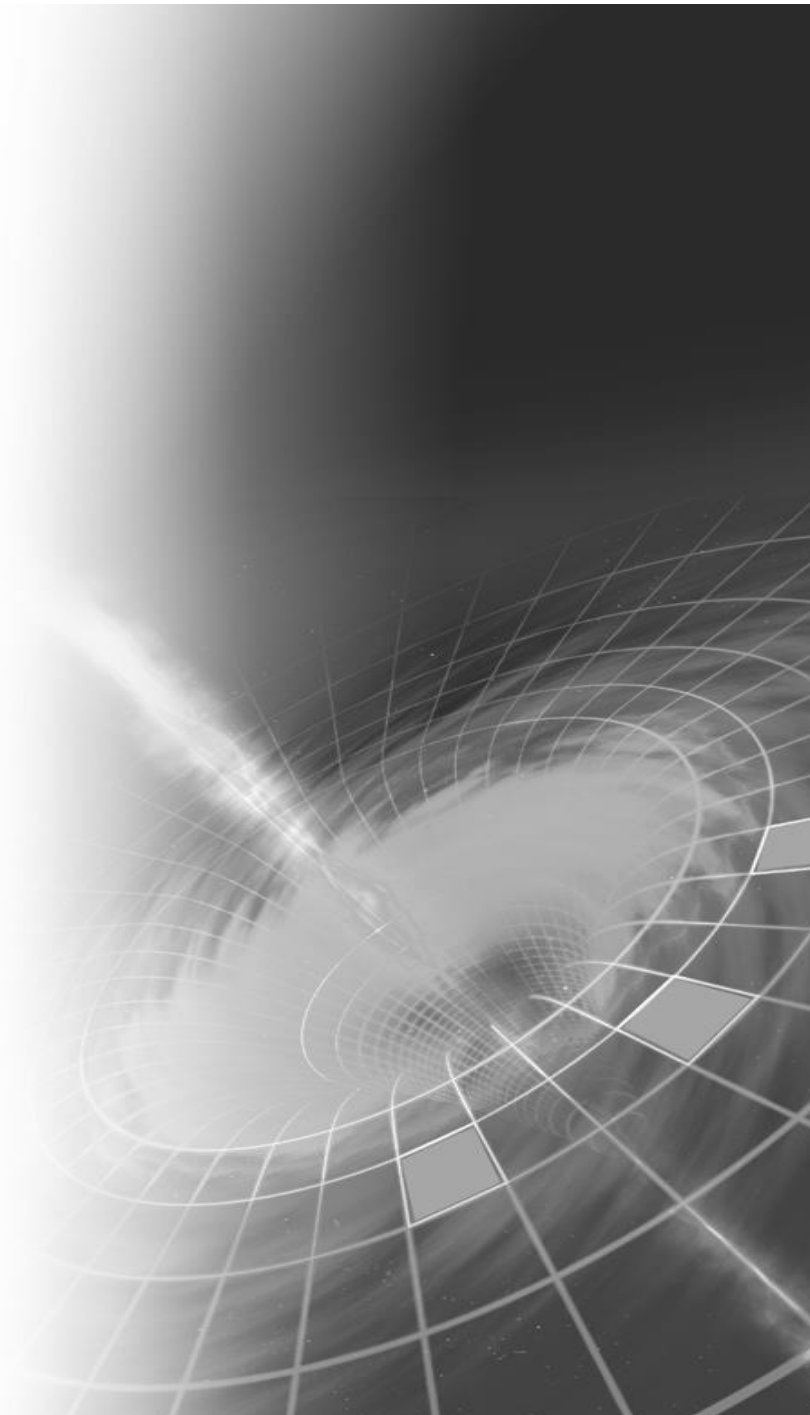
Bachelor-Programm

Compilerbau

im SoSe 2014

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dr. Andreas Kunert
Dipl.-Inf. Ingmar Eveslage

fischer@informatik.hu-berlin.de



4.3 LL-Parser

- Prinzip der prädiktiven Syntaxanalyse

  (LL- und LR-Parser sind prädiktive Parser)
- FIRST-Mengen-Bildung
- FOLLOW-Mengen-Bildung
- LL-Grammatik-Eigenschaften

Motivation FIRST-Mengen

Für bekannte Programmiersprachen gibt es LL(1) und LL(2)-Grammatiken

Wie kann die LL(k)-Eigenschaft einer Grammatik ohne Betrachtung aller möglichen Ableitungen überprüft werden?

- **erstes Hilfsmittel**: **FIRST**-Mengen-Konstruktion
  enthalten Anfangsstücke herleitbarer Terminalworte

Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik, $k \in N$

$\mathbf{FIRST_k}: (V \cup \Sigma)^+ \rightarrow \mathbf{P}(\Sigma)$ ist definiert durch

$$\mathbf{FIRST_k}(\alpha) = \{ k{:}w \mid \alpha \Rightarrow^* w \}$$

1) $\mathbf{FIRST_k}$ ist also die Menge der Anfangsstücke der Länge **k** von Terminalwörtern, die aus α ableitbar sind
2) Falls kürzere Wörter ableitbar sind, dann sind auch diese in der **FIRST**-Menge enthalten, insbesondere ε.

letzte Vorlesung betrachtete den speziellen **Fall: k=1**

Definition einer FIRST-Menge (für k=1)

Def.: FIRST-Menge (formale Definition)

$$FIRST^0 : (V \cup \Sigma)^+ \to \wp(\Sigma)$$

$$FIRST^0(x) := \{y \mid \exists \alpha \in (\Sigma \cup V)^* : x \Rightarrow^* y\alpha, y \in \Sigma\}$$

$$FIRST : (V \cup \Sigma)^+ \to \wp(\Sigma \cup \{\varepsilon\})$$

$$FIRST(x) := \begin{cases} FIRST^0(x) \cup \{\varepsilon\}, wenn & x \Rightarrow^* \varepsilon \\ FIRST^0(x), & sonst \end{cases}$$

Konstruktion von FIRST-Mengen (für k=1)

sei $\alpha \in V$ die rechte Seite einer Produktion

folgende Regeln

werden solange zur Berechnung von $FIRST_1(\alpha)$ angewendet,
bis zu keiner $FIRST_1$-Menge mehr ein neues **Terminalsymbol** oder ϵ hinzukommt:

Regel 1 ist α **Terminalsymbol**, dann ist $FIRST(\alpha) = \{ \alpha \}$;

Regel 2 gibt es eine Produktion $\alpha \rightarrow Y_1 Y_2 ... Y_k$, dann ist

- zunächst $FIRST(Y_1) \setminus \{\epsilon\}$ der Menge $FIRST(\alpha)$ zuzuführen;
- sollte ϵ in $FIRST(Y_1)$ enthalten sein, so ist auch $FIRST(Y_2) \setminus \{\epsilon\}$ der Menge $FIRST(\alpha)$ hinzuzufügen;
- Fortsetzung bis zu einem $i \leq n$, wo Y_i **nicht** ϵ -ableitbar ist;
- sollte ϵ jedoch in **allen** $FIRST(Y_i)$ enthalten sein, ist auch ϵ der Menge $FIRST(\alpha)$ hinzuzufügen;

Regel 3 gibt es eine Produktion $\alpha \rightarrow \epsilon$, dann ist ϵ der Menge $FIRST(\alpha)$ hinzuzufügen.

Beispiel-1: FIRST-Mengen

G-1
S → A B **c**
A → **a**
A → ε
B → **b**
B → ε

Wir betrachten alle Regeln und deren RS, um FIRST der Y_is bestimmen zu können

α ∈ V	FIRST(α) Anfangsschritt	Y_1	Y_2	Y_3	FIRST(α) Final
S	ø	A	B	**c**	
A	ø	**a**			
A	ε	ε			
B	ø	**b**			
B	ε	ε			

Regel 2

Regel 1

Regel 3

Beispiel-1: FIRST-Mengen

G-1
S → A B **c**
A → **a**
A → ε
B → **b**
B → ε

Wir betrachten alle Regeln und deren RS, um FIRST der Y_is bestimmen zu können

α ∈ V	FIRST(α) Anfangsschritt	Y_1	Y_2	Y_3	FIRST(α) Final
S	ø	A	B	**c**	{ **a**, **b**, **c** }
A	ø	**a**			{ **a**, ε}
A	ε	ε			
B	ø	**b**			{ **b**, ε}
B	ε	ε			

Regel 2

Regel 1

Regel 3

Beispiel-4: FIRST-Mengen

G-4

Z → **d**	[1]
Z → XYZ	[2]
Y → ε	[3]
Y → **c**	[4]
X → Y	[5]
X → **a**	[6]

	$\alpha \in V$	$FIRST^0(\alpha)$	Y_1	Y_2	Y_3	$FIRST^1(Y_1)$	$FIRST(Y_2)$	$FIRST(Y_3)$	ε	$FIRST^1(\alpha)$
[1]	Z	ø	**d**			{**d**}	{**a**, **c**, **d** }	{ **a**, **c**, **d** }		
[2]	Z		X	Y	Z	{**a**, **c**, ε }	{ε, **c**}		nein	{**a**, **c**, **d**}
[3]	Y	ε	ε			{ε, **c**}			ja	{ε, **c**}
[4]	Y		**c**							
[5]	X	ø	Y			{ε, **c**, **a**}			ja	{ε, **a**, **c**}
[6]	X		**a**							

Z wird nach Z selbst abgeleitet,
kann also nicht mehr erweitert werden

Prädiktives (vorausschauendes) Parsen

insbesondere können nicht
beide Alternativen zu ε abgeleitet werden

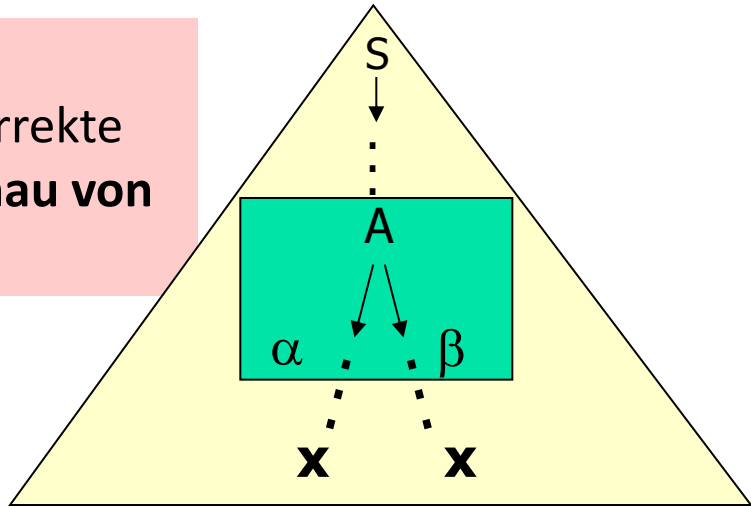
Schlüsseleigenschaft:

Falls zwei Produktionen $A \rightarrow \alpha \mid \beta$

in einer Grammatik vorkommen, soll die Eigenschaft gelten:

$$FIRST(\alpha) \cap FIRST(\beta) = \varnothing$$

Bem.:
Diese Eigenschaft sollte es dem Parser erlauben, eine korrekte Wahl der richtigen Produktionsregel mit einer **Vorausschau von einem Symbol** zu treffen

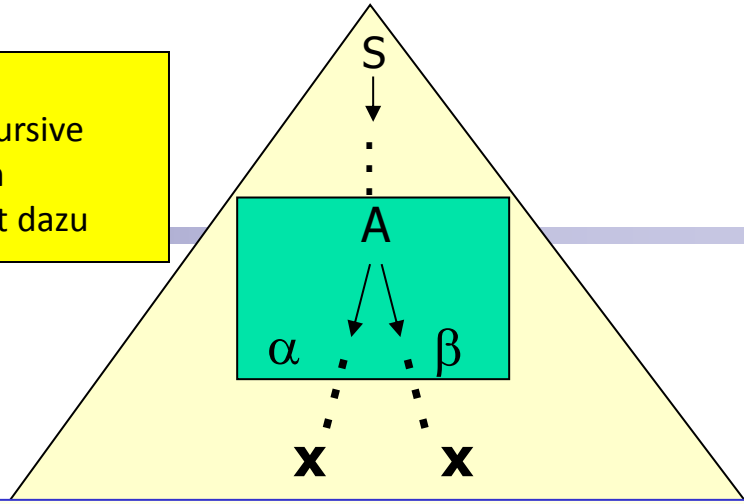


Die Sicherung der FIRST-Mengen-Eigenschaft allein reicht jedoch noch nicht in jedem Fall aus, die Eindeutigkeit des Ableitungsbaumes zu sichern

LL(1)-Eigenschaft

mehrdeutige
oder linksrekursive
Grammatiken
gehören nicht dazu

Eigenschaft einer kfG,
dass deren Sprachen durch einen LL(1)-Parser
analysiert werden können.

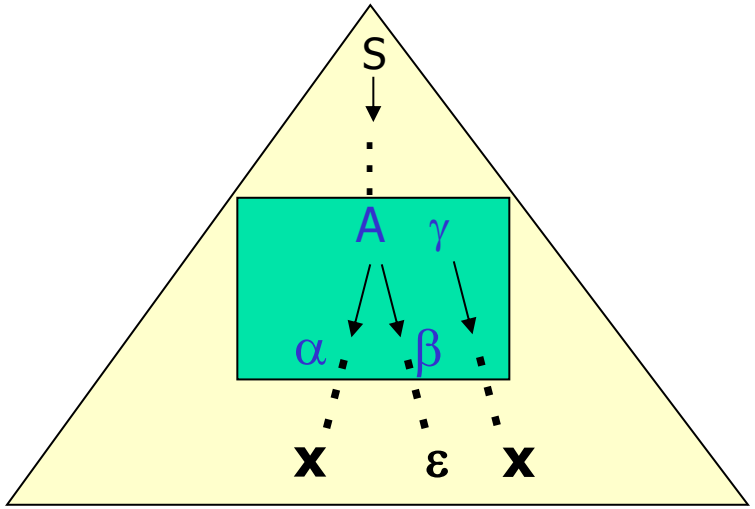


- bilden alle möglichen Ableitungen von α (bzw. β)
- schauen uns dabei nur jeweils das erste Terminalsymbol an
- fassen diese zur Menge FIRST zusammen

Forderung-I:

Falls zwei Produktionen $A \rightarrow \alpha | \beta$
in einer Grammatik vorkommen, gilt:

$$FIRST(\alpha) \cap FIRST(\beta) = \varnothing$$



Forderung-II:

Sollte jedoch eine der beiden Alternativen zu ε werden
(o.B.d.A.: β), gilt zusätzlich

$$FIRST(\alpha) \cap FOLLOW(A) = \varnothing$$

- Terminalsymbole, die potentiell unmittelbar rechts nach dem *A* auftauchen können
(steht man bereits ganz rechts,
wird $ als Repräsentation fürs Eingabeende verwendet)

4.3 LL-Parser

- Prinzip der prädiktiven Syntaxanalyse

  (LL- und LR-Parser sind prädiktive Parser)
- FIRST-Mengen-Bildung
- FOLLOW-Mengen-Bildung
- LL-Grammatik-Eigenschaft
- LL-Analyseverfahren

Definition einer FOLLOW-Menge

Def.: FOLLOW-Menge (informale Definition)

sei A eine Variable,

dann ist FOLLOW(A) die Menge aller Terminalsymbole **x**,
die in einem Resultat von Ableitungsschritten **direkt rechts** neben A stehen dürfen

$ gehört zu FOLLOW(A),
wenn A die am weitesten rechts stehende Meta-Variable ist

$ als eof-Kennung 

Bem.:

1. FOLLOW-Mengen werden ausschließlich über **Variablen** (Nichtterminalsymbole) definiert

2. ACHTUNG: es könnten durchaus zwischen A und **x** während der Bearbeitung Meta-Symbole gestanden haben, die in ϵ übergegangen sind

Konstruktion von FOLLOW-Mengen

folgende Regeln

werden solange zur Berechnung von FOLLOW(X) angewendet,
bis die Menge nicht mehr vergrößert werden kann:

Regel 1 $ gehört zu FOLLOW(X) (falls X=S, das Startsymbol);

Regel 2 gibt es eine Produktion $Y \rightarrow \alpha X \beta$,
wird FIRST(β) \ {ϵ } in FOLLOW(X) aufgenommen;

Regel 3 gibt es Produktionen

$Y \rightarrow \alpha X$ oder

$Y \rightarrow \alpha X \beta$ und FIRST(β) enthält dabei ϵ (d.h. $\beta \Rightarrow^* \epsilon$),

dann wird **jedes** Element von FOLLOW(Y)
auch Element von FOLLOW(X).

FOLLOW: Berechnungsalgorithmus

//Anfangsschritt
$\forall X \in V$: FOLLOW(X) = ø
FOLLOW(S) : FOLLOW(S) = { $ }

do { //Iteration bis sich keine FOLLOW-Menge mehr ändert

do $\forall$ *Regeln* $X \rightarrow \alpha A$ **in** P {
FOLLOW (A)= FOLLOW(A) ∪ FOLLOW(X)
}
do $\forall$ *Regeln* $X \rightarrow \alpha A \beta$ **in** P {
FOLLOW (A)= FOLLOW(A) ∪ (FIRST(β) – {ϵ})
if ($\epsilon \in$ FIRST(β)) **then** FOLLOW(A)= FOLLOW(A) ∪ FOLLOW(X)
}
//Abbruch
if ($\forall$ *Metasymbole* X: FOLLOW(X) unverändert)
then continue;
}

Beispiel-1: FOLLOW-Mengen (a)

Grammatik:

Z → S	[1]
S → S **b**	[2]
S → **b** A **a**	[3]
A → **a** S **c**	[4]
A → **a**	[5]
A → **a** S **b**	[6]

Z → S

Z ist das Startsymbol

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{}
S	{ **b** }	{}
A	{ **a** }	{}

Regel 1/

Produktion 1

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{$}
S	{ **b** }	{}
A	{ **a** }	{}

Beispiel-1: FOLLOW-Mengen (b)

Grammatik:

Z → S	[1]
S → S **b**	[2]
S → **b** A **a**	[3]
A → **a** S **c**	[4]
A → **a**	[5]
A → **a** S **b**	[6]

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{}
S	{ **b** }	{}
A	{ **a** }	{}

Regel 1/

Produktion 1

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{$}
S	{ **b** }	{}
A	{ **a** }	{}

Regel 2/

Produktion 2,3,4,6

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{ **$** }
S	{ **b** }	
A	{ **a** }	

$$\text{FOLLOW}(S) = \text{FOLLOW}(S) \cup \text{FIRST}(c) \setminus \{\epsilon\}$$

A → **a** S **c**

A → **a** S **b**

S → S **b**

S → **b** A **a**

FIRST(**c**) \ {ϵ} = {**c**}

FIRST(**b**) \ {ϵ} = {**b**}

FIRST(**b**) \ {ϵ} = {**b**}

FIRST(**a**) \ {ϵ} = {**a**}

Beispiel-1: FOLLOW-Mengen (b)

Grammatik:

Z → S	[1]
S → S **b**	[2]
S → **b** A **a**	[3]
A → **a** S **c**	[4]
A → **a**	[5]
A → **a** S **b**	[6]

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{}
S	{ **b** }	{}
A	{ **a** }	{}

Regel 1/

Produktion 1

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{$}
S	{ **b** }	{}
A	{ **a** }	{}

Regel 2/

Produktion 2,3,4,6

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{ **$** }
S	{ **b** }	{ **b**, **c** }
A	{ **a** }	{ **a** }

A → **a** S **c**

A → **a** S **b**

S → S **b**

S → **b** A **a**

$$\text{FOLLOW}(S) = \text{FOLLOW}(S) \cup \text{FIRST}(c) \setminus \{\epsilon\}$$

FIRST(**c**) \ {ϵ} = {**c**}
FIRST(**b**) \ {ϵ} = {**b**}

FIRST(**b**) \ {ϵ} = {**b**}

FIRST(**a**) \ {ϵ} = {**a**}

Beispiel-1: FOLLOW-Mengen (c)

Grammatik:

Z → S	[1]
S → S **b**	[2]
S → **b** A **a**	[3]
A → **a** S **c**	[4]
A → **a**	[5]
A → **a** S **b**	[6]

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{}
S	{ **b** }	{}
A	{ **a** }	{}

Regel 1 → Produktion 1

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{$}
S	{ **b** }	{}
A	{ **a** }	{}

Regel 2 → Produktion 2,3,4,6

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{ **$** }
S	{ **b** }	{ **b**, **c** }
A	{ **a** }	{ **a** }

Regel 3 → Produktion 1

X	FIRST(X)	FOLLOW(X)
Z	{ **b** }	{ **$** }
S	{ **b** }	{ **b**, **c**, **$** }
A	{ **a** }	{ **a** }

Z → S

$$\text{FOLLOW}(S) = \text{FOLLOW}(S) \cup \text{FOLLOW}(Z)$$

weitere Anwendung der Regeln erzeugt keine neuen Elemente der FOLLOW-Mengen

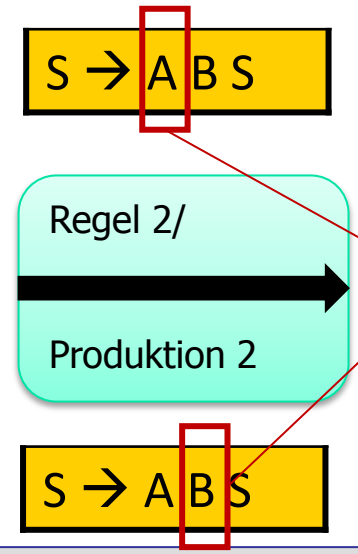
Beispiel-2: FOLLOW-Mengen (a)

Grammatik:

S → **s**	[1]
S → A B S	[2]
B → ϵ	[3]
B → **b**	[4]
A → B	[5]
A → **a**	[6]

S ist das Startsymbol

X	FIRST(X)	FOLLOW(X)
A	{ **a, b**}	{}
B	{ **b**, ϵ }	{}
S	{ **s**, **a**, **b** }	{**$**}



X	FIRST(X)	FOLLOW(X)
A	{ **a, b**}	
B	{ **b**, ϵ }	
S	{ **s**, **a**, **b** }	{**$**}

FOLLOW(A):
Hinzunahme von FIRST(B) \ {ϵ} = {**b**}
da $\epsilon \in$ FIRST(B)
zusätzlich: Hinzunahme von FOLLOW(S)

FOLLOW(B):
Hinzunahme von FIRST(S) \ {ϵ} = {**s**, **a**, **b**}

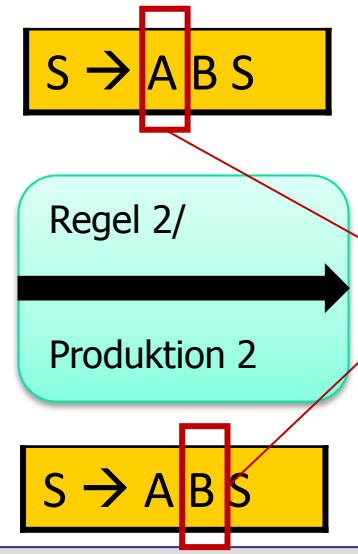
Beispiel-2: FOLLOW-Mengen (a)

Grammatik:

S → **s**	[1]
S → A B S	[2]
B → ϵ	[3]
B → **b**	[4]
A → B	[5]
A → **a**	[6]

S ist das Startsymbol

X	FIRST(X)	FOLLOW(X)
A	{ **a**, **b**}	{}
B	{ **b**, ϵ }	{}
S	{ **s**, **a**, **b** }	{**$**}



X	FIRST(X)	FOLLOW(X)
A	{ **a**, **b**}	{ **b**, **$** }
B	{ **b**, ϵ }	{ **s**, **a**, **b**}
S	{ **s**, **a**, **b** }	{**$**}

FOLLOW(A):
Hinzunahme von FIRST(B) \ {ϵ} = {**b**}
da $\epsilon \in$ FIRST(B)
zusätzlich: Hinzunahme von FOLLOW(S)

FOLLOW(B):
Hinzunahme von FIRST(S) \ {ϵ} = {**s**, **a**, **b**}

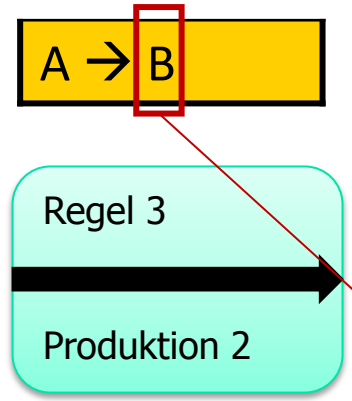
Beispiel-2: FOLLOW-Mengen (b)

Grammatik:

S → **s**	[1]
S → A B S	[2]
B → ϵ	[3]
B → **b**	[4]
A → B	[5]
A → **a**	[6]

S ist das Startsymbol

X	FIRST(X)	FOLLOW(X)
A	{ **a, b**}	{ **b**, **$** }
B	{ **b**, ϵ }	{ **s**, **a**, **b** }
S	{ **s**, **a**, **b** }	{ **$** }



X	FIRST(X)	FOLLOW(X)
A	{ **a, b**}	{ **b**, **$** }
B	{ **b**, ϵ }	{ **s**, **a**, **b**, **?** }
S	{ **s**, **a**, **b** }	{ **$** }

FOLLOW(B):
Hinzunahme von FOLLOW(A): { **b**, **$** }

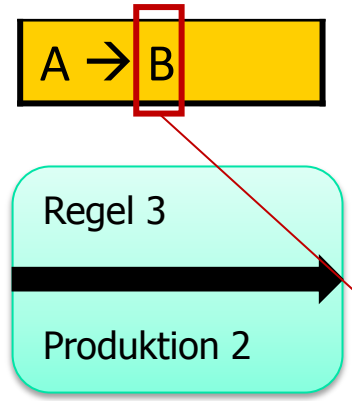
Beispiel-2: FOLLOW-Mengen (b)

Grammatik:

S → **s**	[1]
S → A B S	[2]
B → ϵ	[3]
B → **b**	[4]
A → B	[5]
A → **a**	[6]

S ist das Startsymbol

X	FIRST(X)	FOLLOW(X)
A	{ **a, b**}	{ **b**, **$** }
B	{ **b**, ϵ }	{ **s**, **a**, **b** }
S	{ **s**, **a**, **b** }	{ **$** }



X	FIRST(X)	FOLLOW(X)
A	{ **a, b**}	{ **b**, **$** }
B	{ **b**, ϵ }	{ **s**, **a**, **b**, **$** }
S	{ **s**, **a**, **b** }	{ **$** }

FOLLOW(B):
Hinzunahme von FOLLOW(A): { **b**, **$** }

weitere Anwendung der Regeln erzeugt keine neuen Elemente der FOLLOW-Mengen

Beispiel-3: FOLLOW-Mengen (a)

Grammatik:
S → E
E → TE'
E' → +TE' | ϵ
T → FT'
T' → *FT' | ϵ
F → (E) | id

$\alpha \in V$	Y_1	Y_2	Y_3	FIRST(Y_1)	FIRST^2(Y_1)	ϵ	FIRST(α)
S	E				{ (, id }	nein	{ (, id }
E	T	E'			{ (, id }	nein	{ (, id }
E'	+	T	E'	{ + }		ja	{ϵ, +}
E'	ϵ			{ϵ}			
T	F	T'			{ (, id }	nein	{ (, id }
T'	*	F	T'	{ * }		ja	{ϵ, * }
T'	ϵ			{ϵ}			
F	(	E	)	{ (, id }		nein	{ (, id }
F	id						

benötigen
zunächst
die FIRST-Mengen

Beispiel-3: FOLLOW-Mengen (b)

Grammatik:

S → E	[1]
E → TE'	[2]
E' → +TE' |	[3]
ϵ	[4]
T → FT'	[5]
T' → *FT' |	[6]
ϵ	[7]
F → (E) |	[8]
id	[9]

da S Startsymbol

E → TE'

E' → +TE'

T → FT'

T' → *FT'

F → (E)

Regel 2/

Produktion 2,5,8

X	FIRST(X)	FOLLOW(X)
S	{ (, id }	{ $ }
E	{ (, id }	
E‘	{ϵ, + }	
T	{ (, id }	
T‘	{ϵ, * }	
F	{ (, id }	

X	FIRST(X)	FOLLOW(X)
S	{ (, id }	{ $ }
E	{ (, id }	{) }
E‘	{ϵ, + }	
T	{ (, id }	{ +, }
T‘	{ϵ, * }	
F	{ (, id }	{ *, }

Beispiel-3: FOLLOW-Mengen (c)

Grammatik:

S → E	[1]
E → TE'	[2]
E' → +TE' |	[3]
ϵ	[4]
T → FT'	[5]
T' → *FT' |	[6]
ϵ	[7]
F → (E) |	[8]
id	[9]

da S Startsymbol

E → TE'

E' → +TE'

T → FT'

T' → *FT'

Regel 2

Produktion 2,3,5,6

X	FIRST(X)	FOLLOW(X)
S	{ (, id }	{ $ }
E	{ (, id }	{) }
E‘	{ϵ, + }	
T	{ (, id }	{ +, }
T‘	{ϵ, * }	
F	{ (, id }	{ *, }

X	FIRST(X)	FOLLOW(X)
S	{ (, id }	{ $ }
E	{ (, id }	{) }
E‘	{ϵ, + }	
T	{ (, id }	{ +,), }
T‘	{ϵ, * }	
F	{ (, id }	{ *, +,), }

noch offen

FOLLOW(T): da ϵ FIRST(E') zusätzlich: Hinzunahme von FOLLOW(E) und FOLLOW (E')

FOLLOW(F): da ϵ FIRST(T') zusätzlich: Hinzunahme von FOLLOW(T) und FOLLOW (T')

S → E

E → TE'

E' → +TE'

T → FT'

T' → *FT'

noch offen

Beispiel-3: FOLLOW-Mengen (d)

Grammatik:

S → E	[1]
E → TE'	[2]
E' → +TE' |	[3]
ϵ	[4]
T → FT'	[5]
T' → *FT' |	[6]
ϵ	[7]
F → (E) |	[8]
id	[9]

S → E

E → TE'

E' → +TE'

T → FT'

T' → *FT'

X	FIRST(X)	FOLLOW(X)
S	{ (, id }	{ $ }
E	{ (, id }	{) }
E‘	{ϵ, + }	
T	{ (, id }	{ +,), }
T‘	{ϵ, * }	
F	{ (, id }	{ *, +,), }

Regel 2

Produktion 1,2,3, 5,6

X	FIRST(X)	FOLLOW(X)
S	{ (, id }	{ $ }
E	{ (, id }	{), $}
E‘	{ϵ, + }	{), $}
T	{ (, id }	{ +,), $ }
T‘	{ϵ, * }	{ +,), $ }
F	{ (, id }	{ *, +,), $ }

FOLLOW(E): zusätzlich: Hinzunahme von FOLLOW(S)

FOLLOW(E'): zusätzlich: Hinzunahme von FOLLOW(E) und FOLLOW(E')

FOLLOW(T'): zusätzlich: Hinzunahme von FOLLOW(T) und FOLLOW(T')

weitere Anwendung der Regeln erzeugt keine neuen Elemente der FOLLOW-Mengen

4.3 LL-Parser

- Prinzip der prädiktiven Syntaxanalyse

  (LL- und LR-Parser sind prädiktive Parser)
- FIRST-Mengen-Bildung
- FOLLOW-Mengen-Bildung
- LL-Grammatik-Eigenschaften
- LL-Analyseverfahren

Kriterien für LL(1)- Eigenschaft einer Grammatik

Sei G eine beliebige kontextfreie Grammatik. Dann ist G

eine **LL(1)-Grammatik** g.d.w.

für alle Alternativen

$A \rightarrow \alpha_1 | \ldots | \alpha_n$

gilt:

I. Die Mengen $FIRST(\alpha_1)$, … $FIRST(\alpha_n)$ sind paarweise disjunkt, insbesondere enthält **höchstens** eine der Mengen ϵ und

II. aus $A \Rightarrow^* \epsilon$ folgt:

$FIRST(A) \cap FOLLOW(A) = \emptyset$.

Erfüllung der LL(1)-Eigenschaft

LL(1)-Eigenschaft einer kfG, die garantiert,
dass deren Sprachen durch einen LL(1)-Parser analysiert werden können.

G-1:
$S \rightarrow A$
$S \rightarrow B$
$A \rightarrow \mathbf{a}$
$B \rightarrow \mathbf{b}$

FIRST(A) ∩ FIRST(B) = Ø
{a} {b}

kein *Auswahlkonflikt* :
in keiner
FIRST-Menge
ein ϵ
→ FOLLOW muss
nicht betrachtet
werden

G-2:
$S \rightarrow A\,\mathbf{a}$
$S \rightarrow B$
$A \rightarrow \epsilon$
$A \rightarrow \mathbf{a}$
$B \rightarrow \mathbf{b}$

FIRST(A) ∩ FIRST(B) = Ø
{a, ϵ} {b}

Auswahlkonflikt :
FOLLOW(A)= {a}
keine 1-deutige Regelwahl
möglich

FIRST(A) ∩ FOLLOW(A) ≠ Ø

G-3:
$S \rightarrow A\,\mathbf{a}$
$S \rightarrow \mathbf{b}$
$A \rightarrow \epsilon$
$A \rightarrow \mathbf{c}$

FIRST(A) ∩ FIRST(**b**) = Ø
{c, ϵ} {b}

kein *Auswahlkonflikt* :
FOLLOW(A)= {a}

FIRST(A) ∩ FOLLOW(A) = Ø

Kriterien für Sonderfall von LL(1)-Grammatiken

Sei G eine ϵ -freie kontextfreie Grammatik,
d.h. ohne Produktionen der Form $X \rightarrow \epsilon$,

dann ist G eine **LL(1)-Grammatik** g.d.w.
für jedes Nichtterminal A mit den Alternativen

$A \rightarrow \alpha_1 | \ldots | \alpha_n$

gilt:

die Mengen $FIRST(\alpha_1)$, … $FIRST(\alpha_n)$ sind paarweise disjunkt.

4.3 LL-Parser

- Prinzip der prädiktiven Syntaxanalyse

  (LL- und LR-Parser sind prädiktive Parser)
- FIRST-Mengen-Bildung
- FOLLOW-Mengen-Bildung
- LL-Grammatik-Eigenschaften
- LL-Analyseverfahren

LL(k)- Analyseverfahren

... sind

- Top-Down-Verfahren
- Leserichtung= **links-rechts**, Baumexpansion= **Linksableitung**
- deterministisch, d.h. ohne Rücksetzen und damit effizient (lineare Komplexität)
- prädiktives Verfahren

  für die Auswahl der richtigen Regelalternative reicht für gewisse Grammatiken die Kenntnis **der nächsten k Symbole** (LookAhead) aus

»maßgeschneiderte« Grammatiken für LL(k)-Analyseverfahren

sind LL(k)-Grammatiken

betrachtet hatten wir aber nur den Fall k=1

Eigenschaften von LL(k)-Grammatiken

- nicht mehrdeutig, nicht linksrekursiv
- für eine beliebige kontextfreie Grammatik G ist **nicht entscheidbar**, ob ein k>=1 existiert, so dass G eine LL(k)-Grammatik ist
- **aber** für ein fixiertes k >=1 und eine beliebige kontextfreie Grammatik G ist **entscheidbar**, ob G eine LL(k)-Grammatik ist

Bem.: Programmiersprache Pascal hat eine LL(1)-Grammatik

LL(1)-Analysealgorithmus (1)

Herangehensweise

Benutzung von Parse-Prozeduren, um Terminalsymbole zu erkennen, die durch Meta-Variablen akzeptiert werden

Allgemeines Problem rekursiver Parse-Techniken

Bestimmung der anzuwendenden Produktionsregel

Ausgangssituation

sei G eine LL(1)-Grammatik und

sei der Stand der Analyse mit der Auswertung der Regel

$A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$

bestimmt, wo entschieden werden muss, welche Produktionsregel für A anzuwenden ist

LL(1)-Analysealgorithmus (2)

Entscheidung lässt sich formalisieren

Entscheidungsalgorithmus

- zunächst wird überprüft,
  ob das LookAhead-Symbol in einer der FIRST(α_i)-Mengen enthalten ist
  - wenn **ja** : ist die Wahl der Regel eindeutig
  - wenn **nicht**: …

- … gibt es die Möglichkeit der Ableitung: A → ϵ ?
  - wenn **ja** : sollte FOLLOW(A) die Entscheidung bringen

- wenn **nicht**: (also keine ϵ –Ableitung bzw. keine FOLLOW-Lösung)
  liegt ein syntaktischer Fehler vor

Implementation der LL(1)-Analyse

Implementationsvarianten prädiktiver Top-Down-Verfahren

a) rekursiver Abstieg

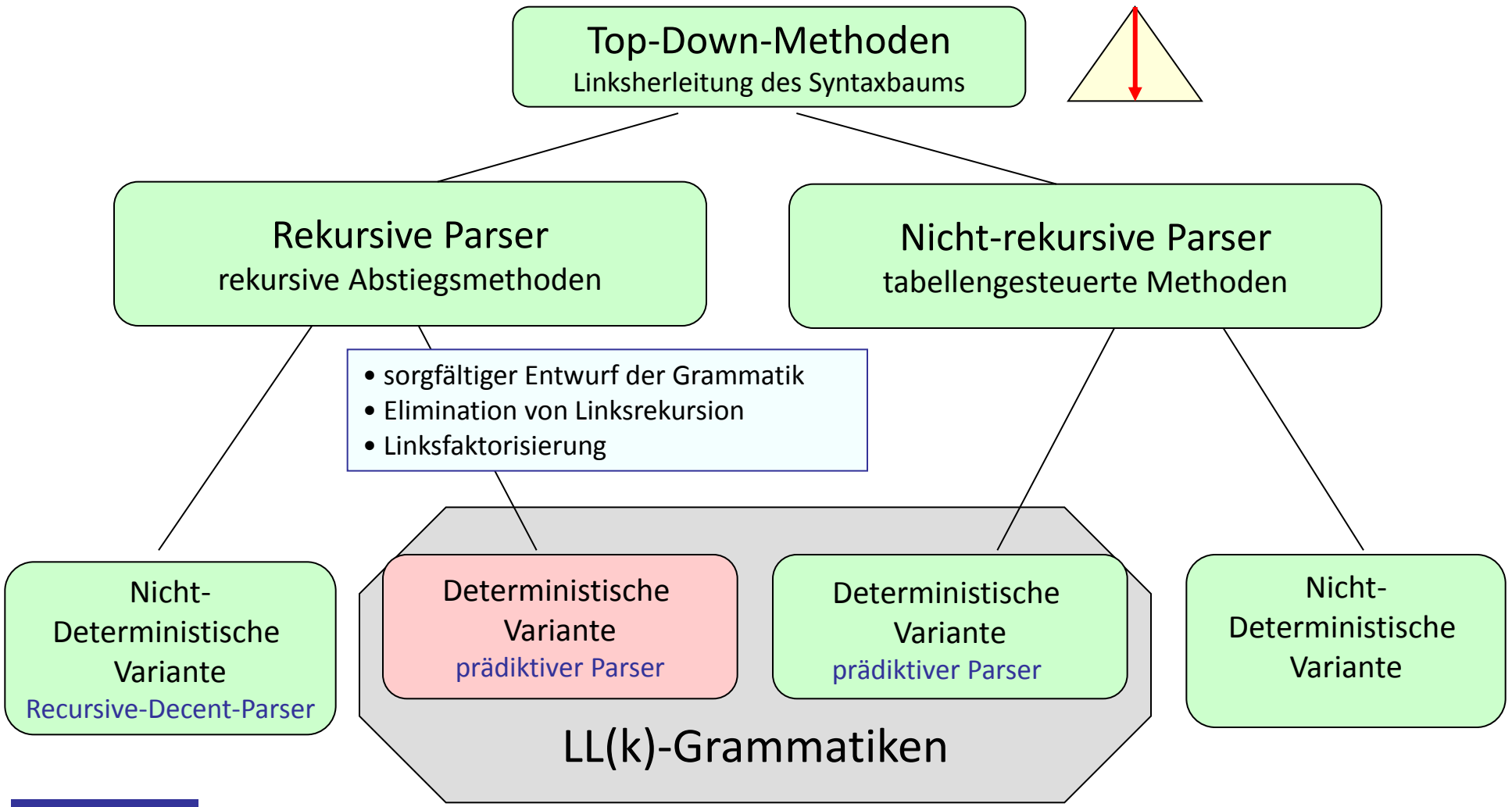
b) nicht-rekursive, tabellengesteuerte Analyse

Bemerkung

Umsetzung der FIRST- und FOLLOW-Mengen

am günstigsten in Form von Mengen als abstrakter Datentyp
(muss in C aber selbst implementiert werden)

Zwischenfazit: Klassen von Top-Down-Methoden



Position

- Teil I
  Die Programmie
- Teil II
  Methodische Gru
- Teil III
  Entwicklung ein

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- Kapitel 3
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Lexikalische Analyse: der Parser
- Kapitel 5
  Parsergeneratoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerierung

- 4.1
  Einführung in die Syntaxanalyse
- 4.2
  Restrukturierung von Grammatiken
- 4.3
  LL-Parser
- 4.4
  Beispiel: Ein-Pass-Compiler
  (Scanner, Parser, Übersetzer)

4.4 Beispiel

Ein einfacher Einpass-Compiler (Front- und Back-End)

als Transformation (s. letzte Vorlesung):

Infix-Ausdrücke ➔ Postfixausdrücke

bei Einsatz

- eines

  handgeschriebenen Scanners

- der prädiktiven Syntaxanalyse-Technik:

  rekursiv absteigender Parser

- der Übersetzungstechnik:

  Übersetzungsschema

Umformung der Grammatik

Ziel: Behandlung mit Prädiktiver Syntaxanalyse

nicht eindeutig

1.	*goal* → *expr*
2.	*expr* → *expr op expr*
3.	| **num**
4.	| **id**
5.	*op* → *
6.	| +
7.	| -
8.	| /
9.	| **DIV**
10.	| **MOD**

1.	
2.	
3.	
4.	*expr* → *expr* **+** *term*
5.	| *expr* **-** *term*
6.	| *term*
7.	*term* → *term* * *factor*
8.	| *term* **/** *factor*
9.	| *term* **DIV** *factor*
10.	| *term* **MOD** *factor*
11.	| *factor*
12.	*factor* → **num**
13.	| **id**

eindeutig, linksrekursive Regeln

Sequenz von expr;

1.	*goal* → *list*
2.	*list* → *expr*; *list*
3.	| ε
4.	*expr* → *expr* **+** *term* {print('+')}
5.	| *expr* **–** *term* {print('-')}
6.	| *term*
7.	*term* → *term* * *factor* {print('*')}
8.	| *term* **/** *factor* {print('/')}
9.	| *term* **DIV** *factor* {print('DIV')}
10.	| *term* **MOD** *factor* {print('MOD')}
11.	| *factor*
12.	*factor* → **num** {print(**num**.value)}
13.	| **id** {print(**id**.lexeme)}

Übersetzungsschema eindeutig, linksrekursive Regeln

Umformung der Grammatik

eindeutig, linksrekursive Regeln

1.	*goal* → *list*
2.	*list* → *expr*; *list*
3.	| ε
4.	*expr* → *expr* **+** *term* {print('+')}
5.	| *expr* **–** *term* {print('-')}
6.	| *term*
7.	*term* → *term* ***** *factor* {print('*')}
8.	| *term* **/** *factor* {print('/')}
9.	| *term* **DIV** *factor* {print('DIV')}
10.	| *term* **MOD** *factor* {print('MOD')}
11.	| *factor*
12.	*factor* → **(** *expr* **)**
13.	| **num** {print(**num**.value)}
14.	| **id** {print(**id**.lexeme)}

rechtsrekursives Übersetzungsschema

1.	*goal* → *list*
2.	*list* → *expr* **;** *list*
3.	| ε
4.	*expr* → *term moreterms*
5	*moreterms* → = **+** *term* {print('+')} *moreterms*
6.	| **–** *term* {print('-')} *moreterms*
7.	| ε
8.	*term* → *factor morefactors*
9.	*morefactors* → ***** *factor* {print('*')} *morefactors*
10.	| **/** *factor* {print('/')} *morefactors*
11.	| **DIV** *factor* {print('DIV')} *morefactors*
12.	| **MOD** *factor* {print('MOD')} *morefactors*
13.	| ε
14.	*factor* → **(** *expr* **)**
15.	| **num** {print(**num**.value)}
16.	| **id** {print(**id**.lexeme)}

Markierung des Eingabeendes

Sonderfall:

eof kann als Token erkannt und weitergereicht werden

- es folgt dem äußersten rechten Symbol der RS der **Startproduktion**
- häufig wird es dennoch in der Grammatik **weggelassen**,
  obwohl es vorhanden ist

➔ Wir fügen die eof-Behandlung explizit ein.

Behandlung von eof

Behandlung von **eof** als Token **$**

```
void parse()
/* analysiert und transformiert Ausdrücke */
{
        lookAhead = lexAnalyse();

        while (lookAhead != DONE) {
                expr();
                match(';');
        }
}
```

1.	*goal* → *list* **$**
2.	*list* → *expr* **;** *list*
3.	| ε
4.	*expr* → *term moreterms*
5	*moreterms* → **+** *term* {print('+')} *moreterms*
6.	| **−** *term* {print('-')} *moreterms*
7.	| ε
8.	*term* → *factor morefactors*
9.	*morefactors* → ***** *factor* {print('*')} *morefactors*
10.	| **/** *factor* {print('/')} *morefactors*
11.	| **DIV** *factor* {print('DIV')} *morefactors*
12.	| **MOD** *factor* {print('MOD')} *morefactors*
13.	| ε
14.	*factor* → **(** *expr* **)**
15.	| **num** {print(**num**.value)}
16.	| **id** {print(**id**.lexeme)}

Bestimmung von FIRST

$\alpha \in V$		Y_1	Y_2	Y_3	FIRST(Y_1)	ε	FIRST(α)
goal		list	**$**		{ ε , **(**, **num**, **id** }	ja	{ ε , **(**, **num**, **id** }
list		expr	**;**	list	{ **(**, **num**, **id** }	nein	
list	ε	ε			{ ε }	ja	{ ε, **(**, **num**, **id** }
expr		term	moreterms		{ **(**, **num**, **id** }	nein	{ **(**, **num**, **id** }
moreterms		**+**	term	moreterms	{ **+** }	nein	
moreterms		**-**	term	moreterms	{ **-** }	nein	
moreterms	ε	ε			{ ε }	ja	{ ε ,**+**, **-** }
term		factor	morefactors		{ **(**, **num**, **id** }	nein	{ **(**, **num**, **id** }
morefactors		*****	factor	morefactors	{ * }	nein	
morefactors		**/**	factor	morefactors	{ **/** }	nein	
morefactors		**DIV**	factor	morefactors	{ **DIV** }	nein	{ ε ,* , **/**, **DIV** ,**MOD** }
morefactors		**MOD**	factor	morefactors	{ **MOD** }	nein	
morefactors	ε	ε			{ ε }	ja	
factor		**(**	expr	**)**	{ **(** }	nein	{ **(**, **num**, **id** }
factor		**num**			{**num** }	nein	
factor		**id**			{**id** }	nein	

Test der LL(1)-Eigenschaft

α ∈ **V**	**FIRST(α)**	**FOLLOW(α)**
goal	{ ε , **(**, **num**, **id** }	{}
list	{ ε, **(**, **num**, **id** }	{ **$** }
expr	{ **(**, **num**, **id** }	{ **;** , **)** }
moreterms	{ ε ,**+**, **-** }	{ **;** , **)** }
term	{ **(**, **num**, **id** }	{ **+**, **-**, **;**, **)** }
morefactors	{ ε ,* , **/**, **DIV** ,**MOD** }	{ **+**, **-**, **;**, **)** }
factor	{ **(**, **num**, **id** }	{ *, **/**, **DIV**, **MOD**,**+**, **-**, **;**, **)** }

FIRST(expr) ∩ FIRST(ε) = ∅ **?**
{ **(**, **num**, **id** } ∩ {ε} = ∅

FIRST(expr) ∩ FOLLOW(list) = ∅ **?**
{ **(**, **num**, **id** } ∩ {**$**} = ∅

FIRST(moreterms) ∩ FOLLOW(moreterms) = ∅ **?**
{ε ,**+**, **-** } ∩ FOLLOW(expr) = ∅ **?**
{ε ,**+**, **-** } ∩ { **)**, **;**} = ∅

1.	*goal* → *list* **$**
2.	*list*→ *expr* **;** *list*
3.	| ε
4.	*expr* → *term moreterms*
5	*moreterms*→ **+** *term* {print('+')} *moreterms*
6.	| **−** *term* {print('-')} *moreterms*
7.	| ε
8.	*term* → *factor morefactors*
9.	*morefactors* → * *factor* {print('*')} *morefactors*
10.	| **/** *factor* {print('/')} *morefactors*
11.	| **DIV** *factor* {print('DIV')} *morefactors*
12.	| **MOD** *factor* {print('MOD')} *morefactors*
13.	| ε
14.	*factor* → **(** *expr* **)**
15.	| **num** {print(**num**.value)}
16.	| **id** {print(**id**.lexeme)}

FIRST(morefactors) ∩ FOLLOW(morefactors) = ∅ **?**
{ε ,* , **/**, **DIV**, **MOD** } ∩ FOLLOW(term) = ∅

Entwurf + Optimierung des Parsers

- ...als rekursives Abstiegsverfahren
  mit Übersetzungsschemata

und **Optimierungsansatz**

manche rekursive Aufrufe lassen sich durch einfache Iterationen ersetzen
(Effekt: Laufzeitverbesserung)

Parser-Änderungen

(1) End-ständige Rekursionen:
Selbstaufrufe (wie moreterms, morefactors) lassen sich durch Iterationen ersetzen

(2) Zusammenführung von
- expr und moreterms
- term und morefactors

zu jeweils einer Funktion

(3) strukturierte Ausgabe: Funktion emit *(Token, Tokenwert)*

1.	*goal* → *list* **$**
2.	*list* → *expr* **;** *list*
3.	| ε
4.	*expr* → *term moreterms*
5	*moreterms* → **+** *term* {print('+')} *moreterms*
6.	| **−** *term* {print('-')} *moreterms*
7.	| ε
8.	*term* → *factor morefactors*
9.	*morefactors* → ***** *factor* {print('*')} *morefactors*
10.	| **/** *factor* {print('/')} *morefactors*
11.	| **DIV** *factor* {print('DIV')} *morefactors*
12.	| **MOD** *factor* {print('MOD')} *morefactors*
13.	| ε
14.	*factor* → **(** *expr* **)**
15.	| **num** {print(**num**.value)}
16.	| **id** {print(**id**.lexeme)}

Vereinfachungen nach Code-Review (1)

Änderungen

(1) End-ständige Rekursionen → Iteration

(2) Zusammenführung von Routinen

(3) strukturierte Ausgabe

```
void expr() {
        term(); moreterms();
}
```

```
void moreterms() {
  if (lookAhead == '+') {
        match('+'); term(); print('+'); moreterms();
  }
  else if (lookAhead == '-') {
        match('-'); term(); print('-'); moreterms();
  else ; /* do nothing */
}
```

1.	*goal* ::= *list* **$**
2.	*list*::= *expr*; *list*
3.	| ε
4.	
5	*moreterms*::= **+** *term* {print('+')} *moreterms*
6.	| **−** *term* {print('-')} *moreterms*
7.	| ε*expr* ::= *term moreterms*
8.	*term ::= factor morefactors*
9.	*more*
10.	
11.	
12.	
13.	
14.	*facto*
15.	*facto*
16.	

```
void expr()  {
  int token;

  term();
  while (1)
      switch (lookAhead) {
            case '+':
            case '-':
                token= lookAhead;
                match(lookAhead);
                term();
                emit(token, NONE);
                continue;
            default:
              return;
            }
}
```

Vereinfachungen nach Code-Review (2)

```
void emit (int token, int tokenValue)   {   /* erzeugt die Ausgabe */
            switch (token) {
                        case '+':
                        case '-':
                        case '*':
                        case '/':     printf("%c\n", token);
                                      break;
                        case DIV: printf("div\n");
                                      break;
                        case MOD: printf("mod\n");
                                      break;
                        case NUM: printf("%d\n", tokenValue);
                                      break;
                        case ID:      printf("%s\n", symbolTable[tokenValue].lexPtr);
                                      break;
                        case ';':     printf(";\n");
                                      break;
                        default:      printf("token %d, tokenValue %d\n", token, tokenValue);
            }
}
```

Vereinfachungen nach Code-Review (3)

```
void term()
{
    factor();
    morefactors();
}
```

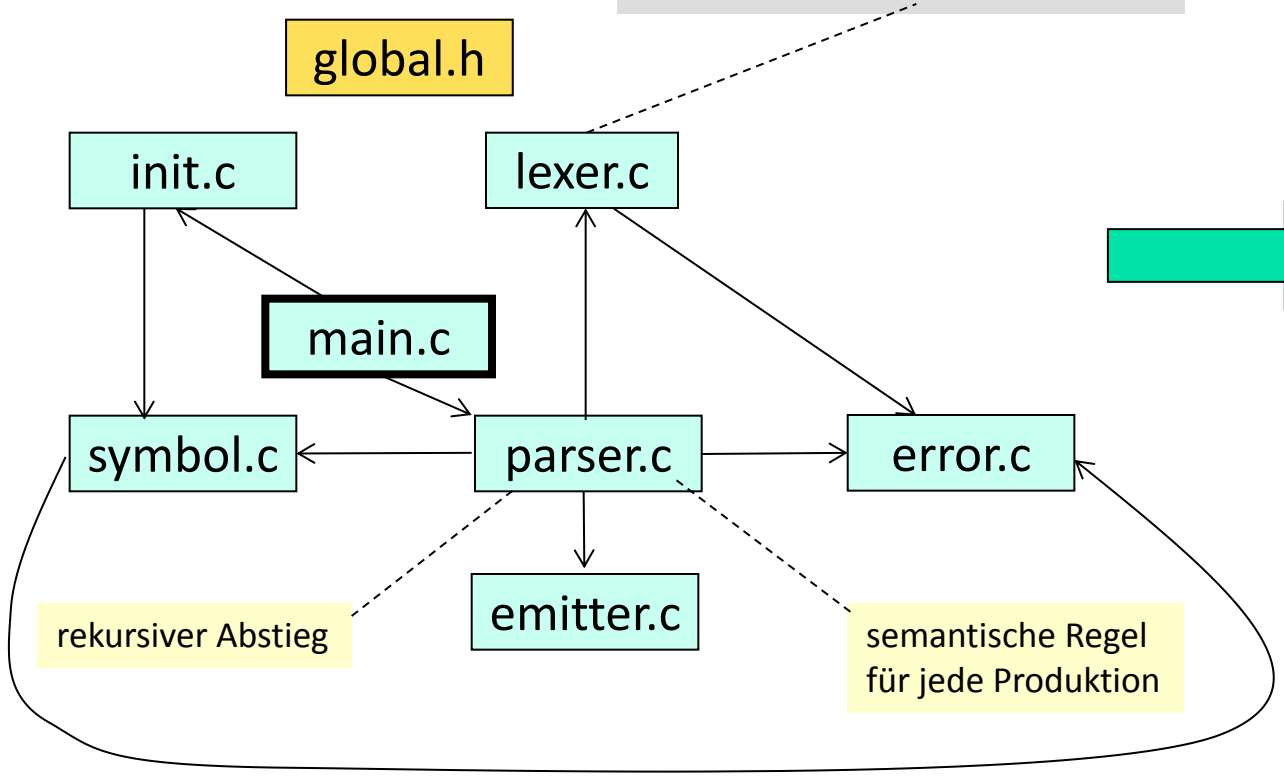
```
void morefactors()
{
   int token;

   switch (lookAhead) {
            case '*':
            case '/':
            case DIV:
            case MOD:
                 token = lookAhead;
                 match(lookAhead);
                 factor();
                 emit(token, NONE);
                 morefactors();
            default:
                 return;
            }
}
```

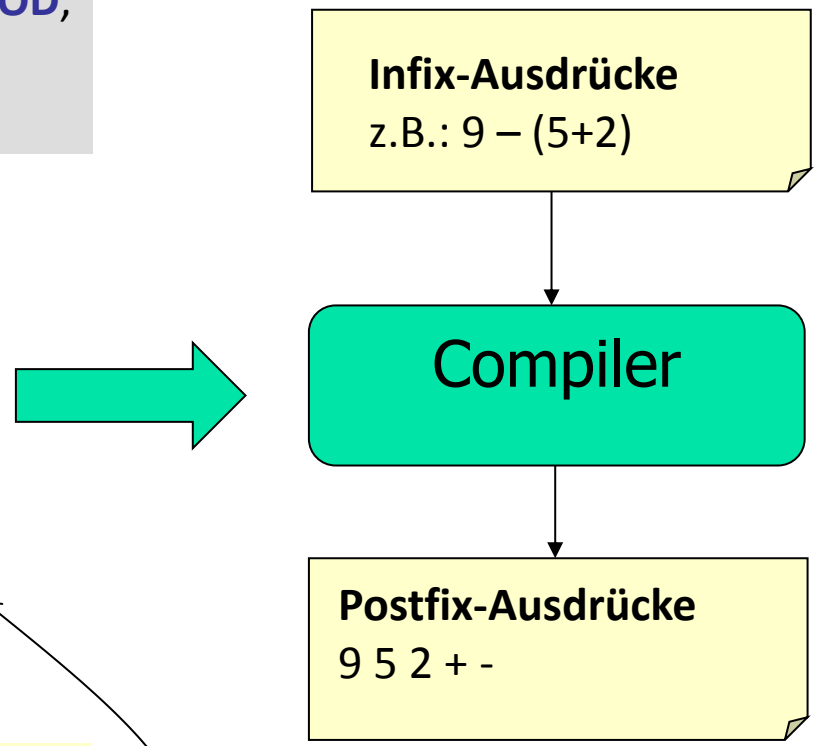
```
void term()
{
   int token;

   factor();
   while (1)
            switch (lookAhead) {
                     case '*':
                     case '/':
                     case DIV:
                     case MOD:
                              token= lookAhead;
                              match(lookAhead);
                              factor();
                              emit(token, NONE);
                              continue;
                     default:  return;
            }
    }
}
```

Umsetzung: Ein einfacher Ein-Pass-Compiler (komplette Handimplementation)

Verzicht auf individuelle Header-Files der einzelnen Module

Token: +, -, *, /, DIV, MOD, (,), ID, NUM, DONE

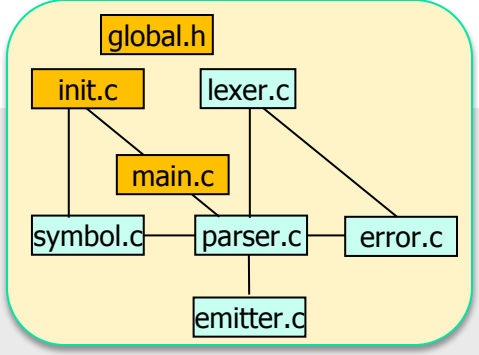




Aufruf-Abhängigkeit

global.h

```
#include <stdio.h>
#include <ctype.h>

#define BUFFERSIZE  128      /* Puffergroesse */
#define NONE                 -1
#define EOS                  '\0'

#define NUM                  256 /* bis 255 einfache Zeichen */
#define DIV                  257
#define MOD                  258
#define ID                   259
#define DONE                 260

int tokenValue;              /* Lexem, Tokenwert */
int lineNo;

typedef struct entry {
        char *lexPtr;
        int token;
} entry;

extern entry symbolTable[];

int insert (char[], int);        /* Op von symbolTabelle */
int lookup (char[]);             /* Op von symbolTabelle
void error (char *, int);
void error1 (char*, int);
void init();                    /*Initialisierung von symbolTabelle*/
void parse();
int lexAnalyse();               /* yylex();  falls generierter Lexer*/
void emit(int, int);
void expr();
void term();
void factor();
void match(int);
```



main.c

```
#include "global.h"

int main()
{
        init();
        parse();
        return 0;              /* erfolgreiche Terminierung */
}
```

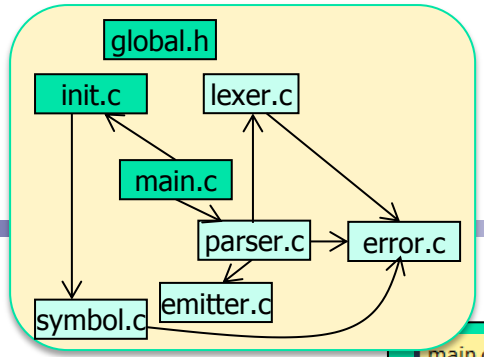
init.c

```
#include "global.h"

entry keywords[]= {
        { "div", DIV },
        { "mod", MOD },
        { 0, 0 }  /* Schlusskennung */
};

void init()  /* laedt die Modul-lokalen Schluesselworte in die
                Symboltabelle */
{
        entry *p;
        for (p= keywords; p->token; p++)
                insert(p->lexPtr, p->token);
}
```

global.h

```c
#include <stdio.h>
#include <ctype.h>

#define BUFFERSIZE  128 /* Puffergroesse */
#define NONE        -1
#define EOS         '\0'

#define NUM         256
#define DIV         257
#define MOD         258
#define ID          259
#define DONE        260

int tokenValue;     /* Lexem, Tokenwert */
int lineNo;

struct entry {
    char *lexPtr;
    int token;
};

extern struct entry symbolTable[];


int insert(char[], int);
void error(char *, int);
int lookup(char[]);
void init();
void parse();
int lexAnalyse();
void emit(int, int);
void expr();
void term();
void factor();
void match(int);
```

yylex();
/*falls generierter Lexer*/



main.c

```c
#include "global.h"

int main()
{
    init();
    parse();
    return(0);  /* erfolgreiche Terminierung */
}
```

init.c

```c
#include "global.h"

struct entry keywords[]= {
    { "div", DIV },
    { "mod", MOD },
    { 0, 0 }
};


void init() /* laedt die Schluesselworte in die Symboltabelle */
{
    struct entry *p;
    for (p= keywords; p->token; p++)
        insert(p->lexPtr, p->token);
}
```

global.h

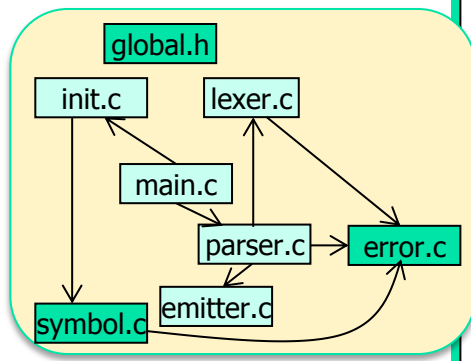
```c
#include <stdio.h>
#include <ctype.h>

#define BUFFERSIZE   128 /* Puffergroesse */
#define NONE         -1
#define EOS          '\0'

#define NUM          256
#define DIV          257
#define MOD          258
#define ID           259
#define DONE         260

int tokenValue;      /* Lexem, Tokenwert */
int lineNo;

struct entry {
    char *lexPtr;
    int token;
};

extern struct entry symbolTable[];


int insert(char[], int);
void error(char *, int);
int lookup(char[]);
void init();
void parse();
int lexAnalyse();
void emit(int, int);
void expr();
void term();
void factor();
void match(int);
```

error.c

```c
#include "global.h"
#include <stdlib.h>
#include <ctype.h>

void error(char *n, int token)
{
    fprintf(stderr, "Zeile %d: %s: %c\n", lineNo, n, token);
    exit(1);    /* keine erfolgreiche Terminierung */
}
```

symbol.c

```c
#include <string.h>
#include "global.h"

#define STRMAX 999 /* Größe des Lexem_arrays */
#define SYMMAX 999 /* Größe der Symboltabelle */

char lexemes[STRMAX];
int lastChar= -1;   /* zuletzt benutzte lexemes-Position */

struct entry symbolTable[SYMMAX];
int lastEntry= 0;   /* zuletzt benutzter SymbolTable-Eintrag */

int lookup(char s[])    /* liefert Index des Eintrages für s */
{
    int p;

    for (p=lastEntry; p>0; p= p-1)
        if (strcmp(symbolTable[p].lexPtr, s) == 0)
            return p;
    return 0;
}

int insert(char s[], int tok)   /* */
{
    int len;
    len= strlen(s);

    if (lastEntry + 1 >= SYMMAX)
        error("interner Fehler: Symboltabelle voll, Anzahl von Einträgen >", SYMMAX);
    if (lastChar + len + 1 >= STRMAX)
        error("interner Fehler: LexemArray voll, Länge >", STRMAX);
    lastEntry++;
    symbolTable[lastEntry].token= tok;
    symbolTable[lastEntry].lexPtr= &lexemes[lastChar + 1];
    lastChar= lastChar + len + 1;
    strcpy(symbolTable[lastEntry].lexPtr, s);
    return lastEntry;
}
```

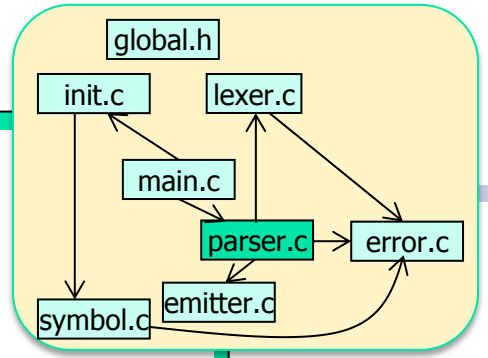


parser.c

```c
#include "global.h"

int lookAhead;

void parse()    /* analysiert und transformiert Ausdrücke */
{
    lookAhead= lexAnalyse();

    while (lookAhead != DONE) {
        expr(); match(';');
    }
}

void expr()
{
    int token;
    term();
    while (1)
        switch (lookAhead) {
            case '+':
            case '-':   token= lookAhead;
                        match(lookAhead);
                        term();
                        emit(token, NONE);
                        continue;
            default:    return;
        }
}

void term()
{
    int token;

    factor();
    while (1)
        switch (lookAhead) {
            case '*':
            case '/':
            case DIV:
            case MOD:
                token= lookAhead;
                match(lookAhead);
                factor();
                emit(token, NONE);
                continue;
            default:    return;
        }
}
```

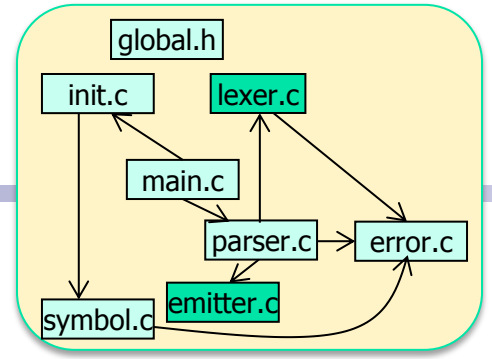
```c
void term()
{
    int token;

    factor();
    while (1)
        switch (lookAhead) {
            case '*':
            case '/':
            case DIV:
            case MOD:
                token= lookAhead;
                match(lookAhead);
                factor();
                emit(token, NONE);
                continue;
            default:    return;
        }
}

void factor()
{
    switch (lookAhead) {
        case '(':
            match('(');
            expr();
            match(')');
            break;
        case NUM:
            emit(NUM, tokenValue);
            match(NUM);
            break;
        case ID:
            emit(ID, tokenValue);
            match(ID);
            break;
        default:
            error("Syntaxfehler: illegales Symbol", lookAhead);
        }
}
```

```c
void match(int token)
{
    if (lookAhead == token){
        if (token == ';') printf(";\n");
        lookAhead= lexAnalyse();
    }
    else
        error("Syntaxfehler", token);
}
```



lexer.c

```c
#include "global.h"

char lexBuffer[BUFFERSIZE];
int lineNo= 1;
int tokenValue= NONE;

int lexAnalyse()    /* Scanner */
{
    int c;

    while (1)
    {
        c= getchar();
        if (c == ' ' || c == '\t')
            ; /* Entfernung von Leerraeumen */
        else if (c == '\n')
            lineNo++;
        else if ( isdigit(c) )
            /*number */
        {
            ungetc (c, stdin);
            scanf("%d", &tokenValue);
            return NUM;
        }
        else if (isalpha(c))
        {
            int p, b =0;
            while (isalnum(c))
            {
                lexBuffer[b]= c;
                c= getchar();
                b++;
                if (b >= BUFFERSIZE)
                    error("Fehler: Bezeichnerlänge größer als",BUFFERSIZE );
            }
            lexBuffer[b]= EOS;
            if (c!= EOF)
                ungetc(c, stdin);
            p= lookup(lexBuffer);
            if (p==0)
                p= insert(lexBuffer, ID);
            tokenValue= p;
            return symbolTable[p].token;
        }
        else if (c==EOF)
            return DONE;
        else {
            tokenValue= NONE;
            return c;
        }
    }
}
```



emitter.c

```c
#include "global.h"

void emit(int token, int tokenValue)    /* erzeugt die Ausgabe */
{
    switch (token) {
        case '+':
        case '-':
        case '*':
        case '/':   printf("%c\n", token);
                    break;
        case DIV:   printf("div\n");
                    break;
        case MOD:   printf("mod\n");
                    break;
        case NUM:   printf("%d\n", tokenValue);
                    break;
        case ID:    printf("%s\n", symbolTable[tokenValue].lexPtr);
                    break;
        case ';':   printf(";\n");
                    break;
        default:    printf("token %d, tokenValue %d\n", token, tokenValue);
    }
}
```

Makefile

make
Realisierung von Übersetzung, ...

Kürzel: zur Auszeichnung der ersten abhängigen Quelle, hier: emitter.c

Einsatz von gcc als Compiler, Linker, ...

alle Warnungen

Sprachstandard C-89

Fehleranzeigemodus

```
CC = gcc
CFLAGS = -Wall -std=c89 -pedantic

OBJECTS = emitter.o error.o init.o lexer.o main.o
          parser.o symbol.o

c0:  $(OBJECTS)
     $(CC) $(CFLAGS) -o c0 $(OBJECTS)
```

Angabe des Zielnamens c0

Symbolische Variable des Make-Kommandos

Name des verbundenen Programms als Ausgabe des Programmverbinders (Linker)

```
emitter.o: emitter.c global.h
           $(CC) $(CFLAGS) -c $<

error.o: error.c global.h
           $(CC) $(CFLAGS) -c $<

init.o: init.c global.h
           $(CC) $(CFLAGS) -c $<

lexer.o: lexer.c global.h
           $(CC) $(CFLAGS) -c $<

main.o: main.c global.h
           $(CC) $(CFLAGS) -c $<

parser.o: parser.c global.h
           $(CC) $(CFLAGS) -c $<

symbol.o: symbol.c global.h
           $(CC) $(CFLAGS) -c $<

clean:
           rm -f *.o c0
```

force: unter allen Umständen

make clean
(Löschen generierter Dateien)