

**Teil 2 Top-Down-Analyse**

Interpretieren wir Programme irgendeiner Programmiersprache als formale Sprache, so ist diese Sprache sicherlich nicht regulär. Schon die Möglichkeit von Klammerungen verhindert die Regularität, wie wir gesehen haben.

Die anstehende Frage ist, wie man algorithmisch beim Analysieren (**Parsen**) von Strings nichtregulärer Sprachen vorgehen soll. Eine häufige Antwort ist: Mit Unterstützung von **Stacks** (Kellerspeichern).

**Beispiel 1** (einfach geschachtelte, korrekte Klammern)

**Grammatik:**  $S \rightarrow \varepsilon \mid (S)$  , Startsymbol  $S$

**erster String:**     (     (     )     )  
                          1     2     3     4

Syntaxanalyse mit Stack (das **top-Element** ist **links** notiert):

Stack	Reststring	benutzte Produktion
S	( ( ) )	$S \rightarrow (S)$
(S)	( ( ) )	
S)	( ) )	$S \rightarrow (S)$
(S))	( ) )	
S))	) )	$S \rightarrow \varepsilon$
)	) )	
)	)	
$\varepsilon$	$\varepsilon$	Stop, String akzeptiert

Wenn wir die jeweils benutzten Produktionen zusammensetzen, so erhalten wir die zugehörige **Linksableitung**:

$$\begin{aligned}
 S &\rightarrow (S) \\
 &\rightarrow ( (S) ) \\
 &\rightarrow ( ( ) )
 \end{aligned}$$

aus der wir auch den **Syntaxbaum** gewinnen können:

```

      S
     / | \
  1(  S  )4
     / | \
    2( S )3
  
```

Der Aufbau des Syntaxbaums beginnt oben (**Top-Down**) und folgt generell dem Prinzip **Suche in die Tiefe** (also "immer links runter, sofern möglich").

**Frage:** Warum sind Stacks (mit ihrer LIFO-Logik, "Last in First Out") für die Analyse erforderlich und nicht etwa Schlangen (mit FIFO-Logik, "First in First Out") ?

**zweiter String:**     (     )     (     )  
                          1     2     3     4

Stack	Reststring	benutzte Produktion
S	(   ) (   )	S -> (S)
(S)	(   ) (   )	
S)	) (   )	S -> $\epsilon$
)	) (   )	
$\epsilon$	(   )	Fehler (Keller leer)

Der Automat "bleibt stecken". Der String wird abgelehnt und zwar zu Recht, denn die Klammern sind zwar korrekt, aber nicht *einfach* geschachtelt.

Bei den obigen beiden Durchläufen war aus der Situation heraus immer klar, was zu tun war, ob also Löschen eines Terminalsymbols (von Keller und Band) oder Pushen einer Produktion auf den Stack (und falls Pushen, dann Pushen welcher Produktion). - Man kann all diese Aktionen tabellarisch codieren, so daß man beim Abarbeiten eines Strings nur in der Tabelle nachsehen muß:

**Analysetabelle** (für Beispiel 1):

	(     )	b	b: Blank-Symbol
S	(S)	$\epsilon$	$\epsilon$

Die Einträge dieser Tabelle enthalten den mittels *push* eingekellerten String, welcher das oberste Kellerelement *top* ersetzt.

Man suche nach Eingabestrings, bei denen das rechte Feld dieser Analysetabelle zum Zuge kommt!

Was auffällt: Für unsere Analyse benötigen wir (abgesehen von Fehlerzustand/Stopzustand) lediglich einen einzigen Zustand. Explizit verwaltete Zustandsmengen werden wir erst bei LR-Parsern (Teil 3) wirklich benötigen.

**Algorithmisches Prinzip der Top-Down-Analyse (LL(1))**

Eine formale Sprache sei gegeben durch ihre Grammatik.

**Start:** Cursor an den Anfang des gegebenen Strings, Startsymbol  $S$  in den Stack.

**Iteration:** Falls der Keller leer ist  $\rightarrow$  Stop

Falls das oberste Kellerelement (top) ein Hilfssymbol  $T$  ist, dann wähle passende Produktion der Grammatik und ersetze  $T$  durch die rechte Seite der Produktion  $\rightarrow$  Iteration

Falls top ein Terminal  $t$  ist, prüfe, ob  $t$  mit dem aktuell gelesenen Zeichen übereinstimmt.

- Falls ja, streiche das top-Element aus dem Keller (also pop) und rücke den Cursor eine Position vor  $\rightarrow$  Iteration.
- Falls nein  $\rightarrow$  Stop

**Stop** Der gegebene String ist akzeptiert, falls er komplett gelesen wurde und außerdem der Keller leer ist.

Die bei der Iteration nötige "Wahl einer passenden Produktion" verspricht uns natürlich Probleme zu machen, wenn es mehrere passende Produktionen zur Auswahl gibt. In der Tat ist dies das Hauptproblem bei der Top-Down-Analyse.

Wir betonen noch einmal die **prinzipielle Arbeitsweise des Top-Down-Parsers:**

Der Parser scannt den Input von links nach rechts, vom aktuellen Reststring kennt er jeweils nur das vordere Zeichen, der übrige Rest ist ihm verborgen.

Bei jedem Schritt der Analyse ist im Stack ein abstraktes Muster des Rest-Inputs abgelegt. Die Aktionen des Parsers sorgen dafür, daß der abstrakte Stackinhalt durch Expansion des aktuell vorderen Stack-Elements (sofern Hilfssymbol) Schritt für Schritt an den unbekannten Reststring angepaßt wird.

**Beispiel 2** (beliebig geschachtelte korrekte Klammern)

**Grammatik:**  $S \rightarrow \varepsilon \mid (S)S$

**Beispielstring:**    (   (   (   )   (   )   )   (   )   )  
                           1   2   3   4   5   6   7   8   9 10

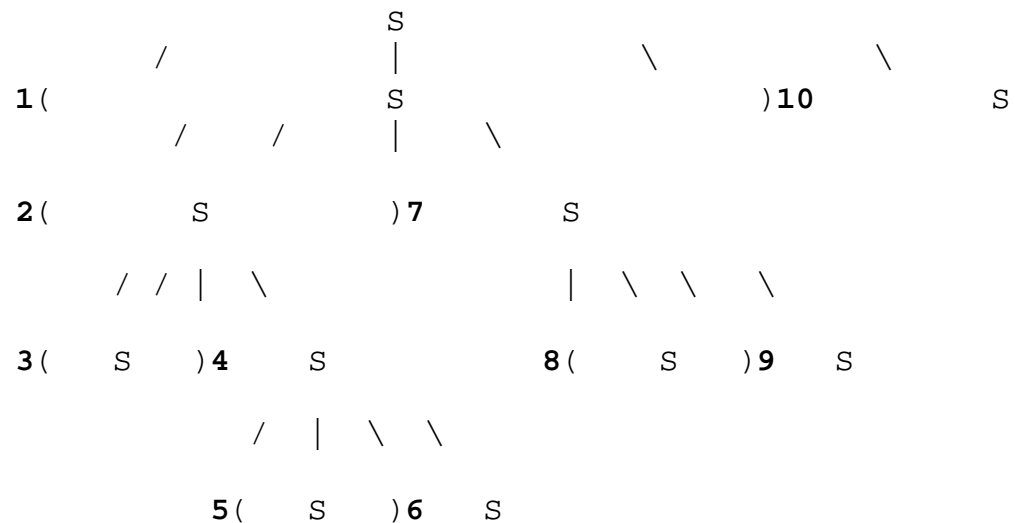
Stack	Reststring	benutzte Produktion
$S$	( ( ( ) ( ) ) ( ) )	$S \rightarrow (S)S$
$(S)S$	( ( ( ) ( ) ) ( ) )	
$S)S$	( ( ) ( ) ) ( ) )	$S \rightarrow (S)S$
$(S)S)S$	( ( ) ( ) ) ( ) )	
$S)S)S$	( ) ( ) ) ( ) )	$S \rightarrow (S)S$
$(S)S)S)S$	( ) ( ) ) ( ) )	
$S)S)S)S$	) ( ) ) ( ) )	$S \rightarrow \varepsilon$

**2 - 4**

)S)S)S	) ( ) ) ( ) )	
S)S)S	( ) ) ( ) )	S → (S)S
(S)S)S)S	( ) ) ( ) )	
S)S)S)S	) ) ( ) )	S → ε
)S)S)S	) ) ( ) )	
S)S)S	) ( ) )	S → ε
)S)S	) ( ) )	
S)S	( ) )	S → (S)S
(S)S)S	( ) )	
S)S)S	) )	S → ε
)S)S	) )	
S)S	)	S → ε
)S	)	
S	ε	S → ε
ε	ε	

Stop, String akzeptiert

Der resultierende **Syntaxbaum** ist:



Und die **Linksableitung**:

```
S -> (S)S
    -> ( (S)S)S
    -> ( ( (S)S)S)S
    -> ( ( ( (S)S)S)S
    -> ( ( ( ( (S)S)S)S)S
    -> ( ( ( ( ( (S)S)S)S)S
    -> ( ( ( ( ( ( (S)S)S)S)S
    -> ( ( ( ( ( ( ( (S)S)S)S)S
    -> ( ( ( ( ( ( ( ( (S)S)S)S)S
    -> ( ( ( ( ( ( ( ( ( (S)S)S)S)S
```

Bei diesem Beispiel wird klar, wie die **Linksableitung** zu ihrem Namen kommt: Stets das am weitesten *links* stehende Hilfssymbol wird ersetzt.

Die Analysetabelle unterscheidet sich nur in einem einzigen Feld von der des ersten Beispiels:

**Analysetabelle** (für Beispiel 2):

	(	)	b
	-----+	-----+	-----
S	(S)S	ε	ε
	-----+	-----+	-----

**Hinweis:** Mit der naheliegenden Grammatik

$$S \rightarrow \varepsilon \mid (S) \mid SS$$

hätte die Syntaxanalyse bei diesem Beispiel nicht so gut funktioniert. Denn bei dieser (im Prinzip ebenfalls "richtigen") Grammatik ist nicht immer klar, welche der Produktionen man nehmen soll. Wählt man die falsche Produktion, so landet man in einer Sackgasse.

Nehmen wir beispielsweise  $()()$  als Inputstring. Der Parser *sieht* davon nur die erste öffnende Klammer und er versucht

```
S -> (S)
  -> ... ?
  -> ( ) ( )
```

mit der Konsequenz, daß der Zielstring  $()()$  nicht mehr erreichbar ist.

Der Fehler war: Statt der Produktion  $S \rightarrow (S)$  hätte zunächst  $S \rightarrow SS$  gewählt werden müssen:

```
S -> SS
  -> (S)S
  -> ( )S
  -> ( )(S)
  -> ( )() (jetzt ok)
```

**Wir bemerken ferner:** Statt der Grammatik

$$S \rightarrow \varepsilon \mid (S)S$$

hätten wir scheinbar genausogut die Grammatik

$$S \rightarrow \varepsilon \mid S(S)$$

nehmen können. - Was wäre dann jedoch das Problem gewesen?

**Beispiel 3** (Nichtleere arithmetische Ausdrücke in + , \* , ( , ) und einer Konstanten a)

**Grammatik:**

A	->	TU	
U	->	$\varepsilon$   +A	Ausdruck ist Summe von Termen
T	->	FG	
G	->	$\varepsilon$   *T	Term ist Produkt von Faktoren
F	->	(A)   a	Faktor ist Terminal oder geklammerter Ausdruck

Diese Sprache ist nicht regulär (was schon aus den Klammern bei der letzten Produktion folgt).

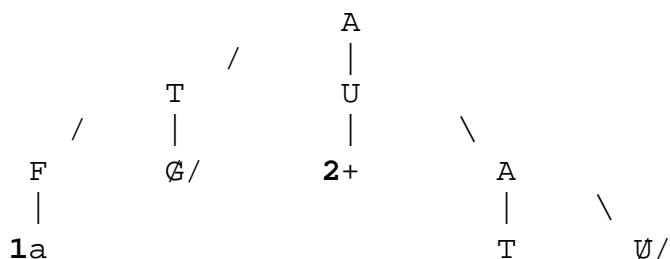
Die etwas seltsam anmutende Konstruktion der Produktionen für "+" und für "\*" hat wieder den Zweck, dem Parser Eindeutigkeit bei der Auswahl der Produktion zu geben (weiteres hierzu siehe unten).

**Beispielstring:**

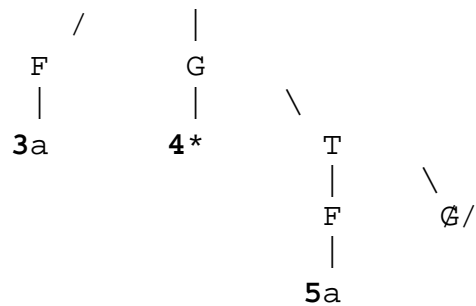
a	+	a	*	a
1	2	3	4	5

Stack	Reststring	benutzte Produktion
A	a + a * a	A -> TU
TU	a + a * a	T -> FG
FGU	a + a * a	F -> a
aGU	a + a * a	
GU	+ a * a	G -> $\varepsilon$
U	+ a * a	U -> +A
+A	+ a * a	
A	a * a	A -> TU
TU	a * a	T -> FG
FGU	a * a	F -> a
aGU	a * a	
GU	* a	G -> *T (!)
*TU	* a	
TU	a	T -> FG
FGU	a	F -> a
aGU	a	
GU	$\varepsilon$	G -> $\varepsilon$
U	$\varepsilon$	U -> $\varepsilon$
$\varepsilon$	$\varepsilon$	

**Syntaxbaum:**



## 2-7



[ anderer Beispielstring:  $a*a+a$  ]

### Bemerkungen

(1) Mit der alternativen und naheliegenden Grammatik

$$\begin{array}{lcl} A & \rightarrow & T+A \mid T \\ T & \rightarrow & F*T \mid F \\ F & \rightarrow & (A) \mid a \end{array}$$

funktioniert die Syntaxanalyse nicht so gut, weil man wieder in Sackgassen landen kann. Denn hier wäre für Hilfssymbol A (und genauso für T) nicht klar, welche der beiden möglichen Produktionen man nehmen soll.

(2) Bei der ebenfalls möglichen, scheinbar eleganten Grammatik

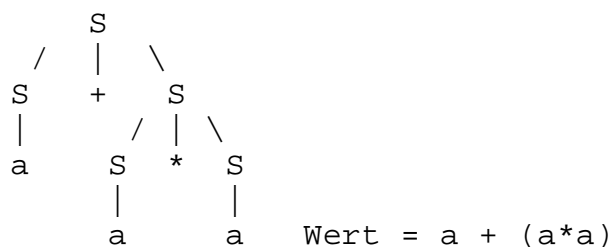
$$S \rightarrow S+S \mid S*S \mid (S) \mid a$$

funktioniert die Analyse erst recht nicht. Zunächst einmal ist diese Grammatik **linksrekursiv** mit der Folge, dass man in Unendlichkeitsschleifen landen kann:

$$\begin{array}{l} S \rightarrow S+S \\ \rightarrow S+S+S \\ \rightarrow S+S+S+S \\ \rightarrow \dots \end{array}$$

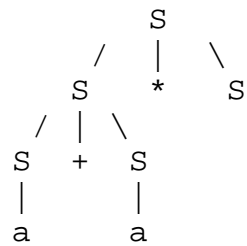
Ferner ist diese Grammatik **semantisch mehrdeutig** mit der völlig unakzeptablen Folge, dass unterschiedliche Analyseläufe für ein- und denselben Eingabestring zu unterschiedlichen Ergebnissen d.h. zu unterschiedlichen Syntaxbäumen (mit unterschiedlichen Werten!) führen können:

Für den Beispielstring  $a+a*a$  zeigen wir zwei Linksableitungen:

$$\begin{array}{l} S \rightarrow S + S \\ \rightarrow a + S \\ \rightarrow a + S*S \\ \rightarrow a + a*S \\ \rightarrow a + a*a \end{array}$$


## 2-8

$S \rightarrow S * S$   
 $\rightarrow S + S * S$   
 $\rightarrow a + S * S$   
 $\rightarrow a + a * S$   
 $\rightarrow a + a * a$



Wert = (a+a) \* a

Der obere Baum würde ausgewertet als  $a+(a*a)$ , der untere als  $(a+a)*a$ , und bei üblicher Interpretation der Operatoren (bei üblicher **Semantik**) wären das unterschiedliche Werte!

Die zuvor notierte Grammatik ist zwar insofern nicht perfekt, als sie in Sackgassen führen kann, aber immerhin wird die „Punkt-vor-Strich“-Priorität von ihr garantiert.

Man vergleiche auch ( $\rightarrow$  Ergänzungen) die mehrdeutige if-else Konstruktion der Programmiersprache C.

-----

Bei der Analyse des letzten Beispiels hatten wir die Zeile

Stack	Reststring	benutzte Produktion
GU	* a	$G \rightarrow *T$ (!)

Von der Grammatik wurden für diese Situation die beiden Produktionen

$G \rightarrow \epsilon \mid *T$

angeboten, und wir hatten wie selbstverständlich die zweite Möglichkeit  $G \rightarrow *T$  genommen.

Tatsächlich hätte die andere Möglichkeit ( $G$  vernichten) nicht zum Erfolg geführt (warum nicht?).

Wir haben hier aber ein generelles Problem: Der Parser *sieht* vom Reststring nur das vordere Zeichen. Wie kann er dann bei Expansion des obersten Kellerelements und bei mehreren angebotenen Produktionen die *richtige* Produktion herausbekommen?

Die im Grunde wenig attraktive Antwort ist: Man muß versuchen, diese Information über Detailanalyse der Grammatik zu gewinnen.

Wir führen diesen Prozeß beim letzten Beispiel (arithmetische Ausdrücke) von Hand vor. In der Praxis hat man heute natürlich Tools, mit denen sich die gewünschten **Analysetabellen** automatisch erstellen lassen.

Zunächst bestimmen wir die **FIRST** und **FOLLOW**-Mengen:

	<b>FIRST</b>	<b>FOLLOW-Mengen</b> (nur relevante)
(1) A $\rightarrow$ TU	a, (	
(2) T $\rightarrow$ FG	a, (	
(3) U $\rightarrow$ +A	+	
(4) $\epsilon$		)
(5) F $\rightarrow$ a	a	
(6) (A)	(	
(7) G $\rightarrow$ *T	*	
(8) $\epsilon$		+, )

Ist  $X \rightarrow \text{str}$  eine Ersetzungsregel, so ist **FIRST( $X \rightarrow \text{str}$ )** definiert als das erste Terminalsymbol, das bei Expansion mit Regeln der Grammatik aus **str** entstehen kann. - Im Beispiel erkennt man der Reihe nach:

$\text{FIRST}(U \rightarrow +A) = \{ + \}$

$\text{FIRST}(F \rightarrow a) = \{ a \}$

$\text{FIRST}(F \rightarrow (A)) = \{ ( \}$

$\text{FIRST}(G \rightarrow *T) = \{ * \}$

und dann:

$\text{FIRST}(T \rightarrow FG) = \{ a, ( \}$

$\text{FIRST}(A \rightarrow TU) = \{ a, ( \}$

Im Falle der Produktionen (4) und (8) kann aus U bzw. aus G gar nichts entstehen, und wir müssen herausbekommen, was *nach* U bzw. *nach* G kommen kann, und das sind die sogenannten **FOLLOW-Mengen**.

Für  $\text{FOLLOW}(G)$  können wir so argumentieren:

$\text{FOLLOW}(G) += \text{FOLLOW}(T) += \text{FIRST}(U) + \text{FOLLOW}(A) = \{ +, ) \}$

(Begründung der einzelnen Schritte in der Vorlesung)

Diese Argumentation enthält Subtilitäten, die man lieber nicht zu oft per Hand, sondern automatisch macht:

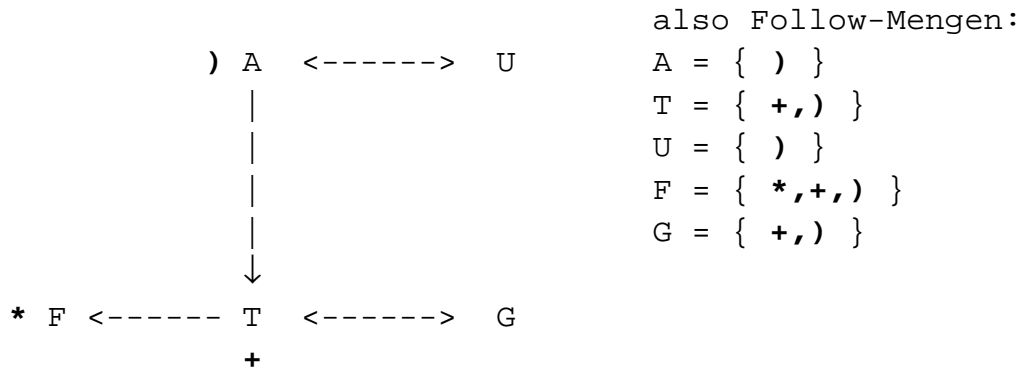
### Algorithmus zur Berechnung der FOLLOW-Mengen

Man wende so oft wie möglich die folgenden Regeln an:

- Für jede Produktion  $A \rightarrow \alpha B \beta$  mit nichtleerem  $\beta$  ergänze  $\text{FOLLOW}(B)$  um  $\text{FIRST}(\beta)$   
Läßt sich  $\beta$  mit Regeln der Grammatik vernichten, dann ergänze  $\text{FOLLOW}(B)$  um  $\text{FOLLOW}(A)$  (Kante von A nach B)
- Für jede Produktion  $A \rightarrow \alpha B$  ergänze  $\text{FOLLOW}(B)$  um  $\text{FOLLOW}(A)$  (Kante von A nach B)

Man kann diesen Prozeß durch einen Graphen visualisieren. Für

unser Beispiel erhalten wir:



Die FOLLOW-Mengen erhält man jetzt durch Sammeln der Markierungen gegen die Pfeilrichtungen, also:

FOLLOW(U) = { ) }

FOLLOW(G) = { + , ) }

Die FIRST- und FOLLOW-Mengen geben nun eindeutige Auskunft über die bei einer Expansion zu wählende Produktion, also etwa:

Falls G auf dem Stack liegt, dann

- expandiere G zu \*T , wenn "\*" aktuell gescannt wird
- vernichte G, wenn "+" oder wenn ")" aktuell gescannt wird

In der **Analysetabelle** sind all diese Informationen zusammengefaßt. Entscheidend bei dieser Tabelle ist die Eindeutigkeit. Für jedes Hilfssymbol auf dem Stack und jedes gescannte Zeichen darf es in der Grammatik höchstens eine mögliche Produktion geben. Oder anders gesagt: An jeder Position der Analysetabelle darf höchstens ein Eintrag stehen.

Eine Grammatik mit einer solchen Analysetabelle ist eine **LL(1)-Grammatik**: von Links lesen, Linksableitung erzeugen, Lookahead 1

	a	+	*	(	)	EoF	
<b>A</b>	1			1			<b>Analysetabelle</b> (arithmetische Ausdrücke)
<b>T</b>	2			2			
<b>U</b>		3			4	4	
<b>F</b>	5			6			
<b>G</b>		8	7		8	8	

**Beispiel 4** (reguläre Ausdrücke)

Reguläre Ausdrücke wie  $a*b$ ,  $(a+b)*c$  oder  $(a*b+ac)d$  dienen in Teil 1 der kompakten Beschreibung regulärer Sprachen.

Wie wir wissen sind Grammatiken mächtiger als reguläre Ausdrücke, und es ist nicht schwer, eine einzige Grammatik anzugeben, mittels der sämtliche regulären Ausdrücke abgeleitet werden können:

$$\begin{aligned} A &\rightarrow T \mid T+A \\ T &\rightarrow F \mid FT \\ F &\rightarrow \langle i \rangle \mid \langle i \rangle^* \mid (A) \mid (A)^* \end{aligned}$$

Hier steht  $\langle i \rangle$  (*Identifizier*) für die Zeichen des Alphabets, aus denen sich der reguläre Ausdruck aufbauen soll, also  $a, b, c$  und  $d$  in den oben notierten Beispielen.

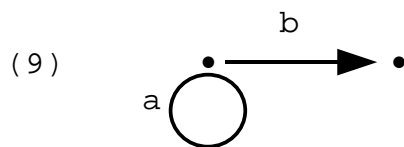
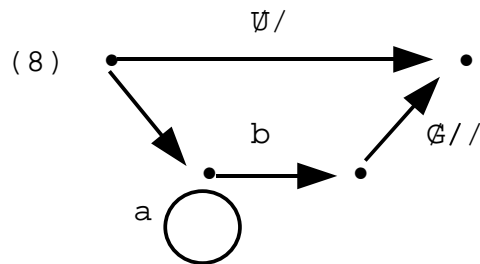
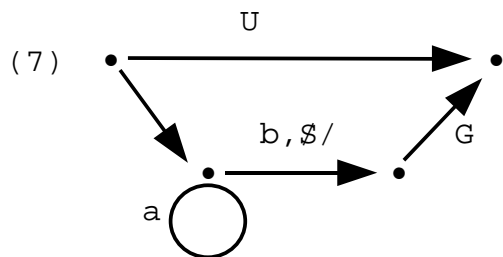
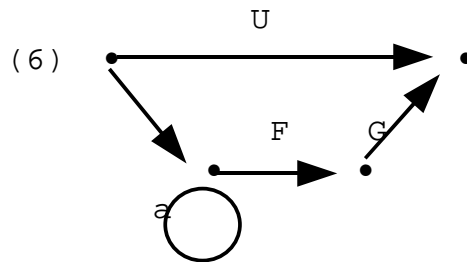
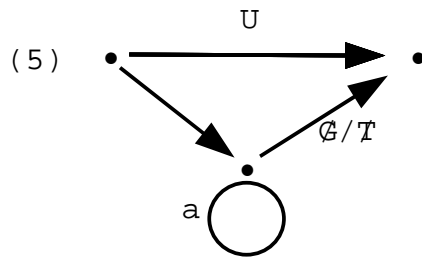
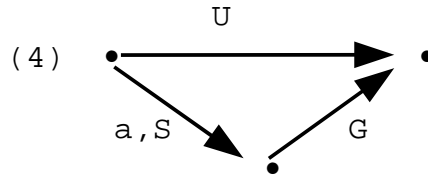
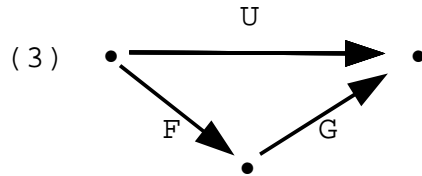
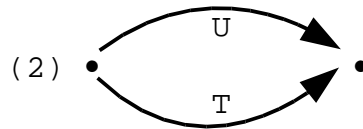
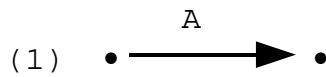
Aus ähnlichen Gründen wie beim letzten Beispiel (mangelnde Eindeutigkeit) ist diese Grammatik jedoch unzureichend. Wir können sie aber genauso wie dort zu einer LL(1)-Grammatik verbessern:

		<b>FIRST</b>	<b>FOLLOW</b> (nur relev.)	<b>Graph</b>
(1)	$A \rightarrow TU$	$\langle i \rangle, ($		
(2)	$U \rightarrow +A$	$+$		$) A \longleftrightarrow U$
(3)	$\epsilon$		$)$	$\downarrow$
(4)	$T \rightarrow FG$	$\langle i \rangle, ($		$+ T \longleftrightarrow G$
(5)	$G \rightarrow T$	$\langle i \rangle, ($		$\downarrow$
(6)	$\epsilon$		$+, )$	$i, ( F$
(7)	$F \rightarrow \langle i \rangle S$	$\langle i \rangle$		$\downarrow$
(8)	$(A)S$	$($		$S$
(9)	$S \rightarrow *$	$*$		
(10)	$\epsilon$		$\langle i \rangle, (, +, )$	

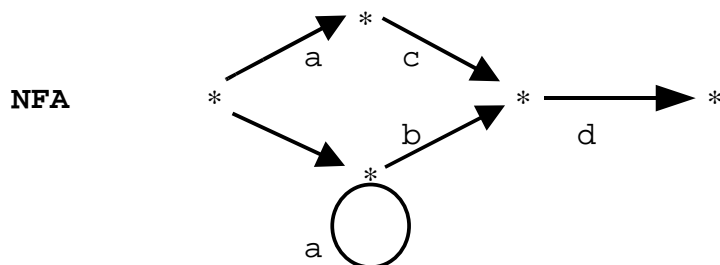
Rechts ist der Graph notiert, mit dessen Unterstützung die FIRST- und FOLLOW-Mengen gewonnen wurden, und hier ist die resultierende Analysetabelle:

	$\langle i \rangle$	$+$	$*$	$($	$)$	EoF	
<b>A</b>	1			1			<b>Analysetabelle</b> (reguläre Ausdrücke)
<b>T</b>	4			4			
<b>U</b>		2			3	3	
<b>F</b>	7			8			
<b>G</b>	5	6		5	6	6	
<b>S</b>	10	10	9	10	10	10	





Wendet man das hier angedeutete Konstruktionsprinzip auf den regulären Ausdruck  $(a*b+ac)d$  an, so erhält man den folgenden, aus Teil 1 in ähnlicher Weise (!) bekannten nichtdeterministischen Automaten:



Übungen zu TI, Teil 2

(1) Auf Basis der Grammatik für einfach geschachtelte Klammern analysiere man die (allesamt "falschen") Strings

( ) ) , ( ( ) , ( ( ) (

(2) Mittels der Grammatik für beliebig geschachtelte Klammern teste man die beiden Strings

( ) ( ( ) ) , ( ) ) (

(3) Für die binären Palindrome mit der zusätzlichen "Mittemarkierung" #, also beispielsweise

01#10 , 1011#1101 , # , 00#00

bestimme man eine geeignete Grammatik. Analysieren Sie sodann einige selbstgewählte Strings.

(4a) Man parse den *arithmetischen Ausdruck*  $a*a + a$  unter Benutzung der Analysetabelle der Vorlesung.

(b) Man parse den arithmetischen Ausdruck  $(a+a)*a$

(5) Man parse den *regulären Ausdruck*  $(a+b)*c$  unter Benutzung der Analysetabelle der Vorlesung (diese Aufgabe ist nicht schwierig, macht aber einige Arbeit).

(6) Für die zum regulären Ausdruck  $(01+010)^*$  gehörende formale Sprache (also Ketten aus 01 und 010) werden drei Grammatiken vorgeschlagen:

**erste Grammatik:**

S  $\rightarrow$  0T  
 $\epsilon$   
 T  $\rightarrow$  1U  
 U  $\rightarrow$  S  
 0S

**zweite Grammatik:**

S  $\rightarrow$  0TS  
 $\epsilon$   
 T  $\rightarrow$  1U  
 U  $\rightarrow$  0  
 $\epsilon$

Man mache sich klar, daß diese beiden Grammatiken beim Versuch, Analysetabellen zu erstellen, zu Konflikten führen.

Die folgende dritte Grammatik hingegen funktioniert. Mit der bestimme man die Analysetabelle und analysiere den String **01001**:

**dritte Grammatik:**

- (1) S  $\rightarrow$  0T
- (2)  $\epsilon$
- (3) T  $\rightarrow$  1U
- (4) U  $\rightarrow$  0V
- (5)  $\epsilon$
- (6) V  $\rightarrow$  0T
- (7) 1U
- (8)  $\epsilon$

(7) Klammernfreie arithmetische Zuweisungen (wie etwa **a:=b\*c+d**) können durch die folgende Grammatik beschrieben werden:

- (1) S  $\rightarrow$  <i>:=A
- (2) A  $\rightarrow$  TU
- (3) U  $\rightarrow$  +A
- (4)  $\epsilon$
- (5) T  $\rightarrow$  FG
- (6) G  $\rightarrow$  \*T
- (7)  $\epsilon$
- (8) F  $\rightarrow$  <i>

Man bestimme die Analysetabelle und analysiere damit die obige Zuweisung.

-----

**Ergänzung:** Bedingte Anweisung in der Programmiersprache C (in **Backus-Naur-Form**):

<Anweisung> ::= <Selektion> | ...

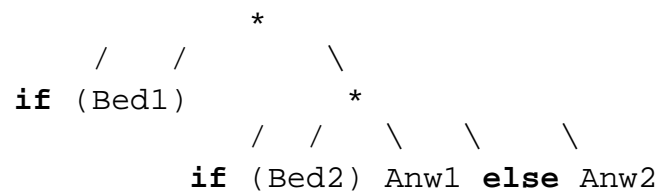
<Selektion> ::= **if** <Bedingung> <Anweisung> |

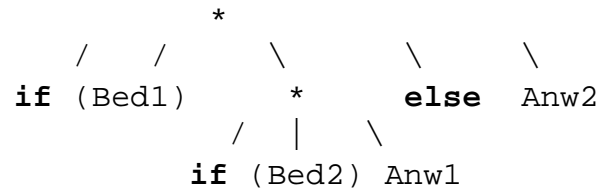
**if** <Bedingung> <Anweisung> **else** <Anweisung>

Diese Grammatikregeln sind semantisch mehrdeutig, wie das folgende Beispiel zeigt:

**if** (Bed1) **if** (Bed2) Anw1 **else** Anw2

Es sind je nach Klammerung zwei unterschiedliche Interpretationen (**Syntaxbäume**) möglich:





Die bei C-Compilern übliche Interpretation ist die Erste:  
Binden des **else** an das letzte **if**.

Im Prinzip ist es kein Problem, die Regeln für die Selektion eindeutig zu machen:

```
if Bool_Expr then Compound [else Compound] end
```

(stammt aus der Beschreibung der Programmiersprache EIFFEL).

Die eckigen Klammern stehen hier für "optional".