



Compiler

Syntaktische  
Analyse

# Kapitel 3a

## Syntaktische Analyse

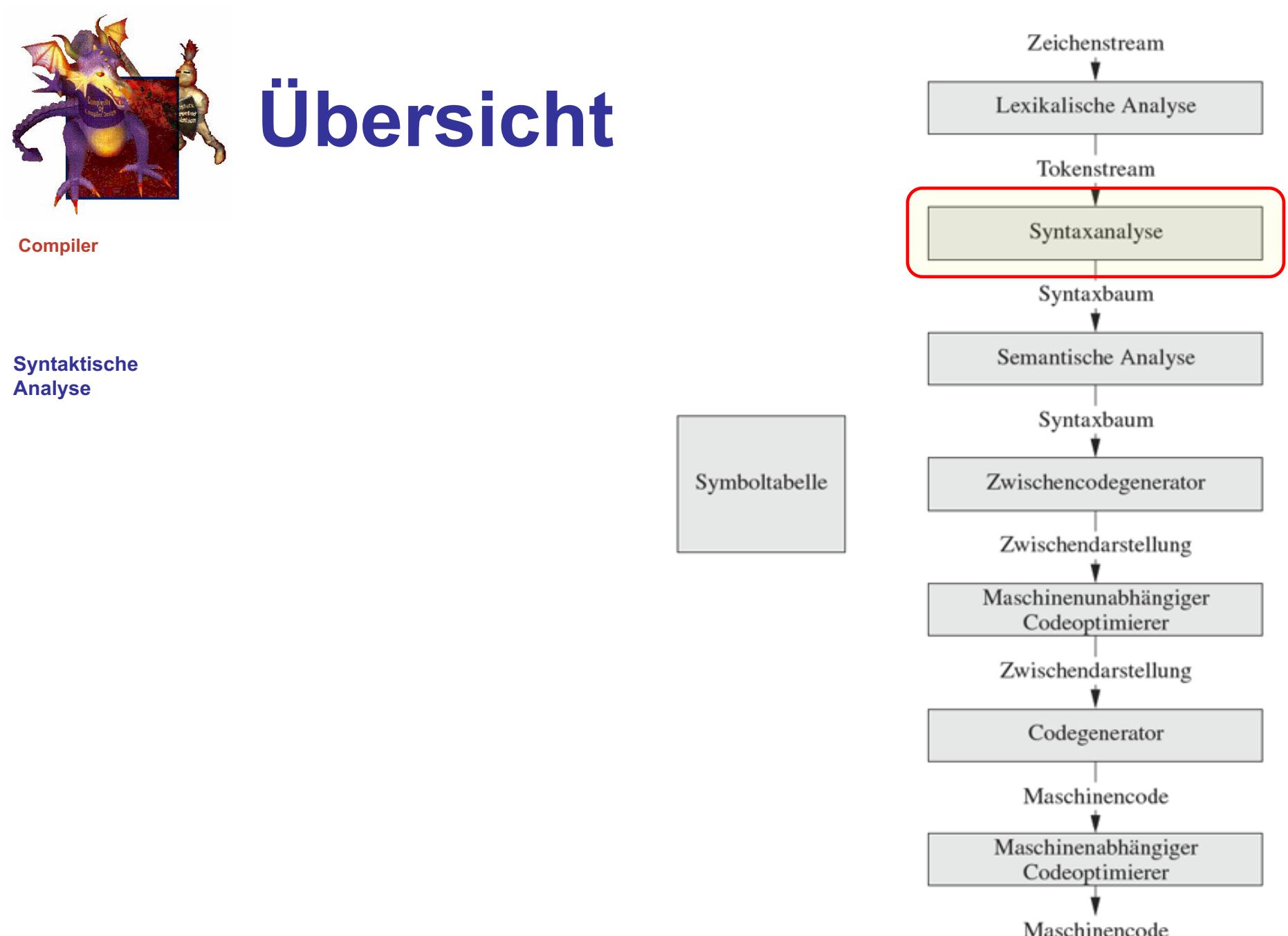


Abbildung 1.6: Phasen eines Compilers

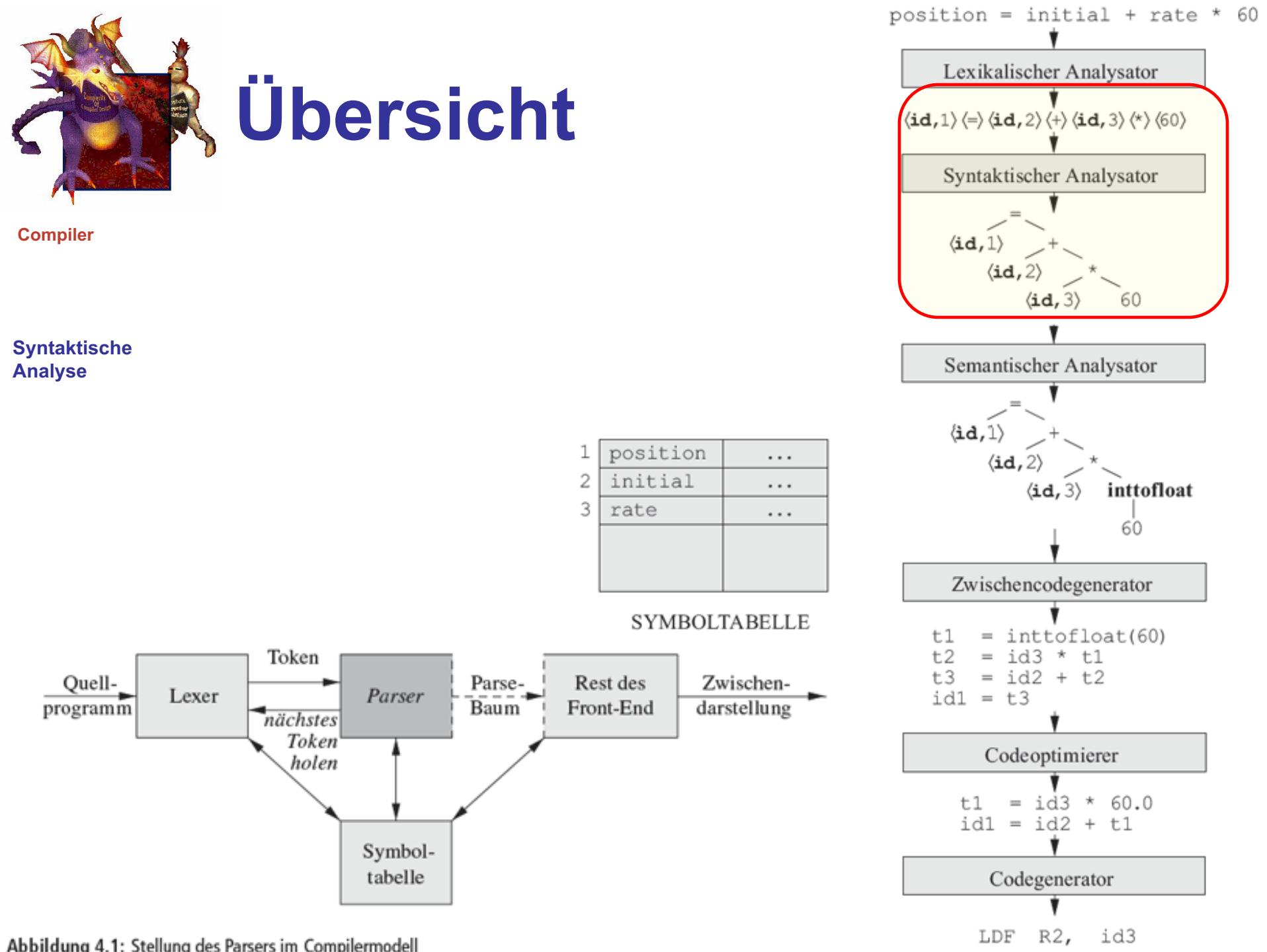


Abbildung 4.1: Stellung des Parsers im Compilermodell



Compiler

Syntaktische  
Analyse

# Syntax: Definition

syn-tax: the way in which words are put together to form phrases, clauses, or sentences.

– *Webster's Dictionary*

Die Syntax ([griechisch σύνταξις](#) ['syntaksis] - die Zusammenstellung) behandelt die Muster und [Regeln](#), nach denen [Wörter](#) zu größeren funktionellen Einheiten wie [Phrasen](#) (Teilsätze) und [Sätzen](#) zusammengestellt und Beziehungen wie Teil-Ganzes, Abhängigkeit etc. zwischen diesen formuliert werden (Satzbau).

– *Wikipedia*



# Natürliche Sprache

Compiler

Syntaktische  
Analyse

this is some text without spaces and punctuation marks which is therefore quite difficult to read by humans lexical analysis will break this text up into words while the parsing phase will extract the grammatical structure of the text

this is some text without spaces and punctuation marks which is therefore quite difficult to read by humans lexical analysis will break this text up into words while the parsing phase will extract the grammatical structure of the text

This **is** some **text** without **spaces** and **punctuation-marks** which **is** therefore quite difficult to **read** by **humans**.

**Lexical-analysis** **will break** this text up into **words** while the **parsing-phase** **will extract** the **grammatical structure** of the **text**.



# Mehrdeutigkeit / Analyse

Compiler

- The boy saw the man with the telescope.

Syntaktische  
Analyse



# Mehrdeutigkeit / Analyse

Compiler

- The boy saw the man with the telescope.
- Subjekt Prädikat Objekt
- The boy saw the man with the telescope.
- Subjekt Prädikat Objekt Adverbial
- The boy saw the man with the telescope
- Wie kann man die Interpretationen darstellen und unterscheiden ?

Syntaktische  
Analyse



Compiler

Syntaktische  
Analyse

# Wie soll die Syntax einer Programmiersprache beschrieben werden?

- Beispiel:
  - If – Then – Else Anweisung



# Deutsche Grammatik

Compiler

Syntaktische  
Analyse

**1. Satzbauplan – Hauptsatz** Der 1. Satzbauplan hat die Reihenfolge:

**Subjekt – finitives Prädikat – indirektes Objekt  
– direktes Objekt – Adverbien –  
Prädikatrest**

Die Satzverneinung steht vor dem Prädikatrest.

Beispiel:

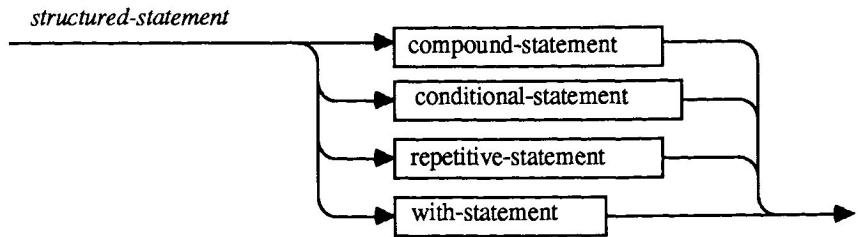
**„der Verkäufer – hatte – seinem Kunden – das  
Buch – gestern – in seinem Laden – (nicht)  
– gegeben.“**

Quelle: Wikipedia.de

# Sample: Part of Pascal Syntax

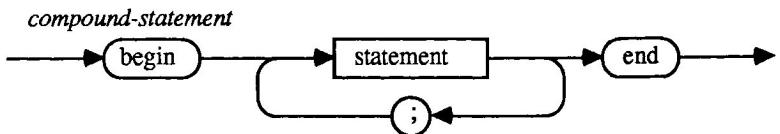
## 6.2 Structured-Statements

Structured-statements are made up of other statements that are to be executed either conditionally (conditional-statements), repeatedly (repetitive-statements), or in sequence (compound-statement or with-statement).



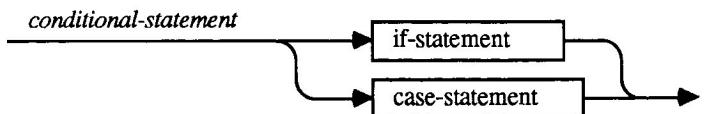
### 6.2.1 Compound-Statement

The compound-statement specifies the execution of its statement-sequence in its textual order.

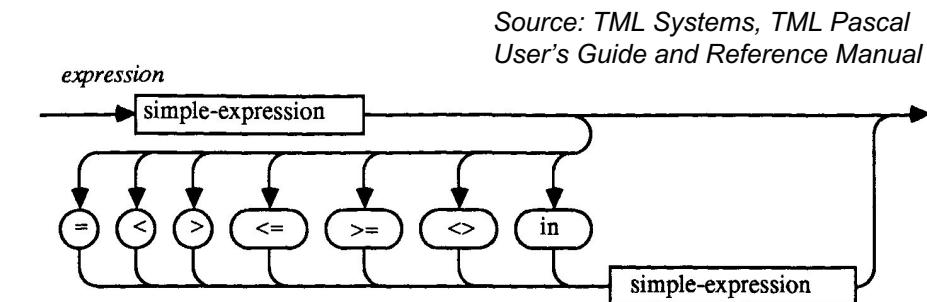
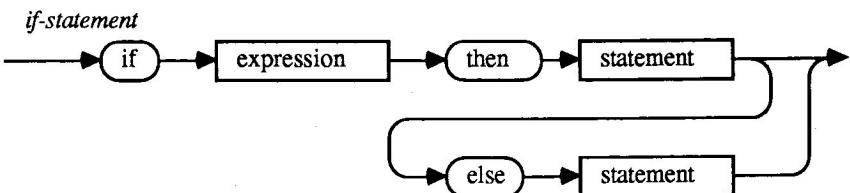


### 6.2.2 Conditional-Statements

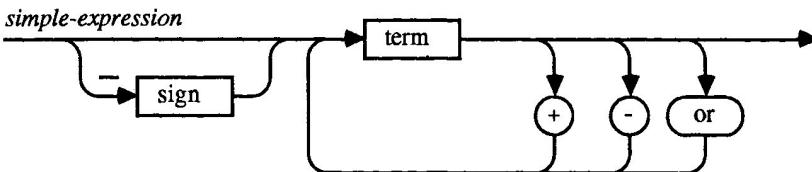
A conditional-statement selects one or none of its component statements for execution.



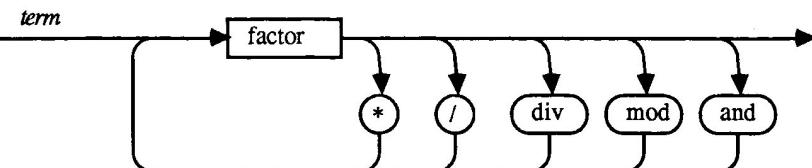
### 6.2.2.1 If-Statement



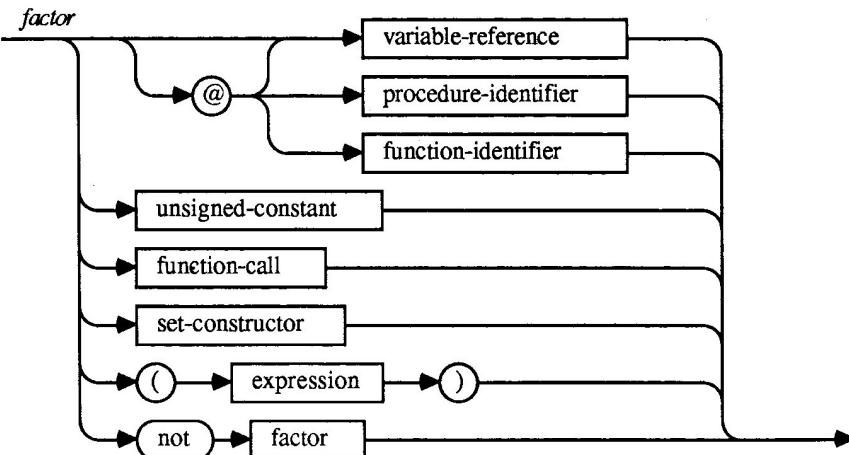
The syntax for a simple-expression allows the *adding* operators and signs to be applied to terms:



The syntax for a term allows the *multiplying* operators to be applied to factors:



The syntax for a factor is as follows:



Source: TML Systems, TML Pascal User's Guide and Reference Manual

*hexadecimal-constant* : (LEXICAL)

*hex-marker*

*hexadecimal-constant hex-digit*

*identifier* : (LEXICAL)

*underscore*

*letter*

*identifier following-character*

*if-else-statement* :

*if ( expression ) statement else statement*

*if-statement* :

*if ( expression ) statement*

*indirect-component-selection* :

*postfix-expression -> name*

*indirection-expression* :

*\* unary-expression*

*initialized-declaration* :

*declaration-specifiers initialized-declarator-list ;*

*initialized-declarator* :

*declarator initializer-part<sub>opt</sub>*

*initialized-declarator-list* :

*initialized-declarator*

*initialized-declarator-list , initialized-declarator*

*initializer* :

*expression*

*{ initializer-list , opt }*

*initializer-list* :

*initializer*

*initializer-list , initializer*

## The if Statement

```
if test:  
    suite  
[elif test:  
    suite]*  
[else:  
    suite]
```

The `if` statement selects from among one or more actions (statement blocks), and it runs the suite associated with the first `if` or `elif` test that is true, or the `else` suite if all are false.

## The while Statement

```
while test:  
    suite  
[else:  
    suite]
```

The `while` loop is a general loop that keeps running the first suite while the test at the top is true. It runs the `else` suite if the loop exits without hitting a `break` statement.



# Syntax Analyse: Übersicht

Compiler

Syntaktische  
Analyse

- Beschreibung der syntaktischen Struktur von Programmen
  - kontext freie Grammatiken (kfG)
  - beschreiben: Zuweisungen, Tests, ..., Programme
- Erkennen der syntaktischen Struktur (“parsing”)
  - Top-Down parsing (LL(1))
  - Bottom-Up Shift/reduce parsing (LR(1))
- yacc/sablecc/... Werkzeuge



# Warum studieren wir Parsing?

Compiler

- Essenziell für die semantische Analyse
- Viele andere Anwendungen:
  - Natural Language Understanding
  - theoretische Informatik
  - ...

Syntaktische  
Analyse



Compiler

Syntaktische  
Analyse

# Wiederholung: Formale Sprachen

- **Alphabet**  $\Sigma$ : endliche Menge von Symbolen
  - $\{0,1\}$ ,  $\{a,b,c,\dots,z\}$ , Ascii, ...
- **String** (Wort): **endliche Folge** von Symbolen  $\in \Sigma$ 
  - 101, helloworld, if a=0 then a:= b
- **Sprache** = **abzählbare Menge** von Strings
  - Primzahlen im Binärformat, English, Java Programme
  - Lexeme für ein Token !!  
**Identifier** :  $\{a,b,\dots,foo,\dots,x\_3,\dots\}$

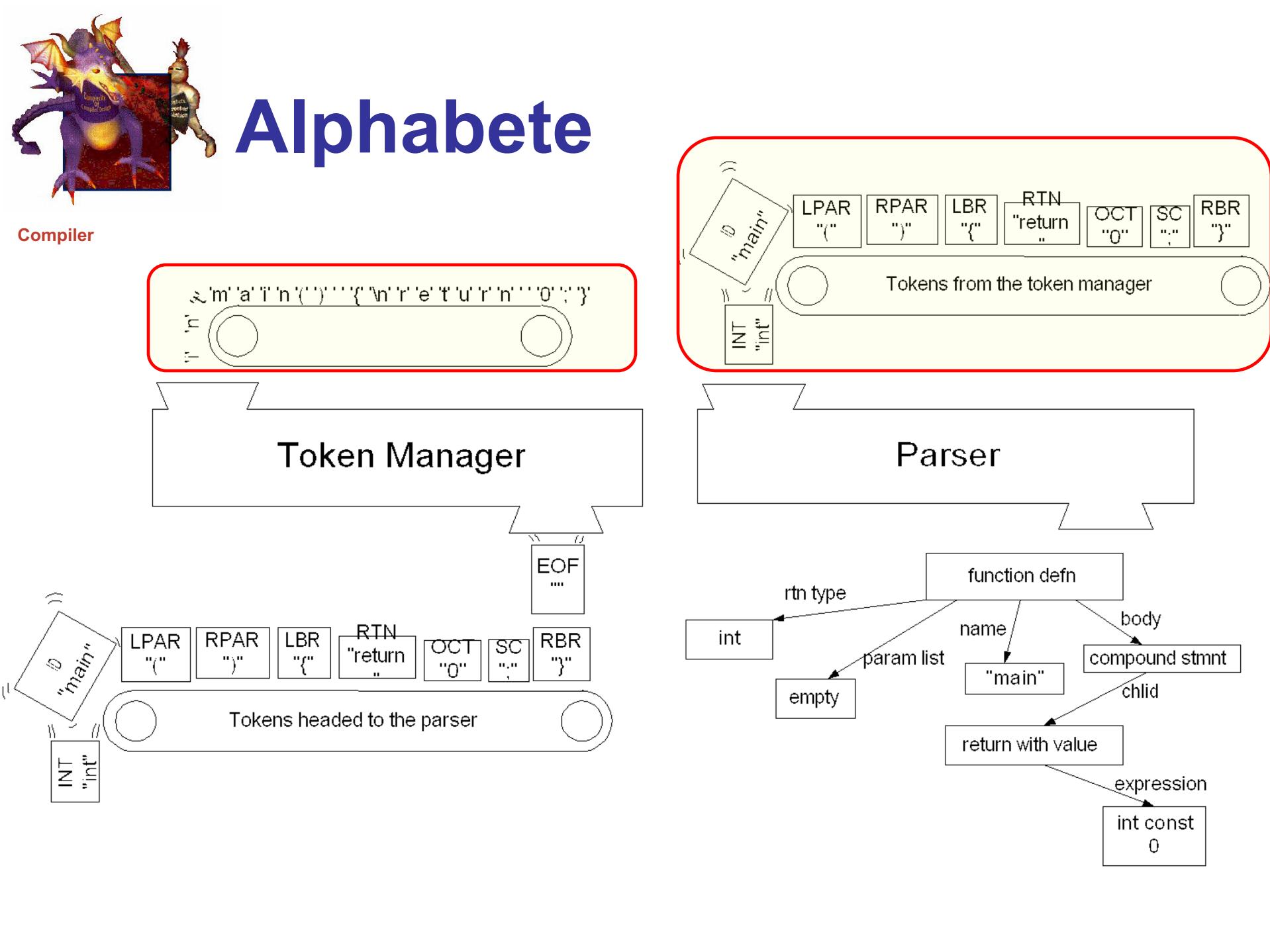


Compiler

Syntaktische  
Analyse

# Anwendung der formalen Sprachen

- Beim Lexing:
  - Zuordnung zu **Token Klassen**
    - $L(\text{num}) = \{\text{"0"}, \text{"1"}, \text{"2"}, \text{"3"}, \dots, \text{"10"}, \text{"11"}, \dots\}$
    - $L(\text{id}) = \{\text{"a"}, \text{"b"}, \dots, \text{"a1"}, \text{"a2"}, \dots, \text{"aa"}, \text{"ab"}, \dots\}$
    - $\Sigma = \text{Ascii oder Unicode}$
- Beim Parsing:
  - Zuordnung zu **grammatischen Konstrukten**
    - $L(\text{assignment}) = \{\text{"id} = \text{id"}, \text{"id} = \text{num"}, \text{"id} = \text{id+id"}, \text{"id} = \text{id+num"}, \text{"id} = \text{num+id"}, \dots\}$
    - $\Sigma = \text{Token (oder Ascii/Unicode wenn kein Lexing)}$





Compiler

Syntaktische  
Analyse

# Beschreibung der Sprachen I

```
if n == ((x+2) *3) then
    return 0
else
    ....
```

## Aufzählung

- Beispiel: Arithmetische Ausdrücke über natürliche Zahlen mit **+, \*, (, )**:

{0,1,2,3,...,0+0,0+1,0+2,...,1+0,1+1,1+2,...,  
0\*0, 0\*1, 0\*2,..., 0+0+0, 0+0+1,...  
0+0\*0, 0+0\*1, 0+0\*2,...  
0\*0+0, 0\*0+1,..., 1\*0+0, 1\*0+1,...  
(0+0),(0+1),(0+2),...,(1+0),(1+1),(1+2),...,  
(0\*0), (0\*1), (0\*2),..., (0+0)+0, (0+0)+1,...}

Können wir reguläre Ausdrücke verwenden ?



Compiler

Syntaktische  
Analyse

# Beschreibung der Sprachen II

Reguläre Ausdrücke:  $a, \epsilon, M|N, MN, M^*$

- Beispiel: Arithmetische Ausdrücke über natürliche Zahlen mit  $+, *, (, )$ :
- $\text{Nat} = 0 \mid [1-9][0-9]^*$   $\{0,1,2,3,\dots\}$
- $\text{Ex} = \text{Nat} ((“+” \mid “ *”) \text{Nat})^*$   $\{0,\dots,0+0,\dots,0^*0,\dots\}$
- Mit Klammern ?



Compiler

Syntaktische  
Analyse

# Quiz: Ein einfacheres Problem (or *Who wants to be a millionaire?*)

- Beschreiben Sie die Sprache  
 $\{\text{“ab”}, \text{“aabb”}, \text{“aaabbb”}, \dots\}$  mit einem regulären Ausdrück
  - **Unmöglich!**
  - Das Gleiche gilt für:
    - Korrekt geklammerte Ausdrücke  $((\dots)\dots)$
    - Korrekt geschachtelte Schleifen,...
- ⇒ Reguläre Ausdrücke reichen zum Parsen nicht aus



# Erklärung

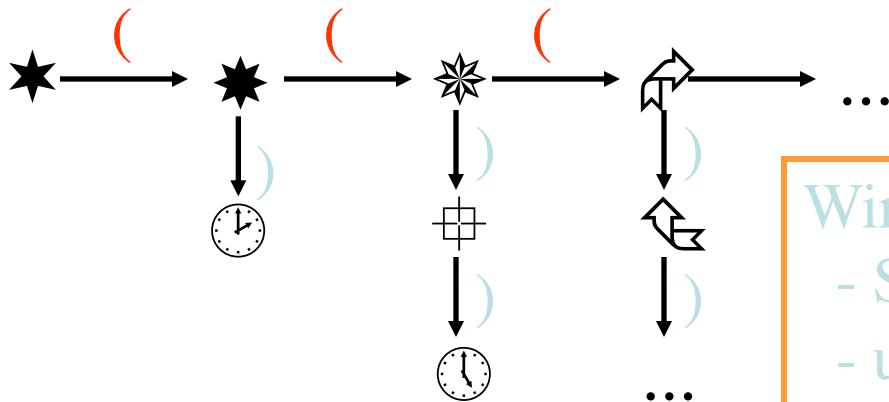
Compiler

Syntaktische  
Analyse

Beschreibung von  $\{“(())”, “((()))”, “(((())”}, \dots\}$   
mit einem regulären Ausdruck:  
**unmöglich!**

Warum:

- Regulärer Ausdruck  $\rightarrow$  endlicher DFA
- DFA: Zustände haben keinen Speicher



Wir brauchen  
- Speicher oder  
- unendlich viele Zustände



# Klammern Problem

Compiler

- $x := (a+b)*2;$  ✓
- $x := (a+b*2;$  ✗

Syntaktische  
Analyse

- if  $x > 2$  then return  $x*x$  else return 0; ✓
- if  $x > 2$  then return  $x*x$  if  $y > 2;$  ✗
- else return 0 then return  $x*x;$  ✗

⇒ kontextfreie Grammatiken (Rekursion)



# Bausteine von kontextfreien Grammatiken

Compiler

Syntaktische  
Analyse

**N** = Menge von **Nichtterminalen**

**T** = Menge von **Terminalen**

**Startsymbol** **S**  $\in$  **N**

Menge P von **Produktionen** der Form:

- $s_0 \rightarrow s_1 \dots s_k$  mit  
 $s_0 \in N, s_i \in T \cup N, k \geq 0$

Notation:

- $s_0 \rightarrow \alpha_1 \mid \dots \mid \alpha_k$  bezeichnet  $\{s_0 \rightarrow \alpha_i \mid 1 \leq i \leq k\}$



# Beschreibung der Sprachen IV

Compiler

Syntaktische  
Analyse

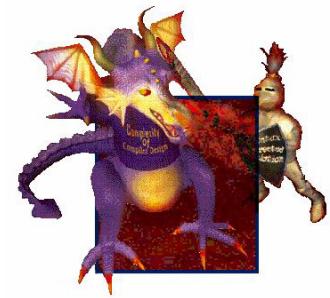
Beispiel: Arithmetische Ausdrücke über  
Bezeichner **id** mit **+, \*, (, )**:

Version1: N={E}, S=E, T = {**id, +, \*, (, )**}

$$E \rightarrow E + E \mid E * E \mid ( E ) \mid id$$

Version 2: N={E,T,F}, S=E, T = {**id, +, \*, (, )**}

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow ( E ) \mid id \end{aligned}$$



Compiler

Syntaktische  
Analyse

# Wo sind die Grammatiken?

*hexadecimal-constant : (LEXICAL)*  
*hex-marker*  
*hexadecimal-constant hex-digit*

*identifier : (LEXICAL)*  
*underscore*  
*letter*  
*identifier following-character*

*if-else-statement :*  
*if ( expression ) statement else statement*

*if-statement :*  
*if ( expression ) statement*

*indirect-component-selection :*  
*postfix-expression -> name*

*indirection-expression :*  
*\* unary-expression*

*initialized-declaration :*  
*declaration-specifiers initialized-declarator-list ;*

*initialized-declarator :*  
*declarator initializer-part<sub>opt</sub>*

*initialized-declarator-list :*  
*initialized-declarator*  
*initialized-declarator-list , initialized-declarator*

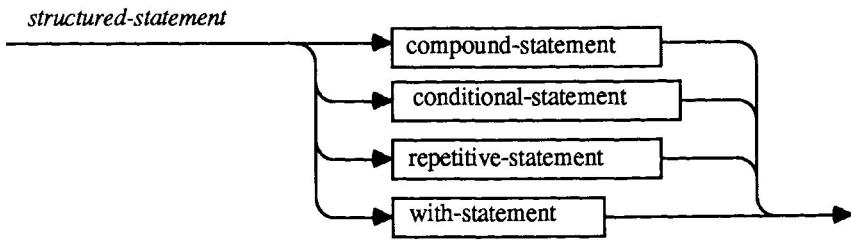
*initializer :*  
*expression*  
*{ initializer-list , opt }*

*initializer-list :*  
*initializer*  
*initializer-list , initializer*

# Grammatiken: wo?

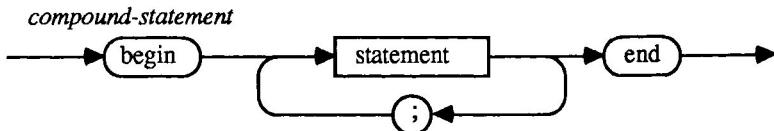
## 6.2 Structured-Statements

Structured-statements are made up of other statements that are to be executed either conditionally (conditional-statements), repeatedly (repetitive-statements), or in sequence (compound-statement or with-statement).



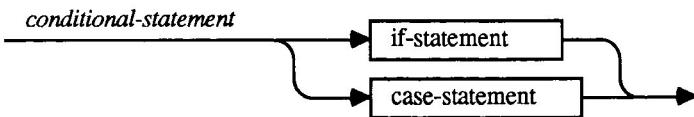
### 6.2.1 Compound-Statement

The compound-statement specifies the execution of its statement-sequence in its textual order.

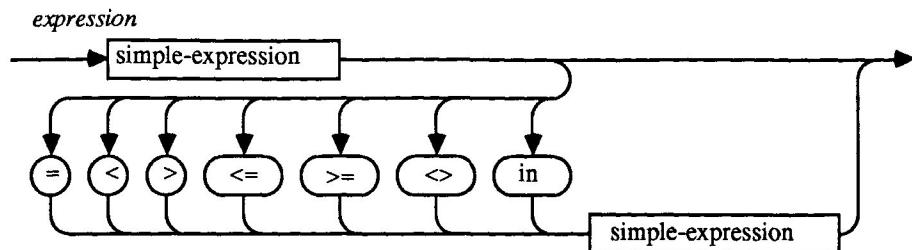
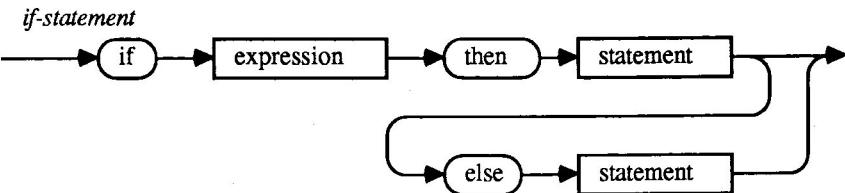


### 6.2.2 Conditional-Statements

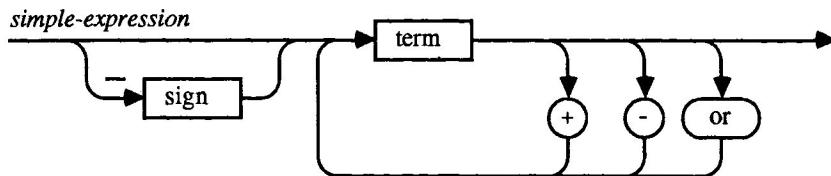
A conditional-statement selects one or none of its component statements for execution.



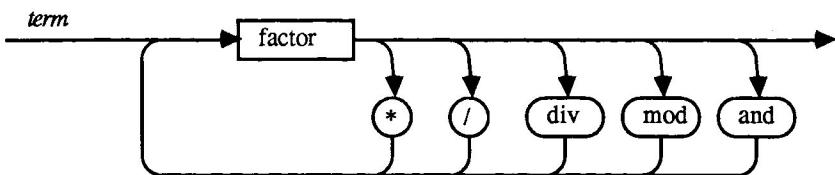
#### 6.2.2.1 If-Statement



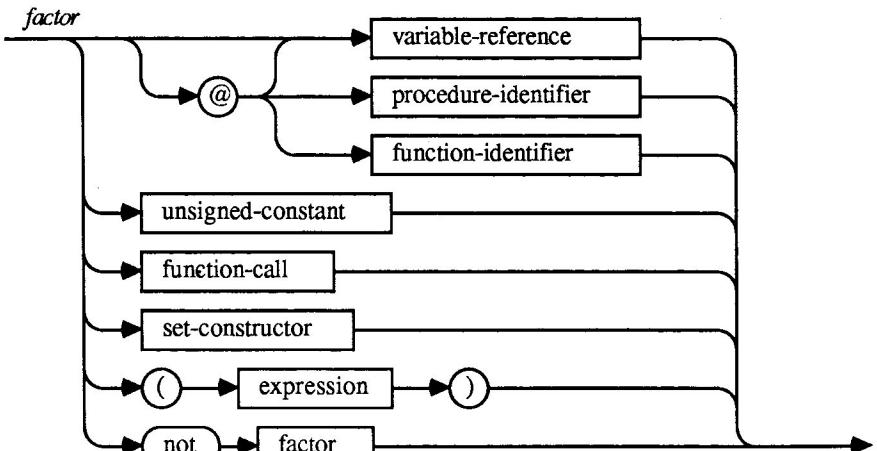
The syntax for a simple-expression allows the *adding* operators and signs to be applied to terms:



The syntax for a term allows the *multiplying* operators to be applied to factors:



The syntax for a factor is as follows:





Compiler

Syntaktische  
Analyse

# Wo sind die Grammatiken?

## The if Statement

```
if test:  
    suite  
[elif test:  
    suite]*  
[else:  
    suite]
```

The if statement selects from among one or more actions (statement blocks), and it runs the suite associated with the first if or elif test that is true, or the else suite if all are false.

## The while Statement

```
while test:  
    suite  
[else:  
    suite]
```

The while loop is a general loop that keeps running the first suite while the test at the top is true. It runs the else suite if the loop exits without hitting a break statement.



Compiler

# Wo ist die kfG? Deutsche Grammatik

1. **Satzbauplan – Hauptsatz** Der 1. Satzbauplan hat die Reihenfolge:

**Subjekt – finitives Prädikat – indirektes Objekt  
– direktes Objekt – Adverbien –  
Prädikatrest**

Die Satzverneinung steht vor dem Prädikatrest.

Beispiel:

**„der Verkäufer – hatte – seinem Kunden – das  
Buch – gestern – in seinem Laden – (nicht)  
– gegeben.“**



Compiler

Syntaktische  
Analyse

# Ableitungsschritt (Ableitung in einem Schritt)

1. Sei  $A \rightarrow \gamma$  eine Produktion
2. Sei  $\alpha A \beta$  ein String über  $N \cup T$
3. Dann schreiben wir:  
 $\alpha A \beta \Rightarrow \alpha \gamma \beta$

Beispiel:

$$E \rightarrow E + E \mid E * E \mid ( E ) \mid \text{id}$$

$$E \Rightarrow (E) \Rightarrow (\text{id})$$



Compiler

# Ableitungen und Satzform

Definition von  $\Rightarrow^*$   
(Ableitung in beliebig vielen Schritten):

1.  $\alpha \xrightarrow{*} \alpha$  für jedes  $\alpha$
2. Wenn  $\alpha \xrightarrow{*} \beta$  und  $\beta \Rightarrow \gamma$ , dann  $\alpha \xrightarrow{*} \gamma$

Wenn  $S \Rightarrow^* \alpha$  gilt, wobei S das Startsymbol von G ist,  
ist  $\alpha$  eine **Satzform** von G



# Ableitungen und Sprache

Compiler

Syntaktische  
Analyse

Definition von  $\Rightarrow^*$ :

1.  $\alpha \xrightarrow{*} \alpha$  für jedes  $\alpha$
2. Wenn  $\alpha \xrightarrow{*} \beta$  und  $\beta \Rightarrow \gamma$ , dann  $\alpha \xrightarrow{*} \gamma$

1. Von Grammatik G **erzeugte Sprache**:  
 $L(G) = \{w \in T^* \mid S \xrightarrow{*} w\}$
2. Zwei Grammatiken mit derselben Sprache werden **äquivalent** genannt



# Die Chomsky Hierarchie

Compiler

Syntaktische  
Analyse

## Type 3: Reguläre Sprachen

- reguläre Ausdrücke, endliche Automaten

## Type 2: kontextfreie Sprachen

- kfG's, Kellerautomaten

## Type 1: kontextsensitive Sprachen

- CSG's (  $AB \rightarrow AC$ ,  $CB \rightarrow CD$ ), LBA's

## Type 0: rekursive aufzählbare Sprachen

- Turingmaschine

$\{a^n b^n c^n | n > 1\}$ : Typ 1 nicht Typ 2,

$\{a^n b^n | n > 1\}$ : Typ 2 nicht Typ 3

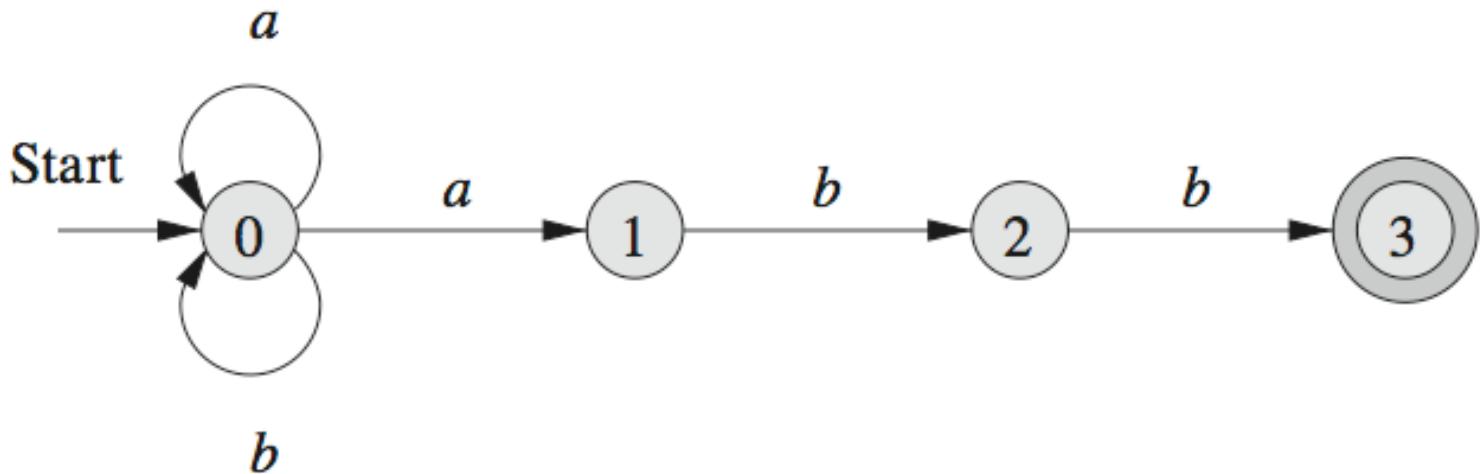


# Reguläre Ausdrücke → kfG

Compiler

- $(a|b)^*abb$

Syntaktische  
Analyse



$$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$$

$$A_1 \rightarrow bA_2$$

$$A_2 \rightarrow bA_3$$

$$A_3 \rightarrow \epsilon$$



Compiler

Syntaktische  
Analyse

# Reguläre Ausdrücke (RE) vs kontextfreie Grammatiken (kfG)

- kfG mächtiger
- RE lassen sich einfacher, automatisch in effiziente Lexer umwandeln
- für Token sind RE im Allgemeinen knapper und leichter verständlich
- kfG sind gut geeignet um verschachtelte Strukturen zu beschreiben
- Trennung des Front Ends in zwei unabhängige Komponenten hat Vorteile



Compiler

Syntaktische  
Analyse

# Quiz

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

$T = \{id, (,), +, *\}, N = \{Ex\}, S = Ex$

Finden Sie (falls möglich) von  $Ex$   
aus Ableitungen für:

- $id + id * id$
- $(id + (+id))$


$$Ex \rightarrow id \mid (Ex) \mid Ex + Ex \mid Ex^* Ex$$

# Lösung

Compiler

Ex

Ex + Ex

id+ Ex

id+ Ex \* Ex

id+ id\* Ex

id+ id\* id

Ex

Ex + Ex

Ex + Ex \* Ex

Ex + Ex \* id

Ex + id\* id

id+ id\* id

Ex

Ex \* Ex

Ex + Ex \* Ex

id+ Ex \* Ex

id+ id\* Ex

id+ id\* id

alle Strings:  
**Satzformen**  
der  
Grammatik



# Links- und Rechtsableitungen

Compiler

Syntaktische  
Analyse

Sei  $A \rightarrow \gamma$  eine Produktion

Linksableitungen:

$\alpha A \beta \Rightarrow_{lm} \alpha \gamma \beta$  wenn  $\alpha A \beta \Rightarrow \alpha \gamma \beta$  und  $\alpha \in T^*$

Rechtsableitungen:

$\alpha A \beta \Rightarrow_{rm} \alpha \gamma \beta$  wenn  $\alpha A \beta \Rightarrow \alpha \gamma \beta$  und  $\beta \in T^*$



# Parsebäume

Compiler

Syntaktische  
Analyse

## Grafische Darstellung einer Ableitung

filtert heraus in welcher Reihenfolge  
Produktionen eingesetzt wurden: kein  
Unterschied zwischen Links- und  
Rechtsableitungen

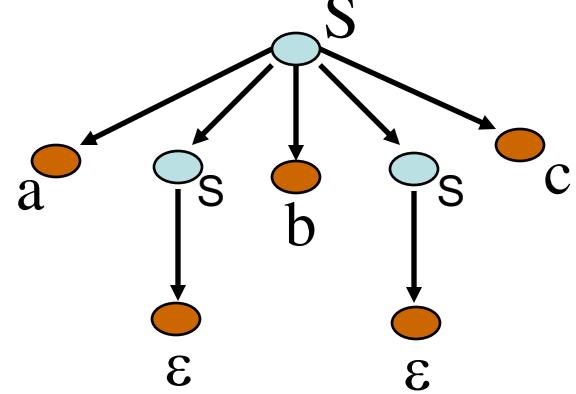


Compiler

Syntaktische  
Analyse

# Parsebäume

$$S \rightarrow aSbSc \mid \varepsilon$$



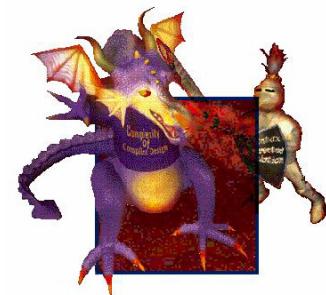
Blätter:

- Terminale oder Nichtterminale
- Grenze: erzeugte Satzform

Innere Knoten:

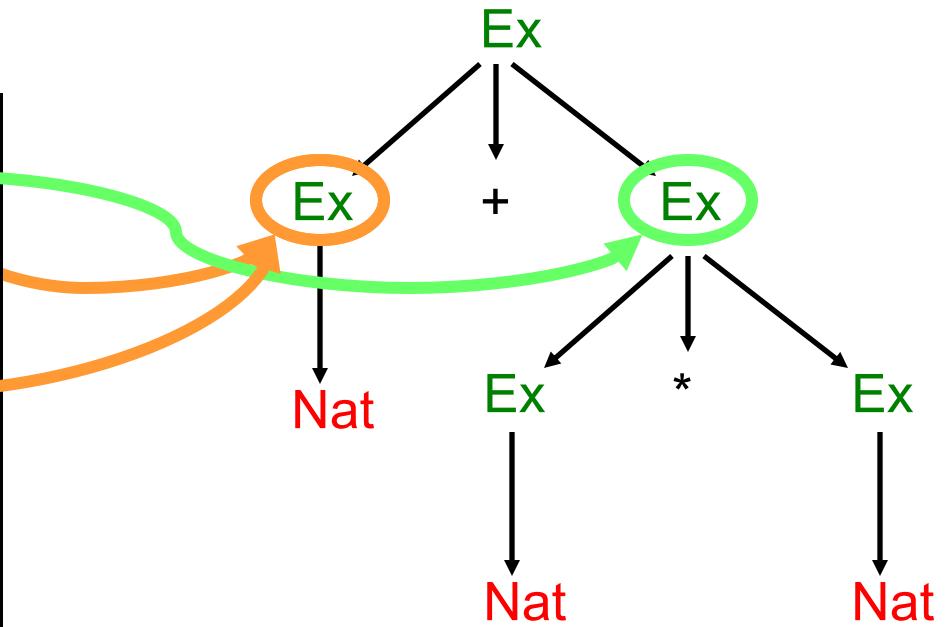
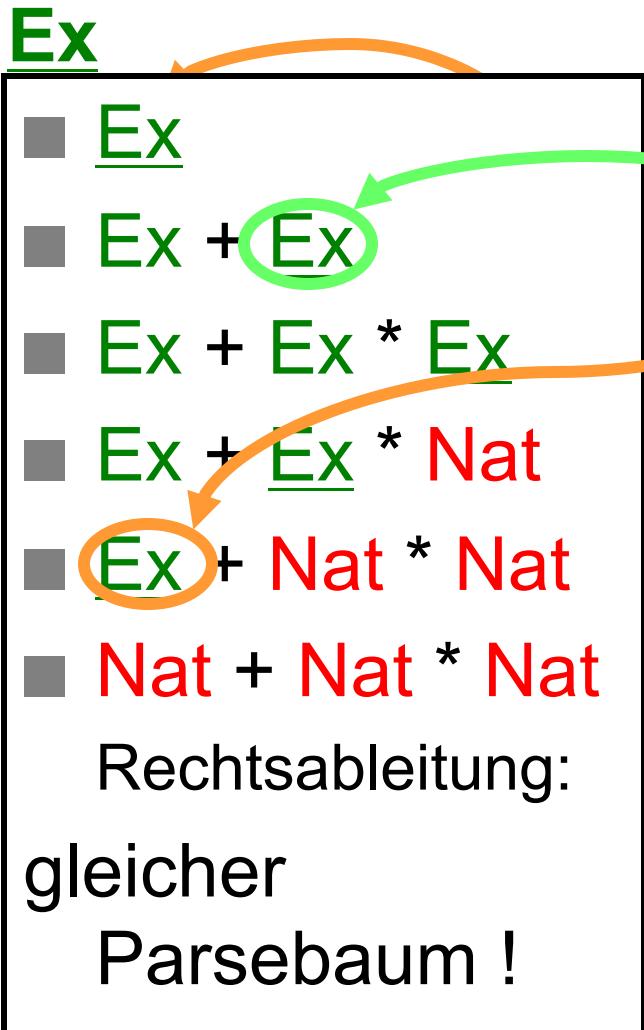
- Nichtterminale
- Kinder: geordnet, jedes Kind wird durch eine Produktion vom Vater erhalten

Wurzel: Startsymbol S

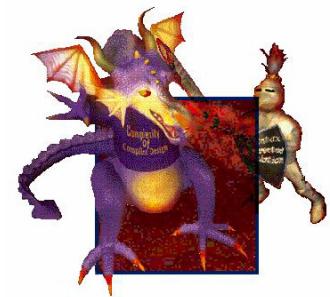

$$Ex \rightarrow Nat \mid (Ex) \mid Ex + Ex \mid Ex * Ex$$

# Parsebäume

Compiler



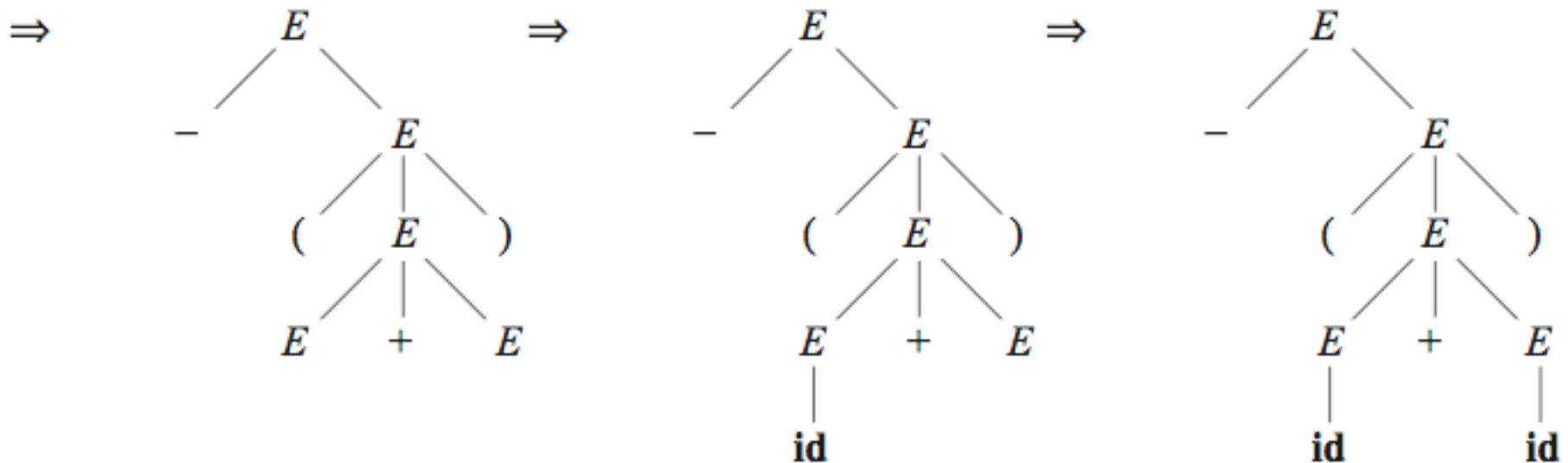
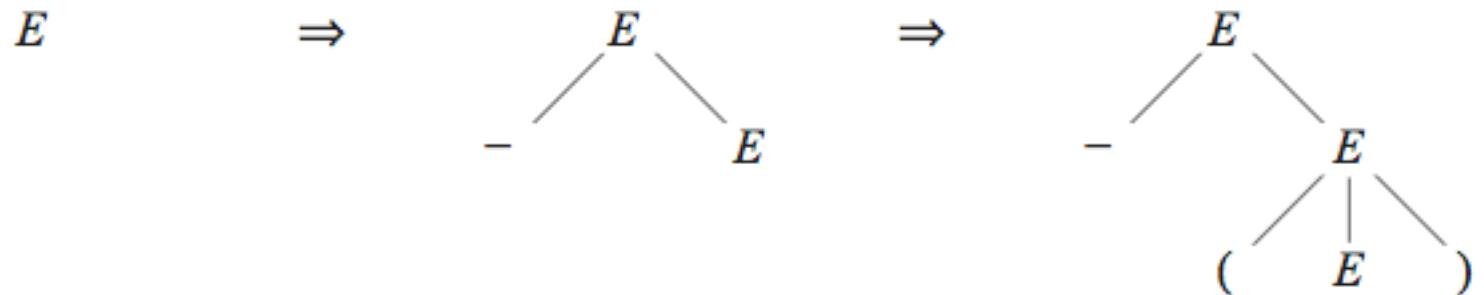
Blätter: erzeugter String  
**Nat + Nat \* Nat**



# Folge von Parse-Bäume

$$E \rightarrow E + E \quad | \quad E * E \quad | \quad -E \quad | \quad ( E ) \quad | \quad \text{id}$$

Compiler

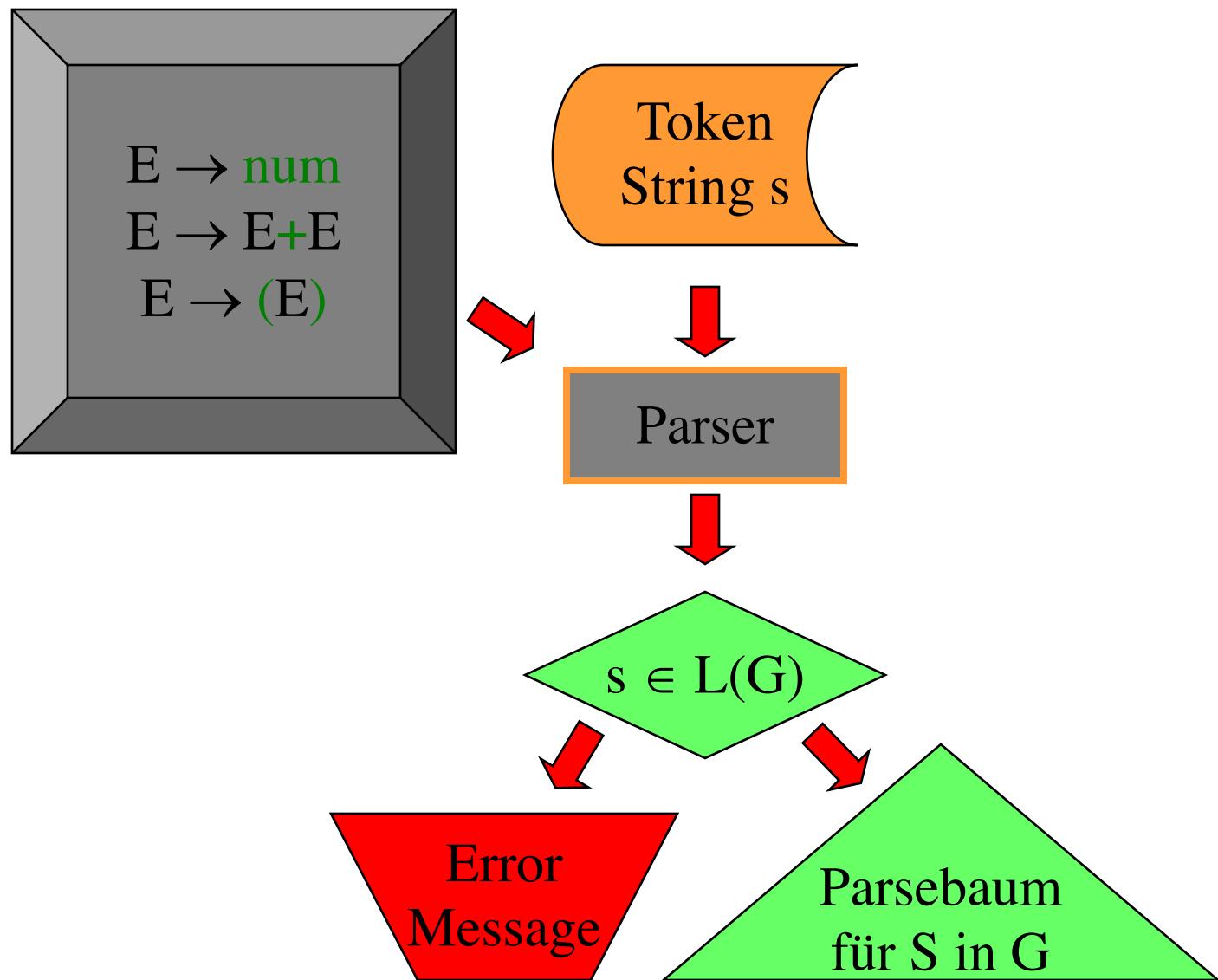


# Was ist Parsing



Compiler

Syntaktische  
Analyse

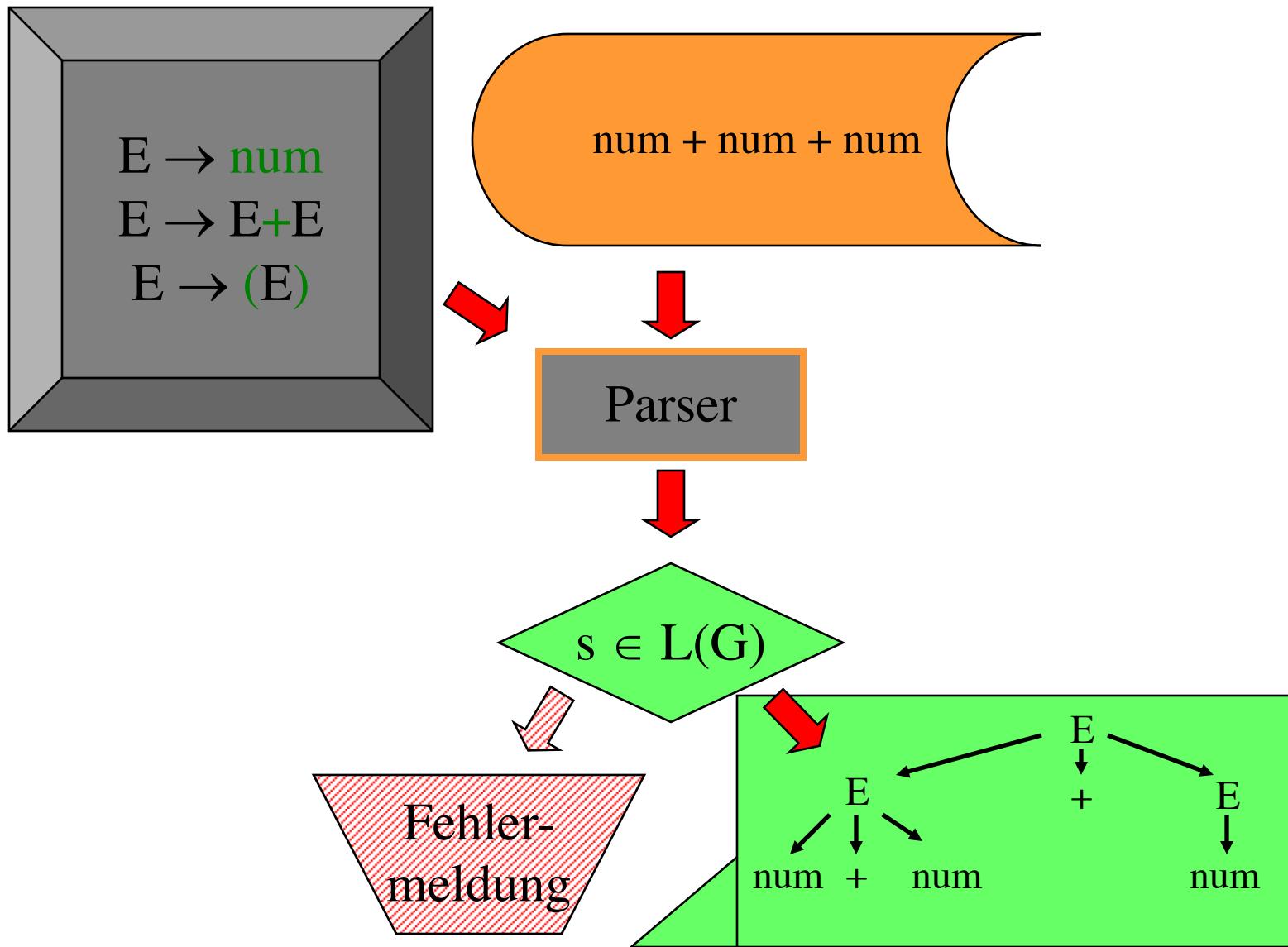


# Was ist Parsing II



Compiler

Syntaktische  
Analyse

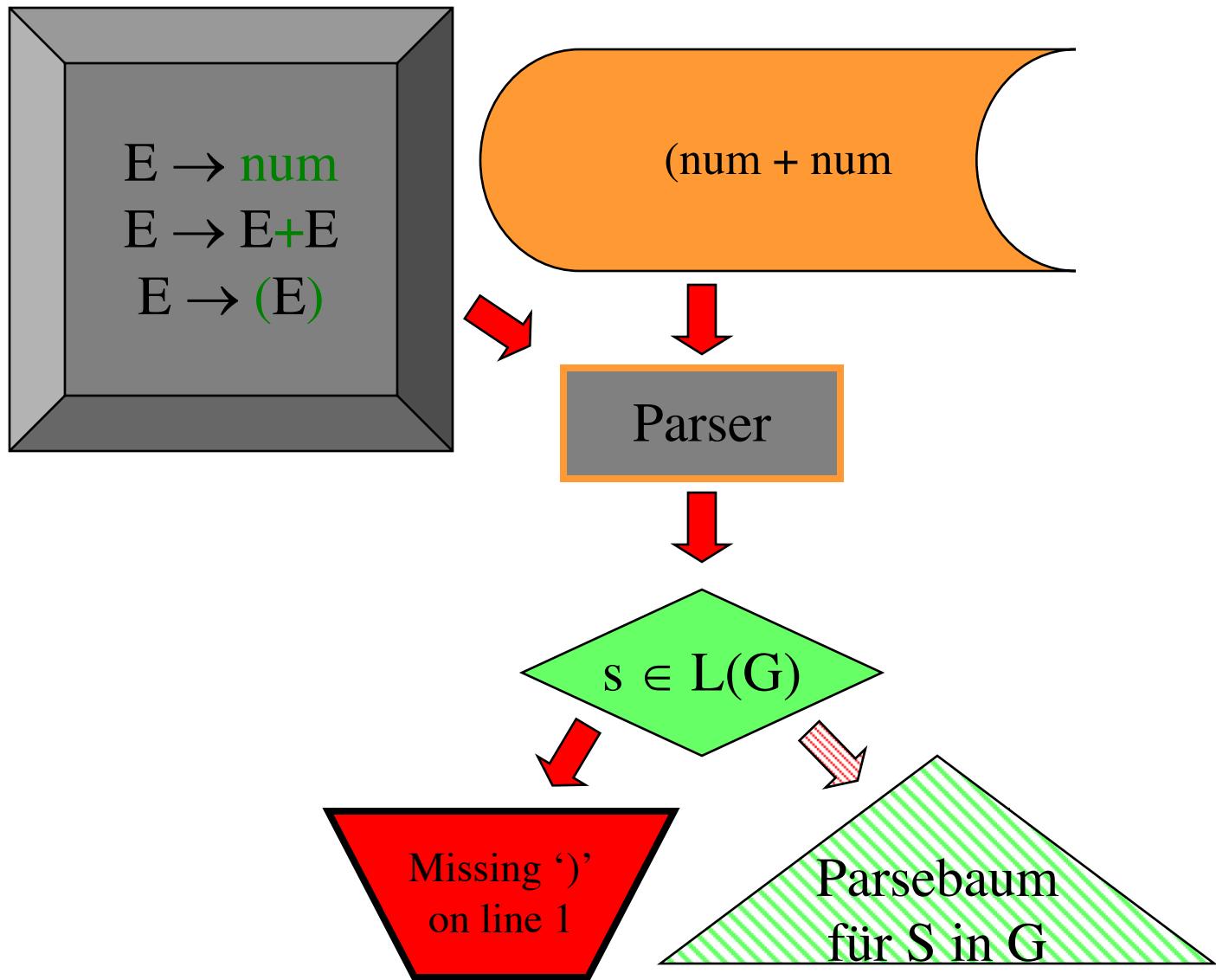


# Was ist Parsing III



Compiler

Syntaktische  
Analyse



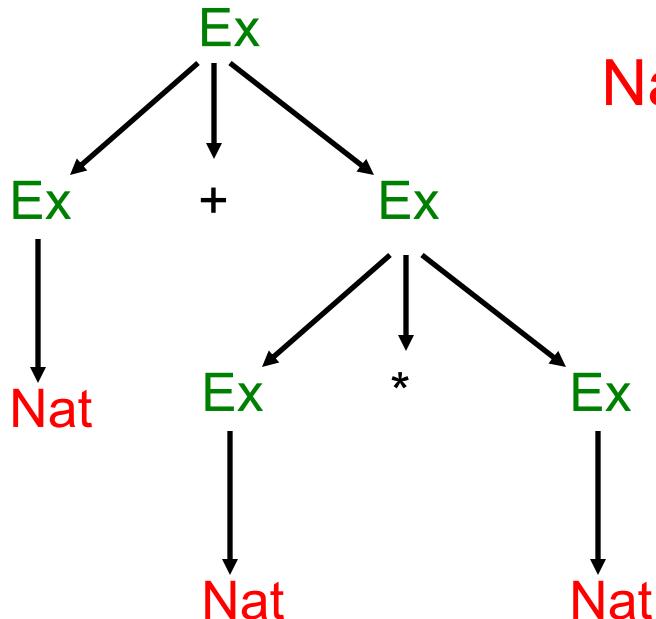

$$Ex \rightarrow \text{Nat} \mid (\text{Ex}) \mid \text{Ex} + \text{Ex} \mid \text{Ex} * \text{Ex}$$

## Mehrdeutige Grammatiken

Compiler

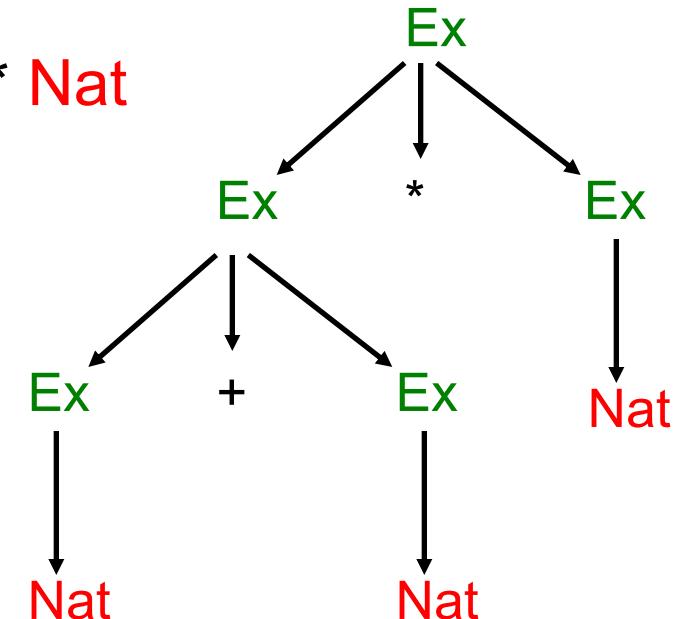
Syntaktische  
Analyse

mehrdeutig: falls es für einen String  $\geq 2$  verschiedene Parsebäume gibt:



$\text{Nat} + (\text{Nat} * \text{Nat})$

$\text{Nat} + \text{Nat} * \text{Nat}$



$(\text{Nat} + \text{Nat}) * \text{Nat}$



# Eliminieren von Mehrdeutigkeiten

Compiler

- $S \rightarrow ab \mid aT$
- $T \rightarrow b \mid bTc$

Syntaktische  
Analyse



# Eliminieren von Mehrdeutigkeiten (4.3.2)

Compiler

Syntaktische  
Analyse

```
stmt → if expr then stmt
      | if expr then stmt else stmt
      | other
```

Gibt es Strings mit zwei unterschiedlichen  
Parsebäumen ?



# Eliminieren von Mehrdeutigkeiten

Compiler

Syntaktische  
Analyse

```
stmt → if expr then stmt
      | if expr then stmt else stmt
      | other
```

ja:

if E<sub>1</sub> then if E<sub>2</sub> then S<sub>1</sub> else S<sub>2</sub>



# Eliminieren von Mehrdeutigkeiten II

Compiler

Syntaktische  
Analyse

$stmt \rightarrow if\ expr\ then\ stmt$   
|  $if\ expr\ then\ stmt\ else\ stmt$   
|  $other$

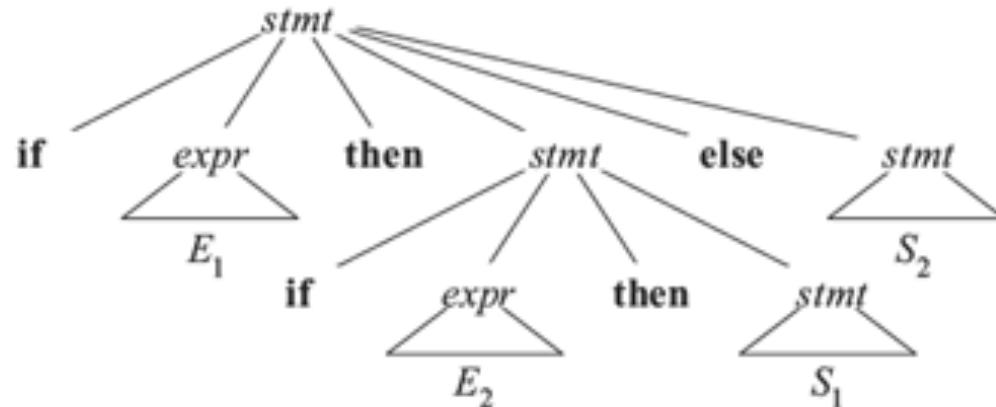
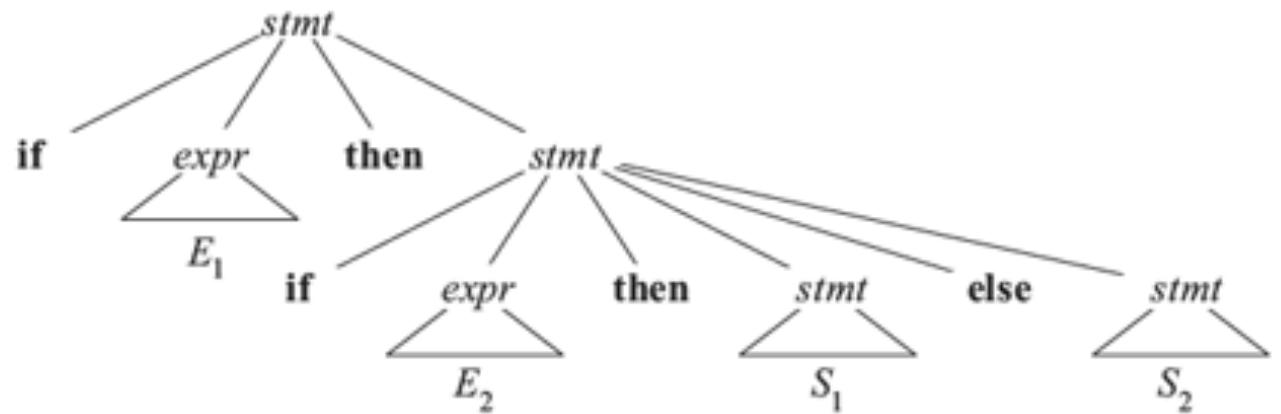


Abbildung 4.9: Zwei Parse-Bäume für einen mehrdeutigen Satz



Compiler

Syntaktische  
Analyse

# Eliminieren von Mehrdeutigkeiten

- Sprache verbessern:
    - $S \rightarrow \text{if } E \text{ then } S \text{ endif}$
    - $S \rightarrow \text{if } E \text{ then } S \text{ else } S \text{ endif}$
- if  $E_1$  then if  $E_2$  then  $S_1$  else  $S_2$
- if  $E_1$  then if  $E_2$  then  $S_1 \text{ endif }$  else  $S_2 \text{ endif }$
- if  $E_1$  then if  $E_2$  then  $S_1$  else  $S_2 \text{ endif }$  **endif**



# Eliminieren von Mehrdeutigkeiten III

Compiler

```
stmt → if expr then stmt  
      | if expr then stmt else stmt  
      | other
```

Syntaktische  
Analyse

```
stmt → matched_stmt  
      | open_stmt  
matched_stmt → if expr then matched_stmt else matched_stmt  
      | other  
open_stmt → if expr then stmt  
      | if expr then matched_stmt else open_stmt
```

if  $E_1$  then if  $E_2$  then  $S_1$  else  $S_2$



# Zusammenfassung

Compiler

## Parsing: Warum, Wann, Was Kontext-freie Grammatiken

- Bestandteile, Ableitungen,, Parse Bäume,  
Mehrdeutigkeit
- Mächtigkeit (im Vergleich zu regulären  
Ausdrücken)

Jetzt:

- **Wie** macht man Parsing