



Compiler

Syntaktische Analyse

# Kapitel 3b

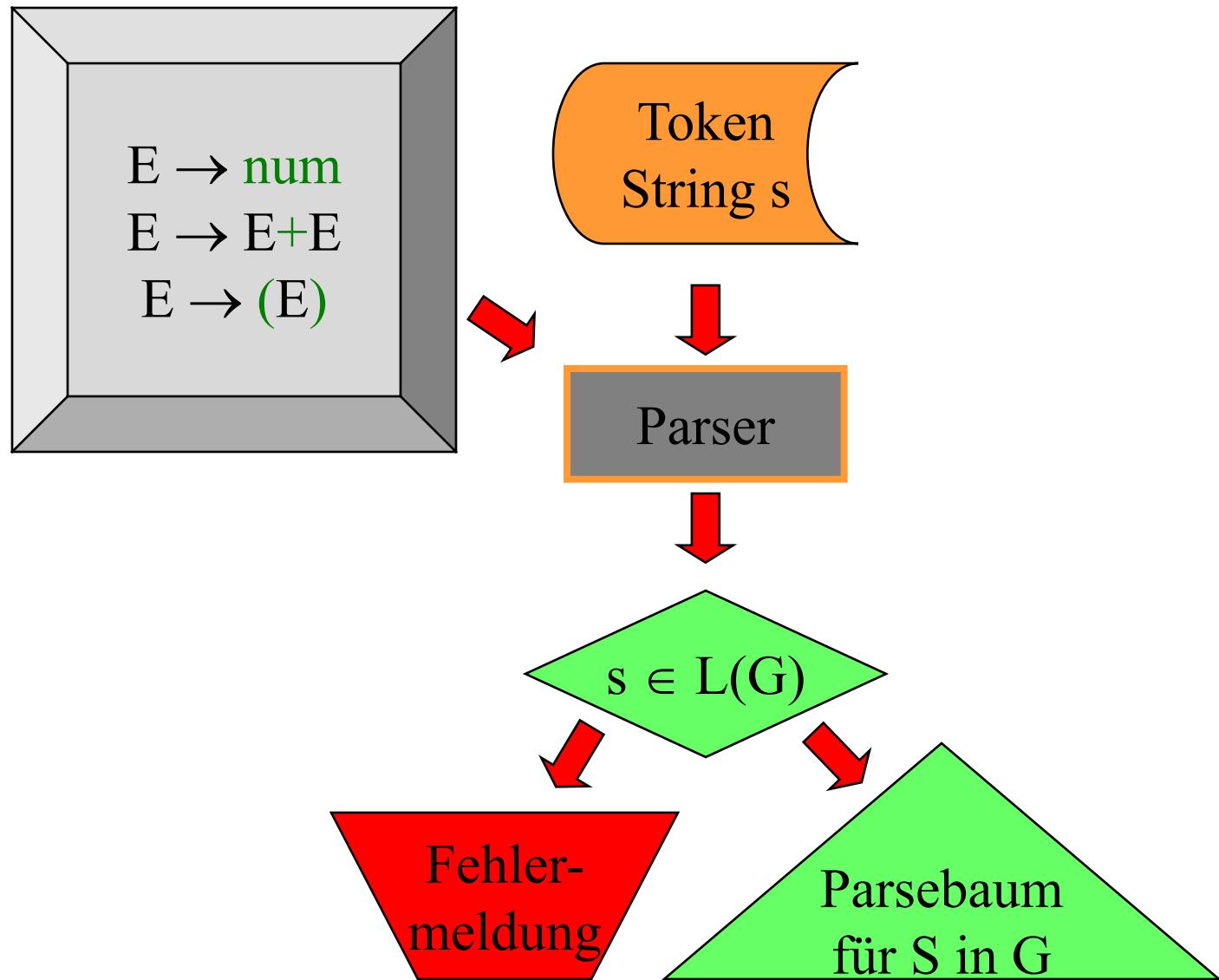
## Syntaktische Analyse: LL Parsing

# Was ist Parsing?



Compiler

Syntaktische Analyse





Compiler

Syntaktische Analyse

# Beispiel

- $S \rightarrow (S)S \mid \epsilon$
- Was ist die generierte Sprache?
- Beweis?



Compiler

Syntaktische Analyse

$$S \rightarrow (S)S \mid \epsilon$$

generiert nur ausgegliche Stringe

- Induktion über Länge der Ableitung
- Basis:  $n = 0$ 
  - $S \Rightarrow \epsilon$  ist ausgeglichen
- Induktion: Ableitungen mit weniger als  $n$  Schritten produzieren nur ausgegliche Stringe
- Induktionsschritt:
  - $S \Rightarrow (S)S \Rightarrow^* (x)S \Rightarrow^* (x)y$   
 $(x)y$  ausgeglichen da  $x$  und  $y$  ausgeglichen sind



Compiler

Syntaktische Analyse

$$S \rightarrow (S)S \mid \epsilon$$

generiert alle ausgeglichenen Strings

- Induktion über Länge der Strings
- Basis:  $n = 0$ 
  - $S \Rightarrow \epsilon$  wird abgeleitet
- Induktion: Alle ausgeglichenen Strings  $w$  der Länge  $< 2n$  werden generiert
- Induktionsschritt: Länge von  $w = 2n$ 
  - String muss mit „(“ anfangen
  - Sei  $(x)$  der kürzeste Präfix von  $w$  mit ausgeglichenen Klammern
  - Sei  $w=(x)y$  wobei  $x$  und  $y$  ausgeglichen sein müssen
  - $S \Rightarrow (S)S \Rightarrow^* (x)S \Rightarrow^* (x)y$   
da  $x$  und  $y$  ausgeglichen und kürzer als  $2n$



Compiler

Syntaktische Analyse

# Top-Down/Bottom-up Parsing

- **Top-down:**
  - Man fängt mit dem “Startsymbol” an
  - Man versucht den String abzuleiten
- **Bottom-up:**
  - Man fängt mit dem String an
  - Man versucht das Startsymbol zu erzeugen
  - Produktionen werden von rechts nach links angewandt

$$S \rightarrow (S)S \mid \epsilon$$

$$Ex \rightarrow Nat \mid (Ex) \mid Ex + Ex \mid Ex * Ex$$

Nat + Nat  
Ex + Nat  
Ex + Ex  
Ex



Compiler

Syntaktische Analyse

# Left-to-right/Right-to-left

- Left-to-right                       $\Rightarrow$ 
  - Untersuche den String von links nach rechts
  
- Right-to-left                       $\Leftarrow$ 
  - Man fängt mit dem letzten Symbol an und arbeitet rückwärts
  - Nicht für Compiler verwendet (Effizienz)



# Einfaches Beispiel

Compiler

- $S \rightarrow a S b \mid a S c \mid d$

Syntaktische Analyse

- Strings
  - aadbc
  - aaadbccb
- Welche Regeln müssen angewandt werden?
- Komplexität?



Compiler

# Top-Down Parsing

## Syntaktische Analyse

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow ( E ) \mid id$$

Was ist der Parsbaum für **id+id\*id** ?



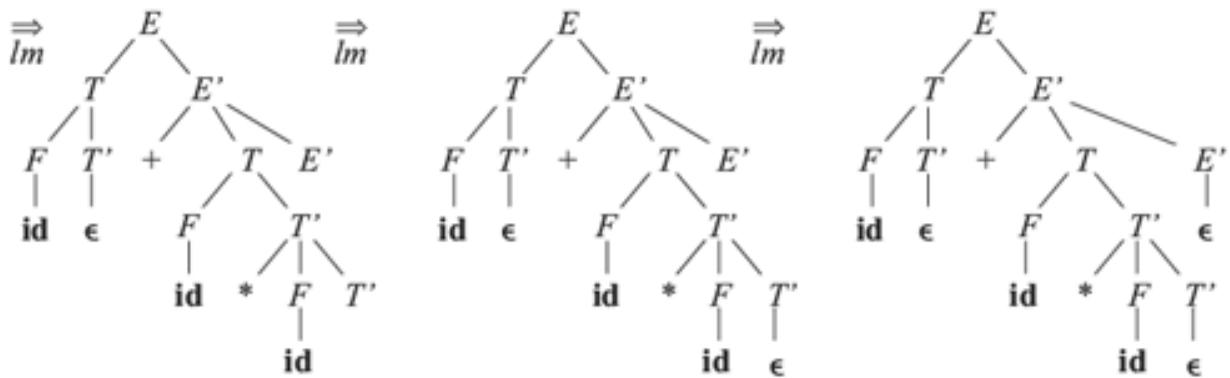
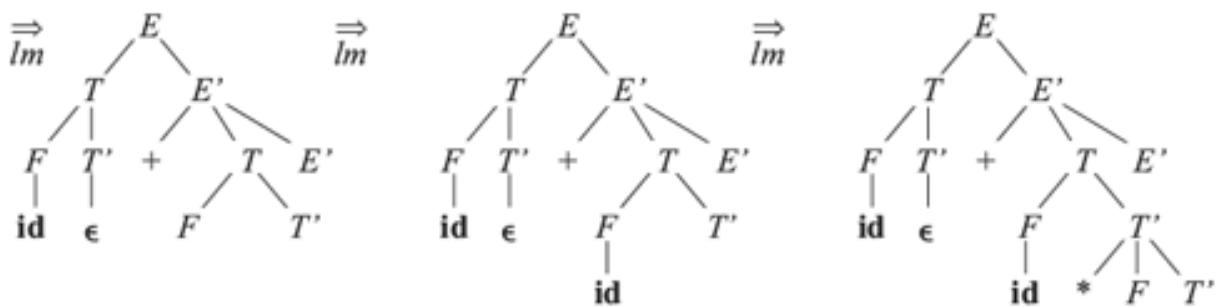
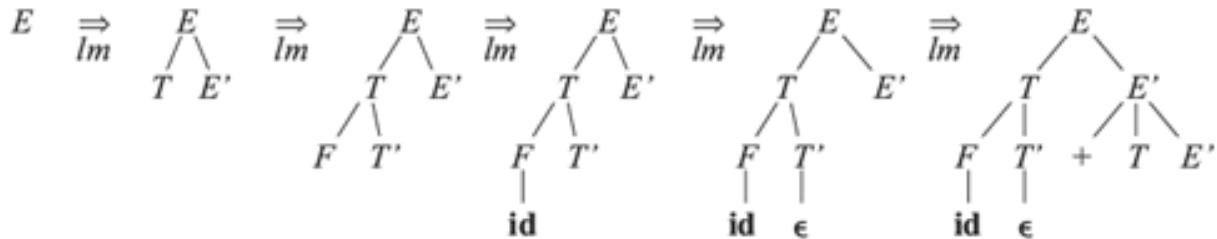
**id+id\*id**

# Top-Down Parsing

$E \rightarrow T E'$   
 $E' \rightarrow + T E' \mid \epsilon$   
 $T \rightarrow F T'$   
 $T' \rightarrow * F T' \mid \epsilon$   
 $F \rightarrow ( E ) \mid id$

## Compiler

## Syntaktische Analyse





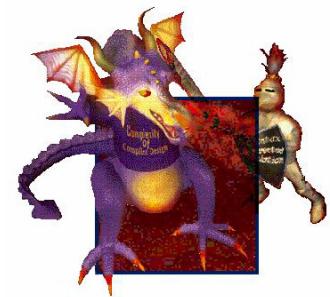
# Rekursiv absteigendes Parsing

Compiler

- 1 Prozedur pro Nichtterminal

Syntaktische Analyse

```
void A() {  
    1)    Wähle eine A-Produktion,  $A \rightarrow X_1 X_2 \dots X_k$ ;  
    2)    for ( i = 1 bis k ) {  
        3)        if (  $X_i$  ist ein Nichtterminal )  
        4)            rufe die Prozedur  $X_i()$  auf;  
        5)        else if (  $X_i$  ist gleich dem aktuellen Eingabesymbol a)  
        6)            setze die Eingabe auf das folgende Symbol;  
        7)        else / mit Backtracking:  
    }  
    }  
    wähle eine andere A-Produktion aus;  
    Fehler falls alle Produktionen probiert
```



# Beispiel

Compiler

- String: “cad”

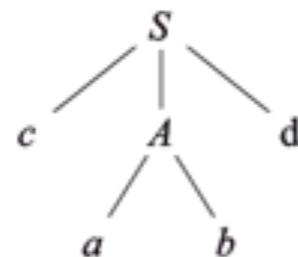
$$S \rightarrow c A d$$

$$A \rightarrow a b \quad | \quad a$$

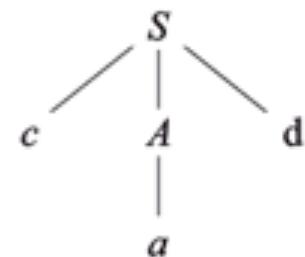
Syntaktische Analyse



a



b



c



Compiler

# Beispiel

- String: “cad”

$$S \rightarrow c A d$$

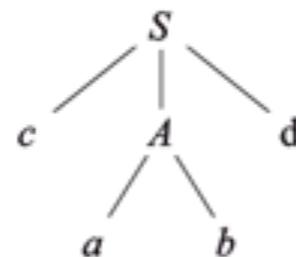
$$A \rightarrow a b \quad | \quad a$$

Syntaktische Analyse

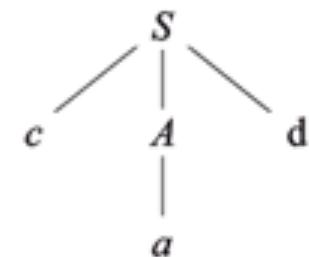
Bei Schritt b ( $A \rightarrow ab$ ) musste Backtracking angewandt werden, da der Baum nicht cad entspricht!



a



b



c



# Prädiktives rekursiv absteigendes Parsing

$\text{Ex} \rightarrow \text{Nat} \mid (\text{Ex}) \mid \text{Ex} + \text{Ex}$

Compiler

Syntaktische Analyse

- Top-down, Linksableitung, left-to-right
- **Voraussage** basierend auf den nächsten k Token: welche Produktion soll verwendet werden
- Funktioniert nur für
  - LL(1),LL(k) Grammatiken
- Einfach per Hand zu schreiben
- Effizient (wenn es klappt):
  - kein Backtracking

$\text{Ex}$   
 $(\text{Ex})$

$(\text{Ex} + \text{Ex})$

( Nat + Nat )



Compiler

Syntaktische Analyse

# Ein kleines Beispiel wo dies funktioniert

- Hier: für jedes Nichtterminal
  - Erstes Symbol der rechten Seite jeweils:  
**eindeutiges Terminal**symbol !

**S** → if E then S else S  
**S** → begin S L  
**S** → print E  
**L** → end  
**L** → ; S L  
**E** → num = num

**S**  
begin S L  
begin print E L  
begin print num = num L  
begin print num = num end

begin print num = num end



# Übersetzung nach Java/C

Compiler

Syntaktische Analyse

- Für jedes Nichtterminal
  - Eine Prozedur mit
  - 1 Switch,  
1 Case pro Produktion

**S** → if E then S else S  
**S** → begin S L  
**S** → print E  
**L** → end  
**L** → ; S L  
**E** → num = num

```
void S() {switch(tok) {  
    case IF: eat(IF) ; E() ;  
        eat(THEN) ; S() ; eat(ELSE) ; S() ; break;  
    case BEGIN: eat(BEGIN) ; S() ; L() ; break;  
    case PRINT: eat(PRINT) ; E() ; break;  
    default: error();  
}
```



# Erweitertes Beispiel

Compiler

Syntaktische Analyse

- Erstes Symbol nicht immer ein **Terminal**
- Prädiktives Parsing trotzdem anwendbar
  - Bestimme die Menge  $\text{FIRST}(\gamma)$ 
    - $\{a \in T \mid \gamma \Rightarrow^* a\alpha\}$
    - $\text{FIRST}(\text{Syscall}) = \{\text{print}, \text{read}, \text{open}, \dots\}$
    - $\text{FIRST}(\text{if } S \text{ then } S \text{ else } S) = \{\text{if}\}$
  - First(.) aller rechten Seiten von Produktionen für das gleiche Nicht-Terminal: sollten disjunkt sein

$S \rightarrow \text{if } S \text{ then } S \text{ else } S$   
 $S \rightarrow \text{begin } S \text{ L}$   
 $S \rightarrow \text{Syscall}$   
 $\text{Syscall} \rightarrow \text{print } E$   
 $\text{Syscall} \rightarrow \text{read } V$   
 $\text{Syscall} \rightarrow \text{open } FD$   
...



Compiler

Syntaktische Analyse

## Beispiel

$$Ex \rightarrow Nat \mid (Ex) \mid Ex + Ex$$

- Berechne  $First(\gamma) = \{a \in T \mid \gamma \Rightarrow^* a\alpha\}$  für alle rechte Seiten
- Sind diese Mengen für jedes Nicht-Terminal paarweise disjunkt?



# Erweitertes Beispiel II

- $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
- $S \rightarrow \text{begin } S \text{ L}$
- $S \rightarrow \text{Syscall}$
- $\text{Syscall} \rightarrow \text{print } E$
- $\text{Syscall} \rightarrow \text{read } V$
- $\text{Syscall} \rightarrow \text{open } FD$
- ...

First(if E then S else S) = {if}  
First(begin S L) = {begin}  
First(Syscall) = {print,read,open}

⇒ C/Java Code für S ?

```
void S() {switch(tok) {
    case IF:   eat(IF); E();
                eat(THEN); S(); eat(ELSE); S(); break;
    case BEGIN: eat(BEGIN); S(); L(); break;
    case PRINT: case READ: case OPEN:
    Syscall(); break;
    default: error();
}
```



# $\epsilon$ -Produktionen: Ein einfaches Beispiel

Compiler

Syntaktische Analyse

- Was passiert hier:
- $T \rightarrow \text{program } S \text{ end}$
- $S \rightarrow \text{if } S \text{ then } S E$
- $S \rightarrow \text{begin } S \text{ end}$
- $S \rightarrow \epsilon$
- $E \rightarrow \text{else } S \text{ fi}$
- $E \rightarrow \text{fi}$
- Wann wendet man Regel 3 für S an ??



Compiler

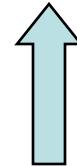
Syntaktische Analyse

# Nullable Symbols

- FIRST reicht nicht aus (bei  $\epsilon$ ):
  - Ableitung
    - FDef
    - Fun ( Arg ) : Type ;
    - **function ID ( Arg ) : Type ;**
  - Welche Regel für Arg ?
    - FIRST(**ID : Type ; Arg**) = {**ID**}, FIRST( $\epsilon$ ) = {}
    - **Muss ein Syntaxfehler ausgegeben werden ??**
    - Nein: **Arg  $\Rightarrow^* \epsilon$  und “)” can auf Arg folgen**

**FDef**  $\rightarrow$  **Fun ( Arg ) : Type ;**  
**Arg**  $\rightarrow$  **ID : Type ; Arg**  
**Arg**  $\rightarrow$   $\epsilon$   
**Type**  $\rightarrow$  **integer**  
**Type**  $\rightarrow$  **real**  
**Type**  $\rightarrow$  **boolean**  
**Fun**  $\rightarrow$  **function ID**

**function ID ( ) : real ;**





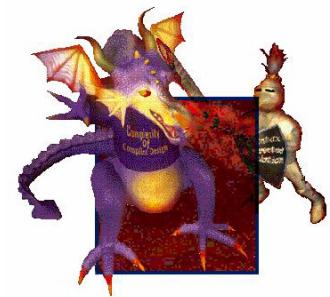
Compiler

Syntaktische Analyse

# Bausteine für prädiktives rekursiv absteigendes Parsing:

- $\text{nullable}(X)$ :
  - wahr falls für Nichtterminal  $X$  gilt:  $X \Rightarrow^* \epsilon$
- $\text{FIRST}(\gamma)$ :
  - Menge der Terminalsymbole mit denen ein von  $\gamma$  abgeleiteter String beginnen kann:
    - $\{a \in T \mid \gamma \Rightarrow^* a\alpha\}$
- $\text{FOLLOW}(X)$ :
  - Menge der Terminalsymbole  $t$  die auf  $X$ ; folgen können:
    - $= \{t \in T \mid S \Rightarrow^* \alpha X t \beta\} \cup \{\$ \mid S \Rightarrow^* \alpha X\}$
    - $\$$  ist immer in  $\text{Follow}(S)$

# First und Follow



Compiler

Syntaktische Analyse

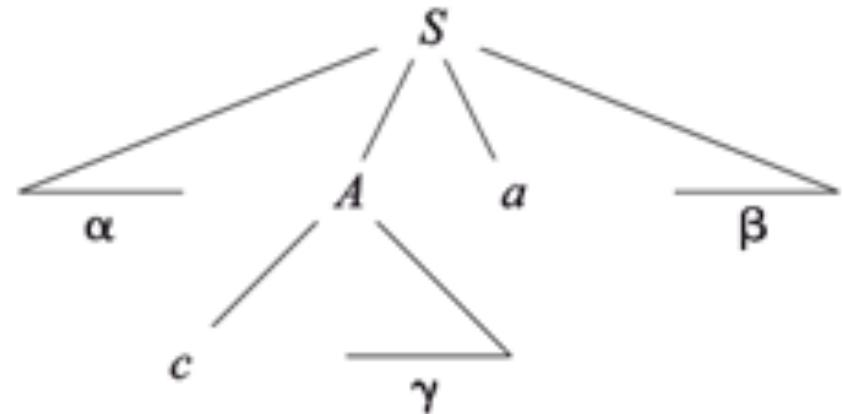


Abbildung 4.15: Terminal  $c$  ist in  $\text{FIRST}(A)$  und  $a$  in  $\text{FOLLOW}(A)$ .

Anmerkung: im Drachenbuch gibt es  
kein nullable, aber  $\epsilon$  kann in FIRST sein:  
nullable(A) entspricht  $\epsilon \in \text{FIRST}(A)$



# LL(1)

Compiler

Syntaktische Analyse

- Grammatik ist LL(1) wenn für alle Produktionen  $A \rightarrow \alpha, A \rightarrow \beta$  gilt:
  - $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
  - $\neg \alpha \Rightarrow^* \epsilon \vee \neg \beta \Rightarrow^* \epsilon$
  - wenn  $\alpha \Rightarrow^* \epsilon$  dann gilt:  
 $\text{FOLLOW}(A) \cap \text{FIRST}(\beta) = \emptyset$
  - wenn  $\beta \Rightarrow^* \epsilon$  dann gilt  
 $\text{FOLLOW}(A) \cap \text{FIRST}(\alpha) = \emptyset$



# Berechnung von nullable(X)

Compiler

**for** all symbols X **do**  
    nullable(X) := **false**;

**repeat**

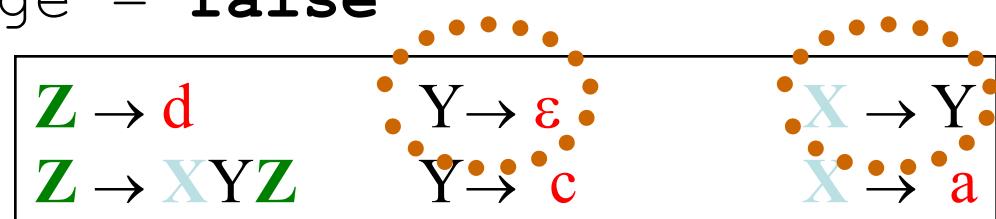
    change := **false**;

**for** every production X  $\rightarrow s_1 \dots s_k$  **do**

**if** nullable(X) = **false** and  
        **if**  $s_1 \dots s_k$  are all nullable **then**  
            nullable(X) := **true**;  
            change := **true**; }

**until** change = **false**

**Wahr für k=0 !**





Compiler

# Berechnung von FIRST

```
for all non-terminals X do FIRST(X) := {};
for all terminals T do FIRST(T) := {T};
```

**repeat**

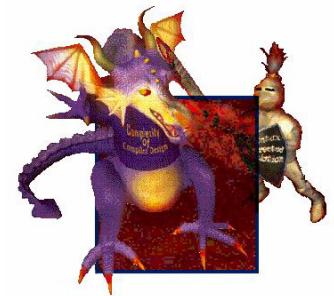
Syntaktische Analyse

```
for every production X → s1 ... sk do
    FIRST(X) := FIRST(X) ∪ FIRST(s1);
    for i := 2 to k do
        if nullable(si-1) then
            FIRST(X) := FIRST(X) ∪ FIRST(si);
        else
            break;
        end if
until no more changes
```

$Y: \{c\}$
$X: \{a,c\}$
$Z: \{a,c,d\}$

Übung:

$Z \rightarrow d$	$Y \rightarrow \varepsilon$	$X \rightarrow Y$
$Z \rightarrow XYZ$	$Y \rightarrow c$	$X \rightarrow a$



Compiler

Syntaktische Analyse

# FIRST( $\gamma$ ) für Strings

- $\text{FIRST}(X\gamma) = \text{FIRST}(X)$   
if not nullable( $X$ )
- $\text{FIRST}(X\gamma) = \text{FIRST}(X) \cup \text{FIRST}(\gamma)$   
if nullable( $X$ )
- Beispiel:  $\text{FIRST}(XYZ) =$ 
  - $\{a,c\} \cup \text{FIRST}(YZ) =$
  - $\{a,c\} \cup \{c\} \cup \text{FIRST}(Z) =$
  - $\{a,c\} \cup \{c\} \cup \{a,c,d\} = \{a,c,d\}$

$Y: \{c\}$   
 $X: \{a,c\}$   
 $Z: \{a,c,d\}$

$Z \rightarrow d$	$Y \rightarrow \epsilon$	$X \rightarrow Y$
$Z \rightarrow XYZ$	$Y \rightarrow c$	$X \rightarrow a$



Compiler

# Berechnung von FOLLOW

FIRST:

Y: {c}

X: {a,c}

Z: {a,c,d}

Y: {a,c,d}

X: {a,c,d}

Z: {\$}

```

for all symbols X do FOLLOW(X) := {};
repeat
  for every production X → s1 ... sk do
    for i := 1 to k do
      FOLLOW(si) := FOLLOW(si) ∪ FIRST(si+1);
    for j := i+2 to k do
      if nullable(si+1) and ... nullable(sj-1) then
        FOLLOW(si) := FOLLOW(si) ∪ FIRST(sj);
      end if
    if i=k or (nullable(si+1) and ... nullable(sk)) then
      FOLLOW(si) := FOLLOW(si) ∪ FOLLOW(X);
    end if
  until no more changes

```

\$ durch Sonderregel  
im Follow des  
Startsymbol

$$Z \rightarrow d$$

$$Z \rightarrow XYZ$$

$$Y \rightarrow \epsilon$$

$$Y \rightarrow c$$

$$X \rightarrow Y$$

$$X \rightarrow a$$



Compiler

Syntaktische Analyse

# Benutzung von nullable,First,Follow: Erstellen eines Parsers

- Erstellen einer **Parsertabelle**:
  - Reihen: Nichtterminale **X**
  - Spalten:
    - Terminalsymbole **c** aus der Eingabe
  - Zellen: Produktion **X**  $\rightarrow$  **γ** anwendbar wenn
    - **c**  $\in$  FIRST(**γ**) oder
    - Nullable(**γ**) und **c**  $\in$  FOLLOW(**X**)
- Übersetzung nach C/Java/....:
  - 1 rekursive Prozedure pro Nichtterminal (siehe vorher)



Compiler

# Unser Beispiel:

FIRST
Y: {c}
X: {a,c}
Z: {a,c,d}

## Prädiktive Parsertabelle:

Syntaktische Analyse

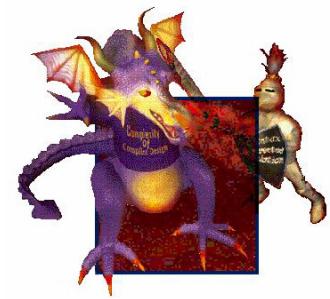
	a	c	d
X	$X \rightarrow a$	$X \rightarrow Y$	$X \rightarrow Y$
Y	$Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$ $Y \rightarrow c$	$Y \rightarrow \epsilon$
Z	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$ $Z \rightarrow d$

FOLLOW
Y: {a,c,d}
X: {a,c,d}
Z: {\$}

Konflikte !!

$Z \rightarrow d$	$Y \rightarrow \epsilon$	$X \rightarrow Y$
$Z \rightarrow XYZ$	$Y \rightarrow c$	$X \rightarrow a$

Hinweis: \$  
Spalte fehlt



# Anderes Beispiel:

Compiler

$$\begin{aligned} E &\rightarrow T \ E' \\ E' &\rightarrow + \ T \ E' \mid \epsilon \\ T &\rightarrow F \ T' \\ T' &\rightarrow * \ F \ T' \mid \epsilon \\ F &\rightarrow ( \ E \ ) \mid \text{id} \end{aligned}$$



Compiler

# Anderes Beispiel:

Nichtterminal	Nullable	First	Follow
E	false	{ id, ( }	{ ), \$ }
E'	true	{ + }	{ ), \$ }
T	false	{ id, ( }	{ +, ), \$ }
T'	true	{ * }	{ +, ), \$ }
F	false	{ id, ( }	{ +, *, ), \$ }

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow ( E ) \mid id$$

Nichtterminal	Eingabesymbol					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Abbildung 4.17: Parsertabelle M zu Beispiel 4.18



Compiler

# Noch ein anderes Beispiel:

Syntaktische Analyse

$$S \rightarrow i \ E \ t \ S \ S' \mid a$$

$$S' \rightarrow e \ S \mid \epsilon$$

$$E \rightarrow b$$

Nichtterminal	Nullable	First	Follow
S	false	{ i, a }	{ e, \$ }
S'	true	{ e }	{ e, \$ }
E	false	{ b }	{ t }

Nichtterminal	Eingabesymbol					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				



# Ein komplettes Beispiel: Parsing von Ausdrücken

Compiler

Syntaktische Analyse

- $E \rightarrow ( E ) EC$
- $E \rightarrow \text{Num } EC$
- $EC \rightarrow + E \mid - E \mid * E \mid / E$  Anmerkung:  
Keine Prioritäten!
- $EC \rightarrow \epsilon$
- $\text{Num} \rightarrow 0 \mid 1 \text{ DigStar} \mid 2 \text{ DigStar} \mid \dots$
- $\text{DigStar} \rightarrow \epsilon$
- $\text{DigStar} \rightarrow 0 \text{ DigStar} \mid 1 \text{ DigStar} \mid \dots$



Compiler

Syntaktische Analyse

# Beispiel: nullable, first, follow

- nullable: EC, DigStar
- $\text{First}(\text{Num}) = \text{First}(\text{DigStar}) = \{0, 1, 2, 3, \dots, 9\}$
- $\text{First}(\text{EC}) = \{+, -, *, /\} \quad \text{First}(\text{E}) = \{(), 0, 1, 2, 3, \dots, 9\}$
- $\text{Follow}(\text{E}) = \{(), \$\} \quad \text{Follow}(\text{EC}) = \{(), \$\}$
- $\text{Follow}(\text{Num}) = \{+, -, *, /, ()\}$
- $\text{Follow}(\text{DigStar}) = \{+, -, *, /, ()\}$

$$E \rightarrow ( E ) EC$$

$$E \rightarrow \text{Num } EC$$

$$EC \rightarrow + E \mid - E \mid * E \mid / E$$

$$EC \rightarrow \epsilon$$

$$\text{Num} \rightarrow 0 \mid 1 \text{ DigStar} \mid 2 \text{ DigStar} \mid \dots$$

$$\text{DigStar} \rightarrow \epsilon$$

$$\text{DigStar} \rightarrow 0 \text{ DigStar} \mid 1 \text{ DigStar} \mid \dots$$



# Übersetzung nach Java: Infrastruktur

Compiler

```
public static void advance() throws java.io.IOException {  
    if(tokasint != -1) { tokasint = System.in.read();  
        tok = (char) tokasint; }  
}
```

Syntaktische Analyse

```
public static void eat(char c) throws java.io.IOException {  
    if(tok==c)  advance();  
    else { System.out.print("*** parser error, expected <");  
        System.out.print(c);  
        System.out.print("> instead of <");  
        if(tokasint== -1)  
            System.out.print("EOF");  
        else  
            System.out.print(tok);  
        System.out.println(">");  
    }
```



# Der resultierende Parser I

Compiler

Syntaktische Analyse

```
public static void Expr() throws java.io.IOException
    {switch(tok) {
        case '(': /* Expr --> ( Expr ) ExprCont */
        eat('('); Expr(); eat(')'); ExprCont();
        break;
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
        /* Expr --> Num ExprCont */
        Num(); ExprCont();
        break;
        default: eat('('); /* generate error message */
    }
}
```



# Der resultierende Parser II

Compiler

Syntaktische Analyse

```
public static void ExprCont() throws java.io.IOException
    {switch(tok) {
        case ')': case (char) (-1):/* eof */
            /* ExprCont --> epsilon */ break;
        case '+': case '*': case '/': case '-':
            /* ExprCont --> (+|*|/-) Expr */
            advance(); Expr();
            break;
        default: eat(')'), /* generate error message */
    }
}
```

Remember:  $\text{Follow}(\text{EC}) = \{ , \$ \}$ ,  $\text{nullable}(\text{EC}) !!$



Compiler

Syntaktische Analyse

# Der resultierende Parser III

```
public static void Num() throws java.io.IOException
    {switch(tok) {
        case '0': advance(); break; /* Num --> 0 */
        case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            /* Num --> (1|2|3|4|5|6|7|8|9) DigStar */
            advance(); DigStar(); break;
        default: eat('0'); /* generate error message */

    }
}
```



Compiler

Syntaktische Analyse

# Der resultierende Parser IV

```
public static void DigStar() throws java.io.IOException
    {switch(tok) {
        case ')': case '+': case '*': case '/': case '-':
        case (char) (-1): /* eof */
            break; /* DigStar --> epsilon */
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            /* DigStar --> (1|2|3|4|5|6|7|8|9) DigStar */
            advance(); DigStar(); break;
        default: eat('0'); /* generate error message */
    }}
```

Remember:  $\text{Follow}(\text{Digstart}) = \{ , +, -, *, / \}$ ,  
 $\text{nullable}(\text{DigStar}) !!$



Compiler

Syntaktische Analyse

# Der resultierende Parser IV

```
public static void main(String[] args)
throws java.io.IOException {
    System.out.println("LL(1) Parser");
    System.out.println("-----");
    advance();
    System.out.println("Starting parse process...");
Expr();
    System.out.println("Finished.");
}
```

**und nun eine Vorführung !**  
(Quellcode im der Ilias erhältlich)



# Probleme für das rekursiv absteigende Parsing

# Compiler

# Syntaktische Analyse



Compiler

Syntaktische Analyse

## Linksfaktorisierung (4.3.4)

- Wiederhole bis keine Änderung mehr:
  - Für jedes Nicht-Terminal X:
    - Finde den längsten gemeinsamen Präfix  $\alpha$  von zwei oder mehr Alternativen
    - Falls  $\alpha \neq \epsilon$ , dann ersetze

$$X \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_k \mid \gamma_1 \mid \dots \mid \gamma_n$$

$$\begin{aligned} X &\rightarrow \alpha X' \mid \gamma_1 \mid \dots \mid \gamma_n \\ X' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_k \end{aligned}$$



# Linksfaktorisierung (Alg. 4.10)

Compiler

**Eingabe:** Eine Grammatik  $G$

**Ausgabe:** Eine äquivalente linksfaktorierte Grammatik

**Methode:** Für jedes Nichtterminal  $A$  suchen wir das längste Präfix  $\alpha$ , das zwei oder mehr seiner Alternativen gemeinsam haben. Ist  $\alpha \neq \epsilon$  – gibt es also ein nicht triviales gemeinsames Präfix –, ersetzen wir alle  $A$ -Produktionen  $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma$  durch Folgendes, wobei  $\gamma$  für alle Alternativen steht, die nicht mit  $\alpha$  beginnen:

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

Hier ist  $A'$  ein neues Nichtterminal. Diese Transformation wird wiederholt angewendet, bis es keine zwei Alternativen für ein Nichtterminal mehr gibt, die ein gemeinsames Präfix haben. □



# Links faktorisierung 2

Compiler

- $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
- $S \rightarrow \text{if } E \text{ then } S$

Syntaktische Analyse

**Nach Linksfaktorisierung:**

$S \rightarrow \text{if } E \text{ then } S \ R$

$R \rightarrow \text{else } S$

$R \rightarrow \varepsilon$

*Immer noch mehrdeutig  
Lösung?*



# Dangling Else Problem

Compiler

- $S \rightarrow$ 
  - Matched |
  - Unmatched
- Matched  $\rightarrow$ 
  - if Expr then** Matched **else** Matched |
  - OtherStmt
- Unmatched  $\rightarrow$ 
  - if Expr then** S |
  - if Expr then** Matched **else** Unmatched



Compiler

Syntaktische Analyse

# Eliminieren der Linksrekursion

- Umschreiben nach Rechtsrekursion:
  - $E \rightarrow E + T$
  - $E \rightarrow T$
  - $T \rightarrow id \mid Num \mid (E)$

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \\ E' &\rightarrow \epsilon \\ T &\rightarrow id \mid Num \mid (E) \end{aligned}$$

- $X \rightarrow X\gamma_1$
- $X \rightarrow X\gamma_2$
- $X \rightarrow \alpha_1$
- $X \rightarrow \alpha_2$

$$\begin{aligned} X &\rightarrow \alpha_1 X' \\ X &\rightarrow \alpha_2 X' \\ X' &\rightarrow \gamma_1 X' \\ X' &\rightarrow \gamma_2 X' \\ X' &\rightarrow \epsilon \end{aligned}$$



# Algorithmus: Eliminieren der Linksrekursion

Compiler

Syntaktische Analyse

- 1) Bringe die Nichtterminale in eine Reihenfolge  $A_1, A_2, \dots, A_n$ .
- 2) **for** ( jedes  $i$  von 1 bis  $n$  ) {
- 3)     **for** ( jedes  $j$  von 1 bis  $i - 1$  ) {
- 4)         ersetze jede Produktion der Form  $A_i \rightarrow A_j \gamma$  durch die Produktionen  
 $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ , wobei  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$   
alle aktuellen  $A_j$ -Produktionen sind
- 5)     }
- 6)     eliminiere die unmittelbare Linksrekursion unter den  $A_i$ -Produktionen
- 7) }



Compiler

# Mehrdeutige Grammatiken

$$Ex \rightarrow \text{Nat} \mid \text{ID} \mid (\text{Ex}) \mid \text{Ex} + \text{Ex} \mid \text{Ex} * \text{Ex}$$

Syntaktische Analyse

- Lösung 1: Parser “hacken”
  - Prioritäten/Konflikte im Parser Explizit behandeln; nicht empfohlen
- Lösung 2: Grammatik umschreiben
  - Verlangt Nachdenken (siehe “dangling else” Problem von vorher)
  - Nicht immer möglich

$$Ex \rightarrow \text{Ex} + \text{Term} \mid \text{Term}$$
$$\text{Term} \rightarrow \text{Term} * \text{Factor} \mid \text{Factor}$$
$$\text{Factor} \rightarrow \text{Nat} \mid \text{ID} \mid (\text{Ex})$$



Compiler

Syntaktische Analyse

# Mehrdeutige Grammatiken

- Operator Präzedenzen in Grammatik kodieren:
  - Ebene1 niedrigste Priorität

$$\begin{aligned}\textbf{Ebene}_1 &\rightarrow \textbf{Ebene}_1 \text{ OP}_1 \textbf{Ebene}_2 \mid \textbf{Ebene}_2 \\ \textbf{Ebene}_2 &\rightarrow \textbf{Ebene}_2 \text{ OP}_2 \textbf{Ebene}_3 \mid \textbf{Ebene}_3 \\ &\dots \\ \textbf{Ebene}_k &\rightarrow \textbf{Ebene}_k \text{ OP}_k \textbf{Factor} \mid \textbf{Factor} \\ \textbf{Factor} &\rightarrow \text{Nat} \mid \text{ID} \mid \dots \mid (\textbf{Ebene1})\end{aligned}$$



# Fehlerbehandlung

Compiler

Syntaktische Analyse

- Exception generieren
- Insertion (Einfügung)
  - e.g., print error, pretend everything ok
- Deletion (Lösung)
  - e.g., skip until a token in FOLLOW is reached

```
void S() {switch(tok) {  
    case IF: eat(IF) ; E() ;  
        eat(THEN) ; S() ; eat(ELSE) ; S() ; break;  
    case BEGIN: eat(BEGIN) ; S() ; L() ; break;  
    case PRINT: eat(PRINT) ; E() ; break;  
    default: error_recovery ; } }
```



# Grammatiken in JavaCC

Compiler

```
options {
    IGNORE_CASE = true;
}
PARSER_BEGIN(MyParser)
class MyParser {
    public static void main(String args[])
        throws ParseException {
    MyParser parser = new MyParser(System.in);
    parser.Start();
}
PARSER_END(MyParser)
```

Syntaktische Analyse

```
TOKEN: {
    < IF: "if" >
    | < #DIGIT: ["0"-"9"]>
    | <ID: ["a"-"z"] ([ "a"-"z" ] | <DIGIT>) * >
    | <NUM: (<DIGIT>) + >
    | <REAL: ( (<DIGIT>) + "." (<DIGIT>) * ) | (( <DIGIT>) * "." (<DIGIT>) + ) >
}
... Parser Grammar Rules
```

Start Symbol der Grammatik



# JavaCC Grammars 2/2

## Compiler

```
void Start() :  
{  
    /* LOCAL VARIABLE DECLARATIONS ... */  
}  
{  NonTerminal() <TOKEN> ... }  
  
void NonTerminal() :  
{  
    /* LOCAL VARIABLE DECLARATIONS ... */  
}  
{  "+" NonTerminal() | ... }    <-- regular expression constructs can be  
                                used
```

## Syntaktische Analyse



# Sehr einfaches Beispiel

Compiler

```
PARSER_BEGIN(Simple3)
public class Simple3 {
    public static void main(String args[]) throws ParseException {
        Simple3 parser = new Simple3(System.in);
        parser.Input();
    }
}
PARSER_END(Simple3)
SKIP :
{ " " | "\t" | "\n" | "\r" }
TOKEN :{ <LBRACE: "{">} <RBRACE: "}">}
void Input() :
{ int count; }
{ count=MatchedBraces() <EOF>
{ System.out.println("The levels of nesting is " + count); }
}
int MatchedBraces() :
{ int nested_count=0; }
{ [<LBRACE> nested_count=MatchedBraces() {nested_count++;} <RBRACE>]
{ return nested_count; }
}
```



Compiler

Syntaktische Analyse

# Zusammenfassung

- Recursive Descent Parsing:
  - **Prinzipien, Einschränkungen**
  - **Berechnung von nullable, First, Follow**
  - **Einen Parser per Hand schreiben**
  - **Was ist LL(1)?**
  - **Probleme: wie kann man diese lösen  
(siehe auch Buch!)**