Kapitel 5: Syntax–Analyse

Aufgabe

Die Token–Folge wird strukturiert in Anweisungen, Ausdrücke etc., um die Semantische Analyse und Code–Erzeugung zu ermöglichen

Themen

- **Kontextfreie Grammatik**
- **Äquivalente Grammatiken**
- **Top Down Analyse**
- **Bottom Up Analyse**

Kontextfreie Grammatik

- Grundlage für die Definition von Programmiersprachen
- Grundlage für die Syntax-Analyse im Übersetzer

Beispiel: Grammatik für Funktionsaufrufe

```
F -> id
F -> id(L)
L -> L,L
L -> num
L -> F
```

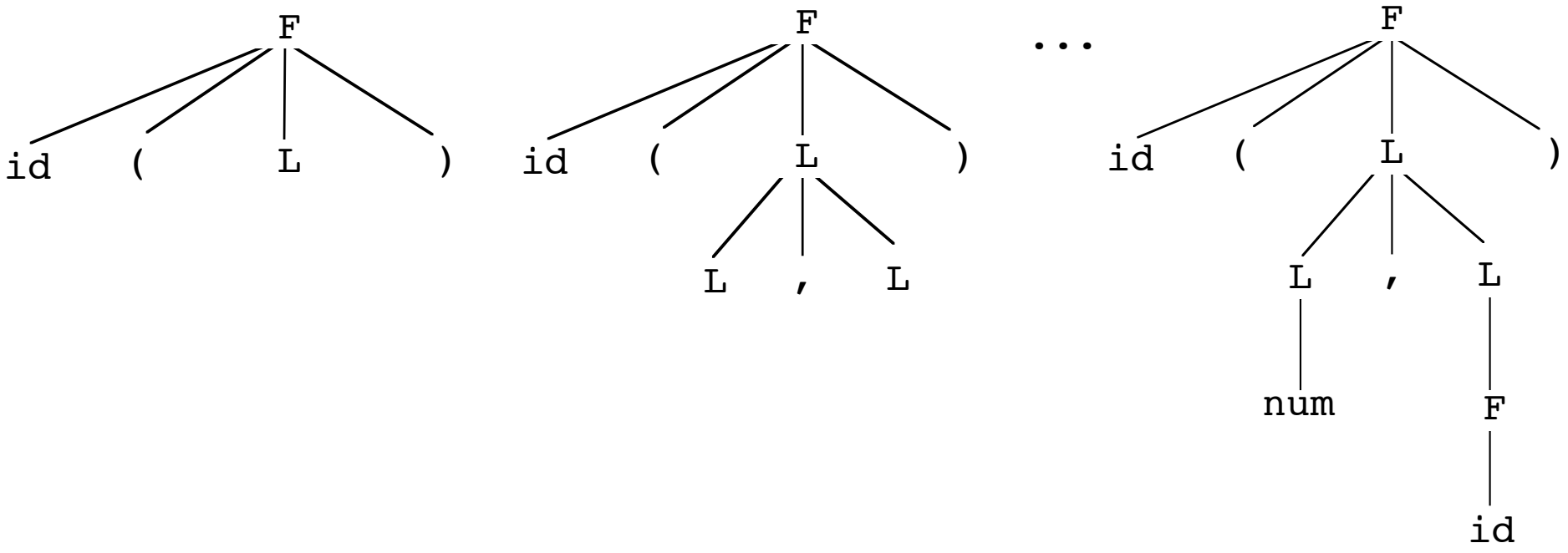
Eine Ableitung

```
F => id(L) => id(L,L) => id(num,L) => id(num,F) => id(num,id)
```

In jedem Ableitungschritt wird eine Produktion der Grammatik angewandt, d.h. ein Nonterminal wird durch eine seiner rechten Seiten ersetzt

Ableitungsbaum

```
F => id(L) => id(L,L) => id(num,L) => id(num,F) => id(num,id)
```



Der Ableitungsbaum zeichnet die Ableitung auf,
d.h. jede angewandte Produktion wird als Teilbaum eingetragen
(Nonterminal als Wurzel, rechte Seite als Nachfolger)

Äquivalenz von Grammatiken

Zwei Grammatiken sind äquivalent, wenn sie dieselbe Sprache beschreiben

Folgende Grammatiken beschreiben z.B. Listen von Zahlen `num,num,...`

```
G1: L -> L , L
    L -> num
```
links- und rechtsrekursiv

```
G2: L -> L , num
    L -> num
```
linksrekursiv

```
G3: L -> num , L
    L -> num
```
rechtsrekursiv

```
G4: L  -> num Lr
    Lr -> , num Lr
    Lr -> ε
```
restrekursiv

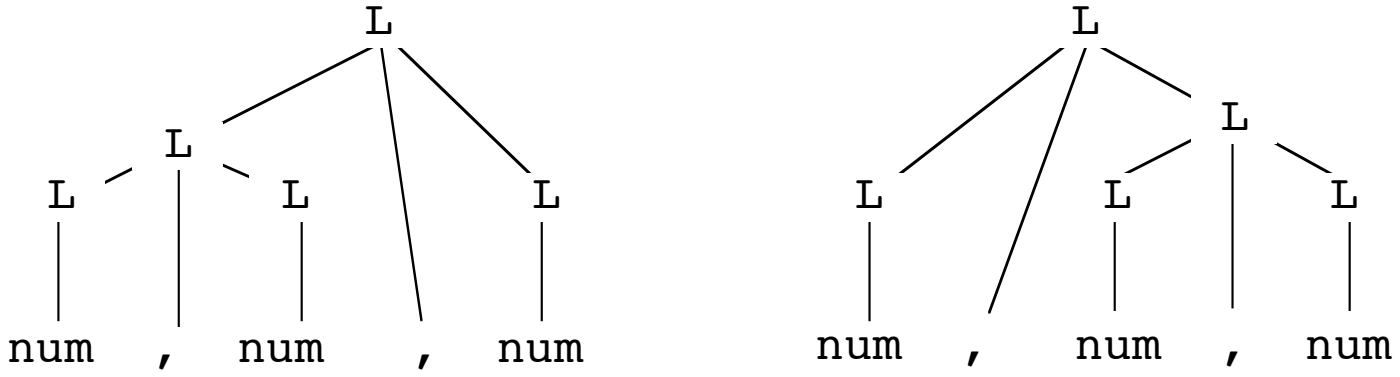
```
G5: L -> num(, num)*
```
Regular Right Part

```
G6: L -> num{, num}
```
Extended BNF

Links– und Rechtsrekursion

Beispiel

```
L -> L , L
L -> num
```

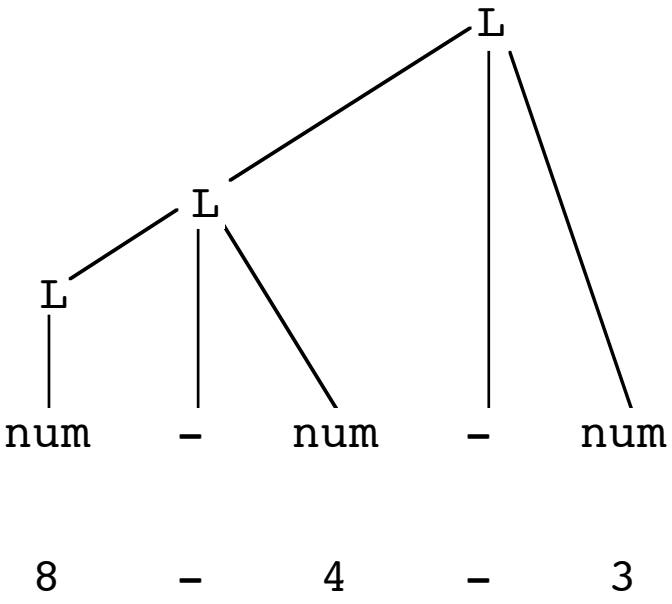


Die Grammatik ist mehrdeutig,
weil es zu einer Satzform verschiedene Ableitungsbäume gibt

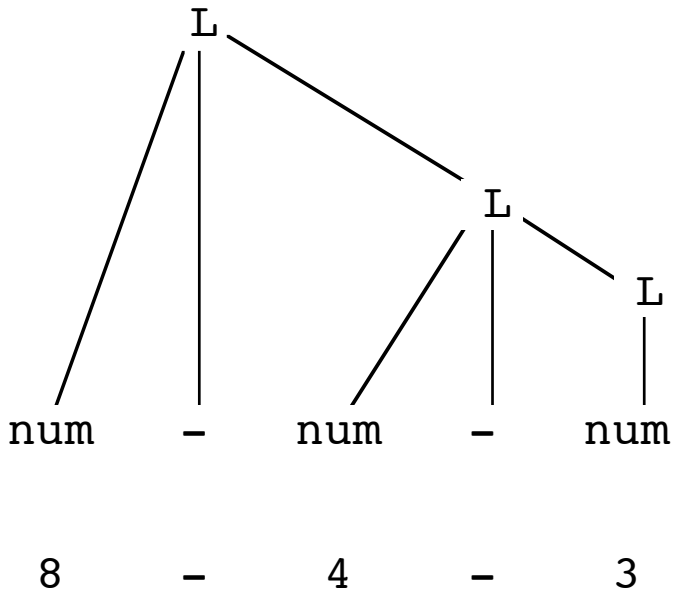
Links– vs. Rechtsrekursion

Beispiel

```
L -> L - num
L -> num
```



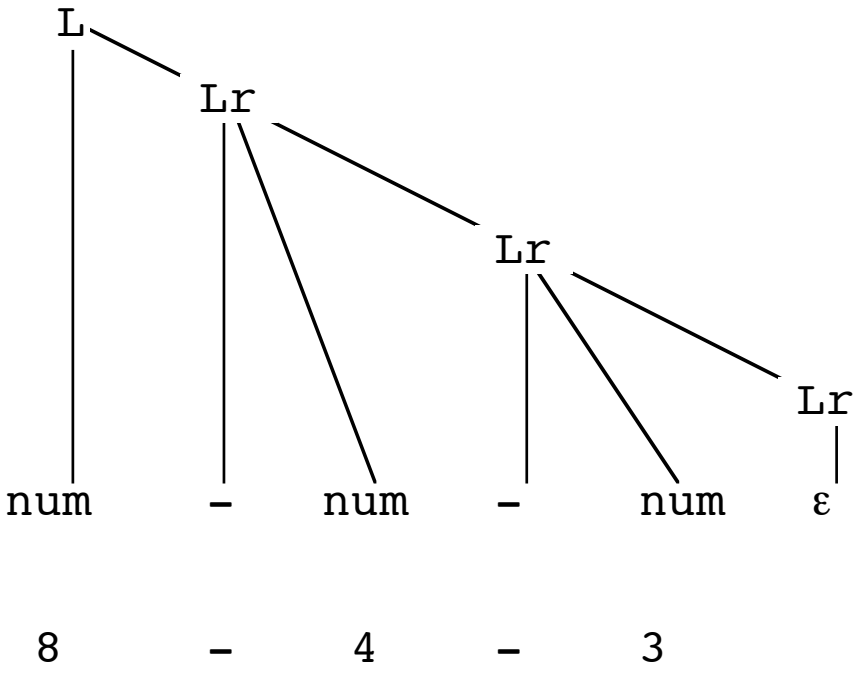

```
L -> num - L
L -> num
```



Restrekursion

Beispiel

```
L  -> num Lr
Lr -> - num Lr
Lr -> ε
```



Zwei äquivalente Grammatiken für Ausdrücke

Linksrekursion

```
S -> E

E -> E + T | T

T -> T * F | F

F -> ( E ) | id
```

<=>

Restrekursion

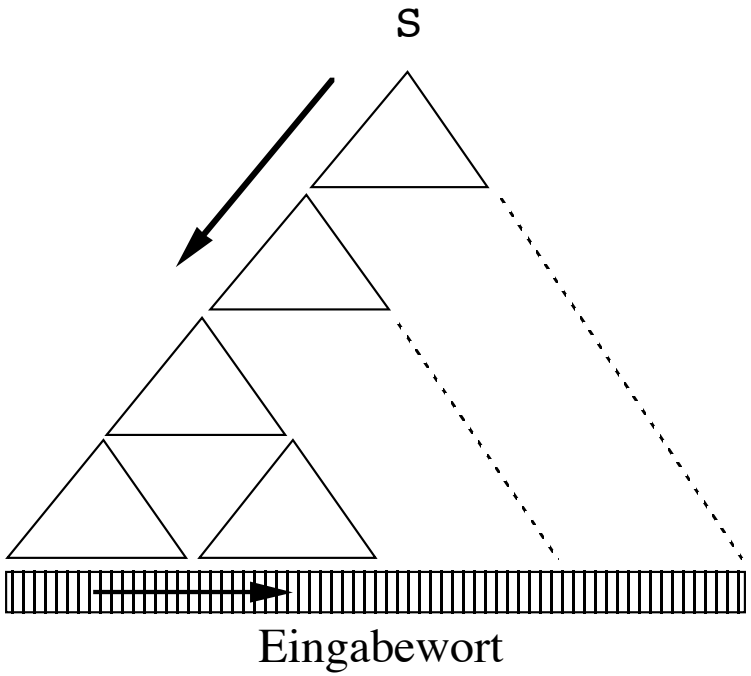
```
S -> E

E  -> T Er
Er -> + T Er | ε

T  -> F Tr
Tr -> * F Tr | ε

F  -> ( E ) | id
```

geeignet für
BOTTOM UP Analyse

geeignet für
TOP DOWN Analyse

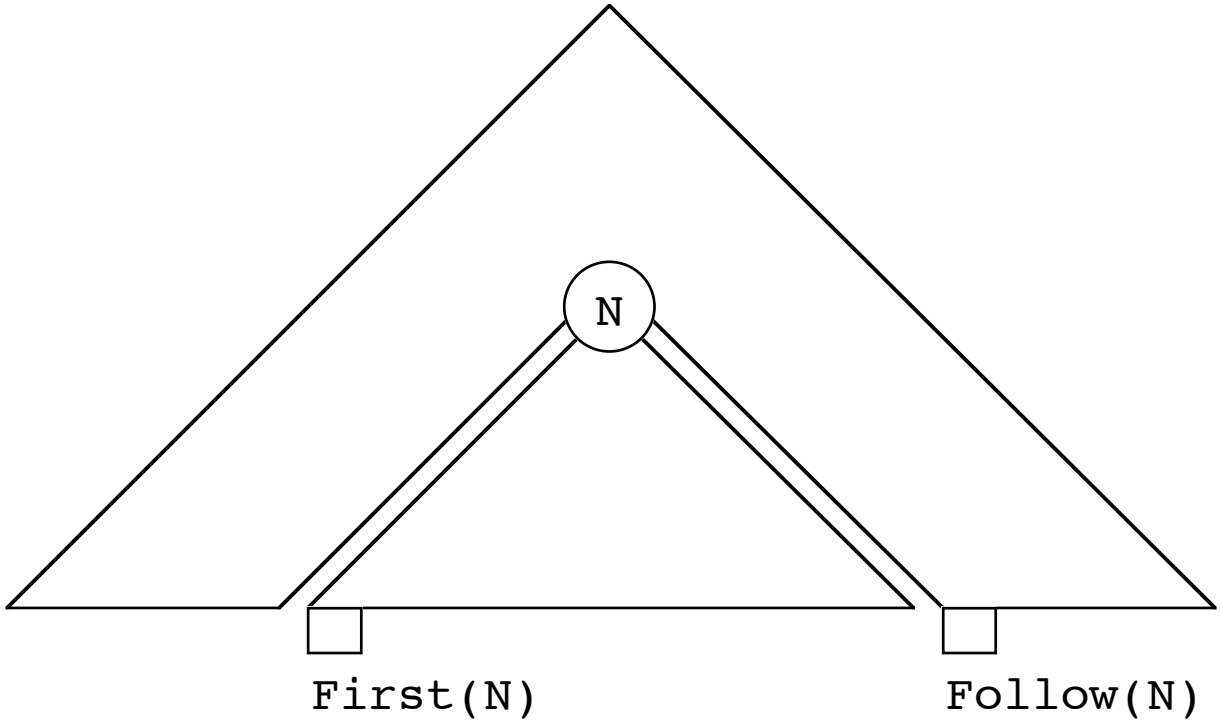
TOP DOWN Syntax-Analyse

Vom Startsymbol ausgehend wird versucht, das Eingabewort abzuleiten



Entscheidungsproblem: Welche Produktion ist bei Alternativen anzuwenden ?

FIRST- und FOLLOW-Mengen



`First(N)` **gibt an, mit welchen Terminalen die aus** `N` **ableitbaren Satzformen beginnen können, bzw. ob** `N` **nach** ε **ableitbar ist**

`Follow(N)` **gibt an, welche Terminalsymbole in beliebigen (aus** `S` **ableitbaren) Satzformen unmittelbar nach** `N` **folgen können, bzw. ob** `$` **folgen kann**

TOP DOWN Analyse-Tabelle

Grammatik

```
S  -> E $              First(S)={id,(}
E  -> T Er             First(E)={id,(}
Er -> + T Er | ε       First(Er)={+,ε}    Follow(Er)={),$}
T  -> F Tr             First(T)={id,(}
Tr -> * F Tr | ε       First(Tr)={*,ε}    Follow(Tr)={),+,$}
F  -> ( E ) | id       First(F)={id,(}
```

Analyse-Tabelle: (Ziel,Eingabesymbol) -> neue Ziele

	id	(	)	+	*	$
S	E	E	.	.	.	.
E	T Er	T Er	.	.	.	.
Er	.	.	ε	+ T Er	.	ε
T	F Tr	F Tr	.	.	.	.
Tr	.	.	ε	ε	* F Tr	ε
F	id	(E)	.	.	.	.

Ablauf einer TOP DOWN Analyse

	id	(	)	+	*	$
S	E	E	.	.	.	.
E	T Er	T Er	.	.	.	.
Er	.	.	ε	+ T Er	.	ε
T	F Tr	F Tr	.	.	.	.
Tr	.	.	ε	ε	* F Tr	ε
F	id	(E)	.	.	.	.

Keller	Eingabe	Produktion (neue Ziele)
$ S	id + id * id $	S -> E
$ E	id + id * id $	E -> T Er
$ Er T	id + id * id $	T -> F Tr
$ Er Tr F	id + id * id $	F -> id
$ Er Tr id	id + id * id $	match id
$ Er Tr	+ id * id $	Tr -> ε (nicht: Tr -> * F Tr)
$ Er	+ id * id $	Er -> + T Er (nicht: Er -> ε)
$ Er T +	+ id * id $	match +
$ Er T	id * id $	T -> F Tr
$ ...		

Rekursive Syntax-Prozeduren

Für jede Zeile der Tabelle (Nonterminal) wird eine Prozedur gebildet

	id	(	)	+	*	$
S	E	E	.	.	.	.
E	T Er	T Er	.	.	.	.
Er	.	.	ε	+ T Er	.	ε
T	F Tr	F Tr	.	.	.	.
Tr	.	.	ε	ε	* F Tr	ε
F	id	(E)	.	.	.	.

```
void E() {
    if (sym=id||sym='(') {T();Er();}
    else Error();
}

void Er() {
    if (sym='+'){nextsym();T();Er();}
    else if (sym=')'||sym='$'){}
    else Error();
}
```

LL(1)–Grammatik

Eine LL(1)– Grammatik erlaubt eine deterministische TOP DOWN Analyse [von Links nach rechts mittels Linksableitung und 1 Symbol Vorausschau]

Jede LL(1)–Grammatik hat eine eindeutige Analyse–Tabelle

Jede LL(1)–Grammatik erfüllt folgende Bedingung:

Für jede Produktion N -> α_1 | α_2 | … gilt

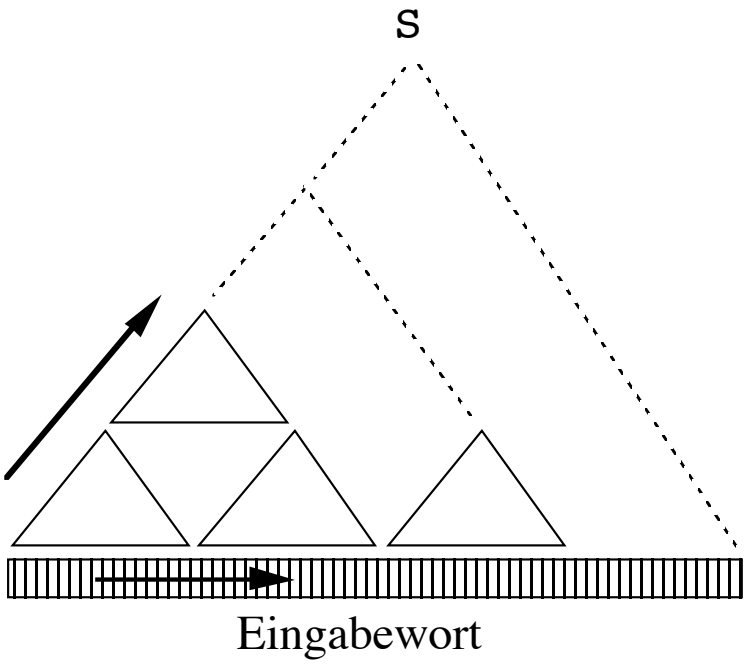
1. First(α_i) ∩ First(α_j) = {} f.a. i,j (i≠j)
2. höchstens ein α_i läßt sich nach ε ableiten
3. falls α_i =*=> ε, dann First(α_j) ∩ Follow(N) = {} f.a. j≠i

Gegenbeispiel: A -> A a | b First(A a) ∩ First(b) = {b} ≠ {}

Linksrekursive Produktionen sind nicht zur TOP DOWN Analyse geeignet

BOTTOM UP Syntax-Analyse

Vom Eingabewort ausgehend wird versucht, zum Startsymbol zu reduzieren



Entscheidungsproblem: Reduzieren oder Weiterlesen?

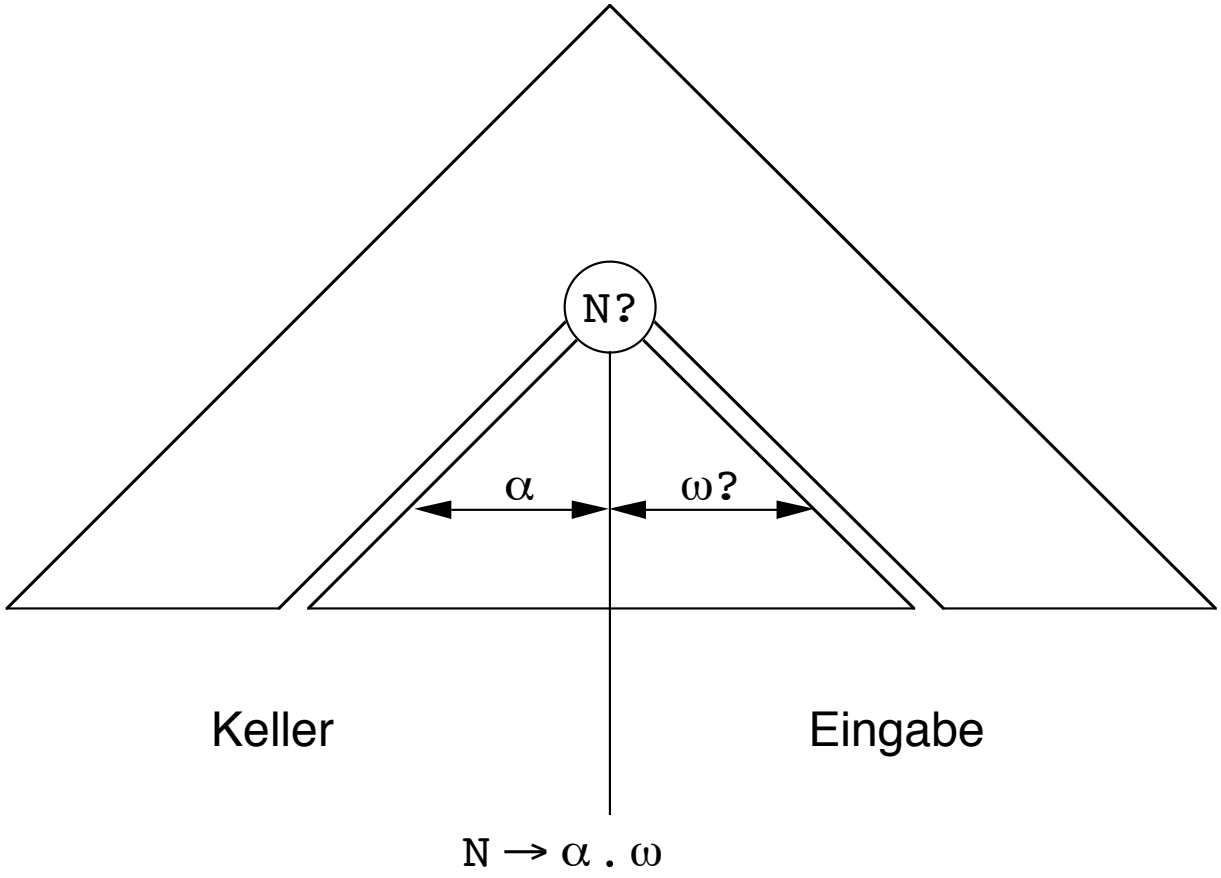
Ablauf einer BOTTOM UP Analyse

Grammatik (linksrekursiv)

```
S -> E
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```

Keller	Eingabe	Aktion
$	id + id * id $	shift id
$ id	+ id * id $	reduce F -> id
$ F	+ id * id $	reduce T -> F
$ T	+ id * id $	reduce E -> T
$ E	+ id * id $	shift + (nicht: reduce S -> E)
$ E +	id * id $	shift id
$ E + id	* id $	reduce F -> id
$ E + F	* id $	reduce T -> F
$ E + T	* id $	shift * (nicht: reduce E -> E + T)
$ E + T *	id $	shift id
$ E + T * id	$	reduce F -> id
$ E + T * F	$	reduce T -> T * F
$ E + T	$	reduce E -> E + T
$ E	$	reduce S -> E
$ S	$	accept

Analysezustand und Item



Items

Ein Item $N \to \alpha \,.\, \omega$ ist eine Produktion, die einem Analysezustand zugeordet ist:
Um N abzuleiten, wurde bereits α abgeleitet und ω muß noch abgeleitet werden

Fallunterscheidungen für ω

(1) $N \to \alpha \,.\, a\beta$

Aktion `shift`, falls `a` in der Eingabe

(2) $N \to \alpha \,.$

Aktion `reduce`, falls ein Element aus `Follow(N)` in der Eingabe

(3) $N \to \alpha \,.\, A\beta$ **mit** $A \to \gamma$

Um A abzuleiten, muß γ abgeleitet werden,
d.h. $A \to .\, \gamma$ ist dem gleichen Zustand zugeordnet (Zustand = Item–Menge)

Konstruktion der Item–Mengen

Grammatik

```
(0) S -> E
(1) E -> E op T
(2) E -> T
(3) T -> ( E )
(4) T -> id
```

Item–Mengen

```
I0 = { S -> . E
       E -> . E op T
       E -> . T
       T -> . ( E )
       T -> . id }
```

```
I1 = { S -> E .                 I1 = goto(I0,E)
       E -> E . op T }
```

```
I2 = { E -> T . }               I2 = goto(I0,T)
```

```
I3 = { T -> ( . E )             I3 = goto(I0,()
       E -> . E op T
       ...
```

BOTTOM UP Analyse-Tabellen

Item-Mengen (Ausschnitt)

```
I1 = { S -> E .                    I1 = goto(I0,E)
       E -> E . op T }
-------------------------------------------------
I2 = { E -> T . }                  I2 = goto(I0,T)
-------------------------------------------------
```

	action-Tabelle					goto-Tabelle					
state	id	op	(	)	$	id	op	(	)	E	T
0	s	.	s	.	.	4	.	3	.	1	2
1	.	s	.	.	r0	.	5	.	.	.	.
2	.	r2	.	r2	r2	.	.	.	.	.	.
3	s	.	s	.	.	4	.	3	.	6	2
4	.	r4	.	r4	r4	.	.	.	.	.	.
5	s	.	s	.	.	4	.	3	.	.	7
6	.	s	.	s	.	.	5	.	8	.	.
7	.	r1	.	r1	r1	.	.	.	.	.	.
8	.	r3	.	r3	r3	.	.	.	.	.	.

Ablauf einer BOTTOM UP Analyse

	action-Tabelle					goto-Tabelle					
state	id	op	(	)	$	id	op	(	)	E	T
0	s	.	s	.	.	4	.	3	.	1	2
1	.	s	.	.	r0	.	5	.	.	.	.
2	.	r2	.	r2	r2	.	.	.	.	.	.
3	s	.	s	.	.	4	.	3	.	6	2
4	.	r4	.	r4	r4	.	.	.	.	.	.
5	s	.	s	.	.	4	.	3	.	.	7
6	.	s	.	s	.	.	5	.	8	.	.
7	.	r1	.	r1	r1	.	.	.	.	.	.
8	.	r3	.	r3	r3	.	.	.	.	.	.

Keller	Eingabe	Aktion
0	id op id op id $	s
0 4	op id op id $	r4 (T -> id)
0 2	op id op id $	r2 (E -> T)
0 1	op id op id $	s
0 1 5	id op id $	s
0 1 5 4	op id $	r4 (T -> id)
0 1 5 7	op id $	r1 (E -> E op T)
0 1	op id $	s
...		

LR(1)–Grammatik

Eine LR(1)– Grammatik erlaubt eine deterministische BOTTOM UP Analyse [von Links nach rechts mittels Rechtsableitung und 1 Symbol Vorausschau]

Jede LR(1)–Grammatik hat eine eindeutige Analyse–Tabelle (action/goto)

LR–Verfahren sind mächtiger als LL–Verfahren in Bezug auf die Grammatiken, die sie verarbeiten können

$$LR(1) \supset LL(1)$$

Bison Beispiel

```
%token   ASSIGNOP PLUSASSIGNOP
%left    PLUSOP
%token   REGISTER NUMBER ';'
%start   stmt
%%

stmt     :   REGISTER ASSIGNOP expr ';'
         |   REGISTER PLUSASSIGNOP expr ';'
         ;
expr     :   REGISTER
         |   constexpr
         ;
constexpr:   NUMBER
         |   constexpr PLUSOP constexpr
         ;
%%
int yyerror(char *e) {
        printf("Parser error: '%s'...\n", e);
        exit(2);
}

int main(void) {
        yyparse();
        return 0;
}
```