! ***");
      WriteLn
   END
END SucheMuster.
```

N

Länge des Musters

a) Fall (schlechtesten Fall „worst case“) :



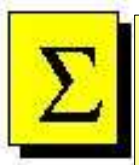
jedes Zeichen des Strings mit jedem Zeichen des Musters getestet werden !

⇒ $M * N$ Operationen

b) Normal-Fall („average case“ ~ mittlerer Aufwand) :

In den meisten Fällen wird der Vergleich des **String**s mit dem **Muster** (~ innere Schleife) frühzeitig abgebrochen !

⇒ $M + N$ Operationen



- Für den Entwurf von Algorithmen ist es sinnvoll, **Modelle** anzuwenden, deren Funktionalität formal beschrieben werden kann !
  Automaten (und ihr Bezug zu den **formalen Sprachen**) dienen als Formalismus zur Beschreibung einfacher Mechanismen als Folge von Zustandsübergängen

- **Grammatiken** bilden Systeme, mit denen Folgen von Zeichenketten **regelhaft** erzeugt werden können; diese Zeichenketten bilden gültige Worte einer durch die Grammatik definierten **Sprache**;
  Grammatiken (und die hieraus definierten Sprachen) bilden je nach Format der **Ersetzungs-** (**Produktions-**) **Regeln** eine Hierarchie (**Chomsky-Hierarchie**)

- Ableitungen **kontextfreier Sprachen** (Typ 2 & 3) lassen sich in Form von **Syntax-** oder **Ableitungsbäumen** darstellen; kompakte Darstellungen sind die **Backus-Naur-Form** (**BNF**) und **EBNF**

- **Endliche Automaten** (deterministisch und ohne Ausgabe) definieren einfach(st)e Maschinen mit Zuständen und Übergängen zwischen Zuständen; diese können **graphisch** mittels **Zustandsdiagrammen** (**-graphen**) dargestellt werden

- Endliche Automaten sind Akzeptoren **regulärer** (Typ 3) **Sprachen**

- Mit **regulären Ausdrücken** lassen sich komplexe Formate beschreiben, die als Muster (z.B. bei Such- und Vergleichsaufgaben, Textersetzungen in Editoren, etc.) dienen