



# Syntactic Analysis I

*Compilers course*

Masters in Informatics and Computing Engineering (MIEIC), 3rd Year



**João M. P. Cardoso**



Universidade do Porto  
**FEUP** Faculdade de Engenharia

Dep. de Engenharia Informática  
Faculdade de Engenharia (FEUP), Universidade do Porto,  
Porto, Portugal  
Email: [jmpc@acm.org](mailto:jmpc@acm.org)

# Syntax in Programming Languages

- Syntax of programming languages cannot be handle by regular expressions (programming languages are not regular languages)
- Why?
  - $(a+(b-c))^*(d-(x-(y-z)))$
  - if  $(x < y)$  if  $(y < z)$   $a = 5$  else  $a = 6$  else  $a = 7$
- Regular languages don't have have the required state to model nesting
- There is none regular expression to specify expressions with parenthesis!

# Solution

- Context Free Grammars (CFGs)
  - Recognition done by Finite Automata with a stack (known as Push Down Automata: PDAs)
  - Or...

# Grammars

- Backus Naur Form (BNF) notation to specify grammars:
  - **Terminal symbols:** Uppercase letters
  - **Non-terminal symbols:** start by an uppercase letters (or delimited by < and >)
  - In a production the left and the right side are separated by  $\rightarrow$  or  $::=$ ,  
E.g.:
    - $\text{Expr} \rightarrow \text{Term OP Term}$
    - $\text{Expr} ::= \text{Term OP Term}$
  - Alternative productions ( $p_1, p_2, p_3, \dots, p_n$ ) are represented by  $p_1 \mid p_2 \mid p_3 \mid \dots \mid p_n$ 
    - E.g.:  $\text{Literal} \rightarrow \text{BINARIO} \mid \text{OCTAL} \mid \text{INT} \mid \text{FLOAT}$
  - If the right hand side of a production does not contain any symbol then we write  $\varepsilon$ 
    - E.g.:  $\text{Palavra} \rightarrow \varepsilon$

# Grammars

- EBNF, or extended BNF
  - Includes { } to represent 0 or more occurrences
  - and [ ] to represent optional elements

BNF: [https://en.wikipedia.org/wiki/Backus%E2%80%93Naur\\_Form](https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form)

EBNF: [https://en.wikipedia.org/wiki/Extended\\_Backus%E2%80%93Naur\\_Form](https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form)

# Context Free Grammars (CFGs)

- Set of terminal symbols  
 $\{ \text{OP}, \text{INT}, \text{OPEN}, \text{CLOSE} \}$   
Each terminal symbol defined by a regular expression
- Set of non-terminal symbols  
 $\{ \text{Start}, \text{Expr} \}$
- Set of productions
  - A single non-terminal symbol in the left hand side (LHS)
  - Sequence of terminal and non-terminal symbols in the right hand side (RHS)

**OP = + | - | \* | /**

**INT = [0-9] [0-9]\***

**OPEN = (**

**CLOSE = )**

**Start  $\rightarrow$  Expr**

**Expr  $\rightarrow$  Expr OP Expr**

**Expr  $\rightarrow$  INT**

**Expr  $\rightarrow$  OPEN Expr CLOSE**

# Production/Derivation Game

Given a string

Repeat until there are none terminal symbols

- Select a non-terminal symbol (start by the non-terminal symbol Start)

- Select a production for that non-terminal symbol

- Substitute the non-terminal symbol with the RHS of the production

Substitute the regular expression with the correspondent strings

Generated string belongs to the language

Note: different selections produce different Strings

# Production/Derivation

**OP = +|-|\*|/**

**INT = [0-9] [0-9]\***

**OPEN = (**

**CLOSE = )**

**1) Start  $\rightarrow$  Expr**

**2) Expr  $\rightarrow$  Expr OP Expr**

**3) Expr  $\rightarrow$  INT**

**4) Expr  $\rightarrow$  OPEN Expr CLOSE**

Start

Expr

Expr OP Expr

OPEN Expr CLOSE OP Expr

OPEN Expr OP Expr CLOSE OP Expr

OPEN INT OP Expr CLOSE OP Expr

OPEN INT OP Expr CLOSE OP INT

OPEN INT OP INT CLOSE OP INT

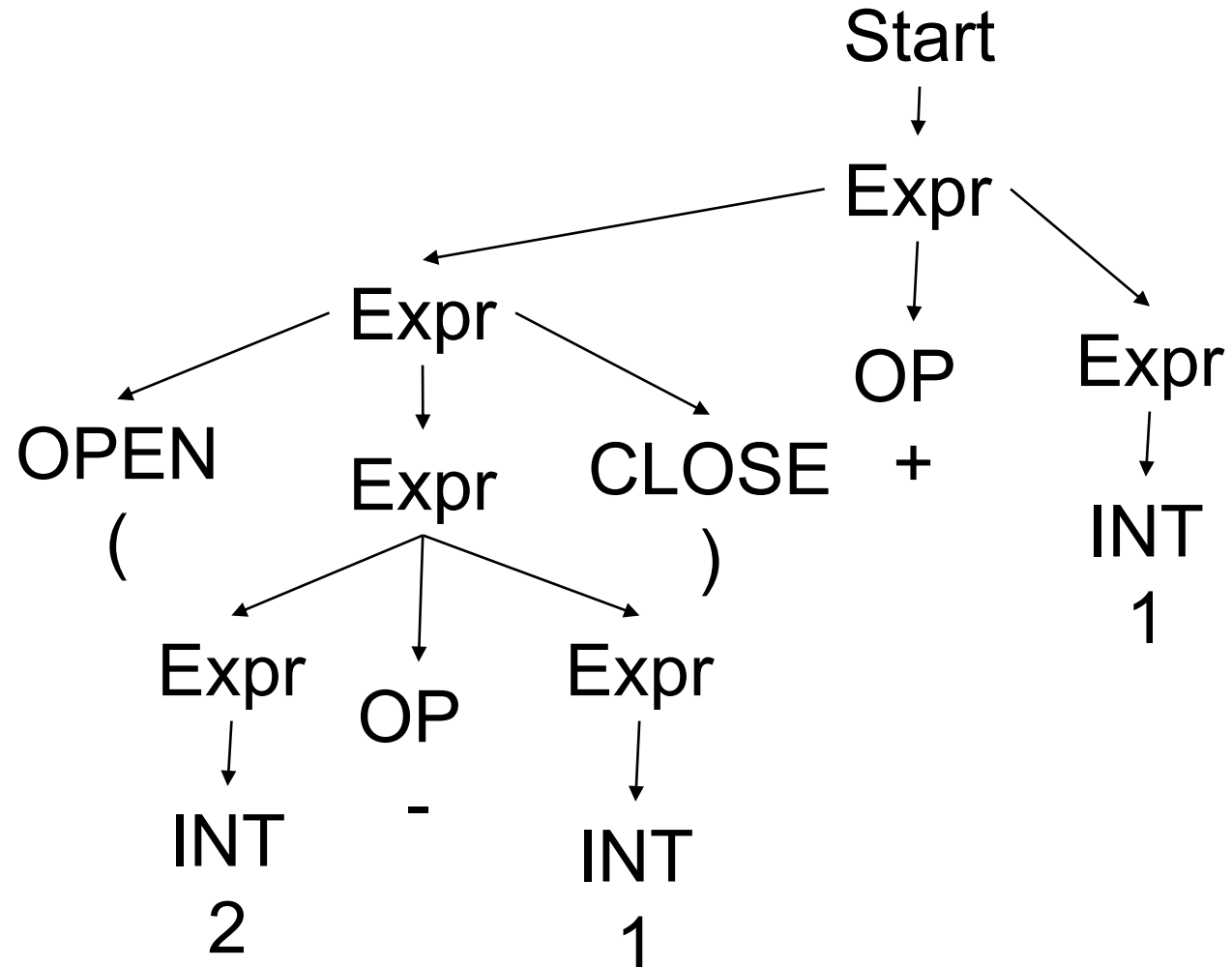
( 2 - 1 ) + 1



# Syntax Tree

- Internal nodes: non-terminal symbols
- Leaves: terminal symbols
- Edges:
  - From non-terminal symbols of the LHS of the production
  - To nodes of the RHS of the production
- Captures the derivation of a String accepted by the language

# Syntax Tree for (2-1)+1



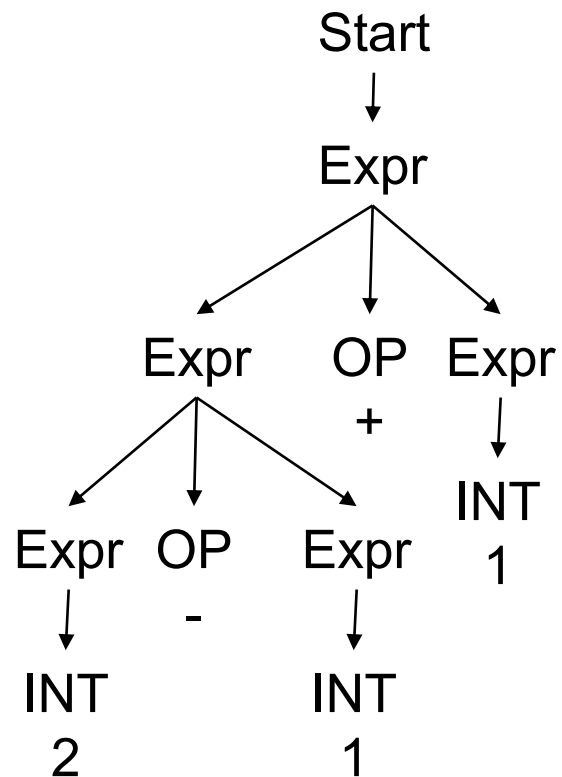
# Ambiguity in a Grammar

- Multiple leftmost or rightmost derivations (implying multiple syntax trees) for the same String
- Syntax tree usually reflect semantic of the program
- Ambiguity in the grammar reflects many times ambiguity in terms of semantic (considered undesirable)

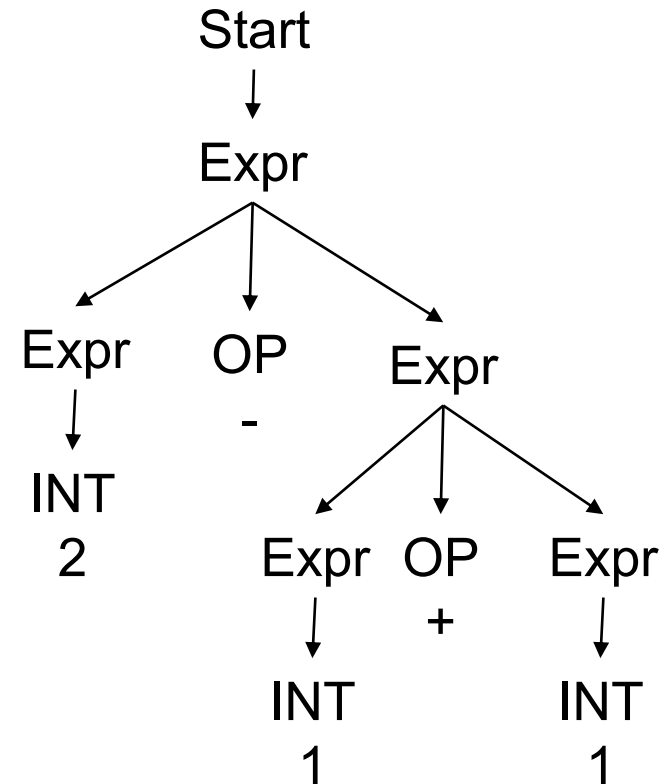
# Example of Ambiguity

- Two syntax tree for  $2-1+1$

Tree corresponding to  $(2-1)+1$



Tree corresponding to  $2-(1+1)$



# Eliminating Ambiguity

- Solution: modify grammar
- All operators with left-associative

## Original Grammar

**Start  $\rightarrow$  Expr**  
**Expr  $\rightarrow$  Expr OP Expr**  
**Expr  $\rightarrow$  INT**  
**Expr  $\rightarrow$  OPEN Expr CLOSE**

## Modified Grammar

**Start  $\rightarrow$  Expr**  
**Expr  $\rightarrow$  Expr OP INT**  
**Expr  $\rightarrow$  INT**  
**Expr  $\rightarrow$  OPEN Expr CLOSE**

Different language!

# Eliminating Ambiguity

- Solution: modify grammar
- All operators with left-associative

## Original Grammar

**Start**  $\rightarrow$  **Expr**  
**Expr**  $\rightarrow$  **Expr OP Expr**  
**Expr**  $\rightarrow$  **INT**  
**Expr**  $\rightarrow$  **OPEN Expr CLOSE**

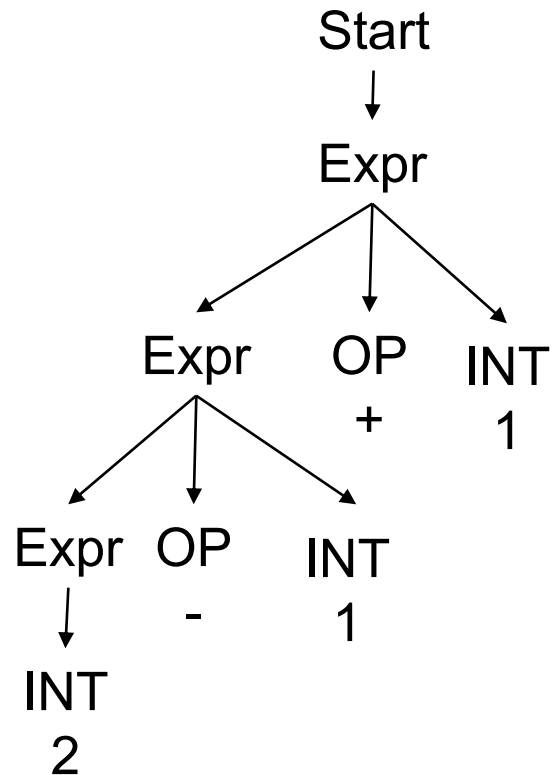
## Modified Grammar

**Start**  $\rightarrow$  **Expr**  
**Expr**  $\rightarrow$  **Expr OP Expr'**  
**Expr**  $\rightarrow$  **INT**  
**Expr**  $\rightarrow$  **OPEN Expr CLOSE**  
**Expr'**  $\rightarrow$  **OPEN Expr CLOSE**  
**Expr'**  $\rightarrow$  **INT**

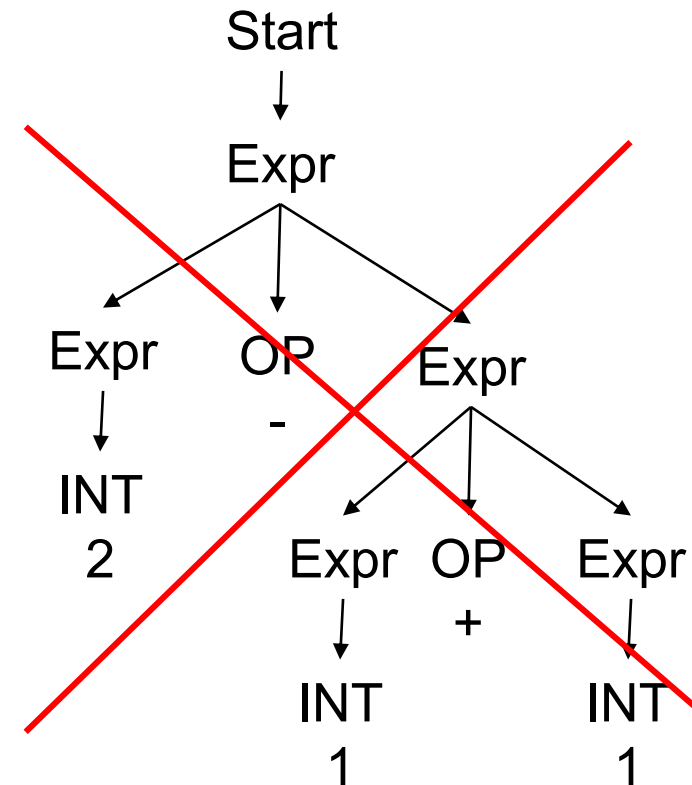
# Syntax Tree

- Only one syntax tree for:  $2-1+1$

Valid syntax tree



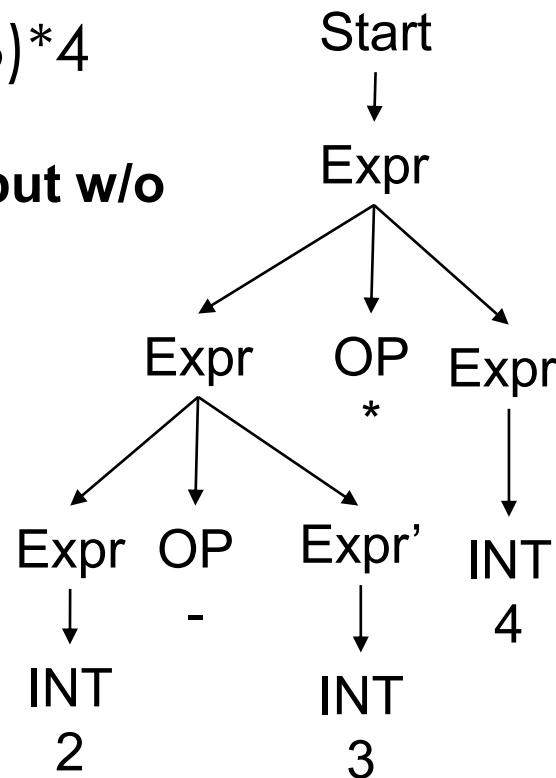
Invalid syntax tree



# Precedence of Operators

- All operators with left-associative
- Without respecting the precedence of \* over +
- 2-3\*4 interpreted as (2-3)\*4

Syntax tree for 2-3\*4



**Modified Grammar (w/o ambiguity but w/o respecting precedences)**

**Start** → **Expr**

**Expr** → **Expr** **OP** **Expr'**

**Expr** → **INT**

**Expr** → **OPEN** **Expr** **CLOSE**

**Expr'** → **OPEN** **Expr** **CLOSE**

**Expr'** → **INT**

**Grammar equivalent:**

**Start** → **Expr**

**Expr** → **Expr** **OP** **Expr'**

**Expr** → **Expr'**

**Expr'** → **OPEN** **Expr** **CLOSE**

**Expr'** → **INT**



# Solution to Precedence

➤ Suppose:

Original Grammar

**OP = + | - | \* | /**

**INT = [0-9] [0-9]\***

**OPEN = (**

**CLOSE = )**

**Start → Expr**

**Expr → Expr OP Expr'**

**Expr → Expr'**

**Expr' → OPEN Expr CLOSE**

**Expr' → INT**

Modified Grammar

**OP1 = + | -**

**OP2 = \* | /**

**INT = [0-9] [0-9]\***

**OPEN = (**

**CLOSE = )**

**Start → Expr**

**Expr → Expr OP1 Term**

**Expr → Term**

**Term → Term OP2 Final**

**Term → Final**

**Final → INT**

**Final → OPEN Expr CLOSE**

# Modification in Syntax Tree

Old syntax tree for

2-3\*4

Start

Expr

Expr

OP

Expr'

\*

Expr

OP

Expr'

INT

Expr'

INT

4

INT

2

3

New syntax tree for

2-3\*4

Start

Expr

Expr

OP1

Term

Term

Term

Final

Final

Final

OP2

Final

INT

2

INT

3

INT

4

Start → Expr

Expr → Expr OP1 Term

Expr → Term

Term → Term OP2 Final

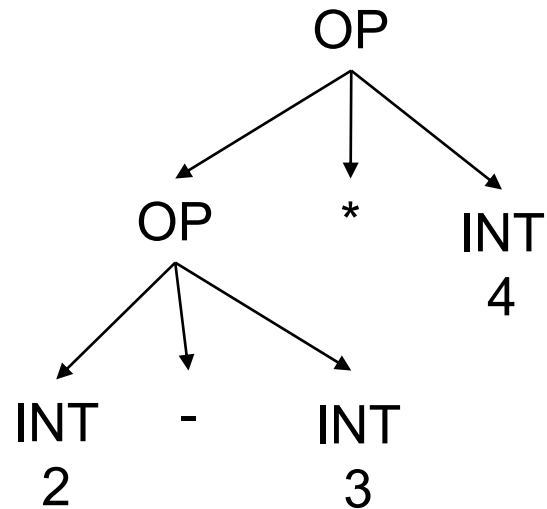
Term → Final

Final → INT

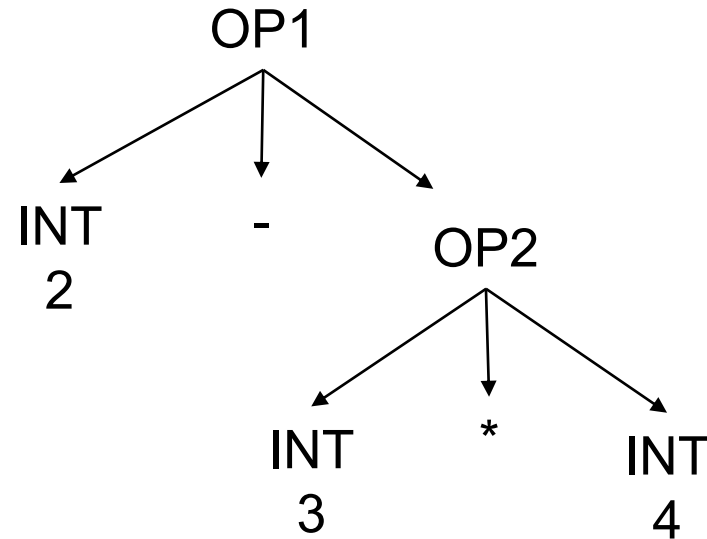
Final → OPEN Expr CLOSE

# Modification in Syntax Tree

Old AST for  
2-3\*4



New AST for  
2-3\*4



# Global Idea

- Group operators by precedence levels
  - \* and / are at the top level
  - + and - are in the next level
- Non-terminal symbol for each precedence level
  - Term is non-terminal for \* and /
  - Expr is non-terminal for + and -
- In each level we can do right or left associativity
- Generalize to more levels of precedence

# Exercise

- Specify using BNF grammars corresponding to the regular expressions:  
 $[0-9]^+ \text{ e } [0-9]^*$
- Given the grammar:

**NUM =  $[0-9]^+$**

**ID =  $[A-Za-Z][0-9A-Za-z]^*$**

**Expr  $\rightarrow$  Expr “+” Term | Expr “-” Term | Term**

**Term  $\rightarrow$  Term “\*” Factor | Term “/” Factor | Factor**

**Factor  $\rightarrow$  Primary “^” Factor | Primary**

**Primary  $\rightarrow$  “-”Primary | Element**

**Element  $\rightarrow$  “(“ Expr “)” | NUM | ID**

- What are the syntax trees for:

- $5-2*3$

- $y^3$

# Handling if-then-else Constructs

Start  $\rightarrow$  Stat

Stat  $\rightarrow$  IF Expr THEN Stat ELSE Stat

Stat  $\rightarrow$  IF Expr THEN Stat

Stat  $\rightarrow$  ...

# Syntax Tree

- Consider the statement:
  - if  $e_1$  then if  $e_2$  then  $s_1$  else  $s_2$

Start  $\rightarrow$  Stat

Stat  $\rightarrow$  IF Expr THEN Stat ELSE Stat

Stat  $\rightarrow$  IF Expr THEN Stat

Stat  $\rightarrow$  ...

# Syntax Tree

Start  $\rightarrow$  Stat

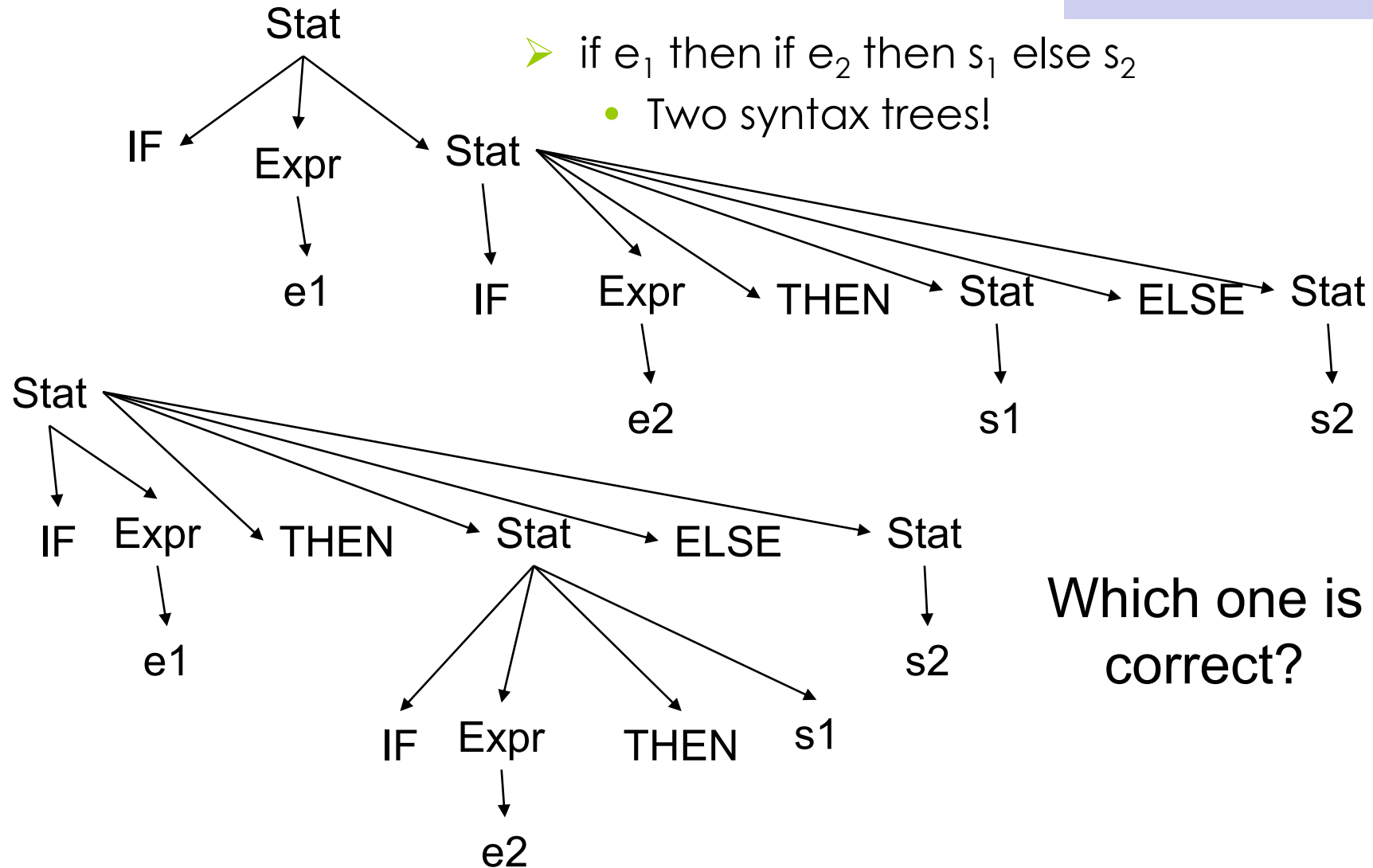
Stat  $\rightarrow$  IF Expr THEN Stat ELSE Stat

Stat  $\rightarrow$  IF Expr THEN Stat

Stat  $\rightarrow$  ...

➤ if  $e_1$  then if  $e_2$  then  $s_1$  else  $s_2$

• Two syntax trees!



Which one is the correct?



# Alternative Interpretations

## ➤ Ambiguous grammar

- For the same statement:
  - if  $e_1$  then if  $e_2$  then  $s_1$  else  $s_2$

- Syntax tree 1

if  $e_1$   
    if  $e_2$   $s_1$   
    else  $s_2$

- Syntax tree 2

if  $e_1$   
    if  $e_2$   $s_1$   
else  $s_2$

# Modified Grammar

Start  $\rightarrow$  Stat

Stat  $\rightarrow$  IF Expr THEN Stat ELSE Stat

Stat  $\rightarrow$  IF Expr THEN Stat

Stat  $\rightarrow$  ...

- Basic Idea: control when an IF without ELSE can occur
  - At the top level of the statements
  - Or as the last in a sequence of statements if then else if then ...

Goal  $\rightarrow$  Stat

Stat  $\rightarrow$  WithElse

Stat  $\rightarrow$  LastElse

WithElse  $\rightarrow$  IF Expr THEN WithElse ELSE WithElse

WithElse  $\rightarrow$  ...

LastElse  $\rightarrow$  IF Expr THEN Stat

LastElse  $\rightarrow$  IF Expr THEN WithElse ELSE LastElse

# Syntactic Analyzer

- Convert programs in a syntax tree
- It can be developed from the scratch!
- Or developed automatically by a parser generator
  - Accept a grammar as input
  - Produce the syntactic analyzer as output
- Parctical problem
  - The syntax tree for the modified grammar can be complex
  - We would like to start with a syntax tree

# Solution

- Abstract vs Concrete Tree
  - Abstract syntax corresponds to an intuitive way to think the program structure
    - Omit details as superfluous keywords
    - Abstract syntax can be ambiguous
  - Concrete syntax corresponds to the complete grammar used to analyze syntactically the language
- The syntax analyzer are many times programmed to generate **Abstract Syntax Trees (ASTs)**

# Abstract Syntax Trees (ASTs)

- Start with an intuitive grammar but possibly ambiguous
- Modify the grammar to make it non-ambiguous
  - Concrete Syntax Trees
  - Less intuitive
- Convert the concrete syntax tree (CST) in ASTs
  - They correspond to the intuitive grammar for the language
  - Simpler to manipulate by the compiler

# Example

Non-ambiguous grammar:

OP1 = + | -

OP2 = \* | /

INT = [0-9] [0-9]\*

OPEN = (

CLOSE = )

Start  $\rightarrow$  Expr

Expr  $\rightarrow$  Expr OP1 Term

Expr  $\rightarrow$  Term

Term  $\rightarrow$  OPEN Expr CLOSE

Term  $\rightarrow$  Term OP2 INT

Term  $\rightarrow$  INT

Intuitive grammar but ambiguous:

OP = \* | / | + | -

INT = [0-9] [0-9]\*

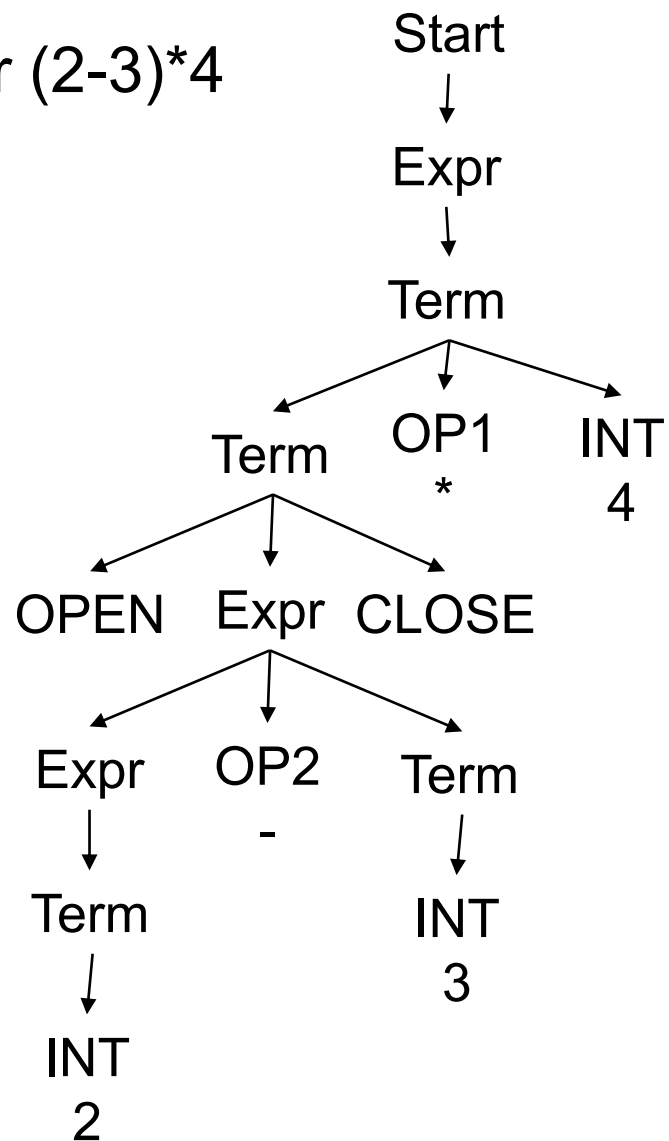
Start  $\rightarrow$  Expr

Expr  $\rightarrow$  Expr OP Expr

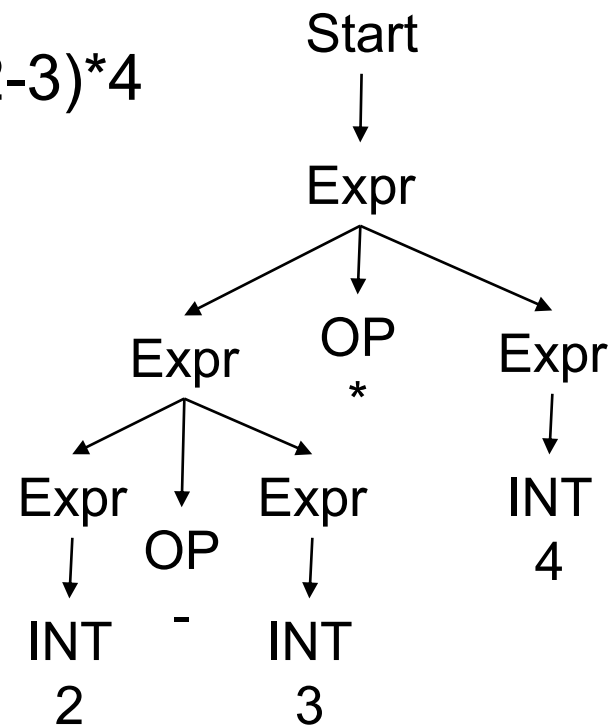
Expr  $\rightarrow$  INT

# Example

CST for  $(2-3)*4$



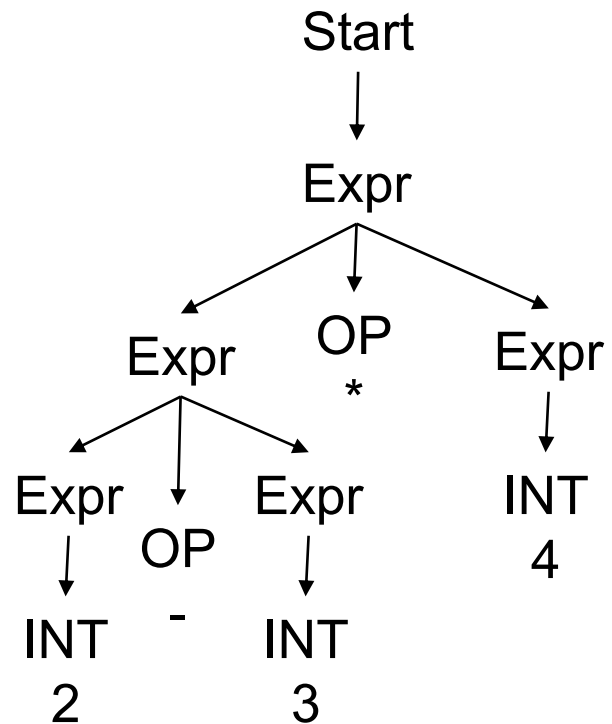
AST for  $(2-3)*4$



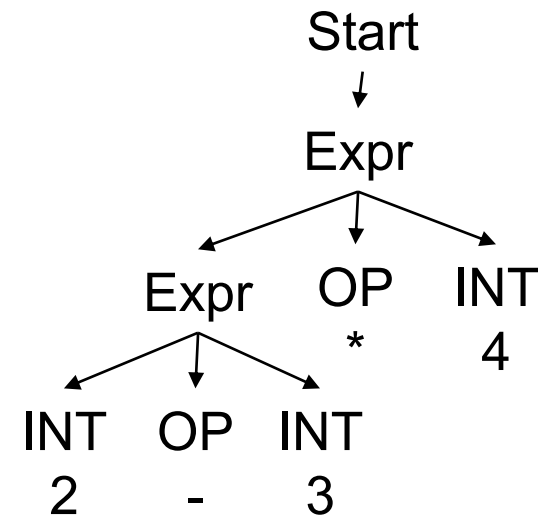
- Close to intuitive grammar
- Eliminates superfluous tokens
  - OPEN, CLOSE, etc.

# Example

AST for  $(2-3)*4$



AST for  $(2-3)*4$   
(even more simplified!)





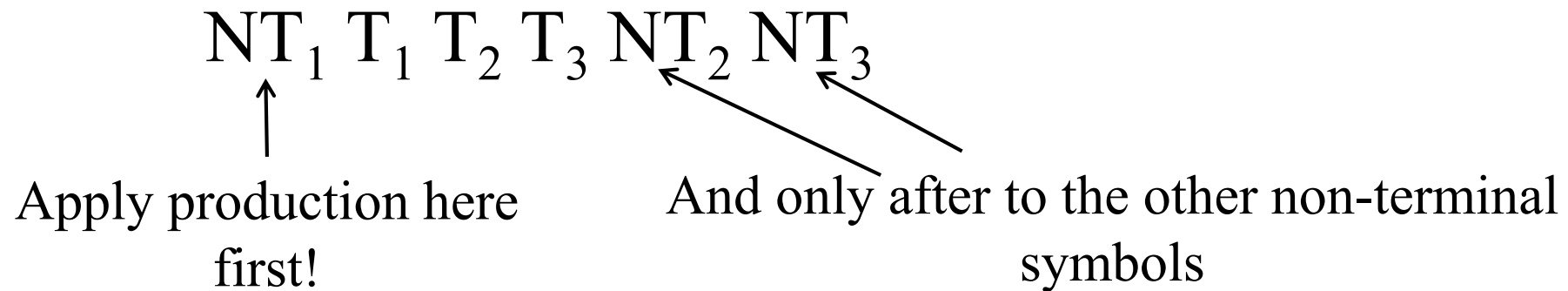
# Summary

- Levels of lexical and syntactic structures
  - Lexical – regular expressions
  - Syntactic– grammars
- Ambiguities in the grammar
  - Modified grammars
- Abstract Syntax Tree (ASTs)
- Generative vs recognizing roles
  - Generative more convenient for specifications
  - Recognizing required for implementation

# Grammar Vocabulary

## ➤ Leftmost derivation

- Always expand the non-terminal symbol that remains in the left



## ➤ Rightmost derivation

- Always expand the non-terminal symbol that remains in the right

# Initial Point

- Assume that the lexical analysis produced a sequence of tokens (terminal symbols)
  - Each token has a type and a value
  - Types correspond to terminal symbols
  - Values correspond to the content of the token read
- Example
  - INT(549) – token that identifies an integer with value read 549
  - IF – keyword “if” without necessity of value
  - OP(+) – Operator with value: +

# Basic Approach

- Start with the start variable
- Build a *leftmost* derivation
  - If the *leftmost* symbol is a non-terminal, select a production and apply it
  - If *leftmost* symbol is terminal, try match with the input
  - If all the terminals were matched there was found a derivation that accepts the String!
  - Key: find the correct productions for the non-terminal symbols

# Grammar Example

INT = [0-9]<sup>+</sup>

Start  $\rightarrow$  Expr

Expr  $\rightarrow$  Expr "+" Term

Expr  $\rightarrow$  Expr "-" Term

Expr  $\rightarrow$  Term

Term  $\rightarrow$  Term "\*" INT

Term  $\rightarrow$  Term "/" INT

Term  $\rightarrow$  INT

- Set of tokens (non-terminal symbols):

{ +, -, \*, /, INT }

# Syntactic Analyzer for the Grammar Example

Syntactic Tree

Start

Input not processed

$2-2*2$

Sentencial form

Start

*Current position in the syntax tree*

# Syntax Analyzer for the Grammar

## Example

Syntax tree

Start



Expr



*Current position in the syntax tree*

Input not processed

2-2\*2

Sentential Form

Expr

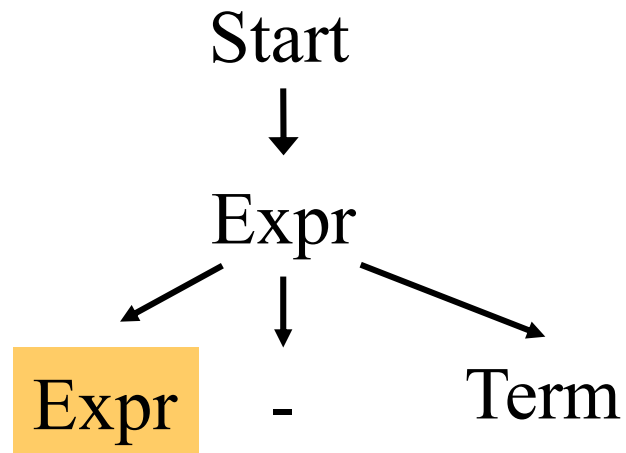
Applied production

Start  $\rightarrow$  Expr

# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Input not processed

2-2\*2

Sentential form

Expr - Term

Applied production

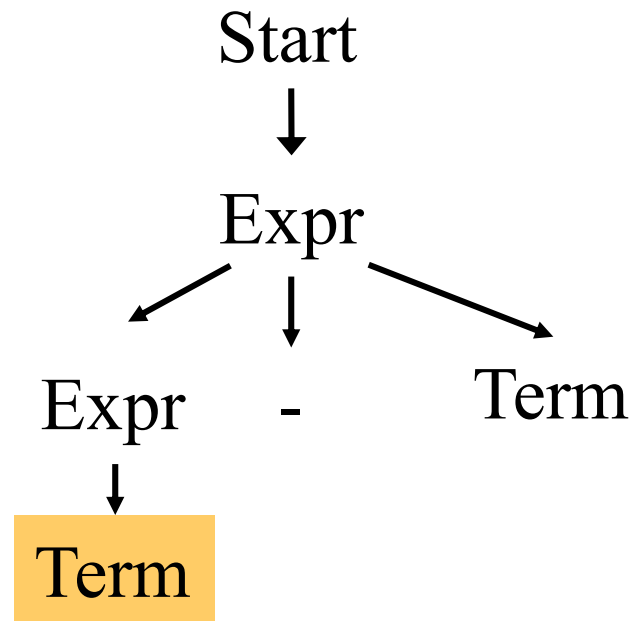
$\text{Expr} \rightarrow \text{Expr} - \text{Term}$



# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Input not processed

2-2\*2

Sentential form

Term - Term

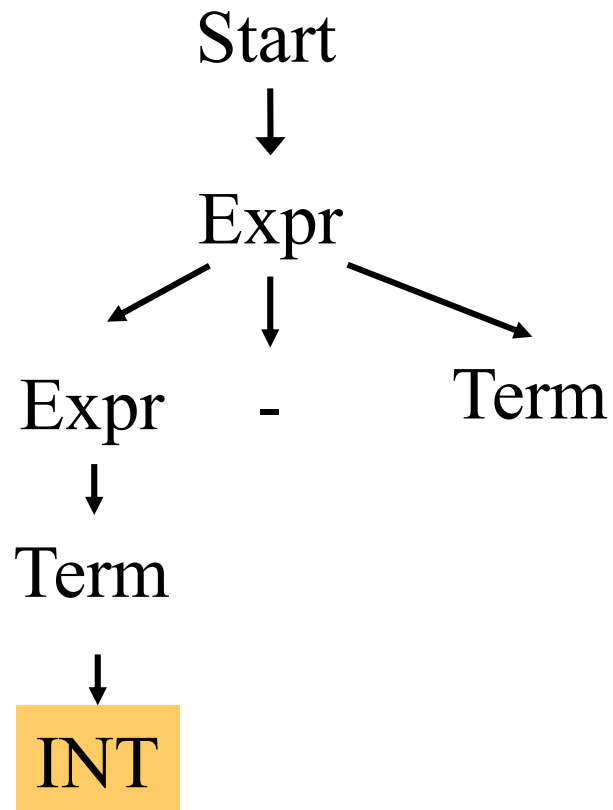
Applied production

$\text{Expr} \rightarrow \text{Term}$

# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Input not processed

2-2\*2

Sentential form

INT - Term

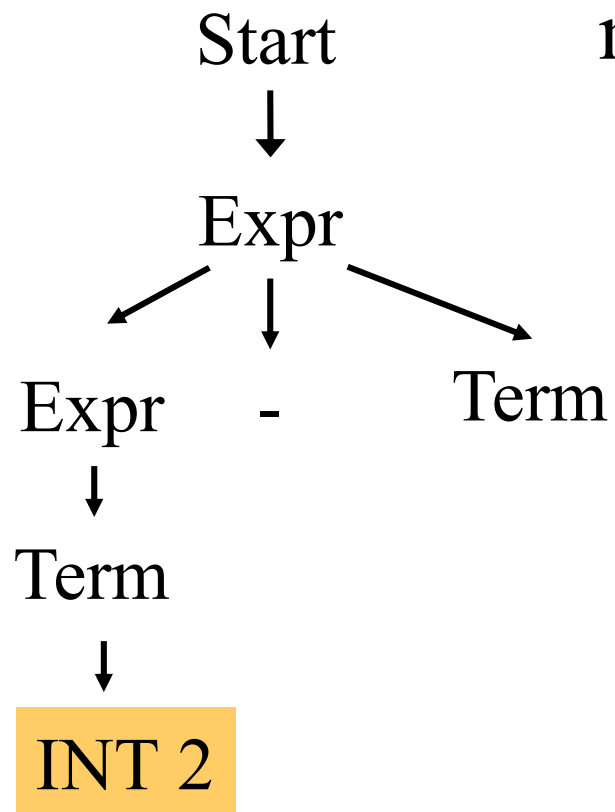
Applied production

Term  $\rightarrow$  INT

# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Token  
matches!

Input not processed

2-2\*2

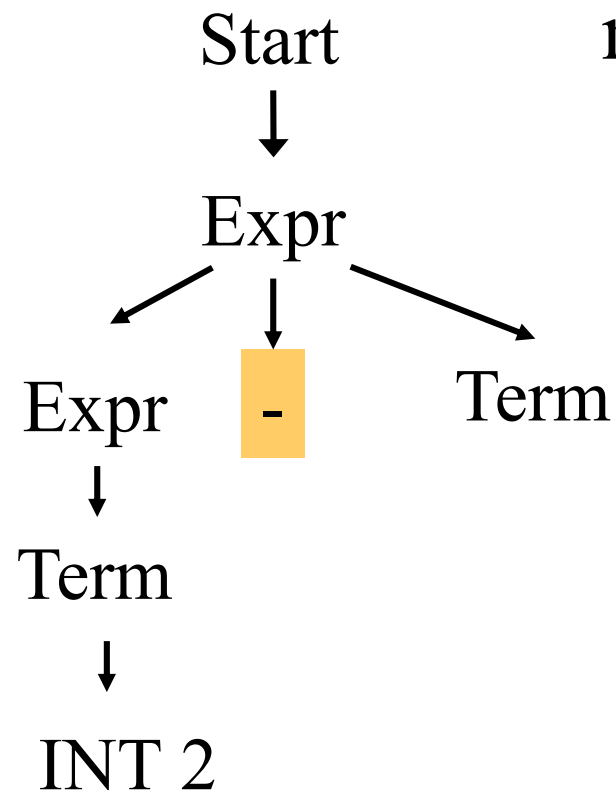
Sentencial form

2 - Term

# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Token  
matches!

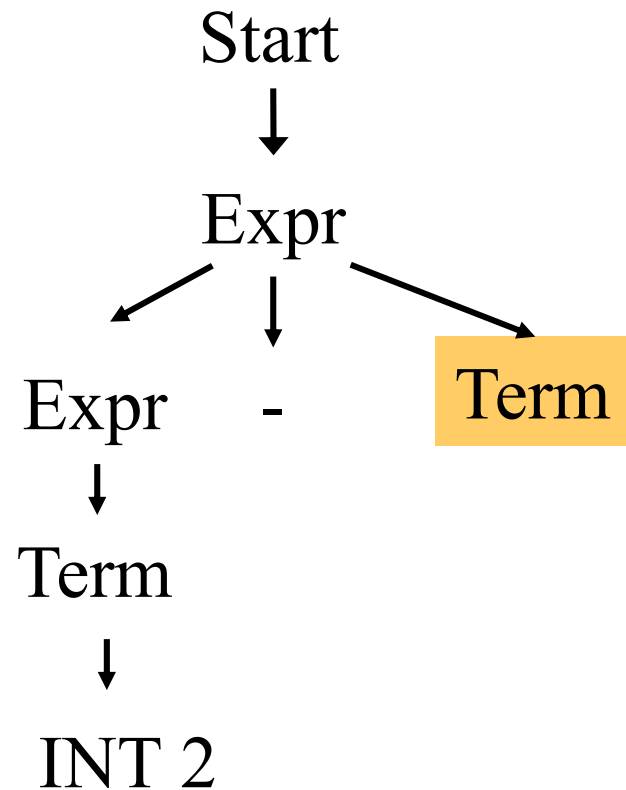
Input not processed  
-2\*2

Sentencial form  
2 - Term

# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Input not processed

$2*2$

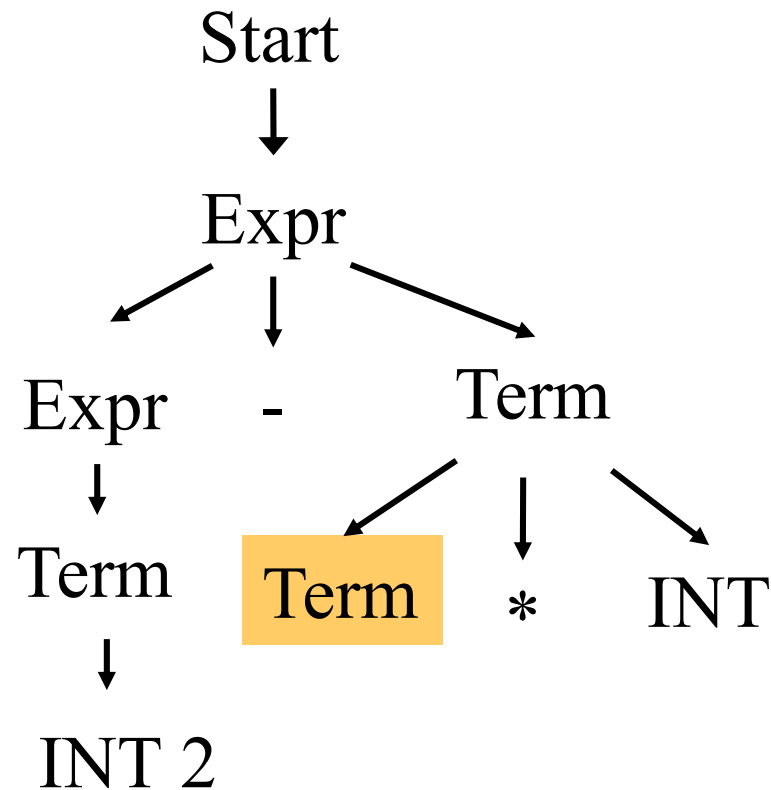
Sentencial form

$2 - \text{Term}$

# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Input not processed

2\*2

Sentential form

2 – Term\*INT

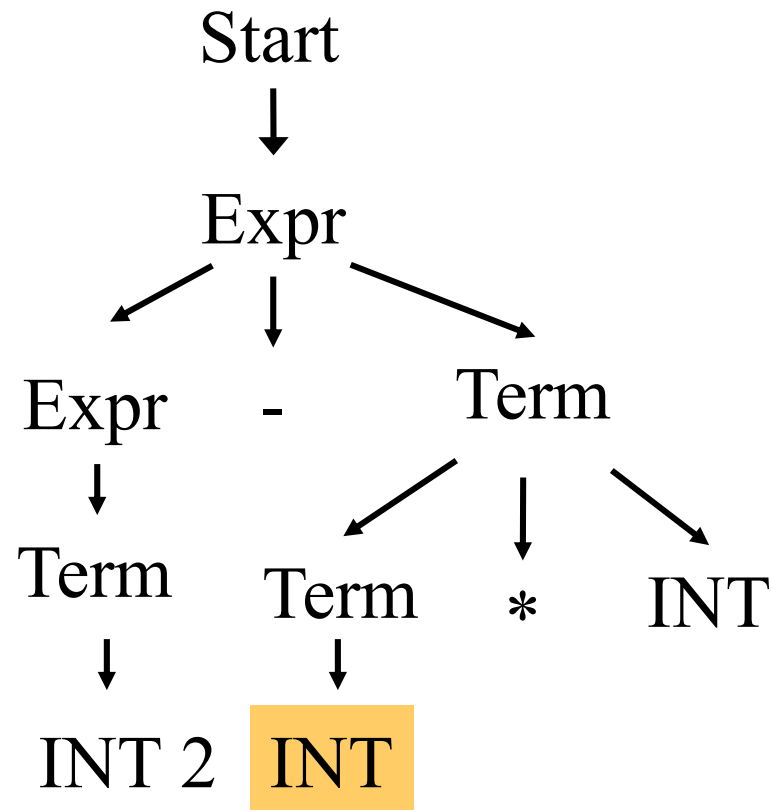
Applied production

Term  $\rightarrow$  Term \* INT

# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Input not processed

$2 * 2$

Sentential form

$2 - \text{INT} * \text{INT}$

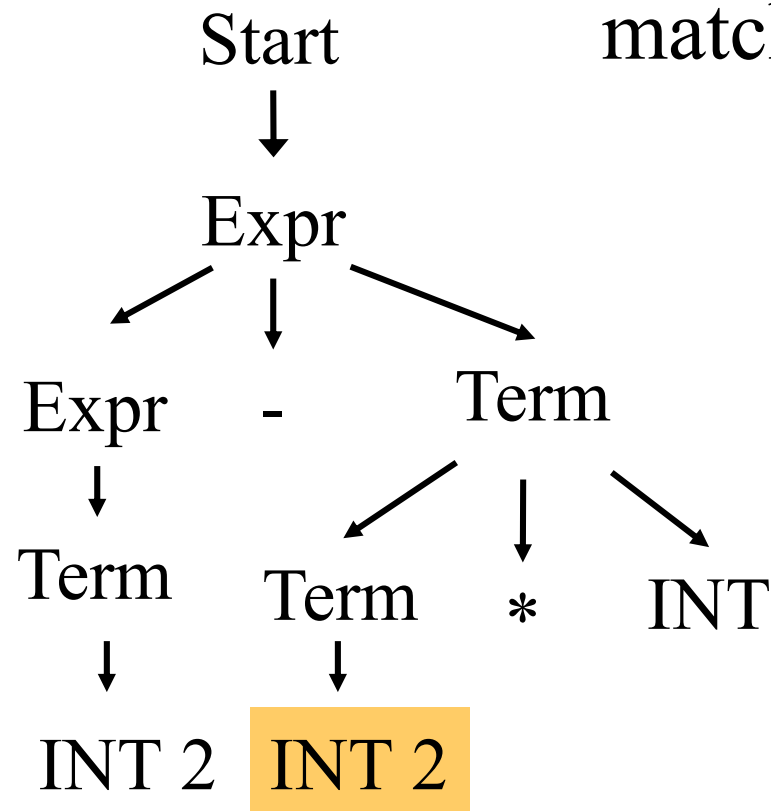
Applied production

$\text{Term} \rightarrow \text{INT}$

# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Token  
matches!

Input not processed  
 $2*2$

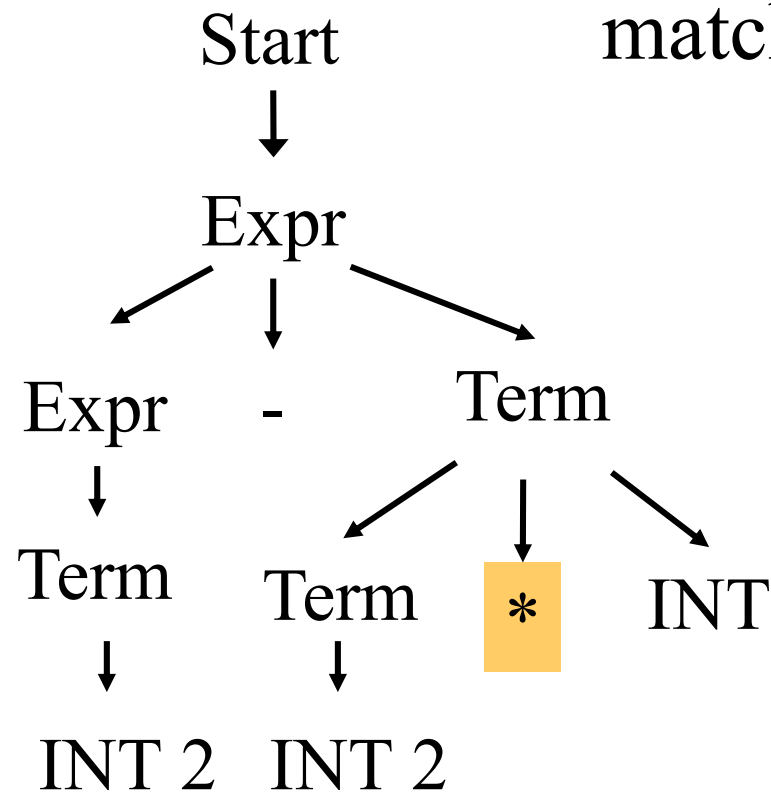
Sentential form  
 $2 - 2*INT$



# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Token  
matches!

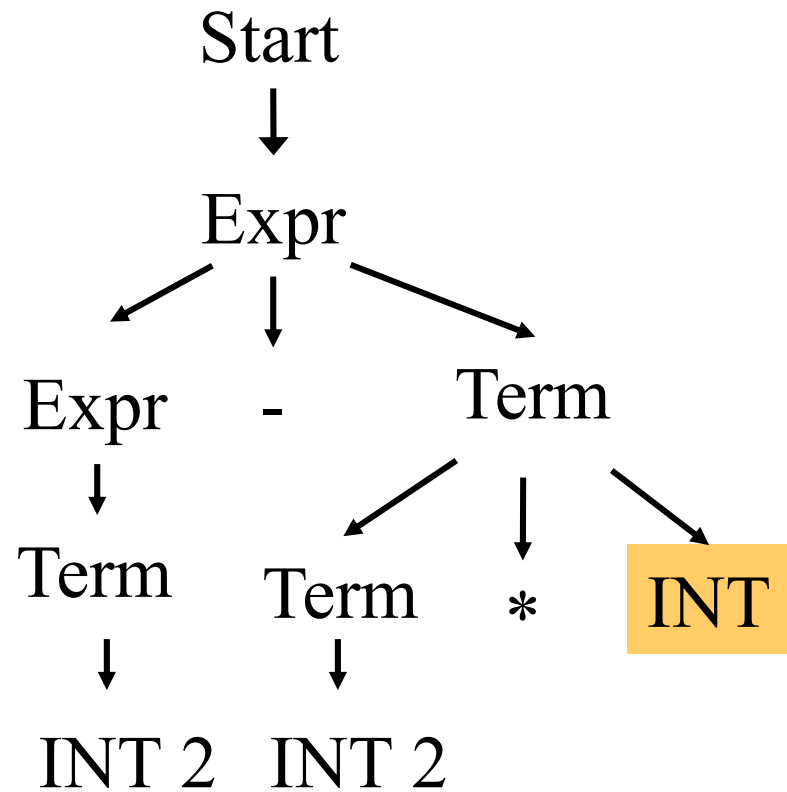
Input not processed  
\*2

Sentential form  
 $2 - 2 * \text{INT}$

# Syntax Analyzer for the Grammar

## Example

Syntax Tree



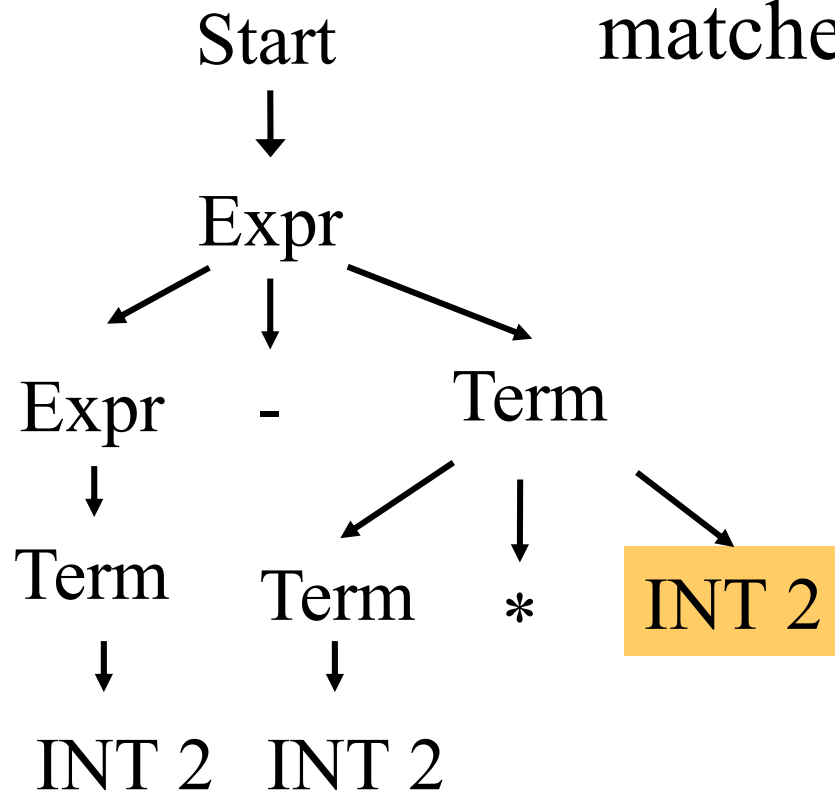
Input not processed  
2

Sentential form  
 $2 - 2 * \text{INT}$

# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Token  
matches!

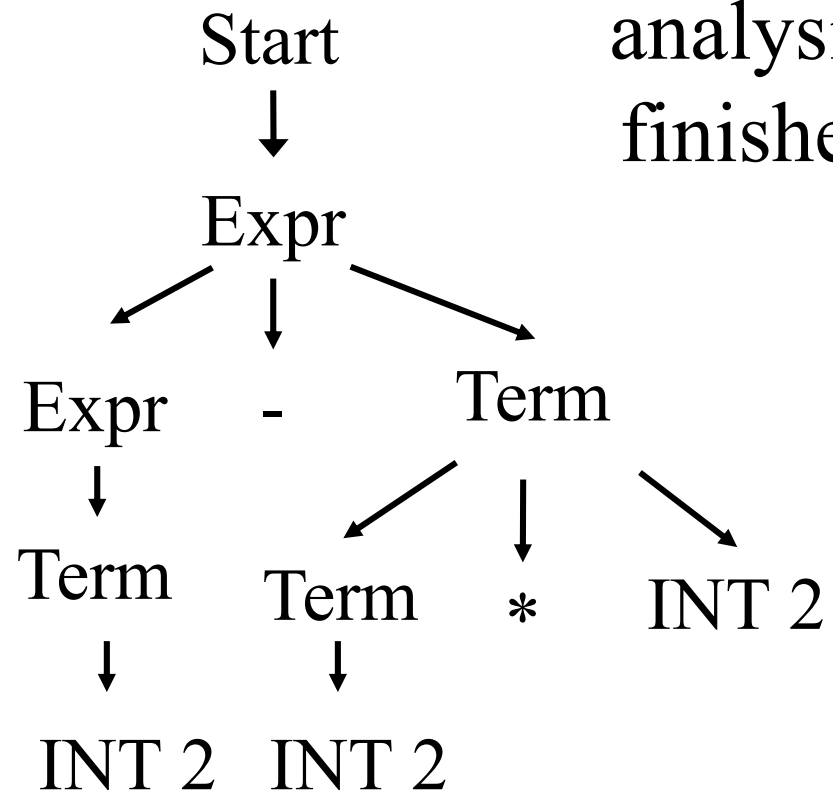
Input not processed  
2

Sentential form  
 $2 - 2 * 2$

# Syntax Analyzer for the Grammar

## Example

Syntax Tree



Syntactic  
analysis  
finished

Input not processed

Sentential form  
 $2 - 2 * 2$

# Summary

- Three actions (mechanisms)
  - Apply production to expand the current non-terminal symbol at the syntax tree
  - Match the current terminal symbol
  - Accept the syntax analysis as correct
- What is the production to be used for each non-terminal symbols?
- An approach: *Backtracking*
  - Try an alternative
  - When it is clear the alternative failed then try another alternative

# Predictive Syntactic Analyzer

- Alternative to *backtracking*
- Very useful for programming languages that can be designed to make easier the analysis
- Basic idea:
  - Lookahead in the sequence of tokens
  - Decision about the production to apply is based on the following tokens
  - We use the token level in the lookahead mechanism

# Grammar Example

Start  $\rightarrow$  Expr

Expr  $\rightarrow$  Term Expr'

Expr'  $\rightarrow$  "+" Term Expr'

Expr'  $\rightarrow$  "-" Term Expr'

Expr'  $\rightarrow \epsilon$

Term  $\rightarrow$  INT Term'

Term'  $\rightarrow$  "\*" INT Term'

Term'  $\rightarrow$  "/" INT Term'

Term'  $\rightarrow \epsilon$

INT = [0-9]<sup>+</sup>

o Set of tokens (terminal symbols):

{ +, -, \*, /, INT }

# Points of Selection

- Assume that Term' is the current position in the syntax tree
- 3 different productions to apply
  - Term'  $\rightarrow$  "\*" INT Term'
  - Term'  $\rightarrow$  "/" INT Term'
  - Term'  $\rightarrow \epsilon$
- Use the next Token to decide
  - If next token is \*, apply Term'  $\rightarrow$  \* Int Term'
  - If next token is /, apply Term'  $\rightarrow$  / Int Term'
  - Otherwise, apply Term'  $\rightarrow \epsilon$



# Multiple Productions with the Same RHS Prefix

- Grammar example:

$N_t \rightarrow \text{IF THEN}$

$N_t \rightarrow \text{IF THEN ELSE}$

- Assume that  $N_t$  is the syntax tree and IF is the next token
- What is the production to apply?

# Solution: Factoring the Grammar

- New grammar include the common prefix in a single production:  
     $Nt \rightarrow \text{IF THEN } Nt'$   
     $Nt' \rightarrow \text{ELSE}$   
     $Nt' \rightarrow \varepsilon$
- None selection when next token is an IF
- Alternatives were unified in a single production

# Non-Terminal Symbols

- And the productions with non-terminal symbols?  
 $Nt \rightarrow Nt_1 \alpha_1$   
 $Nt \rightarrow Nt_2 \alpha_2$
- We have to select based on the first possible terminals  
 $Nt_1$  and  $Nt_2$  can generate
- And if  $Nt_1$  or  $Nt_2$  can generate  $\varepsilon$ ?
  - We have to select based on  $\alpha_1$  and  $\alpha_2$

# Definitions: First() and Follow() Sets

## Notation

$T$  is a terminal,  
 $NT$  is a non-terminal,  
 $S$  is terminal or non-terminal,  
and  $\alpha$  and  $\beta$  represent sequences with terminals and/or non-terminals

- First( $\beta$ ) Set
  - Set of leftmost terminal symbols in all possible derivation trees of  $\beta$
  - $T \in \text{First}(\beta)$  if  $T$  can appear as a first symbol of a derivation string in  $\beta$
  - Start with the concept of  $NT$  deriving  $\varepsilon$ :
    - $NT \rightarrow \varepsilon$  implies that  $NT$  derives  $\varepsilon$
    - $NT \rightarrow NT_1 \dots NT_n$  and if all  $NT_i$  ( $1 \leq i \leq n$ ) derive  $\varepsilon$  implies that  $NT$  derives  $\varepsilon$

# Definitions: Rules for First()

## Notation

$T$  is a terminal,

$NT$  is a non-terminal,

$S$  is terminal or non-terminal,

and  $\alpha$  and  $\beta$  represent sequences with terminals and/or non-terminals

$$1) T \in \text{First}(T)$$

$$2) \text{First}(S) \subseteq \text{First}(S \beta)$$

3)  $NT$  derives  $\varepsilon$  implies:

$$\text{First}(\beta) \subseteq \text{First}(NT \beta)$$

4)  $NT \rightarrow S \beta$  implies:

$$\text{First}(S \beta) \subseteq \text{First}(NT)$$

# Definitions: First() Example

➤ First(Term')?

Grammar

$\text{Term}' \rightarrow * \text{INT Term}'$

$\text{Term}' \rightarrow / \text{INT Term}'$

$\text{Term}' \rightarrow \varepsilon$

Solution

**$\text{First}(\text{Term}') = \{*, /, \varepsilon\}$**

$\text{First}(* \text{INT Term}') = \{*\}$

$\text{First}( / \text{INT Term}') = \{/ \}$

$\text{First}(*) = \{*\}$

$\text{First}( / ) = \{/ \}$

# Definitions: First() Set

- If two or more different productions for the same non-terminal symbol have First sets with common terminal symbols then:
  - The grammar cannot be analysed with a predictive LL(1) parser without backtracking
    - Example:
      - $S \rightarrow X \$$
      - $X \rightarrow a$
      - $X \rightarrow a b$
    - $\text{First}(X \rightarrow a) = \{ a \}$
    - $\text{First}(X \rightarrow a b) = \{ a \}$
    - **Which production to choose when the current symbol is a?**

# Definitions: Follow() Set

- For the non-terminal  $A$ ,  $\text{Follow}(A)$  is the set of the first terminals that can follow after  $A$  in a derivation
- Rules for  $\text{Follow}()$ 
  - $\$ \in \text{Follow}(S)$ , where  $S$  is the start symbol
  - If  $A \rightarrow \alpha B \beta$  is a production then
$$\text{First}(\beta) \subseteq \text{Follow}(B)$$
  - If  $A \rightarrow \alpha B$  is a production then
$$\text{Follow}(A) \subseteq \text{Follow}(B)$$
  - If  $A \rightarrow \alpha B \beta$  is a production and  $\beta$  derives  $\varepsilon$  then
$$\text{Follow}(A) \subseteq \text{Follow}(B)$$



# Definitions: Algorithm for Follow()

for all nonterminals NT

Follow(NT) = {}

Follow(S) = {\$}

while Follow sets keep changing

for all productions  $A \rightarrow \alpha B \beta$

Follow(B) = Follow(B)  $\cup$  First( $\beta$ )

if ( $\beta$  derives  $\varepsilon$ ) Follow(B) = Follow(B)  $\cup$  Follow(A)

for all productions  $A \rightarrow \alpha B$

Follow(B) = Follow(B)  $\cup$  Follow(A)

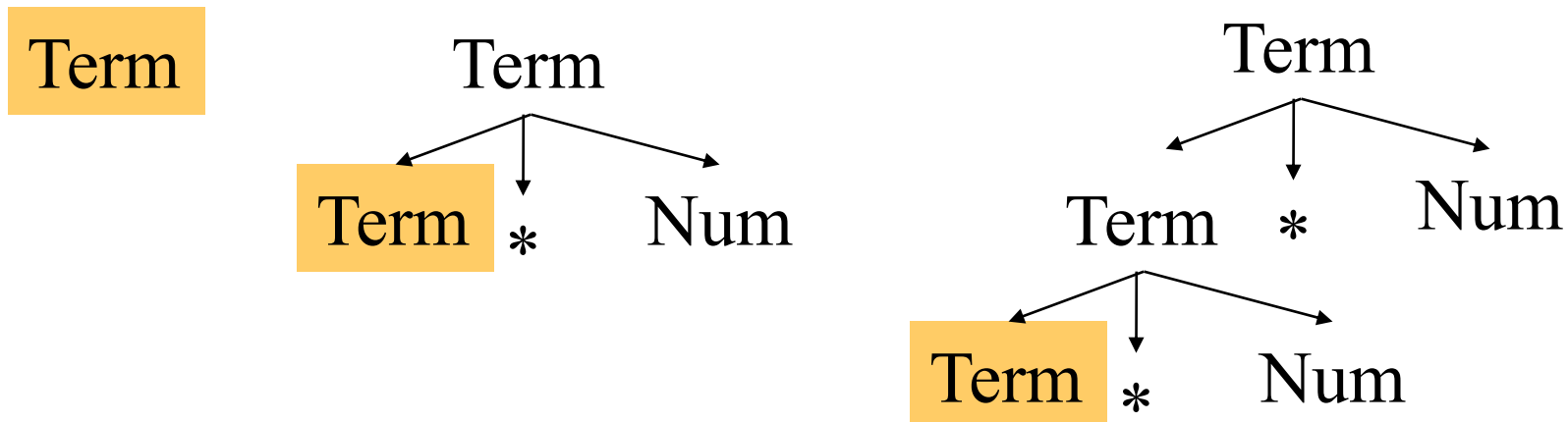
# Definitions: Example Follow()

## ➤ Grammar examples:

- $S \rightarrow X \$$   
 $X \rightarrow a$   
 $X \rightarrow a b$ 
  - $\text{Follow}(S) = \{ \$ \}$
  - $\text{Follow}(X) = \{ \$ \}$
- $S \rightarrow X \$$   
 $X \rightarrow "(X)"$   
 $X \rightarrow \epsilon$ 
  - $\text{Follow}(S) = \{ \$ \}$
  - $\text{Follow}(X) = \{ ")", \$ \}$

# Descendent Syntactic Analyzer

- Left recursion may lead to infinite loops!
- Example of production:
  - $\text{Term} \rightarrow \text{Term} * \text{Num}$
- Potential steps of the analysis:

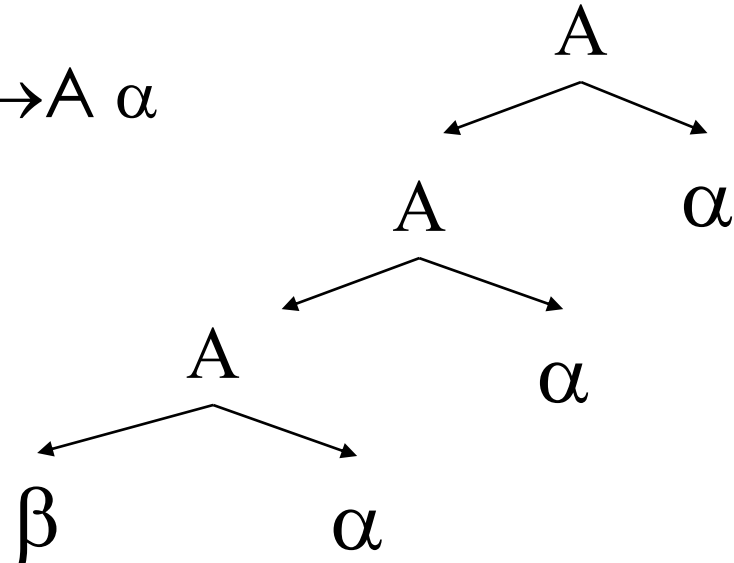


# Descendent Syntactic Analyzer

- Left recursion may lead to infinite loops!
- Solution: modify grammar to eliminate left recursion

# Eliminate Left Recursion

- Given the productions of the form:
  - $A \rightarrow A \alpha$
  - $A \rightarrow \beta$
  - Sequences  $\alpha, \beta$  of terminal and non-terminal symbols which do not start with  $A$
- Repetition of the derivation:  $A \rightarrow A \alpha$  forms the syntax tree:

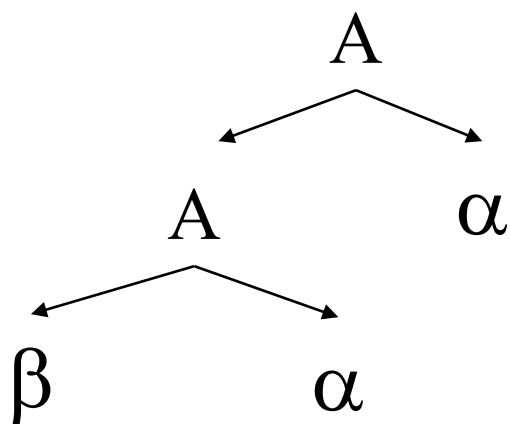


# Eliminate Left Recursion

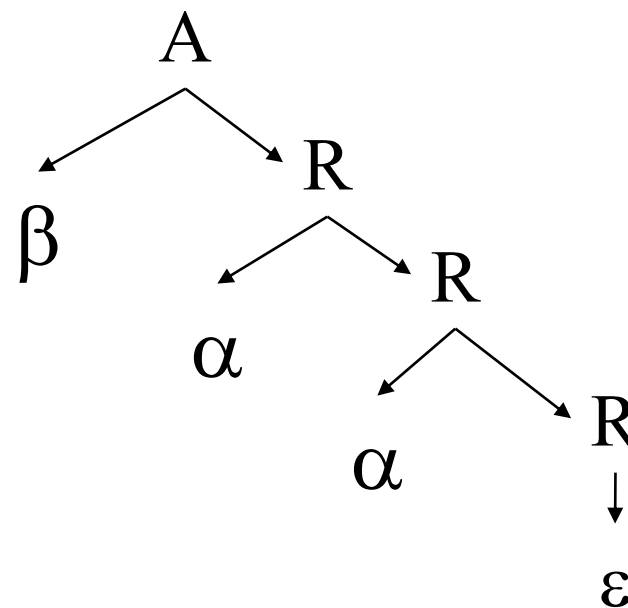
## ➤ Productions for substitutions

- $A \rightarrow A \alpha$      $A \rightarrow \beta R$      $R$  is a new non-terminal symbol
- $A \rightarrow \beta$      $R \rightarrow \alpha R$
- $R \rightarrow \varepsilon$

Initial tree



New tree



# Grammar Example

INT = [0-9]<sup>+</sup>

Start  $\rightarrow$  Expr

Expr  $\rightarrow$  Expr “+” Term

Expr  $\rightarrow$  Expr “-” Term

Expr  $\rightarrow$  Term

Term  $\rightarrow$  Term “\*” INT

Term  $\rightarrow$  Term “/” INT

Term  $\rightarrow$  INT

○ Set of tokens (terminal symbols):

{ +, -, \*, /, INT }

# Modified Grammar

Part of the original grammar

$\text{Term} \rightarrow \text{Term} "*" \text{INT}$

$\text{Term} \rightarrow \text{Term} "/" \text{INT}$

$\text{Term} \rightarrow \text{INT}$

Part of the modified grammar

$\text{Term} \rightarrow \text{INT Term}'$

$\text{Term}' \rightarrow "*" \text{INT Term}'$

$\text{Term}' \rightarrow "/" \text{INT Term}'$

$\text{Term}' \rightarrow \epsilon$



# Modified Grammar

INT = [0-9]<sup>+</sup>

Start → Expr

Expr → Expr “+” Term

Expr → Expr “-” Term

Expr → Term

Term → Term “\*” INT

Term → Term “/” INT

Term → INT

INT = [0-9]<sup>+</sup>

Start → Expr

Expr → Term Expr’

Expr’ → “+” Term Expr’

Expr’ → “-” Term Expr’

Expr’ → ε

Term → INT Term’

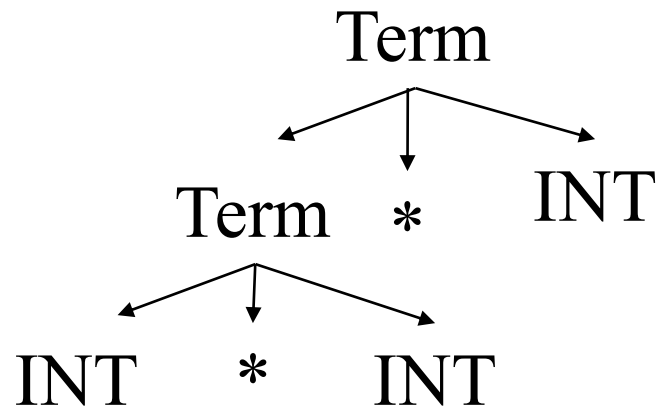
Term’ → “\*” INT Term’

Term’ → “/” INT Term’

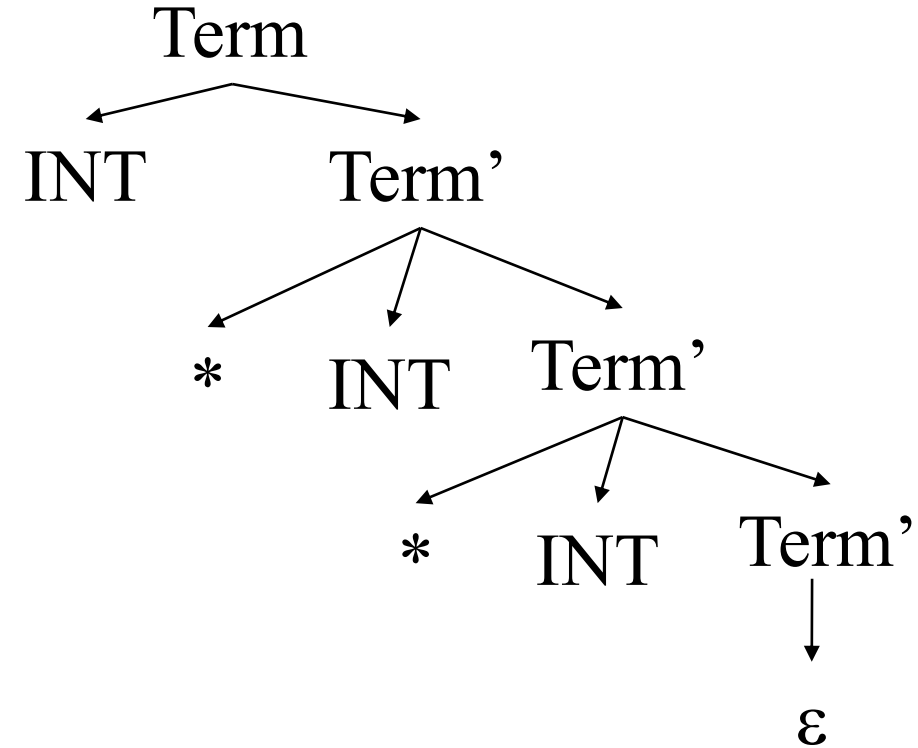
Term’ → ε

# Comparing the Syntax Trees

Original grammar



Modified grammar

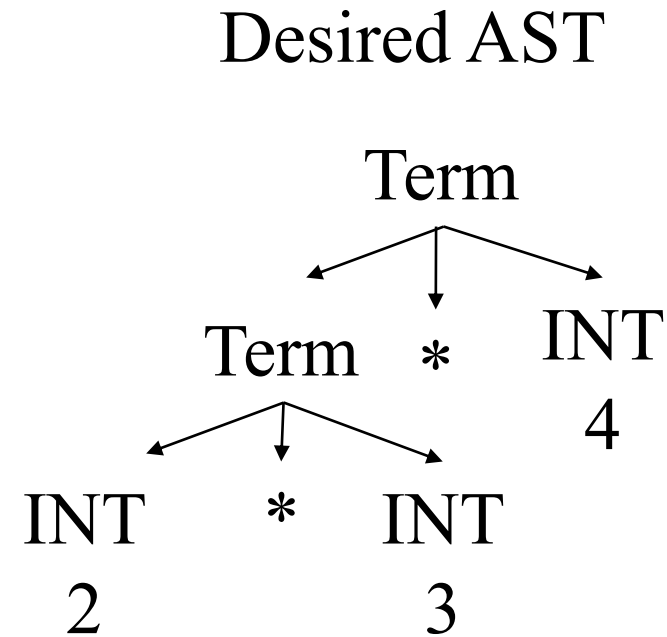
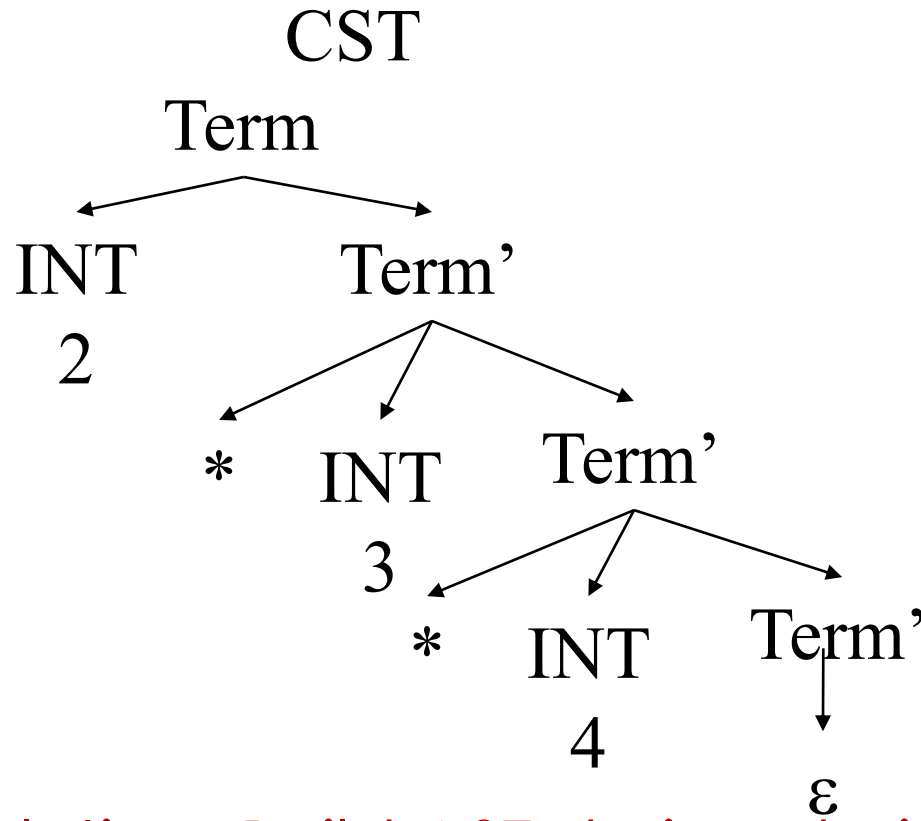


# Eliminate Left Recursion

- Necessary in the predictive syntactic analysis
- Modify the search algorithm in the space of productions
  - Eliminate direct infinite recursion
  - However: modified grammar is less intuitive
- Requires more transformations to achieve the desired AST

# Requires more transformations to achieve the desired AST

- Syntax tree for:  $2*3*4$



- Solution: Build AST during derivation!

# Summary

- Descendent Syntactic Analyser (top down parser)
- Use *Lookahead* to avoid *Backtracking*
- Modify grammar to avoid the necessity to inspect many subsequent tokens (*lookahead*): factorization
- Modify grammar to avoid infinite loops
- How to implement a descendent syntactic analyser?

# Modified Grammar

Part of the original grammar

$\text{Term} \rightarrow \text{Term} "*" \text{INT}$

$\text{Term} \rightarrow \text{Term} "/" \text{INT}$

$\text{Term} \rightarrow \text{INT}$

Part of the modified grammar

$\text{Term} \rightarrow \text{INT Term}'$

$\text{Term}' \rightarrow "*" \text{INT Term}'$

$\text{Term}' \rightarrow "/" \text{INT Term}'$

$\text{Term}' \rightarrow \varepsilon$

# Syntactic Analyser Manually Developed

- One procedure per non-terminal symbol
  - Analyses the current input symbol
  - Calls recursively procedures for the RHS of the selected production
- The procedures return true if the syntactic analysis was well succeeded and false, otherwise

# Example


Productions for the symbol non-terminal Term:

**Term** → **INT Term**'

Function NextToken() steps one token position in the sequence of tokens generated by the lexical analysis and returns the token in the new position.

➤ Procedure for the non-terminal symbol Term:

```
Term() {  
    if (token == INT) {  
        token = NextToken();  
        return TermPrime();  
    } else return false;  
}
```





# Example

- Procedure for the non-terminal symbol Term':

Term  $\rightarrow$  INT Term'

**Term'  $\rightarrow$  "\*" INT Term'**

**Term'  $\rightarrow$  "/" INT Term'**

**Term'  $\rightarrow \epsilon$**

```
TermPrime() {  
    if(token == '*') {  
        token = NextToken();  
        if (token == INT) {  
            token = NextToken();  
            return TermPrime();  
        } else return false;  
    } else if(token == '/') {  
        token = NextToken();  
        if (token == INT) {  
            token = NextToken();  
            return TermPrime();  
        } else return false;  
    }  
    } else return true;  
}
```

```
TermPrime() {  
    if((token == '*') || (token == '/')) {  
        token = NextToken();  
        if (token == INT) {  
            token = NextToken();  
            return TermPrime();  
        } else return false;  
    } else return true;  
}
```

# Example

- Pseudo-code for the part of the program responsible for the syntactic analyser:

...

```
token = NextToken();
```

```
Term();
```

...

# Construction of the Syntax Tree

- Each procedure returns the part of the tree for the part of the String that analysed
- Use exceptions to make clear the code structure (other option is to use an error function)
- Generally, we can use the syntactic analyser algorithm for different goals (besides the recognition or not of the input String)
  - Typically, it produces an AST instead of the CST

# Construction of the Syntax Tree

➤ With generation of exceptions:

```
Term() {  
    if (token == INT) {  
        oldToken = token;  
        token = NextToken();  
        node = TermPrime();  
        if (node == NULL) return oldToken;  
        else return new TermNode(oldToken, node);  
    } else throw SyntaxError;  
}
```

$\text{Term} \rightarrow \text{INT Term}'$

$\text{Term}' \rightarrow "*" \text{INT Term}'$

$\text{Term}' \rightarrow "/" \text{INT Term}'$

$\text{Term}' \rightarrow \varepsilon$

# Construction of the Syntax Tree

- With generation of exceptions:

```
TermPrime() {  
    if ((token == '*' || (token == '/')) {  
        first = token;  
        next = NextToken();  
        if (next == INT) {  
            token = NextToken();  
            return new TermPrimeNode(first, next, TermPrime());  
        } else throw SyntaxError;  
    } else return NULL;  
}
```

$\text{Term} \rightarrow \text{INT Term}'$

$\text{Term}' \rightarrow "*" \text{INT Term}'$

$\text{Term}' \rightarrow "/" \text{INT Term}'$

$\text{Term}' \rightarrow \varepsilon$

# Construction of the Syntax Tree

➤ Without generation of exceptions

```
Term() {  
    if (token == INT) {  
        oldToken = token;  
        token = NextToken();  
        node = TermPrime();  
        if (node == NULL) return oldToken;  
        else return new TermNode(oldToken, node);  
    } else error();  
}  
TermPrime() {  
    if ((token == '*') || (token == '/')) {  
        first = token; next = NextToken();  
        if (next == INT) {  
            token = NextToken();  
            return new TermPrimeNode(first, next, TermPrime());  
        } else error();  
    } else return NULL;  
}
```

$\text{Term} \rightarrow \text{INT Term}'$

$\text{Term}' \rightarrow "*" \text{INT Term}'$

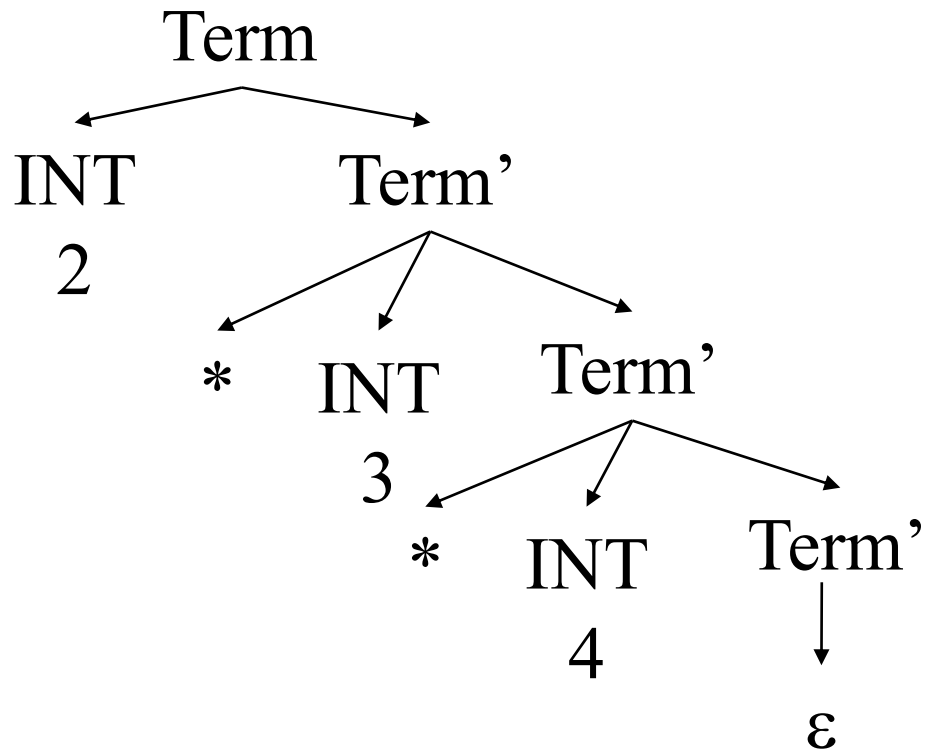
$\text{Term}' \rightarrow "/" \text{INT Term}'$

$\text{Term}' \rightarrow \varepsilon$

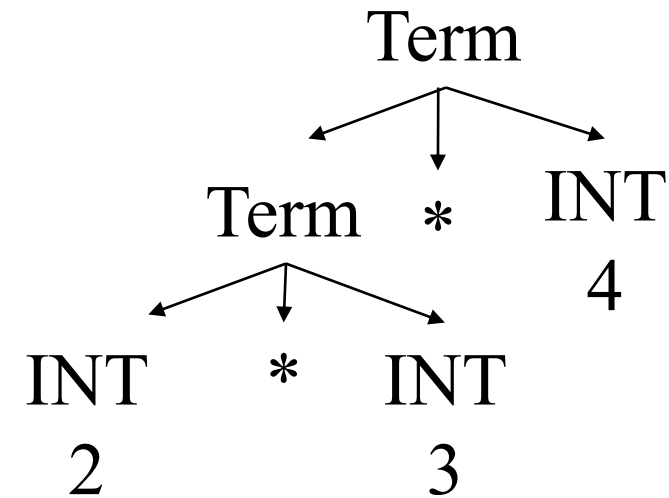
# Syntax Tree for 2\*3\*4

Term  $\rightarrow$  INT Term'  
Term'  $\rightarrow$  "\*" INT Term'  
Term'  $\rightarrow$  "/" INT Term'  
Term'  $\rightarrow \epsilon$

CST

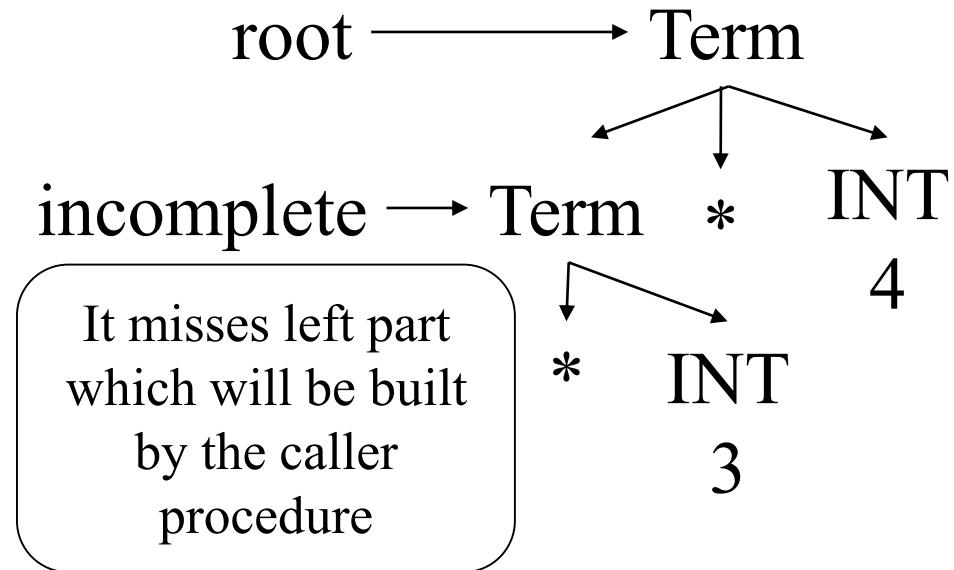


AST desired



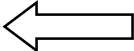
# Direct Generation of the AST

- TermPrime builds an incomplete tree
  - It lacks *leftmost child*
  - Returns the root and the incomplete node
- (root, incomplete) = TermPrime()
  - Call with token: \*
  - Tokens missing: 3 \* 4

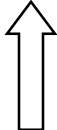





# Code for Term

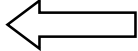
```
Term() {  
    if (token == INT) {   
        leftmostInt = token;  
        token = NextToken();  
        (root, incomplete) = TermPrime();  
        if (root == NULL) return leftmostInt;  
        incomplete.leftChild = leftmostInt;  
        return root;  
    } else throw SyntaxError;  
}
```

Input

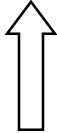
2\*3\*4  


token  INT  
2

# Code for Term

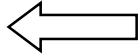
```
Term() {  
    if (token == INT) {  
        leftmostInt = token;   
        token = NextToken();  
        (root, incomplete) = TermPrime();  
        if (root == NULL) return leftmostInt;  
        incomplete.leftChild = leftmostInt;  
        return root;  
    } else throw SyntaxError;  
}
```

Input

2\*3\*4  


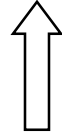
token  $\longrightarrow$   $\begin{matrix} \text{INT} \\ 2 \end{matrix}$

# Code for Term

```
Term() {  
    if (token == INT) {  
        leftmostInt = token;  
        token = NextToken();   
        (root, incomplete) = TermPrime();  
        if (root == NULL) return leftmostInt;  
        incomplete.leftChild = leftmostInt;  
        return root;  
    } else throw SyntaxError;  
}
```

leftmostInt  $\longrightarrow$   $\begin{matrix} \text{INT} \\ 2 \end{matrix}$

Input

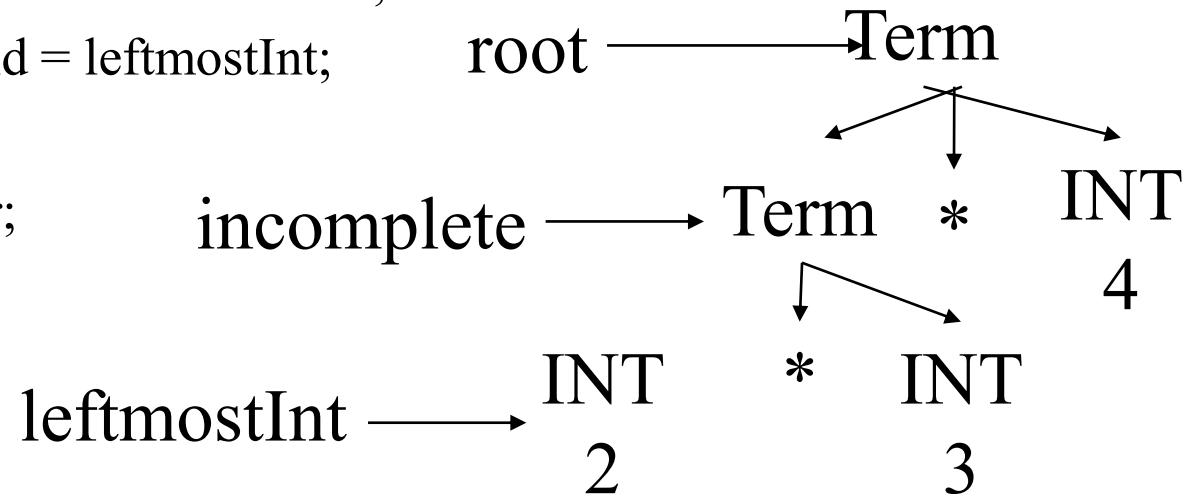
2\*3\*4  


# Code for Term

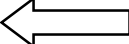
```
Term() {  
    if (token == INT) {  
        leftmostInt = token;  
        token = NextToken();  
        (root, incomplete) = TermPrime();  
        if (root == NULL) return leftmostInt;  
        incomplete.leftChild = leftmostInt;  
        return root;  
    } else throw SyntaxError;  
}
```

Input

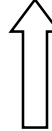
2\*3\*4  
↑

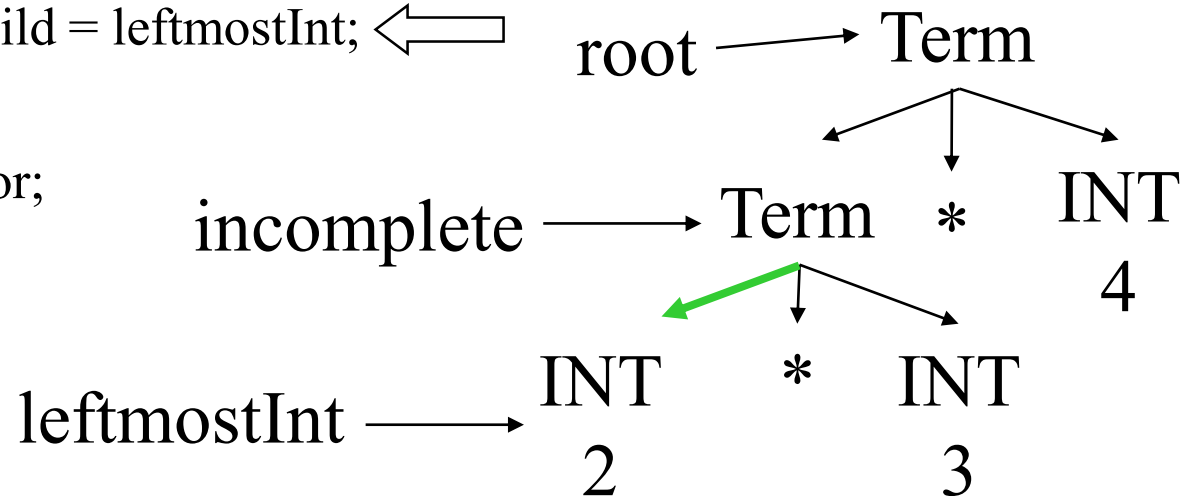


# Code for Term

```
Term() {  
    if (token == INT) {  
        leftmostInt = token;  
        token = NextToken();  
        (root, incomplete) = TermPrime();  
        if (root == NULL) return leftmostInt;  
        incomplete.leftChild = leftmostInt;   
        return root;  
    } else throw SyntaxError;  
}
```

Input

2\*3\*4  


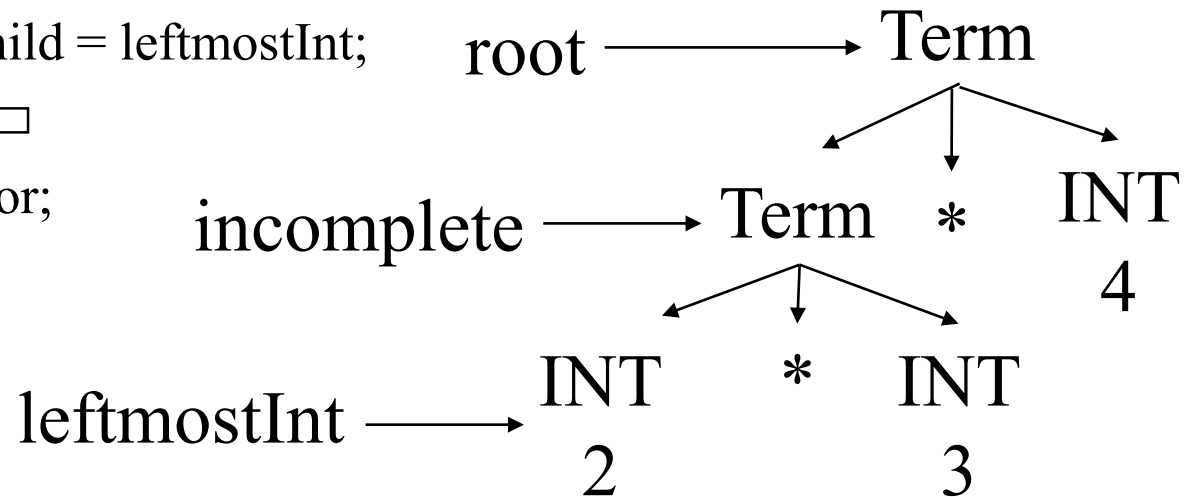


# Code for Term

```
Term() {  
    if (token == INT) {  
        leftmostInt = token;  
        token = NextToken();  
        (root, incomplete) = TermPrime();  
        if (root == NULL) return leftmostInt;  
        incomplete.leftChild = leftmostInt;  
        return root;   
    } else throw SyntaxError;  
}
```

Input

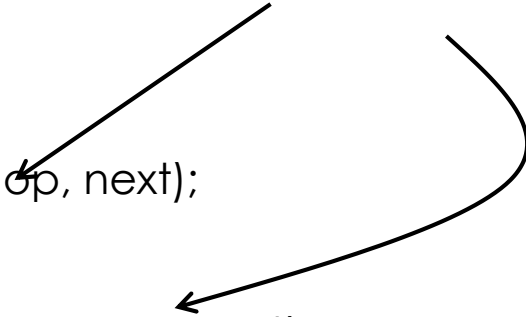
2\*3\*4  
↑



# Code for TermPrime

```
TermPrime() {  
    if((token == '*') || (token == '/')) {  
        op = token;  
        next = NextToken();  
        if (next == INT) {  
            token = NextToken();  
            (root, incomplete) = TermPrime();  
            if (root == NULL) {  
                root = new ExprNode(NULL, op, next);  
                return(root, root);  
            } else {  
                newChild = new ExprNode(NULL, op, next);  
                incomplete.leftChild = newChild;  
                return(root, newChild);  
            }  
        } else throw SyntaxError;  
    } else return(NULL, NULL);  
}
```

Left child to be  
placed by the caller  
procedure



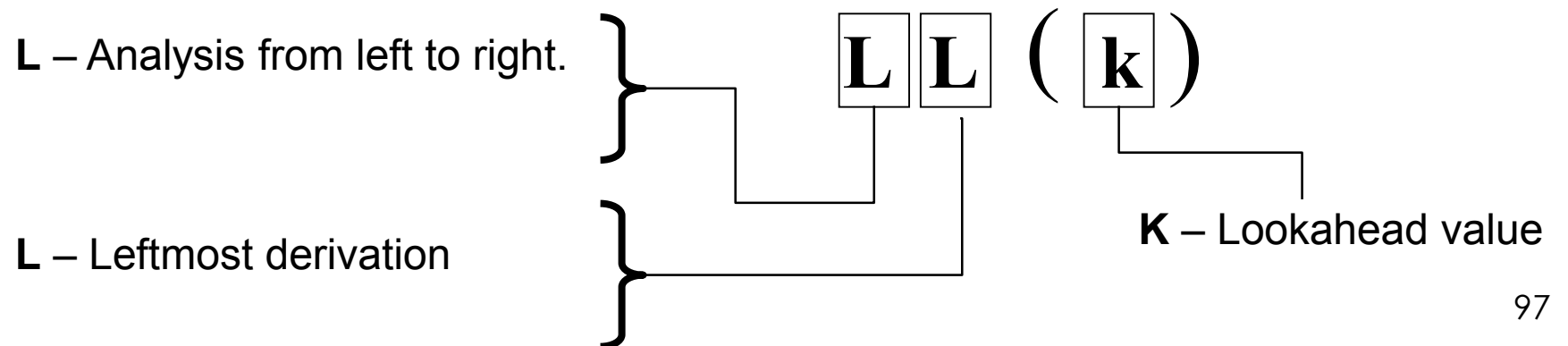
# Summary

- Descendent syntactic analyser (*Top-Down Parser*)
- Use *Lookahead* to avoid *Backtracking*
- The *parser* consists of a set of procedures mutually recursive



# Terminology

- Many techniques for syntactic analysis
  - Each one can handle a set of CFGs (context free grammars)
  - Categorization of the techniques
- Examples: LL(1), LL(2)
- LL(k)
  - Descendent (*top-down*), predictive
  - Construct derivation *leftmost* and from top to bottom



# Classify a Grammar as LL(1)

- How to verify if a grammar is LL(1)?
  - If the syntactic table does not have more than one production in each cell
- Syntactic table of the Predictive Analyzer
  - One row per non-terminal
  - One column per terminal
  - Put  $X \rightarrow \gamma$  in row  $X$ , column  $T$ , for each  $T \in \text{First}(\gamma)$
  - If  $\gamma$  can derive  $\varepsilon$  then put production  $X \rightarrow \gamma$  in row  $X$ , column  $T$ , for each  $T \in \text{Follow}(X)$

# Classify a Grammar as LL(1)

Grammar:

$Z \rightarrow \text{"d"}$

$Z \rightarrow X Y Z$

$Y \rightarrow \varepsilon$

$Y \rightarrow \text{"c"}$

$X \rightarrow Y$

$X \rightarrow \text{"a"}$

- Put production  $X \rightarrow \gamma$  in row  $X$ , column  $T$ , for each  $T \in \text{First}(\gamma)$
- If  $\gamma$  can derive  $\varepsilon$  then put production  $X \rightarrow \gamma$  in row  $X$ , column  $T$ , for each  $T \in \text{Follow}(X)$

| Não-terminais | Terminais |     |     |
|---------------|-----------|-----|-----|
|               | "d"       | "c" | "a" |
| Z             |           |     |     |
| Y             |           |     |     |
| X             |           |     |     |

# Classify a Grammar as LL(1)

Grammar:

$Z \rightarrow \text{"d"}$

$Z \rightarrow X Y Z$

$Y \rightarrow \varepsilon$

$Y \rightarrow \text{"c"}$

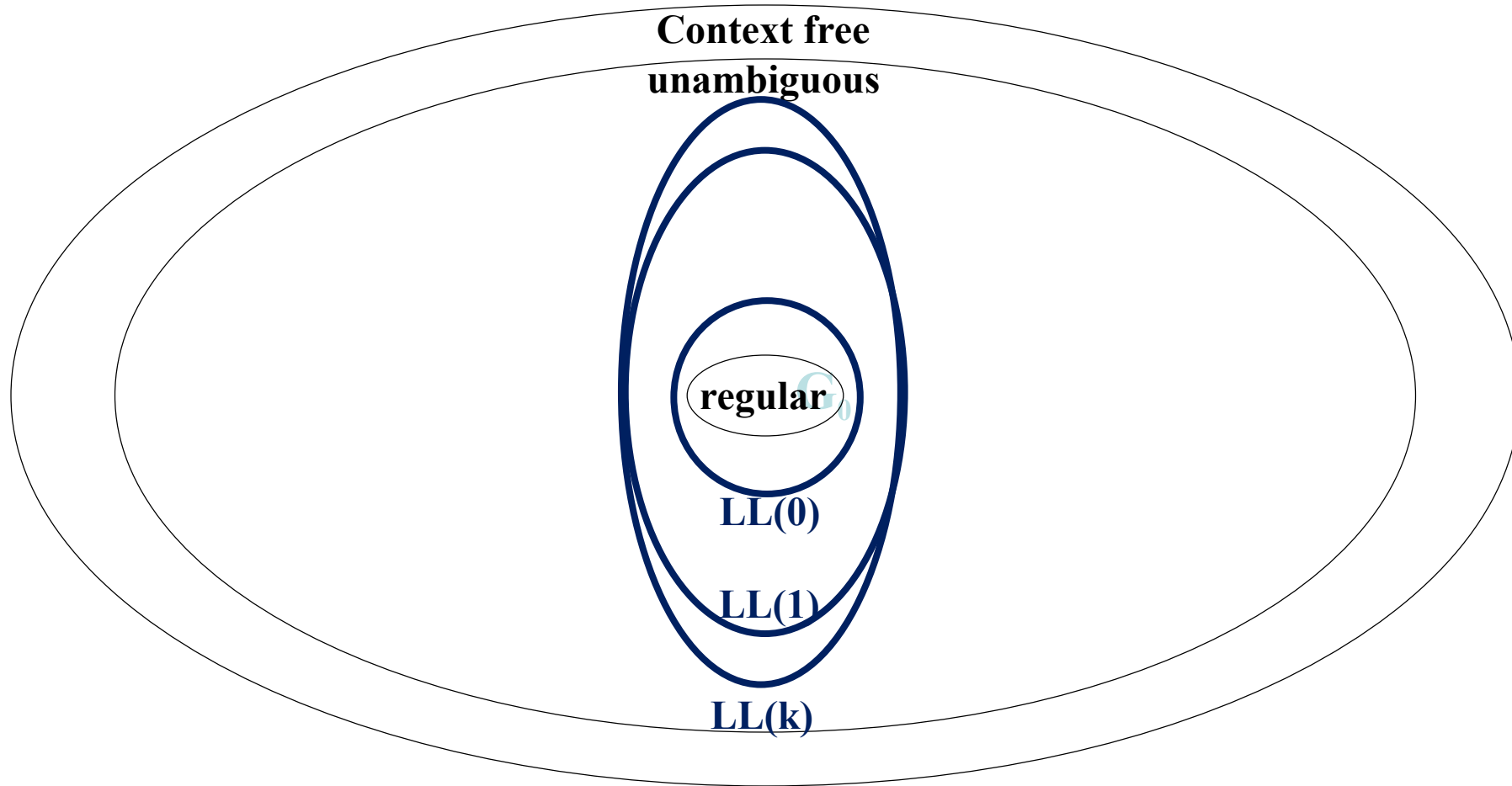
$X \rightarrow Y$

$X \rightarrow \text{"a"}$

- How to verify if a grammar is LL(1)?
  - If the syntactic table does not have more than one production per cell
- This grammar is not LL(1)

| Não-terminais | Terminais   |   |   |
|---------------|---|---|---|
|               | "d"   | "c"   | "a"   |
| <b>Z</b>      | $Z \rightarrow X Y Z$<br>$Z \rightarrow \text{"d"}$ | $Z \rightarrow X Y Z$                                     | $Z \rightarrow X Y Z$                           |
| <b>Y</b>      | $Y \rightarrow \varepsilon$                         | $Y \rightarrow \varepsilon$<br>$Y \rightarrow \text{"c"}$ | $Y \rightarrow \varepsilon$                     |
| <b>X</b>      | $X \rightarrow Y$                                   | $X \rightarrow Y$   | $X \rightarrow Y$<br>$X \rightarrow \text{"a"}$ |

# Grammar Classification



# Lookahead Extensions

- Syntactic Lookahead
- Semantic Lookahead
- Both included in the JavaCC parser generator:
  - Syntactic:

```
LOOKAHEAD("(" Type1() "["  
          "(" Type1() "[" Other1()  
          | "(" Type2() "(" Other2()
```

- Semantic:

```
LOOKAHEAD( { getToken(1).kind == C && getToken(2).kind != C } )  
<C:"c">
```

# Parser Generators

- Generate C, <http://dinosaur.compilertools.net/>
  - Lex & Yacc
  - flex e bison
- Generate Java:
  - JLex e CUP
    - <http://www.cs.princeton.edu/~appel/modern/java/JLex/>
    - <http://www.cs.princeton.edu/~appel/modern/java/CUP/>
  - SableCC, <http://sablecc.org/>
  - JavaCC (version 6 includes C++ generation):  
<http://www.experimentalstuff.com/Technologies/JavaCC/index.html>
- ANTLR Parser Generator (generates Java, C#, JavaScript, Python):
  - <http://www.antlr.org/>
- List with other parser generators
  - <http://catalog.compilertools.net/lexparse.html>
  - <http://catalog.compilertools.net/java.html>