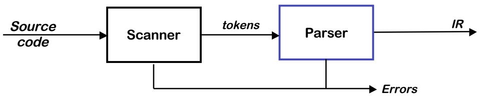
Syntactic Analysis

Introduction

Copyright 2009, Pedro C. Diniz, all rights reserved.
Students enrolled in Compilers class at Instituto Superior Técnico (IST/UTL) have explicit permission to make copies of these materials for their personal use.

The Front End



Parser

- Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

The Study of Parsing

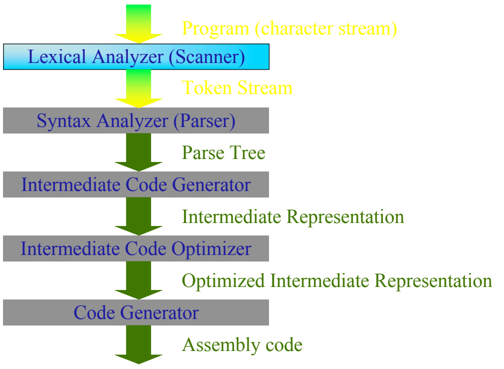
The process of discovering a *derivation* for some sentence

- Need a mathematical model of syntax — a grammar *G*
- Need an algorithm for testing membership in *L(G)*
- Need to keep in mind that our goal is building parsers, not studying the mathematics of arbitrary languages

Outline

- Overview of Lexical Analysis
- What is Syntax Analysis?
- Context-Free Grammars
- Derivation and Parse Trees
- Top-down vs. Bottom-up Parsing
- Ambiguous Grammars
- Implementing a Parser

Anatomy of a Compiler



Program (character stream)

Lexical Analyzer (Scanner)

Token Stream

Syntax Analyzer (Parser)

Parse Tree

Intermediate Code Generator

Intermediate Representation

Intermediate Code Optimizer

Optimized Intermediate Representation

Code Generator

Assembly code

Overview of Lexical Analysis

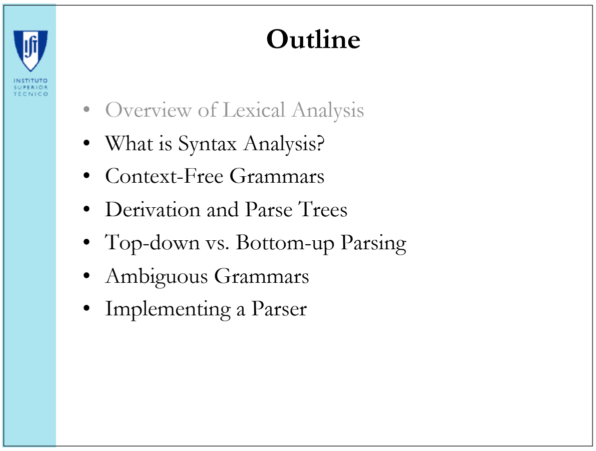
- Lexical Analyzer Creates Tokens out of a Text Stream
- Tokens are defined using Regular Expressions

Regular Expressions, Grammars and Languages

- A regular expression can be written using only:
  - characters in the alphabet
  - regular expression operators: ‘*’ ‘ · ’ ‘|’ ‘+’ ‘?’ ‘(‘ ‘)’
  - Example:
    (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ · (. ·(0|1|2|3|4|5|6|7|8|9)*)?
- Regular language is a language defined by a regular expression

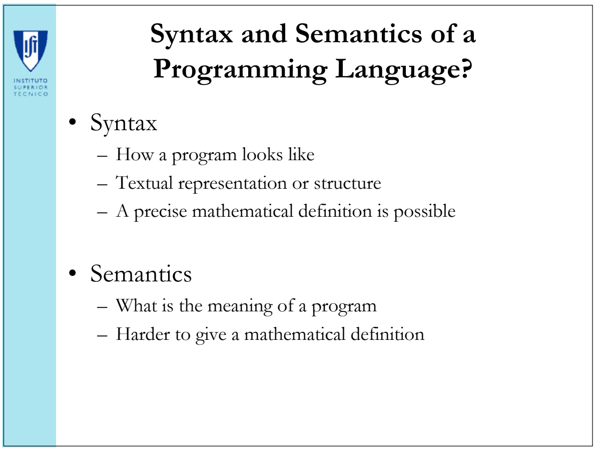
Regular Expressions, Grammars and Languages

- What about symbolic variables?
  - Example:
    num = 0|1|2|3|4|5|6|7|8|9
    posint = *num* · *num**
    int = (ϵ | -) · *posint*
    real = *int* · (ϵ | (. · *posint*))
- They are just shorthand or “syntactic sugar”
  - Example:
    (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ · (. ·(0|1|2|3|4|5|6|7|8|9)*)?



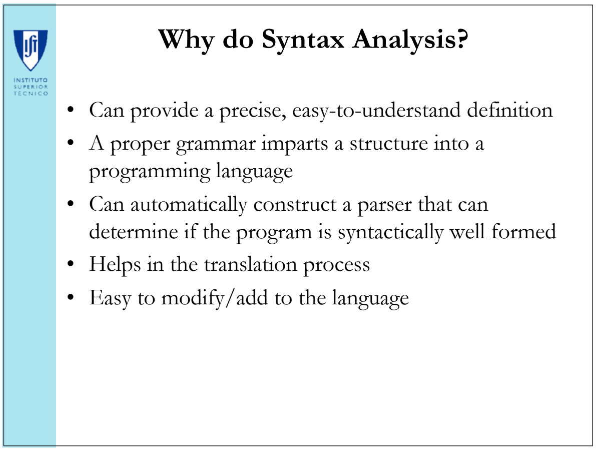
Outline

- Overview of Lexical Analysis
- What is Syntax Analysis?
- Context-Free Grammars
- Derivation and Parse Trees
- Top-down vs. Bottom-up Parsing
- Ambiguous Grammars
- Implementing a Parser



Syntax and Semantics of a Programming Language?

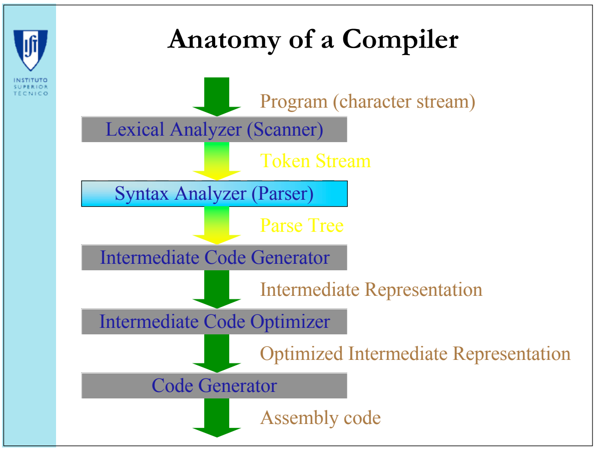
- Syntax
  - How a program looks like
  - Textual representation or structure
  - A precise mathematical definition is possible
- Semantics
  - What is the meaning of a program
  - Harder to give a mathematical definition

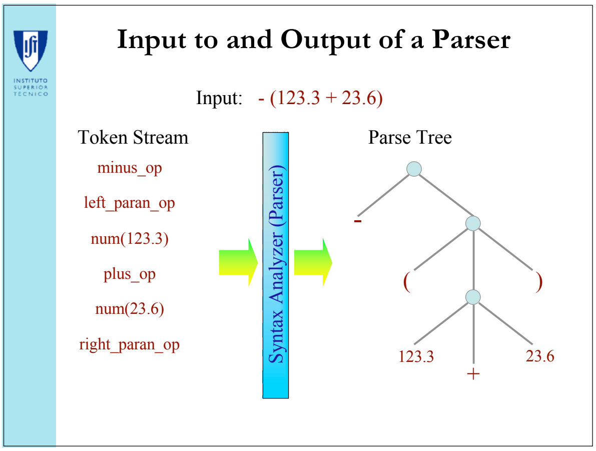


Why do Syntax Analysis?

- Can provide a precise, easy-to-understand definition
- A proper grammar imparts a structure into a programming language
- Can automatically construct a parser that can determine if the program is syntactically well formed
- Helps in the translation process
- Easy to modify/add to the language

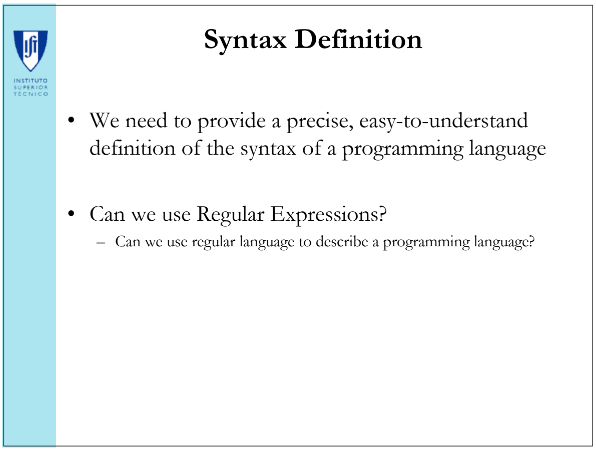
Anatomy of a Compiler


Program (character stream)
Lexical Analyzer (Scanner)
Token Stream
Syntax Analyzer (Parser)
Parse Tree
Intermediate Code Generator
Intermediate Representation
Intermediate Code Optimizer
Optimized Intermediate Representation
Code Generator
Assembly code



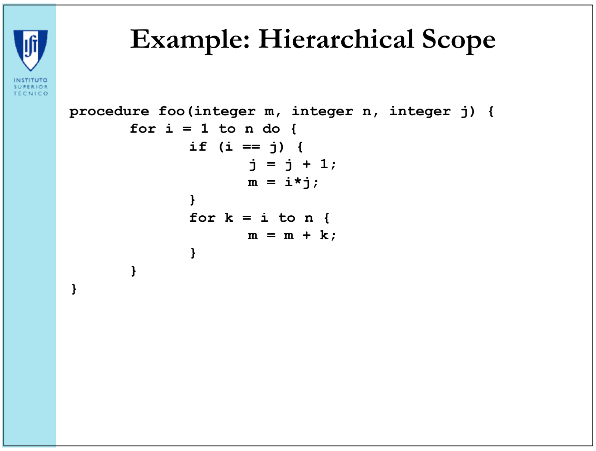
Input to and Output of a Parser

Input: - (123.3 + 23.6)



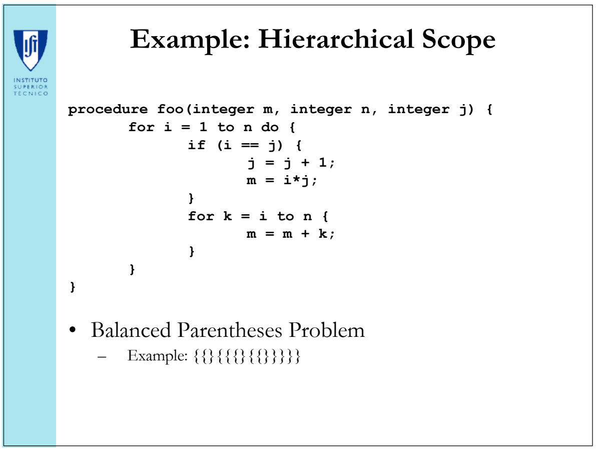
Syntax Definition

- We need to provide a precise, easy-to-understand definition of the syntax of a programming language
- Can we use Regular Expressions?
  - Can we use regular language to describe a programming language?



Example: Hierarchical Scope

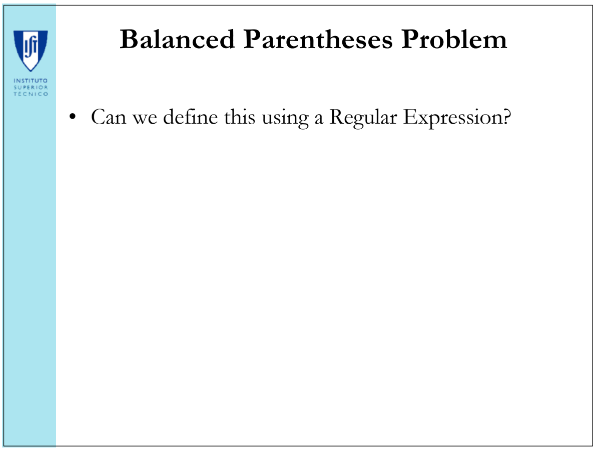
```
procedure foo(integer m, integer n, integer j) {
        for i = 1 to n do {
                if (i == j) {
                        j = j + 1;
                        m = i*j;
                }
                for k = i to n {
                        m = m + k;
                }
        }
}
```



Example: Hierarchical Scope

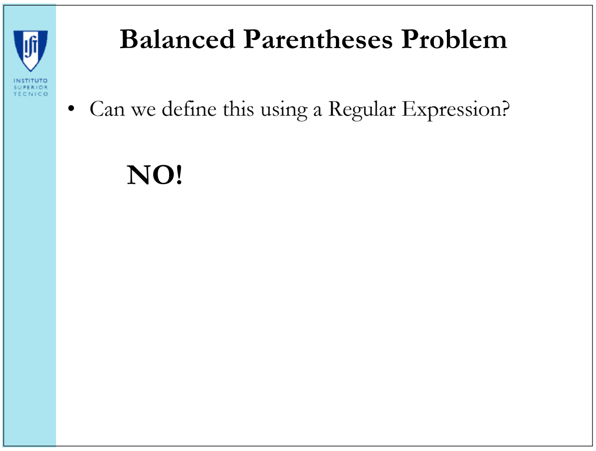
```
procedure foo(integer m, integer n, integer j) {
        for i = 1 to n do {
                if (i == j) {
                        j = j + 1;
                        m = i*j;
                }
                for k = i to n {
                        m = m + k;
                }
        }
}
```

- Balanced Parentheses Problem
  - Example: {{}{{{}{{}}}}}



Balanced Parentheses Problem

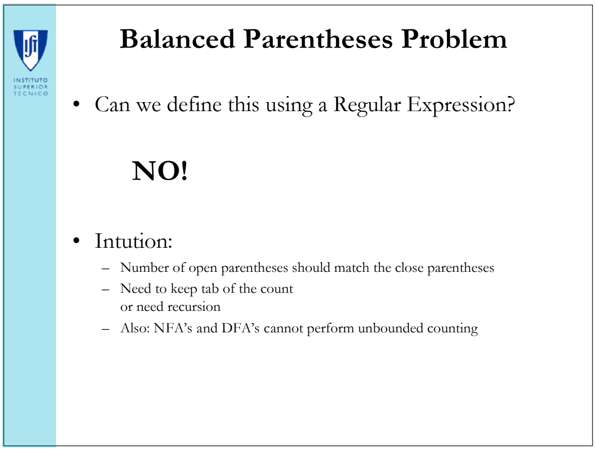
- Can we define this using a Regular Expression?



Balanced Parentheses Problem

- Can we define this using a Regular Expression?

NO!

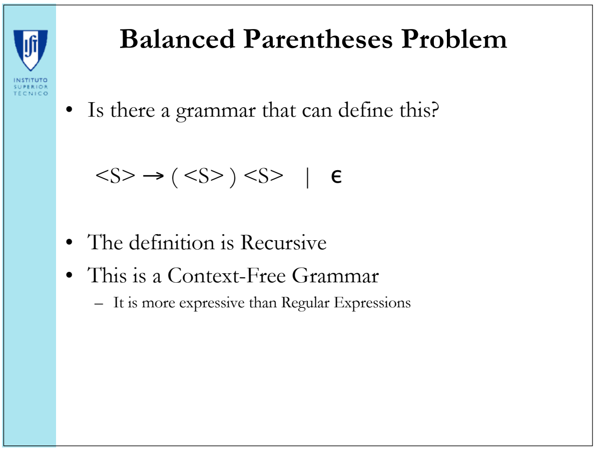


Balanced Parentheses Problem

- Can we define this using a Regular Expression?

NO!

- Intution:
  - Number of open parentheses should match the close parentheses
  - Need to keep tab of the count
    or need recursion
  - Also: NFA's and DFA's cannot perform unbounded counting

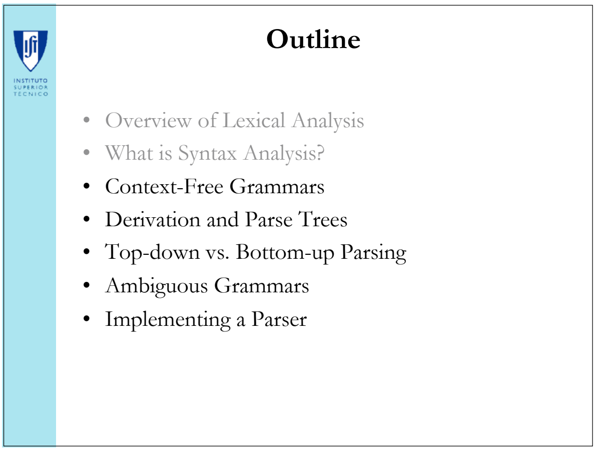


Balanced Parentheses Problem

- Is there a grammar that can define this?

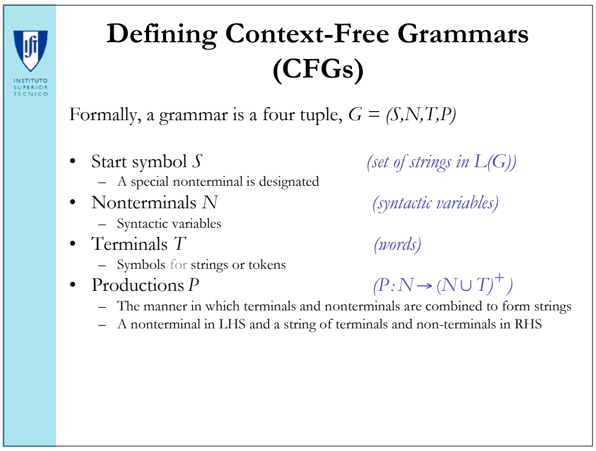
<S> → (<S>) <S> | ϵ

- The definition is Recursive
- This is a Context-Free Grammar
  - It is more expressive than Regular Expressions



Outline

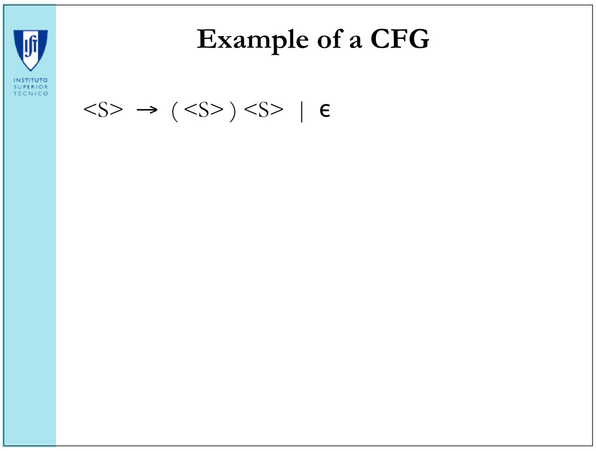
- Overview of Lexical Analysis
- What is Syntax Analysis?
- Context-Free Grammars
- Derivation and Parse Trees
- Top-down vs. Bottom-up Parsing
- Ambiguous Grammars
- Implementing a Parser



Defining Context-Free Grammars (CFGs)

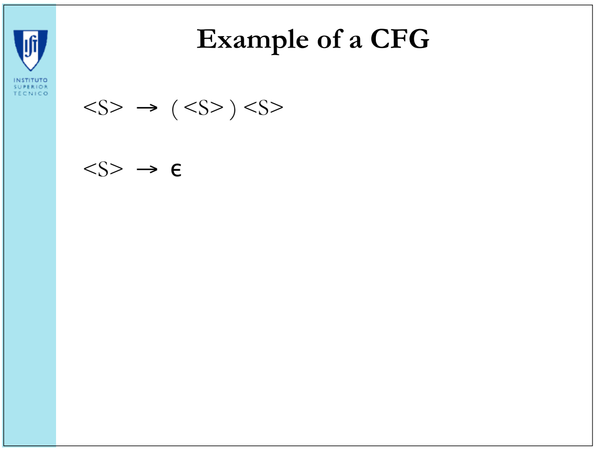
Formally, a grammar is a four tuple, *G = (S,N,T,P)*

- Start symbol *S* *(set of strings in L(G))*
  - A special nonterminal is designated
- Nonterminals *N* *(syntactic variables)*
  - Syntactic variables
- Terminals *T* *(words)*
  - Symbols for strings or tokens
- Productions *P* $(P: N \rightarrow (N \cup T)^{+})$
  - The manner in which terminals and nonterminals are combined to form strings
  - A nonterminal in LHS and a string of terminals and non-terminals in RHS



Example of a CFG

<S> → (<S>) <S> | ϵ

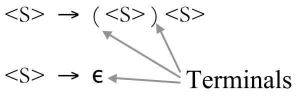


Example of a CFG

<S> → (<S>) <S>

<S> → ϵ

Example of a CFG

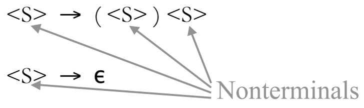


<S> → (<S>) <S>

<S> → ϵ

Terminals

Example of a CFG



<S> → (<S>) <S>

<S> → ϵ

Nonterminals

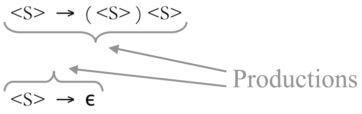
Example of a CFG

<S> → (<S>) <S>

<S> → ϵ

Start Symbol: <S>

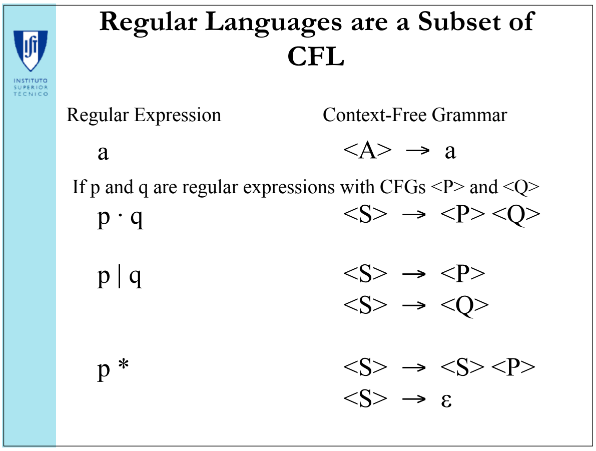
Example of a CFG



<S> → (<S>) <S>

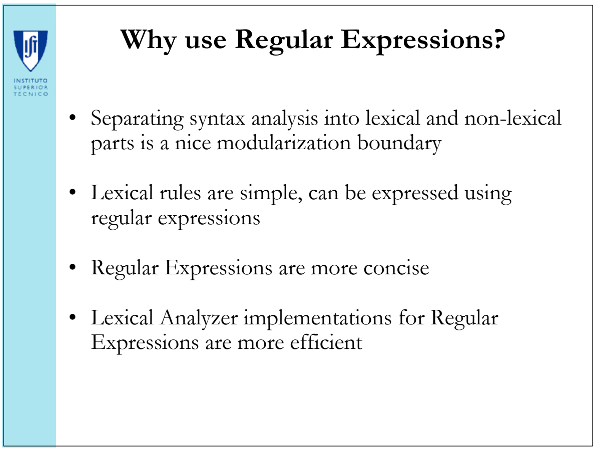
<S> → ϵ

Productions



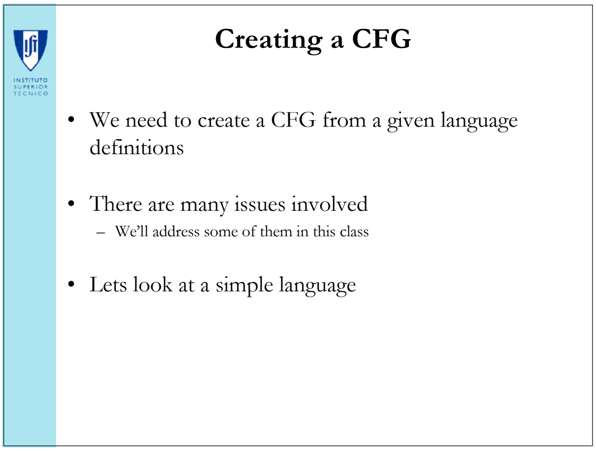
Regular Languages are a Subset of CFL

Regular Expression	Context-Free Grammar
a	<A> → a
If p and q are regular expressions with CFGs <P> and <Q>	
p · q	<S> → <P> <Q>
p | q	<S> → <P> <S> → <Q>
p *	<S> → <S> <P> <S> → ε



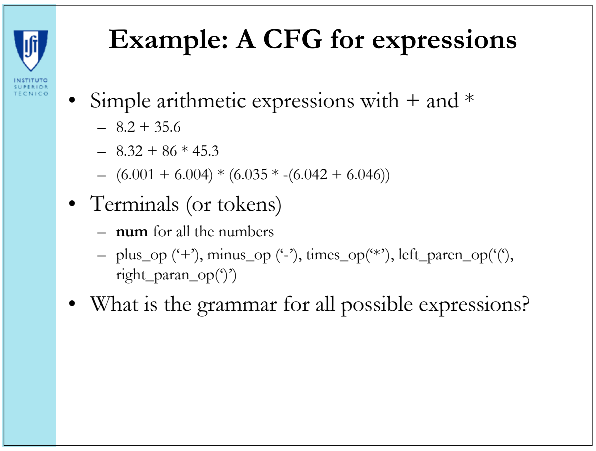
Why use Regular Expressions?

- Separating syntax analysis into lexical and non-lexical parts is a nice modularization boundary
- Lexical rules are simple, can be expressed using regular expressions
- Regular Expressions are more concise
- Lexical Analyzer implementations for Regular Expressions are more efficient



Creating a CFG

- We need to create a CFG from a given language definitions
- There are many issues involved
  - We'll address some of them in this class
- Lets look at a simple language



Example: A CFG for expressions

- Simple arithmetic expressions with + and *
  - 8.2 + 35.6
  - 8.32 + 86 * 45.3
  - (6.001 + 6.004) * (6.035 * -(6.042 + 6.046))
- Terminals (or tokens)
  - **num** for all the numbers
  - plus_op ('+'), minus_op ('-'), times_op('*'), left_paren_op('('), right_paran_op(')')
- What is the grammar for all possible expressions?

Example: A CFG for expressions

<expr> → <expr> <op> <expr>
<expr> → (<expr>)
<expr> → - <expr>
<expr> → num
<op> → +
<op> → *

Example: A CFG for expressions

<expr> → <expr> <op> <expr>
<expr> → **(** <expr> **)**
<expr> → **-** <expr>
<expr> → **num**
<op> → **+**
<op> → *

Terminals

Example: A CFG for expressions

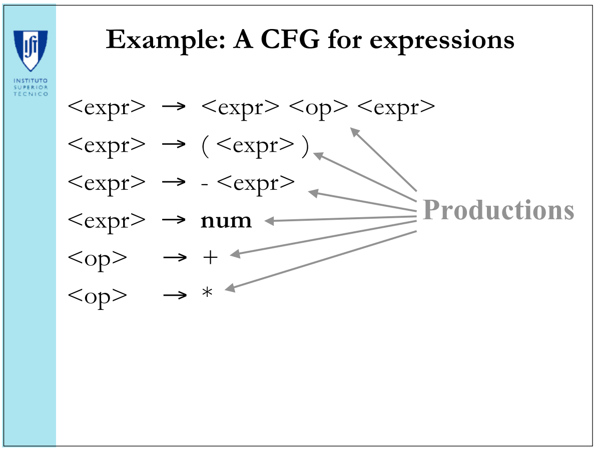
<**expr**> → <**expr**> <**op**> <**expr**>
<**expr**> → (<**expr**>)
<**expr**> → - <**expr**>
<**expr**> → num
<**op**> → +
<**op**> → *

Nonterminals

Example: A CFG for expressions

<expr> → <expr> <op> <expr>
<expr> → (<expr>)
<expr> → - <expr>
<expr> → num
<op> → +
<op> → *

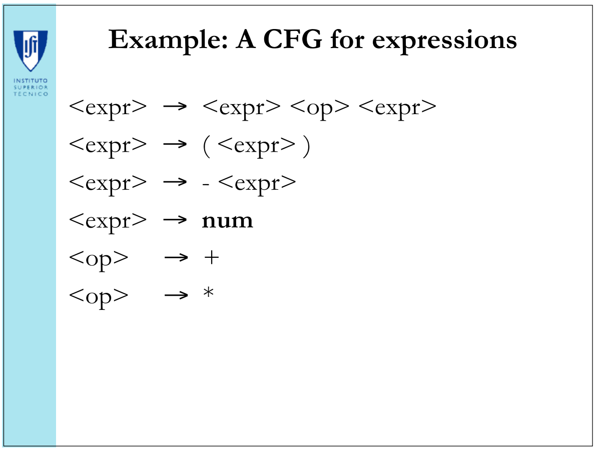
Start Symbol:
<expr>



Example: A CFG for expressions

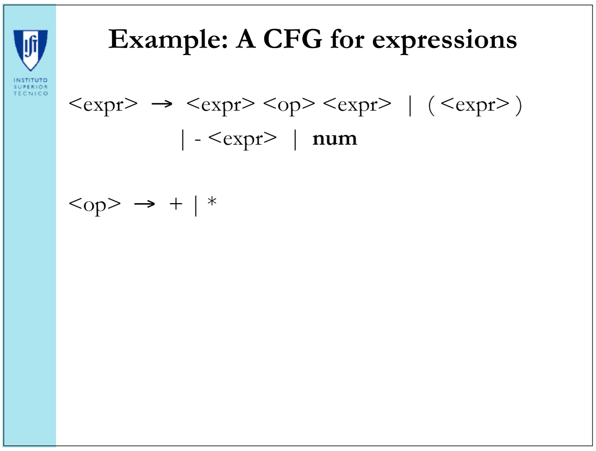
<expr> → <expr> <op> <expr>
<expr> → (<expr>)
<expr> → - <expr>
<expr> → **num**
<op> → +
<op> → *

Productions



Example: A CFG for expressions

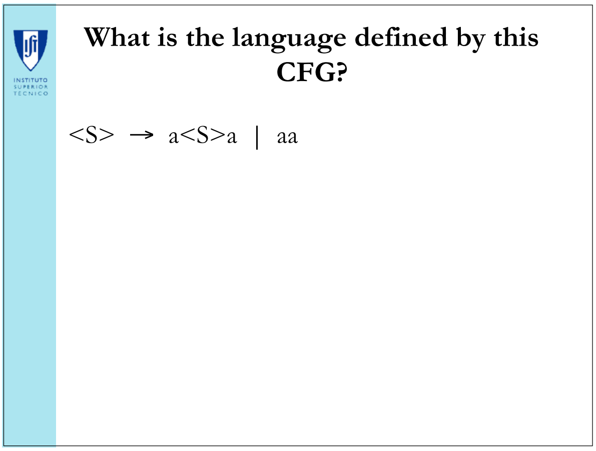
<expr> → <expr> <op> <expr>
<expr> → (<expr>)
<expr> → - <expr>
<expr> → **num**
<op> → +
<op> → *



Example: A CFG for expressions

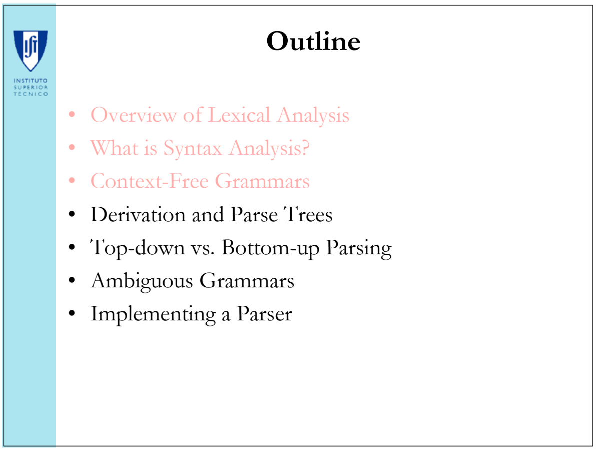
<expr> → <expr> <op> <expr> | (<expr>)
| - <expr> | **num**

<op> → + | *



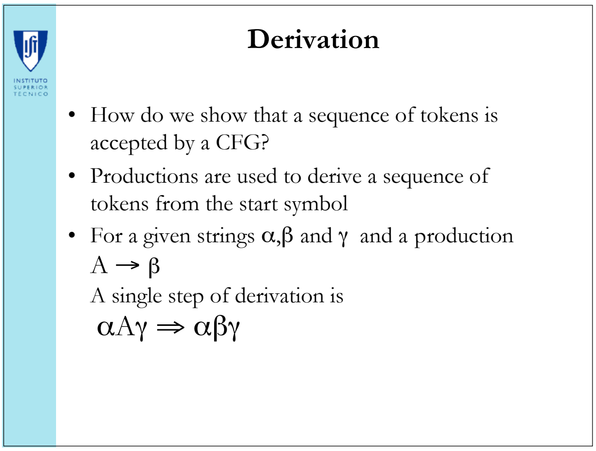
What is the language defined by this CFG?

<S> → a<S>a | aa



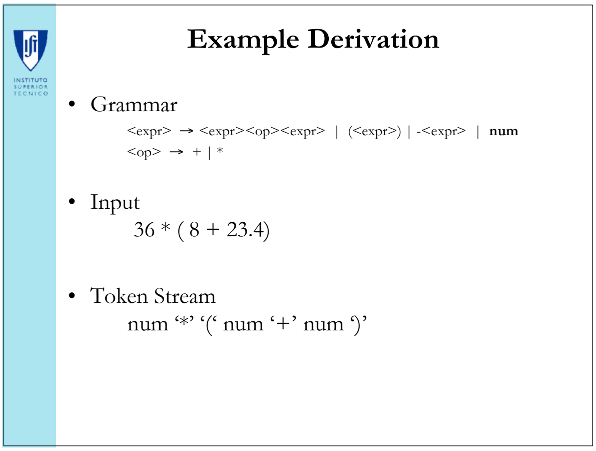
Outline

- Overview of Lexical Analysis
- What is Syntax Analysis?
- Context-Free Grammars
- Derivation and Parse Trees
- Top-down vs. Bottom-up Parsing
- Ambiguous Grammars
- Implementing a Parser



Derivation

- How do we show that a sequence of tokens is accepted by a CFG?
- Productions are used to derive a sequence of tokens from the start symbol
- For a given strings α,β and γ and a production $A \rightarrow \beta$
  A single step of derivation is
  $\alpha A\gamma \Rightarrow \alpha\beta\gamma$

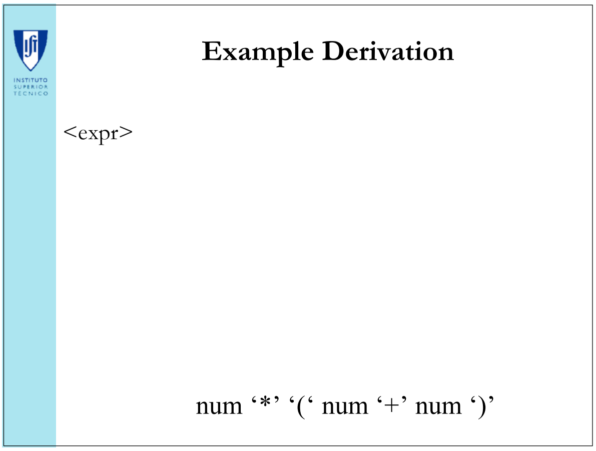


Example Derivation

- Grammar
  <expr> → <expr><op><expr> | (<expr>) | -<expr> | **num**
  <op> → + | *

- Input
  36 * (8 + 23.4)

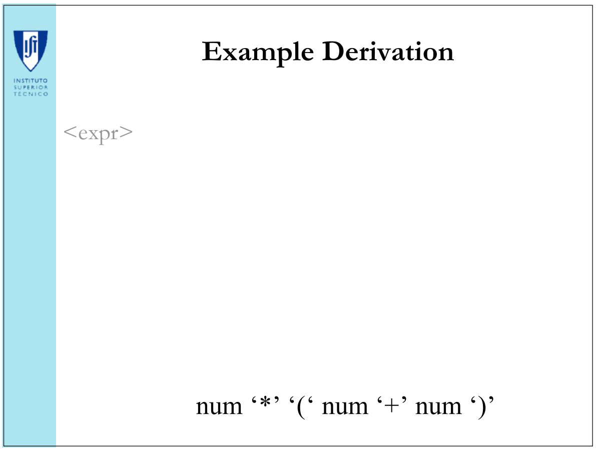
- Token Stream
  num '*' '(' num '+' num ')'



Example Derivation

<expr>

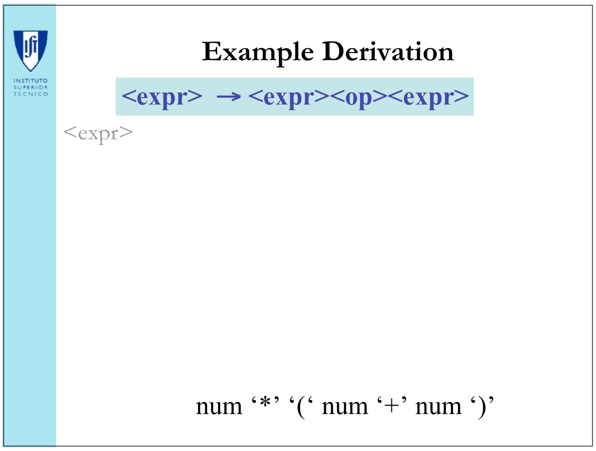
num '*' '(' num '+' num ')'



Example Derivation

<expr>

num '*' '(' num '+' num ')'

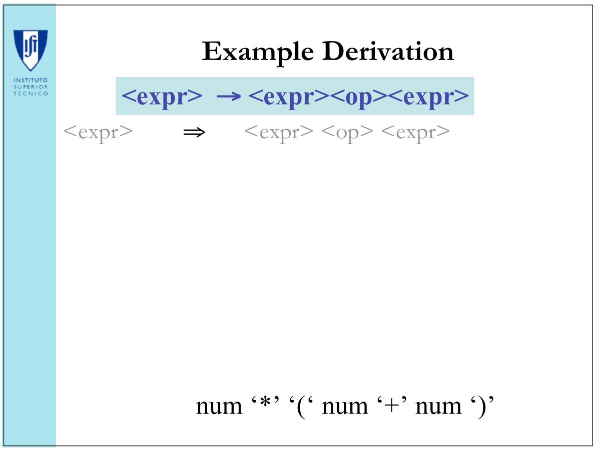


Example Derivation

<expr> → <expr><op><expr>

<expr>

num '*' '(' num '+' num ')'

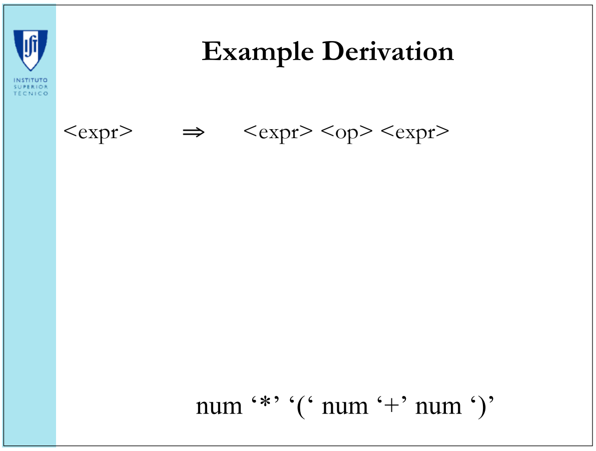


Example Derivation

<expr> → <expr><op><expr>

<expr> ⇒ <expr> <op> <expr>

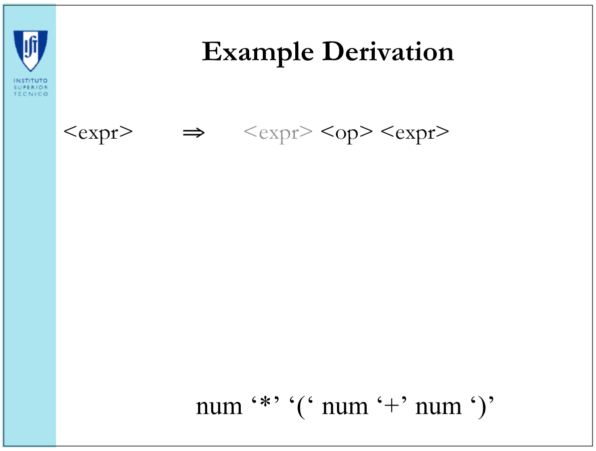
num '*' '(' num '+' num ')'



Example Derivation

<expr> ⇒ <expr> <op> <expr>

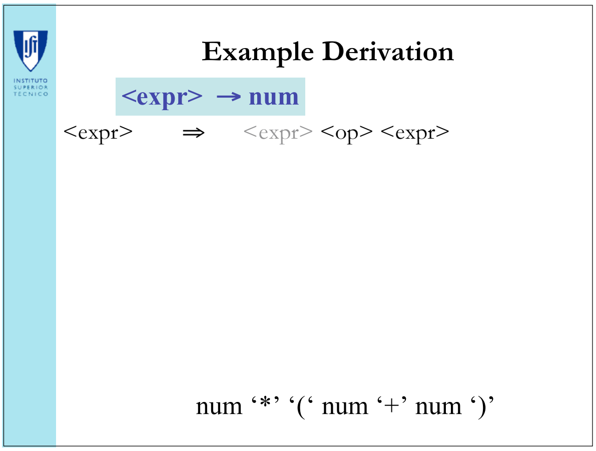
num '*' '(' num '+' num ')'



Example Derivation

<expr> ⇒ <expr> <op> <expr>

num ‘*’ ‘(‘ num ‘+’ num ‘)’

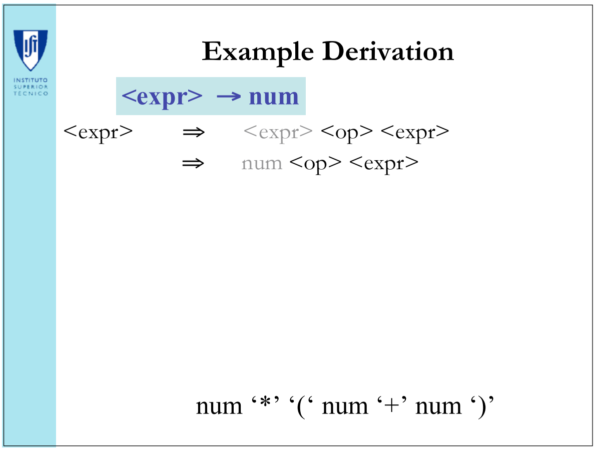


Example Derivation

<expr> → num

<expr> ⇒ <expr> <op> <expr>

num ‘*’ ‘(‘ num ‘+’ num ‘)’

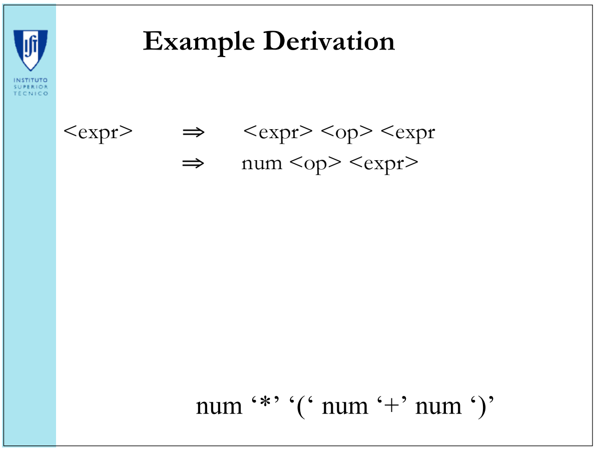


Example Derivation

<expr> → num

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>

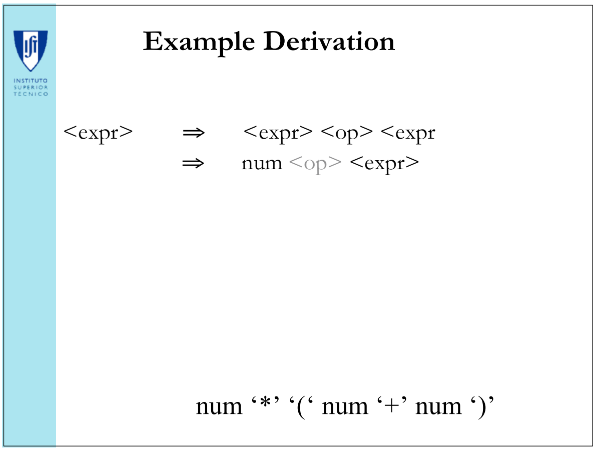
num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> ⇒ <expr> <op> <expr
⇒ num <op> <expr>

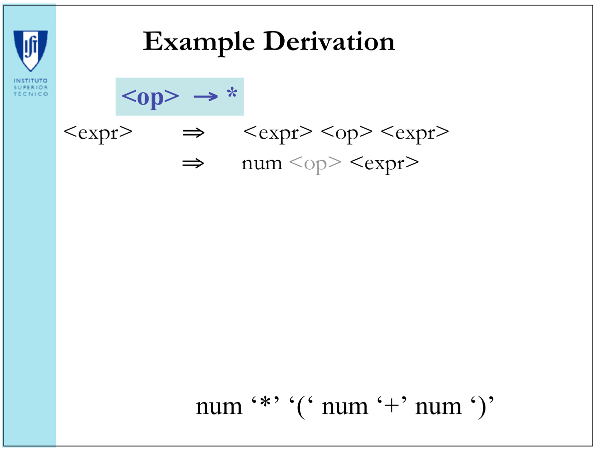
num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> ⇒ <expr> <op> <expr
⇒ num <op> <expr>

num ‘*’ ‘(‘ num ‘+’ num ‘)’

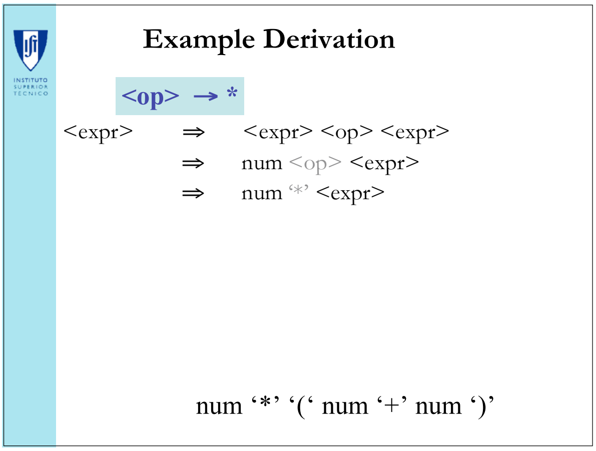


Example Derivation

<op> → *

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>

num ‘*’ ‘(‘ num ‘+’ num ‘)’

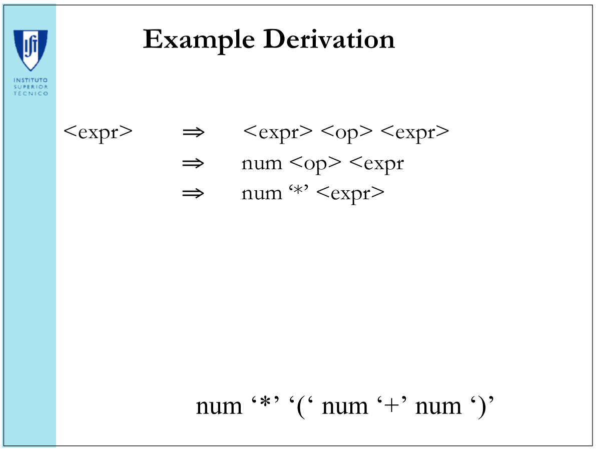


Example Derivation

<op> → *

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num ‘*’ <expr>

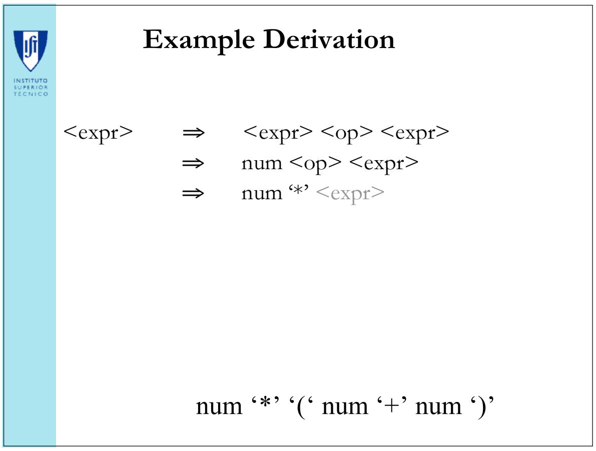
num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr
⇒ num ‘*’ <expr>

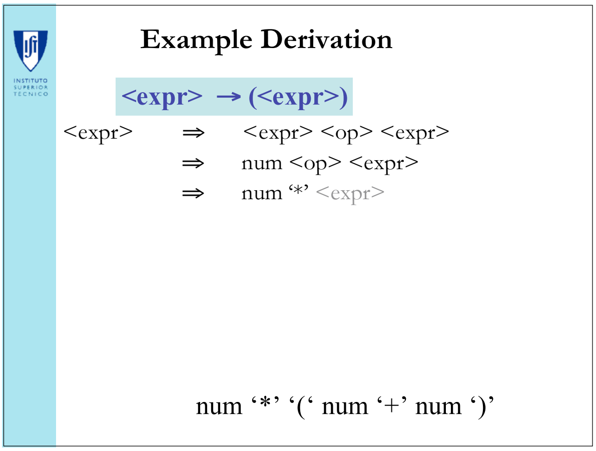
num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num '*' <expr>

num '*' '(' num '+' num ')'

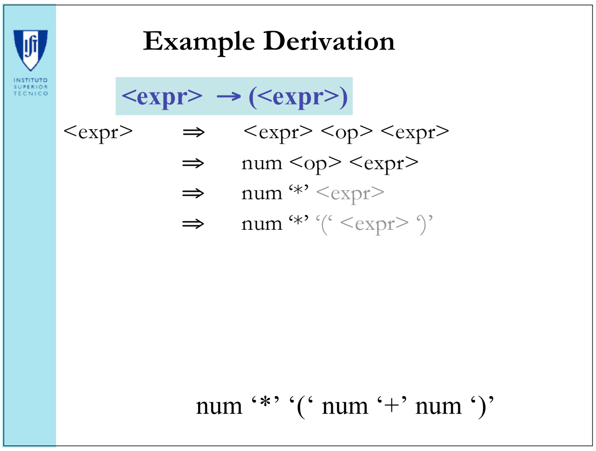


Example Derivation

<expr> → (<expr>)

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num '*' <expr>

num '*' '(' num '+' num ')'

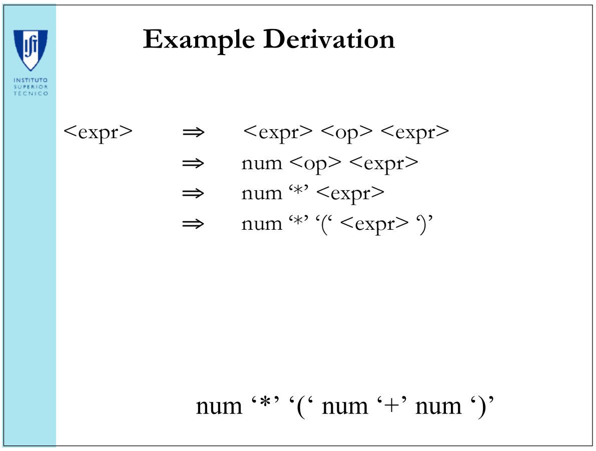


Example Derivation

<expr> → (<expr>)

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num '*' <expr>
⇒ num '*' '(' <expr> ')'

num '*' '(' num '+' num ')'



Example Derivation

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num '*' <expr>
⇒ num '*' '(' <expr> ')'

num '*' '(' num '+' num ')'

Example Derivation

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num ‘*’ <expr>
⇒ num ‘*’ ‘(‘ <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’

Example Derivation

<expr> → <expr><op><expr>

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num ‘*’ <expr>
⇒ num ‘*’ ‘(‘ <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’

Example Derivation

<expr> → <expr><op><expr>

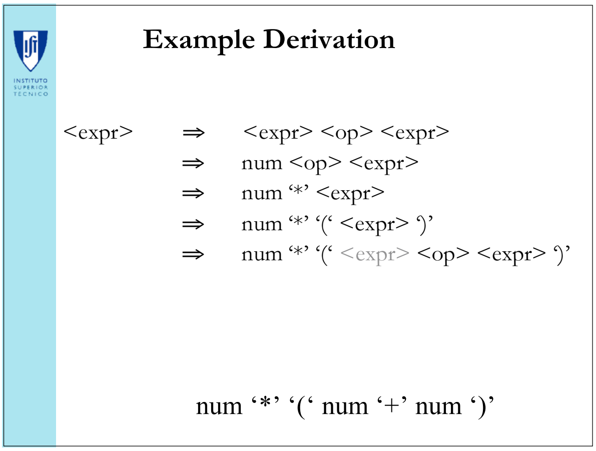
<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num ‘*’ <expr>
⇒ num ‘*’ ‘(‘ <expr> ‘)’
⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’

Example Derivation

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num ‘*’ <expr>
⇒ num ‘*’ ‘(‘ <expr> ‘)’
⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> ⇒ <expr> <op> <expr>

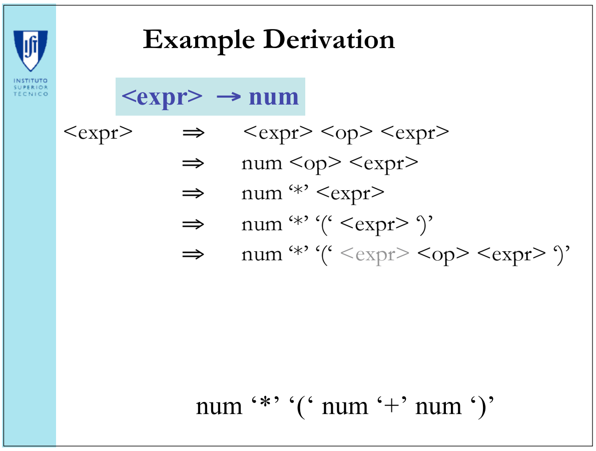
⇒ num <op> <expr>

⇒ num ‘*’ <expr>

⇒ num ‘*’ ‘(‘ <expr> ‘)’

⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> → num

<expr> ⇒ <expr> <op> <expr>

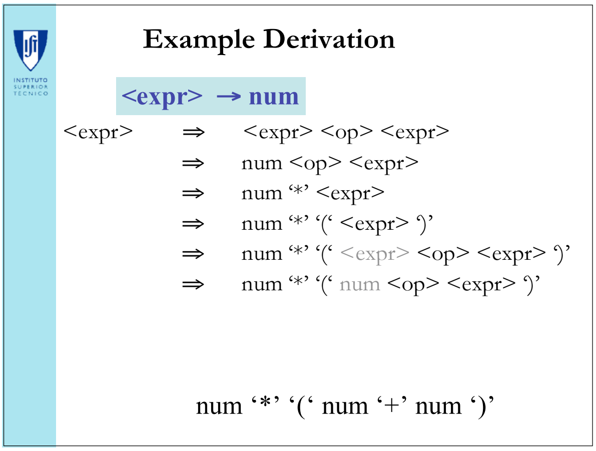
⇒ num <op> <expr>

⇒ num ‘*’ <expr>

⇒ num ‘*’ ‘(‘ <expr> ‘)’

⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> → num

<expr> ⇒ <expr> <op> <expr>

⇒ num <op> <expr>

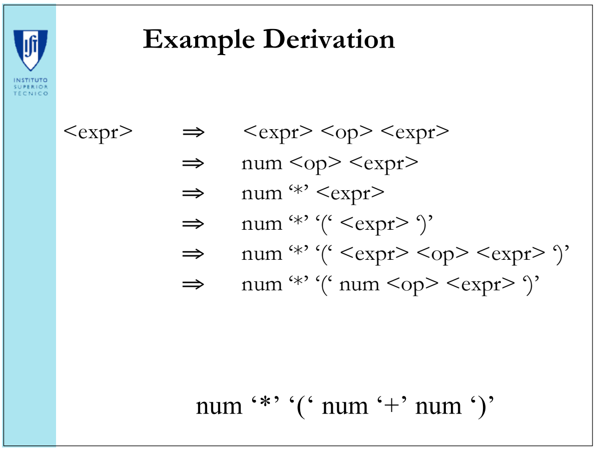
⇒ num ‘*’ <expr>

⇒ num ‘*’ ‘(‘ <expr> ‘)’

⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’

⇒ num ‘*’ ‘(‘ num <op> <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> ⇒ <expr> <op> <expr>

⇒ num <op> <expr>

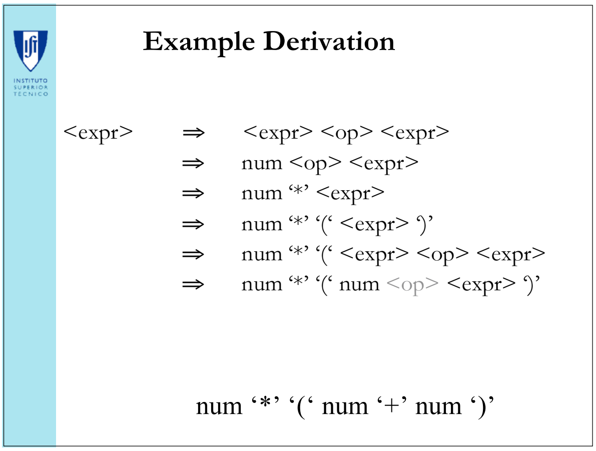
⇒ num ‘*’ <expr>

⇒ num ‘*’ ‘(‘ <expr> ‘)’

⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’

⇒ num ‘*’ ‘(‘ num <op> <expr> ‘)’

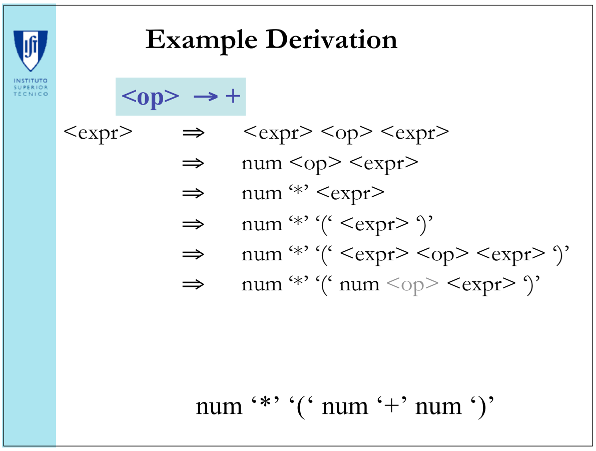
num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num ‘*’ <expr>
⇒ num ‘*’ ‘(‘ <expr> ‘)’
⇒ num ‘*’ ‘(‘ <expr> <op> <expr>
⇒ num ‘*’ ‘(‘ num <op> <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’

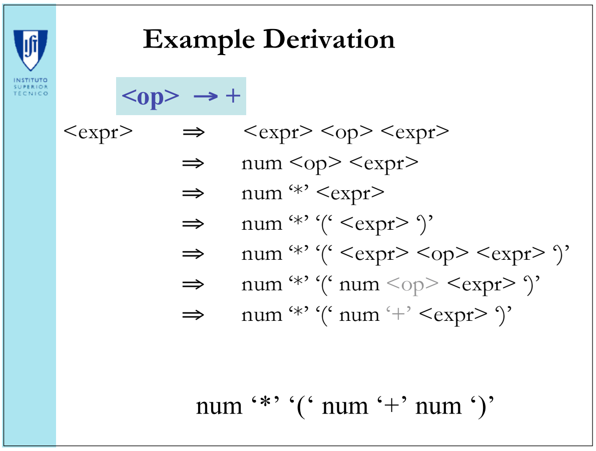


Example Derivation

<op> → +

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num ‘*’ <expr>
⇒ num ‘*’ ‘(‘ <expr> ‘)’
⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’
⇒ num ‘*’ ‘(‘ num <op> <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’

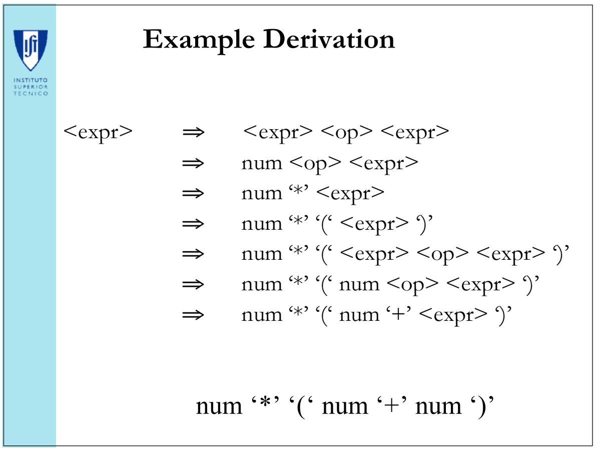


Example Derivation

<op> → +

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num ‘*’ <expr>
⇒ num ‘*’ ‘(‘ <expr> ‘)’
⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’
⇒ num ‘*’ ‘(‘ num <op> <expr> ‘)’
⇒ num ‘*’ ‘(‘ num ‘+’ <expr> ‘)’

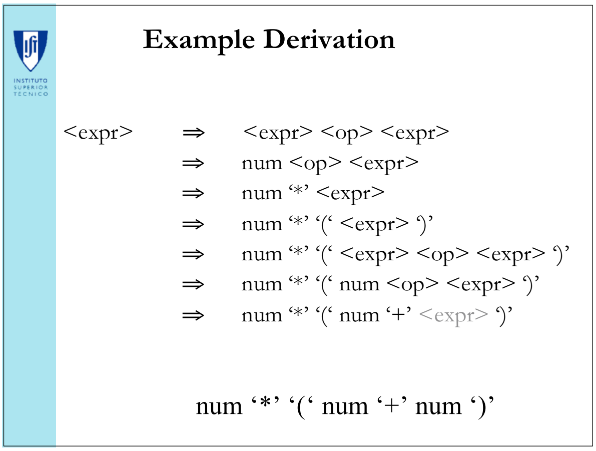
num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> ⇒ <expr> <op> <expr>
⇒ num <op> <expr>
⇒ num ‘*’ <expr>
⇒ num ‘*’ ‘(‘ <expr> ‘)’
⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’
⇒ num ‘*’ ‘(‘ num <op> <expr> ‘)’
⇒ num ‘*’ ‘(‘ num ‘+’ <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> ⇒ <expr> <op> <expr>

⇒ num <op> <expr>

⇒ num ‘*’ <expr>

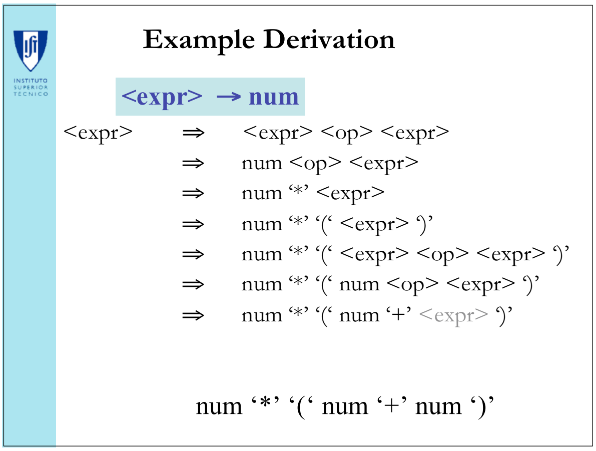
⇒ num ‘*’ ‘(‘ <expr> ‘)’

⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’

⇒ num ‘*’ ‘(‘ num <op> <expr> ‘)’

⇒ num ‘*’ ‘(‘ num ‘+’ <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> → num

<expr> ⇒ <expr> <op> <expr>

⇒ num <op> <expr>

⇒ num ‘*’ <expr>

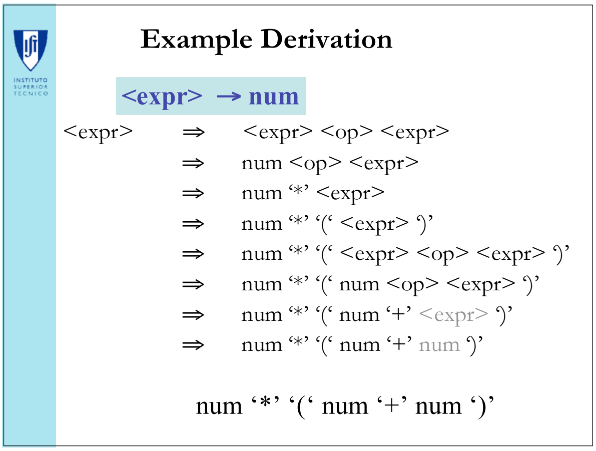
⇒ num ‘*’ ‘(‘ <expr> ‘)’

⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’

⇒ num ‘*’ ‘(‘ num <op> <expr> ‘)’

⇒ num ‘*’ ‘(‘ num ‘+’ <expr> ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> → num

<expr> ⇒ <expr> <op> <expr>

⇒ num <op> <expr>

⇒ num ‘*’ <expr>

⇒ num ‘*’ ‘(‘ <expr> ‘)’

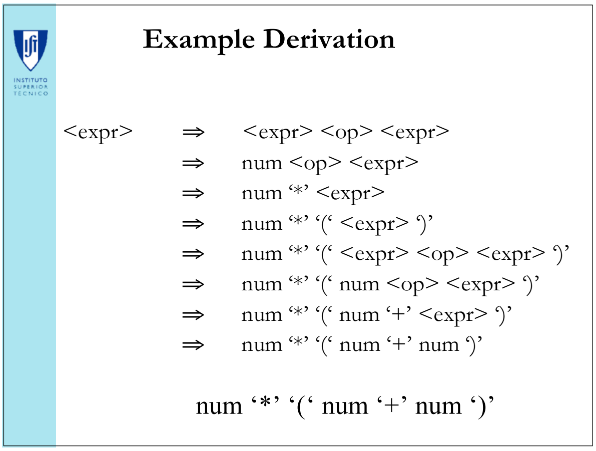
⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’

⇒ num ‘*’ ‘(‘ num <op> <expr> ‘)’

⇒ num ‘*’ ‘(‘ num ‘+’ <expr> ‘)’

⇒ num ‘*’ ‘(‘ num ‘+’ num ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’



Example Derivation

<expr> ⇒ <expr> <op> <expr>

⇒ num <op> <expr>

⇒ num ‘*’ <expr>

⇒ num ‘*’ ‘(‘ <expr> ‘)’

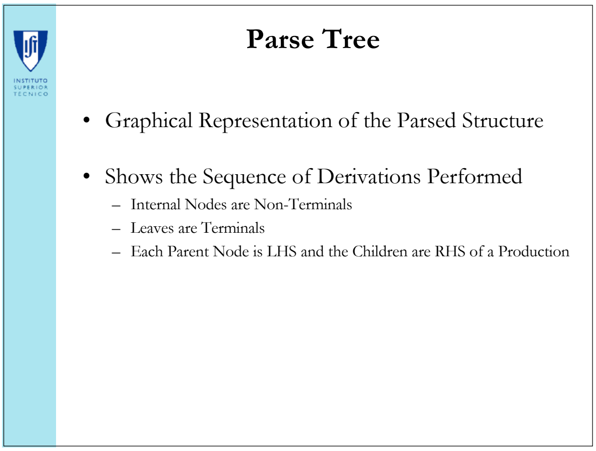
⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’

⇒ num ‘*’ ‘(‘ num <op> <expr> ‘)’

⇒ num ‘*’ ‘(‘ num ‘+’ <expr> ‘)’

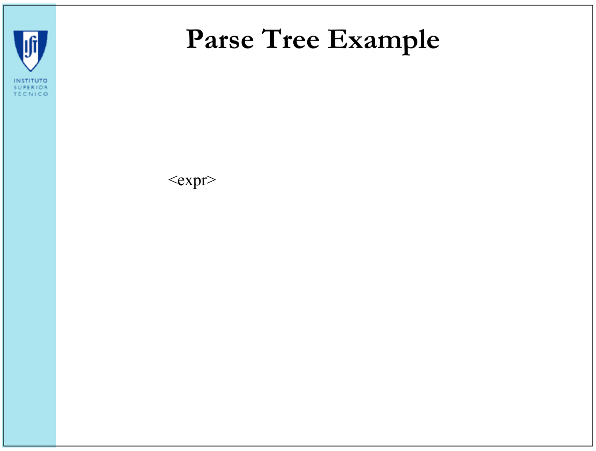
⇒ num ‘*’ ‘(‘ num ‘+’ num ‘)’

num ‘*’ ‘(‘ num ‘+’ num ‘)’

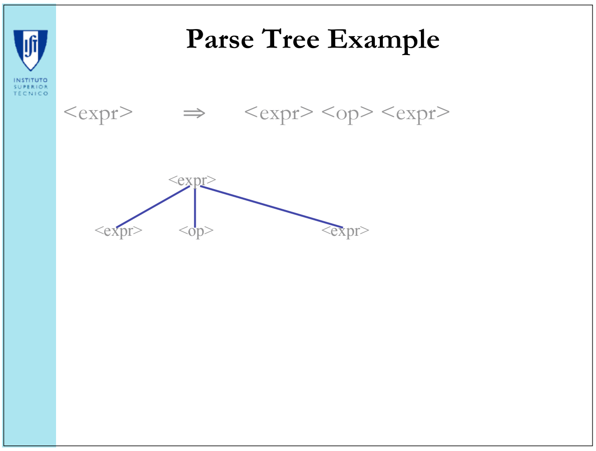


Parse Tree

- Graphical Representation of the Parsed Structure
- Shows the Sequence of Derivations Performed
  - Internal Nodes are Non-Terminals
  - Leaves are Terminals
  - Each Parent Node is LHS and the Children are RHS of a Production

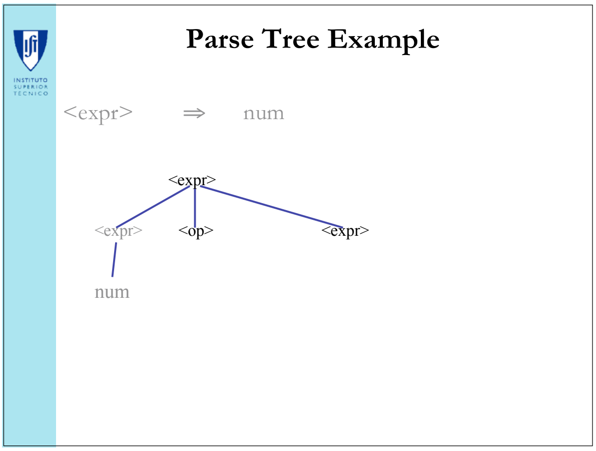


Parse Tree Example



Parse Tree Example

<expr> ⇒ <expr> <op> <expr>

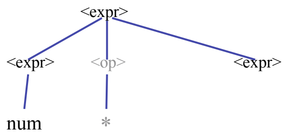


Parse Tree Example

<expr> ⇒ num

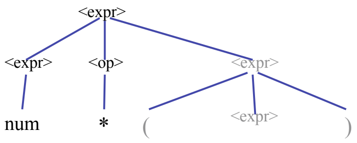
Parse Tree Example

<op> ⇒ ‘*’



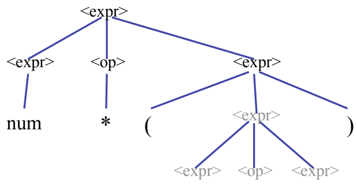
Parse Tree Example

<expr> ⇒ ‘(‘ <expr> ‘)’



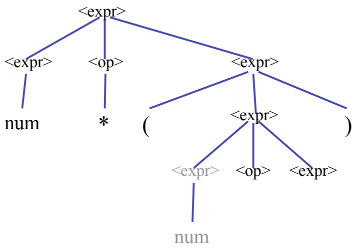
Parse Tree Example

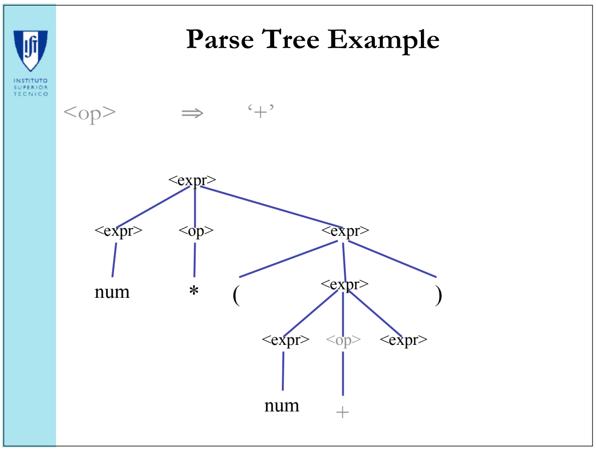
<expr> ⇒ <expr> <op> <expr>



Parse Tree Example

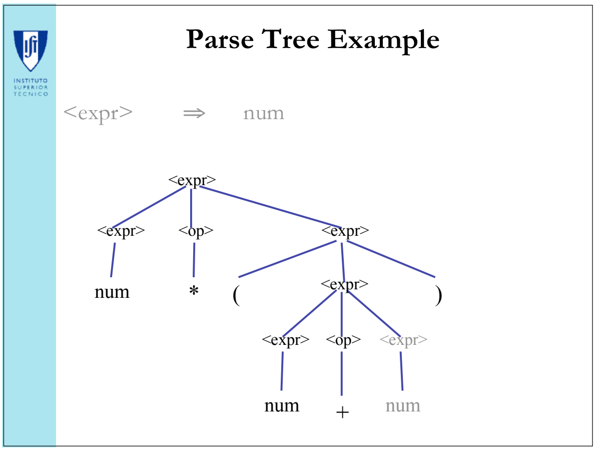
<expr> ⇒ num





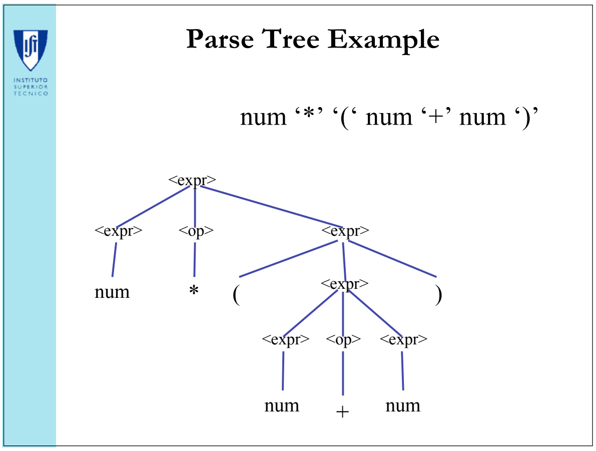
Parse Tree Example

<op> ⇒ ‘+’



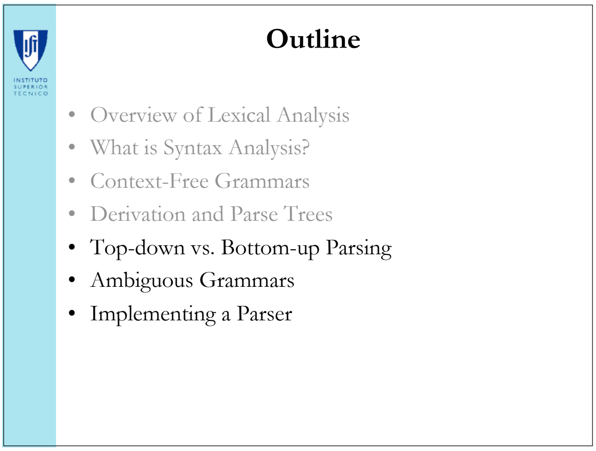
Parse Tree Example

<expr> ⇒ num



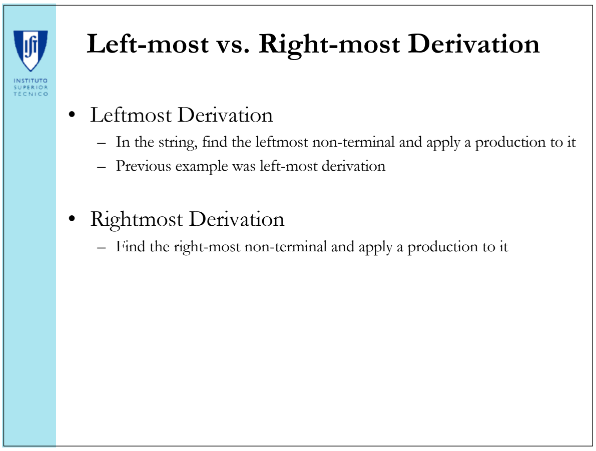
Parse Tree Example

num ‘*’ ‘(‘ num ‘+’ num ‘)’



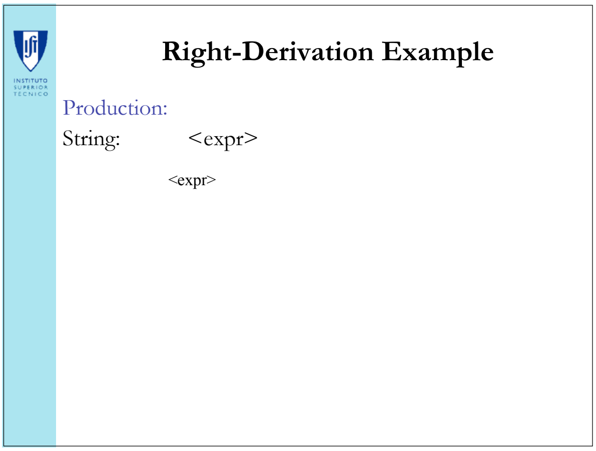
Outline

- Overview of Lexical Analysis
- What is Syntax Analysis?
- Context-Free Grammars
- Derivation and Parse Trees
- Top-down vs. Bottom-up Parsing
- Ambiguous Grammars
- Implementing a Parser



Left-most vs. Right-most Derivation

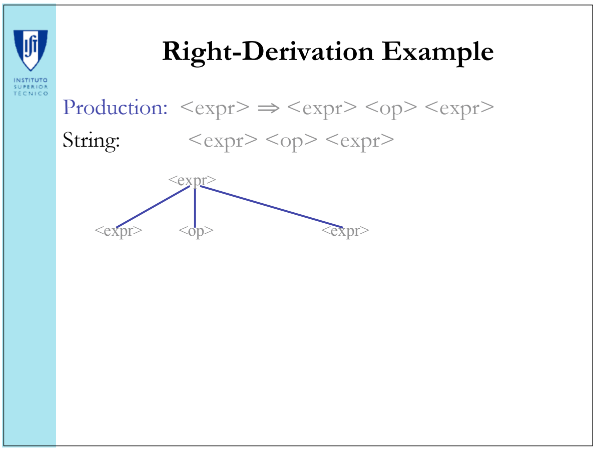
- Leftmost Derivation
  - In the string, find the leftmost non-terminal and apply a production to it
  - Previous example was left-most derivation
- Rightmost Derivation
  - Find the right-most non-terminal and apply a production to it



Right-Derivation Example

Production:

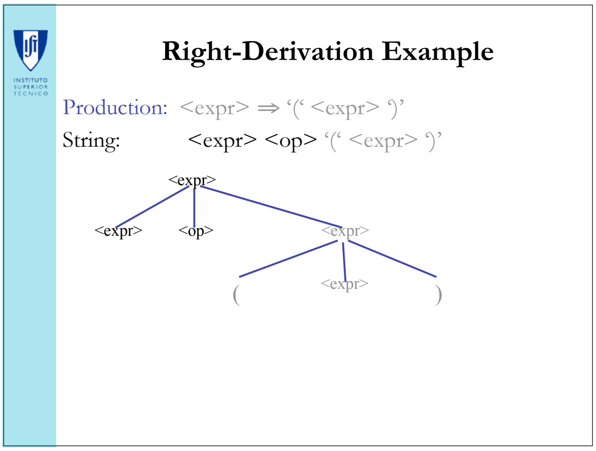
String: <expr>



Right-Derivation Example

Production: <expr> ⇒ <expr> <op> <expr>

String: <expr> <op> <expr>



Right-Derivation Example

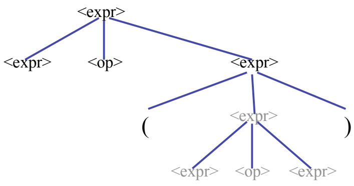
Production: <expr> ⇒ '(' <expr> ')'

String: <expr> <op> '(' <expr> ')'

Right-Derivation Example

Production: <expr> ⇒ <expr> <op> <expr>

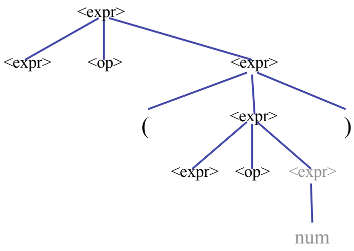
String: <expr> <op> ‘(‘ <expr> <op> <expr> ‘)’



Right-Derivation Example

Production: <expr> ⇒ num

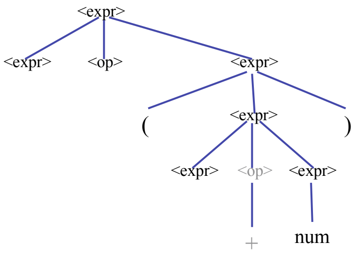
String: <expr> <op> ‘(‘ <expr> <op> num ‘)’



Right-Derivation Example

Production: <op> ⇒ ‘+’

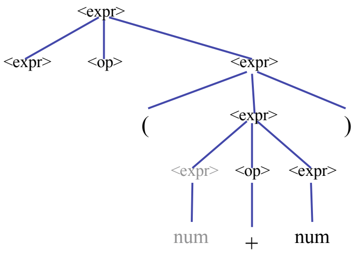
String: <expr> <op> ‘(‘ <expr> ‘+’ num ‘)’

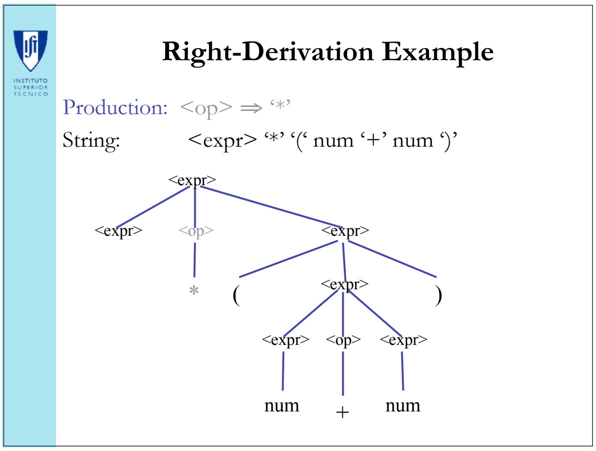


Right-Derivation Example

Production: <expr> ⇒ num

String: <expr> <op> ‘(‘ num ‘+’ num ‘)’

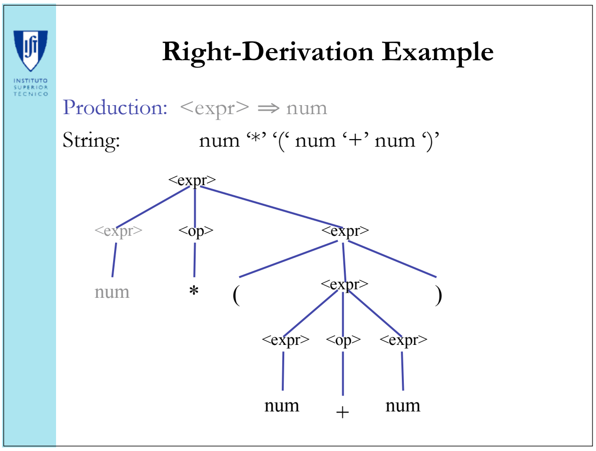




Right-Derivation Example

Production: <op> ⇒ ‘*’

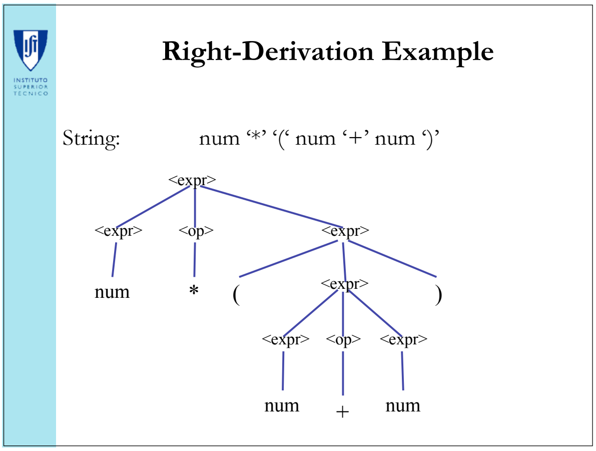
String: <expr> ‘*’ ‘(‘ num ‘+’ num ‘)’



Right-Derivation Example

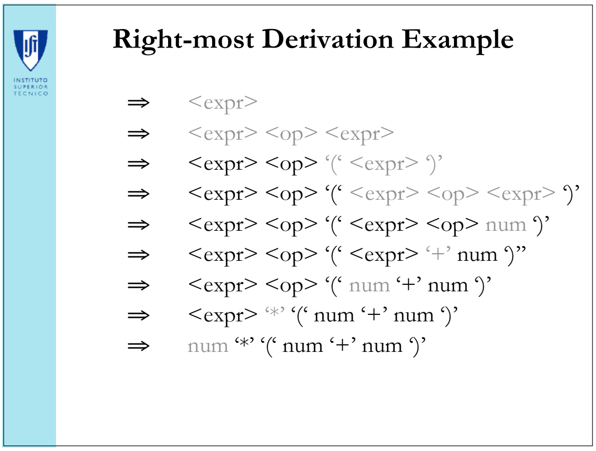
Production: <expr> ⇒ num

String: num ‘*’ ‘(‘ num ‘+’ num ‘)’



Right-Derivation Example

String: num ‘*’ ‘(‘ num ‘+’ num ‘)’



Right-most Derivation Example

⇒ <expr>

⇒ <expr> <op> <expr>

⇒ <expr> <op> ‘(‘ <expr> ‘)’

⇒ <expr> <op> ‘(‘ <expr> <op> <expr> ‘)’

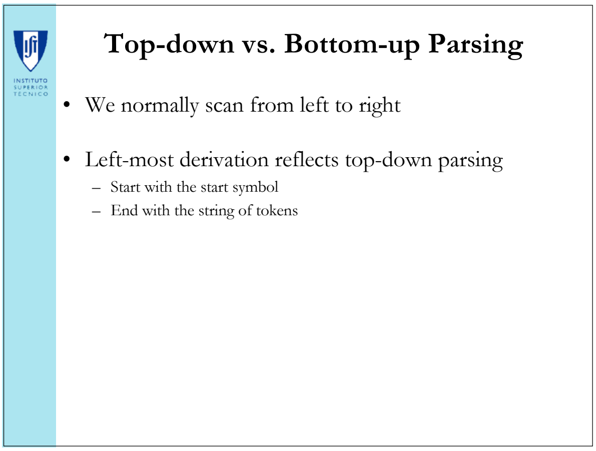
⇒ <expr> <op> ‘(‘ <expr> <op> num ‘)’

⇒ <expr> <op> ‘(‘ <expr> ‘+’ num ‘)”

⇒ <expr> <op> ‘(‘ num ‘+’ num ‘)’

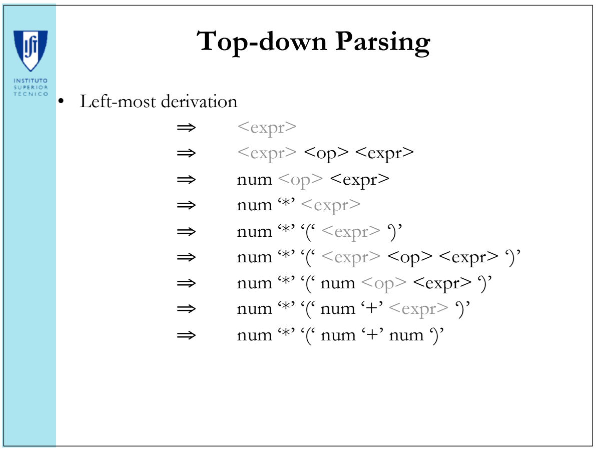
⇒ <expr> ‘*’ ‘(‘ num ‘+’ num ‘)’

⇒ num ‘*’ ‘(‘ num ‘+’ num ‘)’



Top-down vs. Bottom-up Parsing

- We normally scan from left to right
- Left-most derivation reflects top-down parsing
  - Start with the start symbol
  - End with the string of tokens



Top-down Parsing

- Left-most derivation

⇒ <expr>

⇒ <expr> <op> <expr>

⇒ num <op> <expr>

⇒ num ‘*’ <expr>

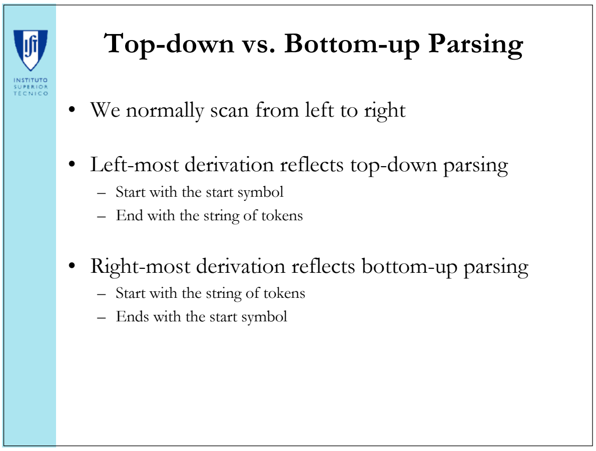
⇒ num ‘*’ ‘(‘ <expr> ‘)’

⇒ num ‘*’ ‘(‘ <expr> <op> <expr> ‘)’

⇒ num ‘*’ ‘(‘ num <op> <expr> ‘)’

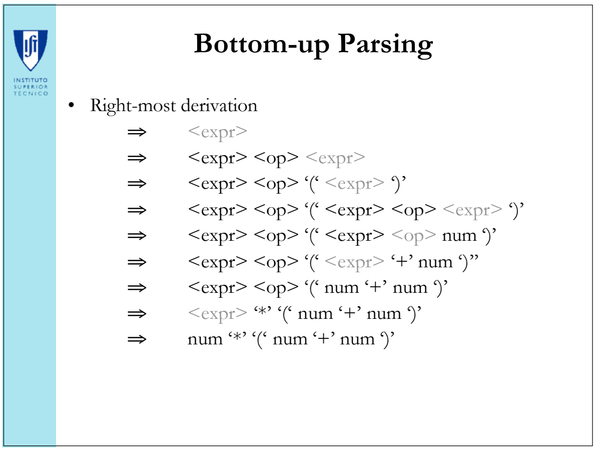
⇒ num ‘*’ ‘(‘ num ‘+’ <expr> ‘)’

⇒ num ‘*’ ‘(‘ num ‘+’ num ‘)’



Top-down vs. Bottom-up Parsing

- We normally scan from left to right
- Left-most derivation reflects top-down parsing
  - Start with the start symbol
  - End with the string of tokens
- Right-most derivation reflects bottom-up parsing
  - Start with the string of tokens
  - Ends with the start symbol



Bottom-up Parsing

- Right-most derivation

⇒ <expr>

⇒ <expr> <op> <expr>

⇒ <expr> <op> ‘(‘ <expr> ‘)’

⇒ <expr> <op> ‘(‘ <expr> <op> <expr> ‘)’

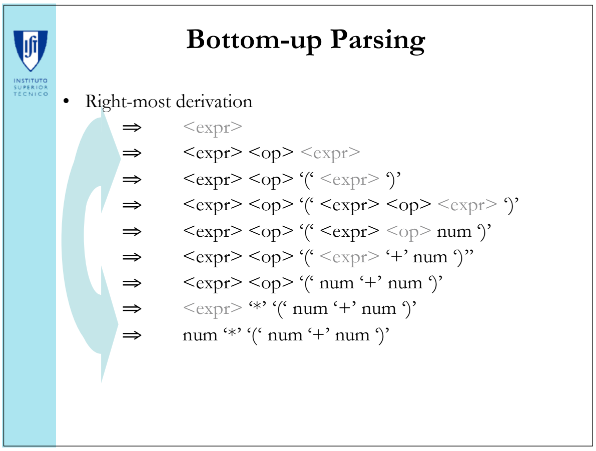
⇒ <expr> <op> ‘(‘ <expr> <op> num ‘)’

⇒ <expr> <op> ‘(‘ <expr> ‘+’ num ‘)”

⇒ <expr> <op> ‘(‘ num ‘+’ num ‘)’

⇒ <expr> ‘*’ ‘(‘ num ‘+’ num ‘)’

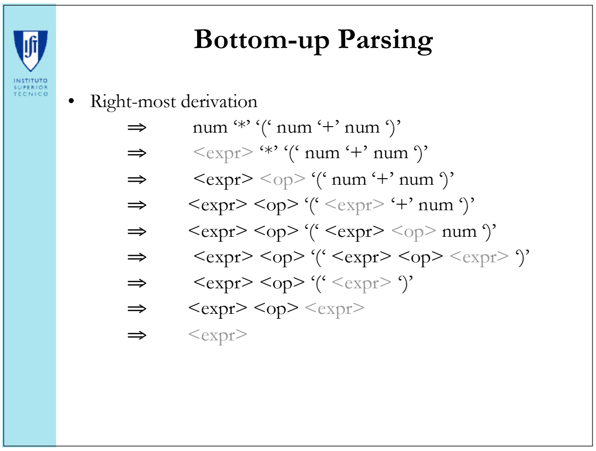
⇒ num ‘*’ ‘(‘ num ‘+’ num ‘)’



Bottom-up Parsing

- Right-most derivation

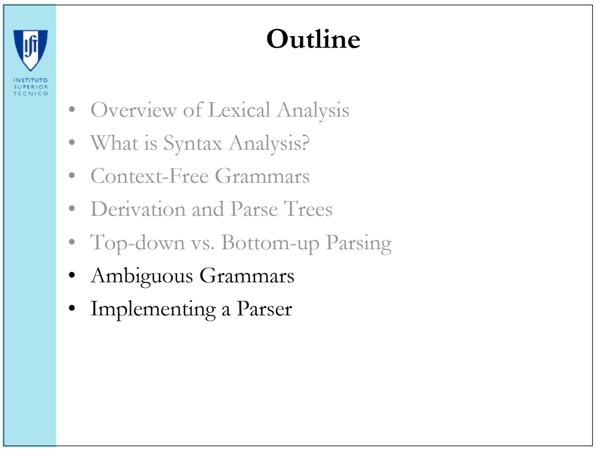
⇒ <expr>
⇒ <expr> <op> <expr>
⇒ <expr> <op> ‘(‘ <expr> ‘)’
⇒ <expr> <op> ‘(‘ <expr> <op> <expr> ‘)’
⇒ <expr> <op> ‘(‘ <expr> <op> num ‘)’
⇒ <expr> <op> ‘(‘ <expr> ‘+’ num ‘)”
⇒ <expr> <op> ‘(‘ num ‘+’ num ‘)’
⇒ <expr> ‘*’ ‘(‘ num ‘+’ num ‘)’
⇒ num ‘*’ ‘(‘ num ‘+’ num ‘)’



Bottom-up Parsing

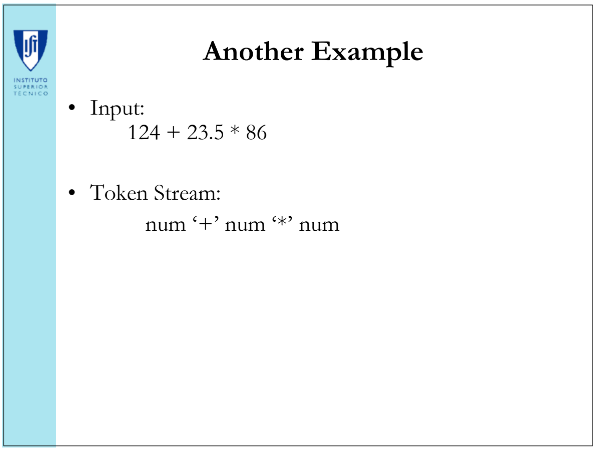
- Right-most derivation

⇒ num ‘*’ ‘(‘ num ‘+’ num ‘)’
⇒ <expr> ‘*’ ‘(‘ num ‘+’ num ‘)’
⇒ <expr> <op> ‘(‘ num ‘+’ num ‘)’
⇒ <expr> <op> ‘(‘ <expr> ‘+’ num ‘)’
⇒ <expr> <op> ‘(‘ <expr> <op> num ‘)’
⇒ <expr> <op> ‘(‘ <expr> <op> <expr> ‘)’
⇒ <expr> <op> ‘(‘ <expr> ‘)’
⇒ <expr> <op> <expr>
⇒ <expr>



Outline

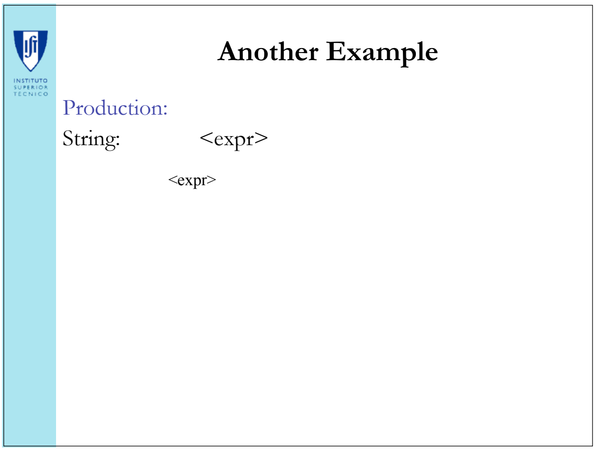
- Overview of Lexical Analysis
- What is Syntax Analysis?
- Context-Free Grammars
- Derivation and Parse Trees
- Top-down vs. Bottom-up Parsing
- Ambiguous Grammars
- Implementing a Parser



Another Example

- Input:
  124 + 23.5 * 86

- Token Stream:
  num ‘+’ num ‘*’ num

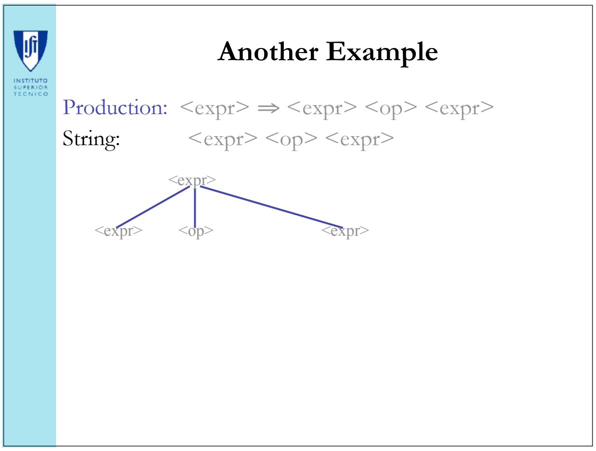


Another Example

Production:

String: <expr>

<expr>



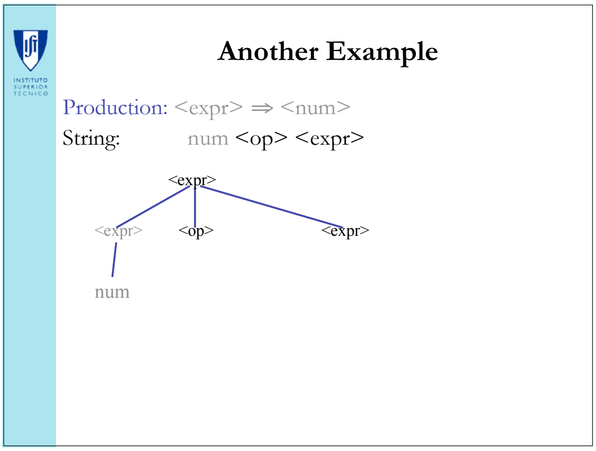
Another Example

Production: <expr> ⇒ <expr> <op> <expr>

String: <expr> <op> <expr>

<expr>

<expr> <op> <expr>



Another Example

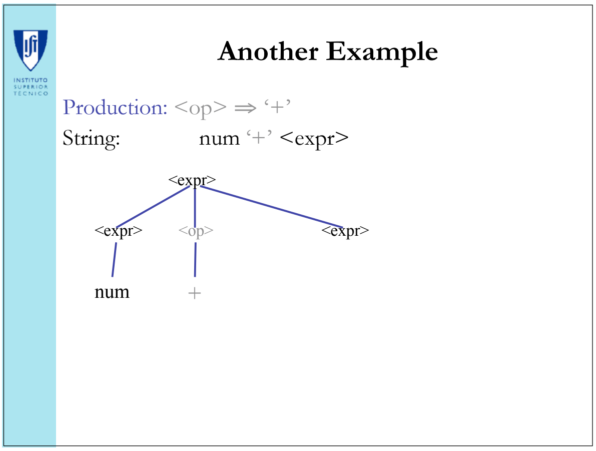
Production: <expr> ⇒ <num>

String: num <op> <expr>

<expr>

<expr> <op> <expr>

num



Another Example

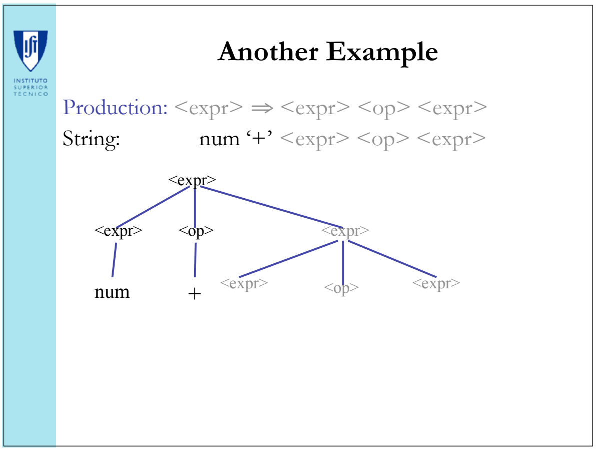
Production: <op> ⇒ '+'

String: num '+' <expr>

<expr>

<expr> <op> <expr>

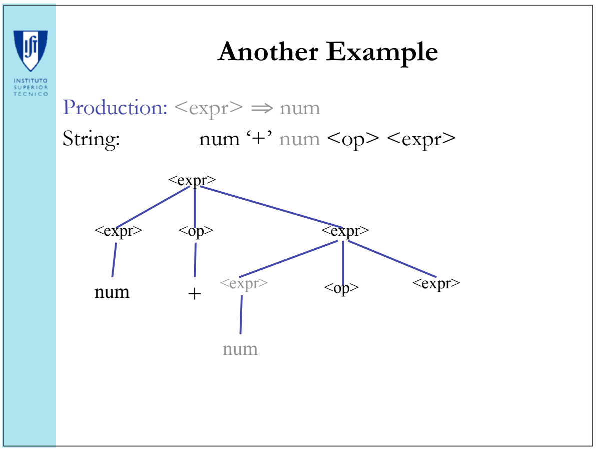
num +



Another Example

Production: <expr> ⇒ <expr> <op> <expr>

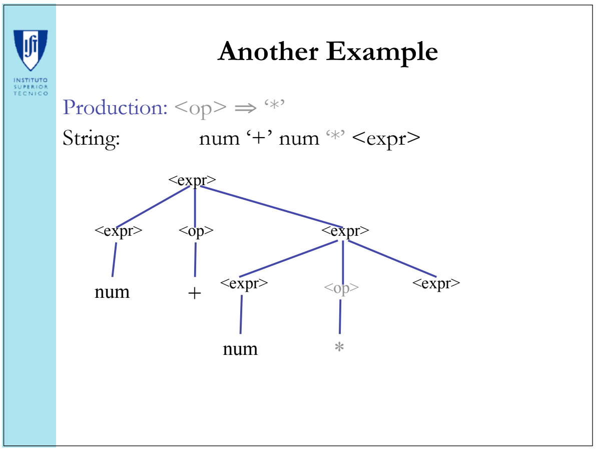
String: num '+' <expr> <op> <expr>



Another Example

Production: <expr> ⇒ num

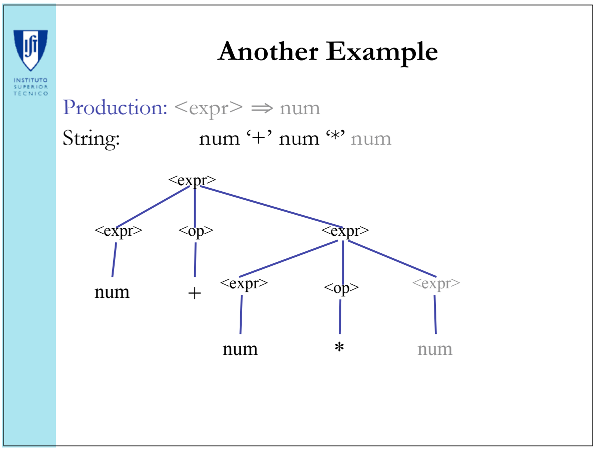
String: num '+' num <op> <expr>



Another Example

Production: <op> ⇒ '*'

String: num '+' num '*' <expr>



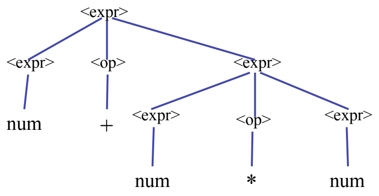
Another Example

Production: <expr> ⇒ num

String: num '+' num '*' num

Another Example

String: num '+' num '*' num



Another Example

String: num '+' num '*' num

- How about a different order of derivation

Another Example

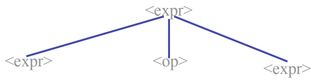
String: <expr>

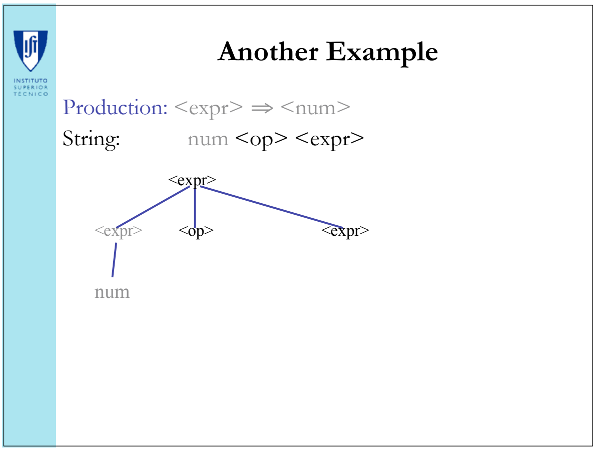
<expr>

Another Example

Production: <expr> ⇒ <expr> <op> <expr>

String: <expr> <op> <expr>

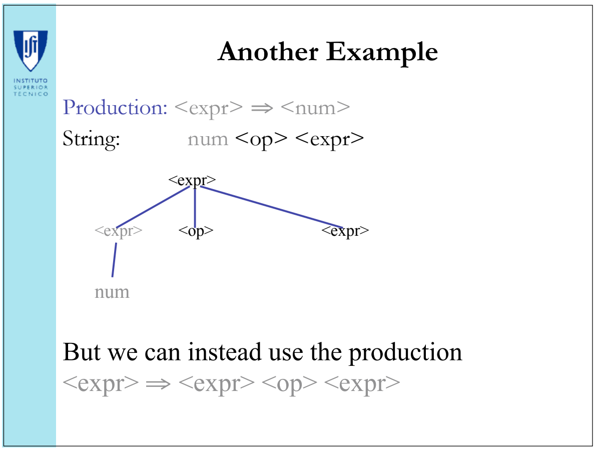




Another Example

Production: <expr> ⇒ <num>

String: num <op> <expr>

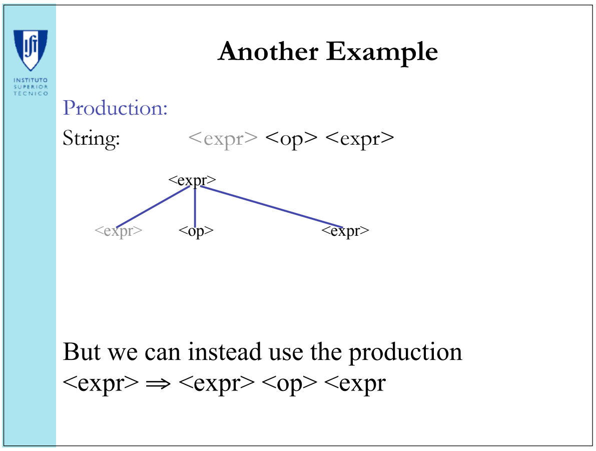


Another Example

Production: <expr> ⇒ <num>

String: num <op> <expr>

But we can instead use the production
<expr> ⇒ <expr> <op> <expr>

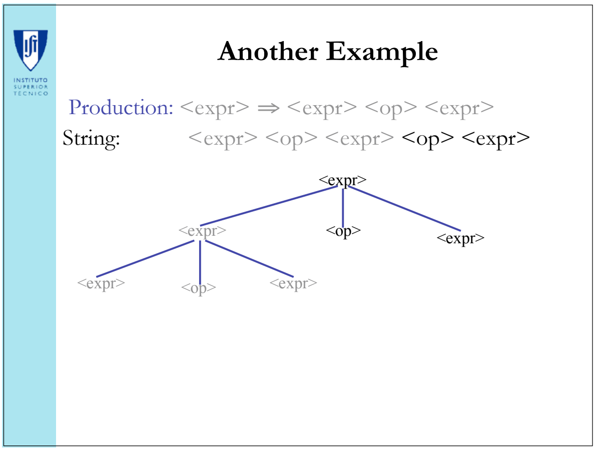


Another Example

Production:

String: <expr> <op> <expr>

But we can instead use the production
<expr> ⇒ <expr> <op> <expr



Another Example

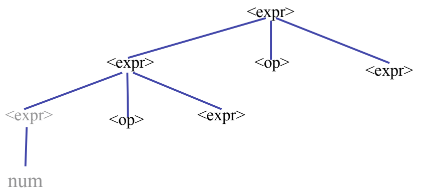
Production: <expr> ⇒ <expr> <op> <expr>

String: <expr> <op> <expr> <op> <expr>

Another Example

Production: <expr> ⇒ <num>

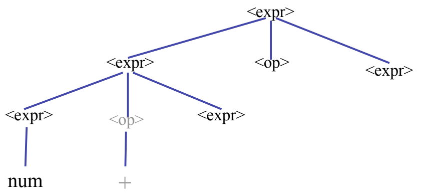
String: num <op> <expr> <op> <expr>



Another Example

Production: <op> ⇒ <+>

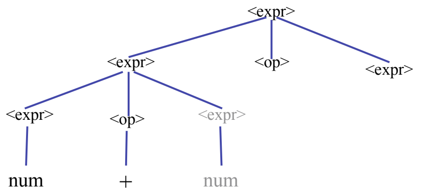
String: num '+' <expr> <op> <expr>



Another Example

Production: <expr> ⇒ <num>

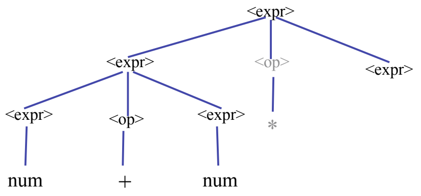
String: num '+' num <op> <expr>

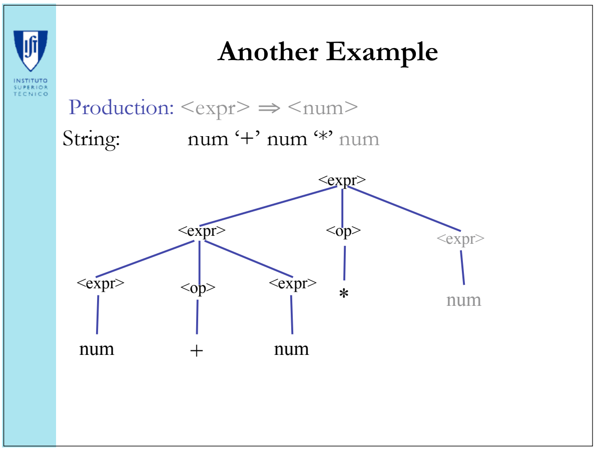


Another Example

Production: <op> ⇒ '*'

String: num '+' num '*' <expr>

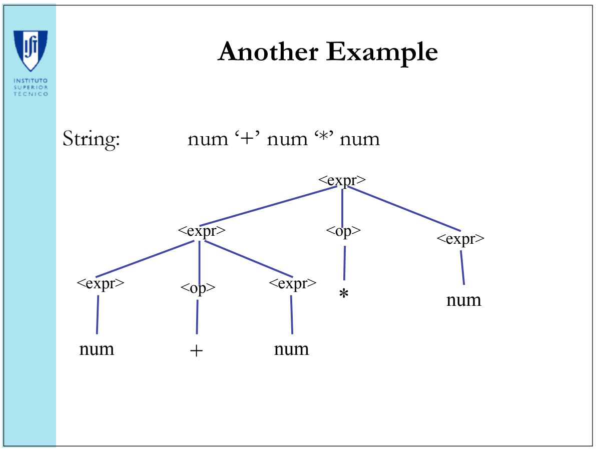




Another Example

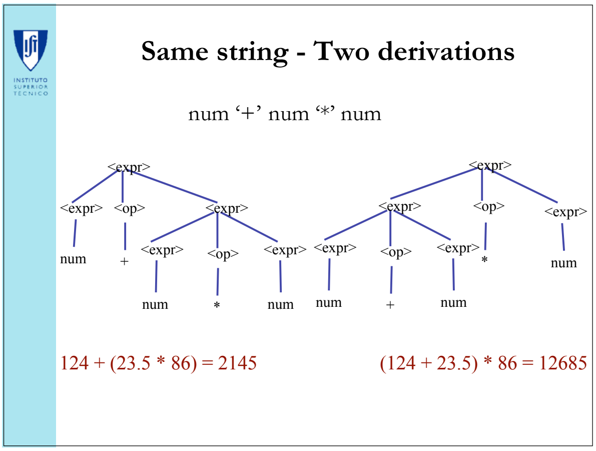
Production: <expr> ⇒ <num>

String: num '+' num '*' num



Another Example

String: num '+' num '*' num

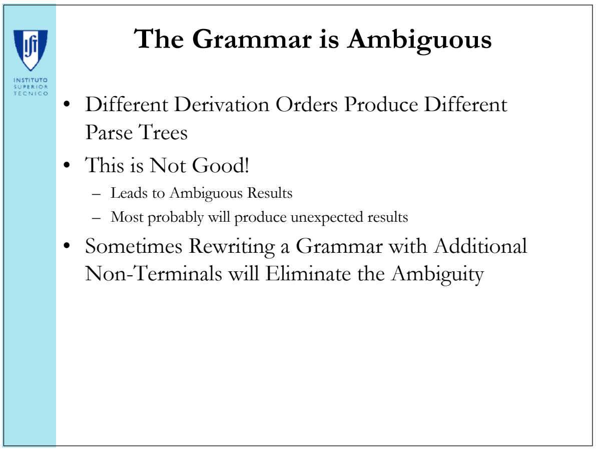


Same string - Two derivations

num '+' num '*' num

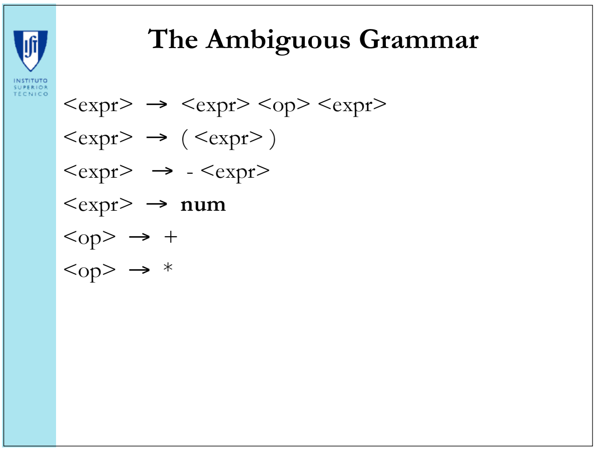
124 + (23.5 * 86) = 2145

(124 + 23.5) * 86 = 12685



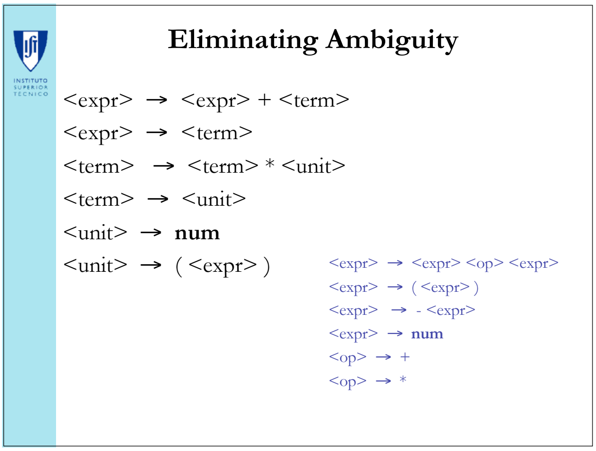
The Grammar is Ambiguous

- Different Derivation Orders Produce Different Parse Trees
- This is Not Good!
  - Leads to Ambiguous Results
  - Most probably will produce unexpected results
- Sometimes Rewriting a Grammar with Additional Non-Terminals will Eliminate the Ambiguity



The Ambiguous Grammar

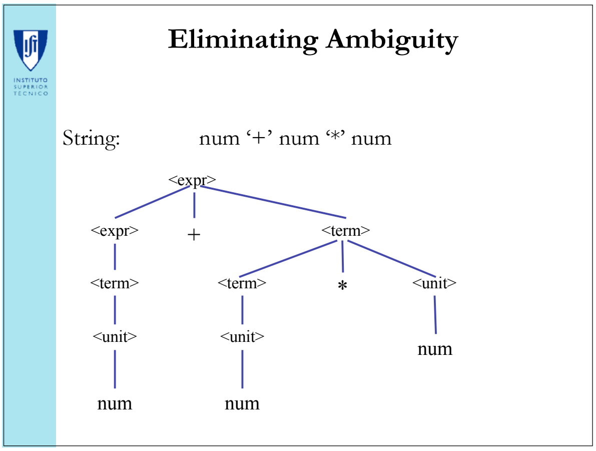
<expr> → <expr> <op> <expr>
<expr> → (<expr>)
<expr> → - <expr>
<expr> → **num**
<op> → +
<op> → *



Eliminating Ambiguity

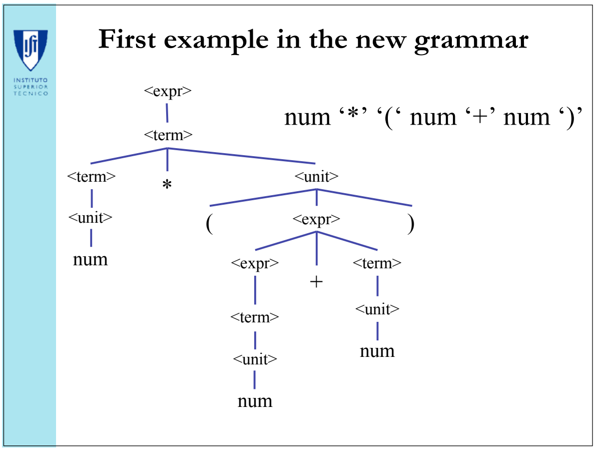
<expr> → <expr> + <term>
<expr> → <term>
<term> → <term> * <unit>
<term> → <unit>
<unit> → **num**
<unit> → (<expr>)

<expr> → <expr> <op> <expr>
<expr> → (<expr>)
<expr> → - <expr>
<expr> → **num**
<op> → +
<op> → *



Eliminating Ambiguity

String: num '+' num '*' num



First example in the new grammar

num '*' '(' num '+' num ')'

Question: Is this Grammar ambiguous?

<stmt> → if <expr> then <stlist>

<stmt> → if <expr> then <stlist> else <stlist>

How do you make this unambiguous?

<stmt> → if <expr> then <stlist>

<stmt> → if <expr> then <stlist> else <stlist>

Ambiguous Grammars

Definitions

- If a grammar has more than one leftmost derivation for a single *sentential form*, the grammar is *ambiguous*
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is *ambiguous*
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar

Classic example — the *if*-*then*-*else* problem

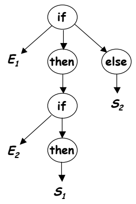
Stmt → if *Expr* then *Stmt*
| if *Expr* then *Stmt* else *Stmt*
| … *other stmts* …

This ambiguity is entirely grammatical in nature

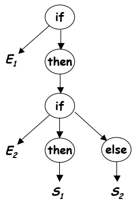
Ambiguity

This sentential form has two derivations

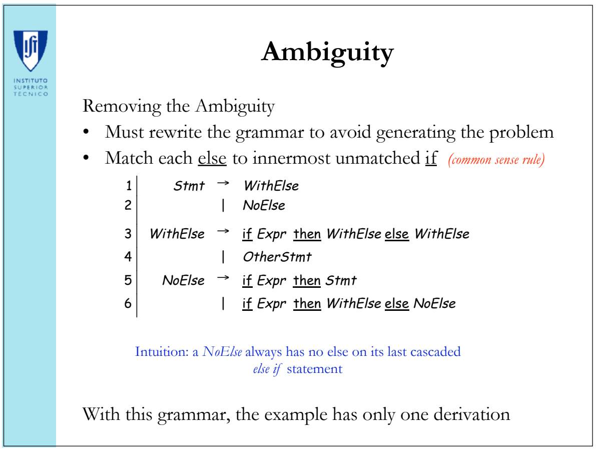
if $Expr_1$ then if $Expr_2$ then $Stmt_1$ else $Stmt_2$



production 2, then production 1



production 1, then production 2



Ambiguity

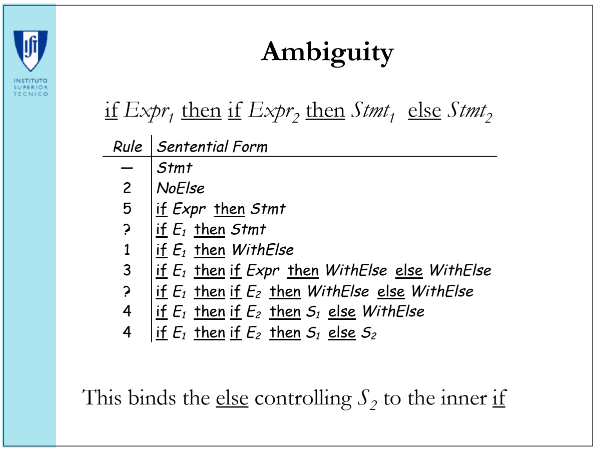
Removing the Ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each else to innermost unmatched if *(common sense rule)*

1	*Stmt*	→	*WithElse*
2		|	*NoElse*
3	*WithElse*	→	if *Expr* then *WithElse* else *WithElse*
4		|	*OtherStmt*
5	*NoElse*	→	if *Expr* then *Stmt*
6		|	if *Expr* then *WithElse* else *NoElse*

Intuition: a *NoElse* always has no else on its last cascaded *else if* statement

With this grammar, the example has only one derivation

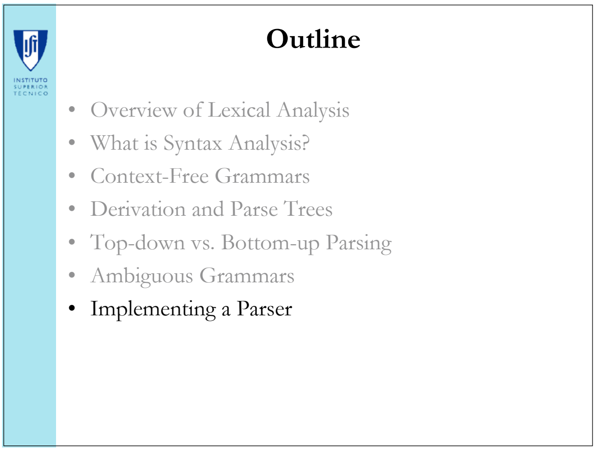


Ambiguity

if $Expr_1$ then if $Expr_2$ then $Stmt_1$ else $Stmt_2$

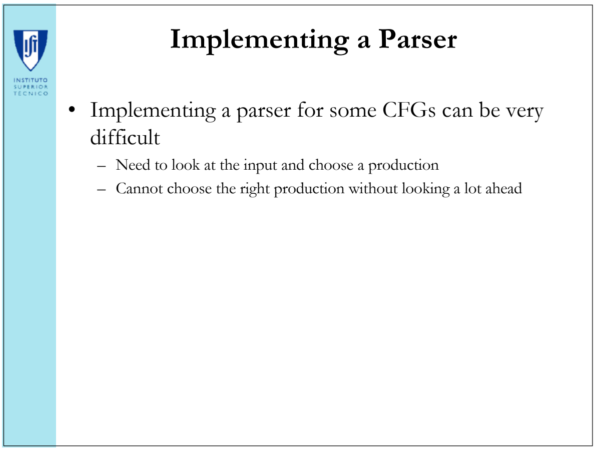
Rule	Sentential Form
—	*Stmt*
2	*NoElse*
5	if *Expr* then *Stmt*
?	if E_1 then *Stmt*
1	if E_1 then *WithElse*
3	if E_1 then if *Expr* then *WithElse* else *WithElse*
?	if E_1 then if E_2 then *WithElse* else *WithElse*
4	if E_1 then if E_2 then S_1 else *WithElse*
4	if E_1 then if E_2 then S_1 else S_2

This binds the else controlling S_2 to the inner if



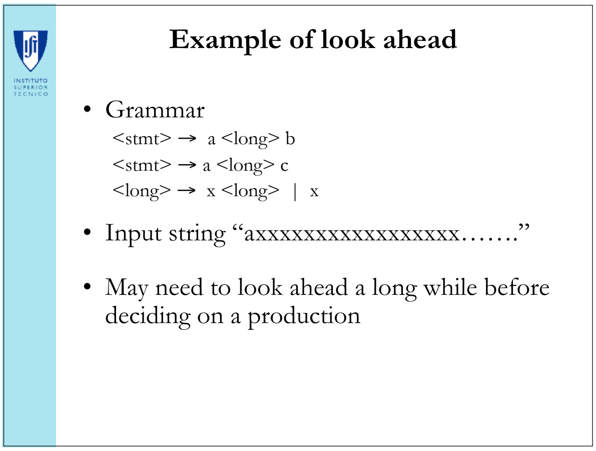
Outline

- Overview of Lexical Analysis
- What is Syntax Analysis?
- Context-Free Grammars
- Derivation and Parse Trees
- Top-down vs. Bottom-up Parsing
- Ambiguous Grammars
- Implementing a Parser



Implementing a Parser

- Implementing a parser for some CFGs can be very difficult
  - Need to look at the input and choose a production
  - Cannot choose the right production without looking a lot ahead

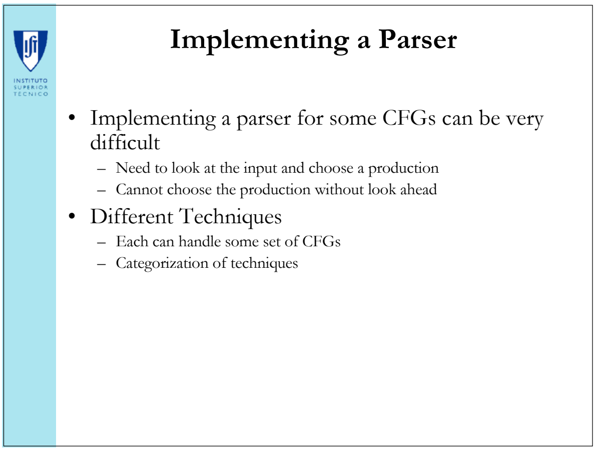


Example of look ahead

- Grammar

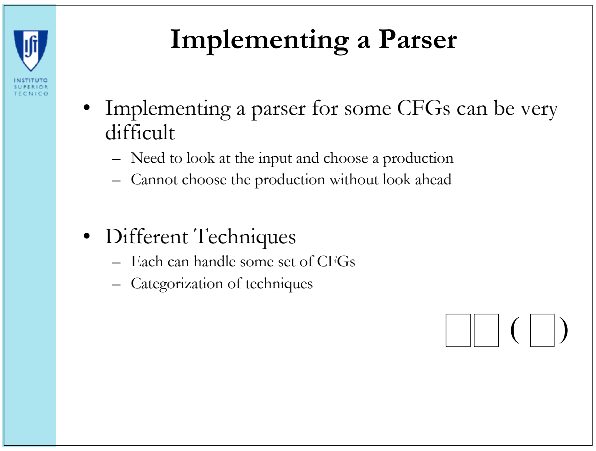
  <stmt> → a <long> b
  <stmt> → a <long> c
  <long> → x <long> | x

- Input string “axxxxxxxxxxxxxxxxxx…….”
- May need to look ahead a long while before deciding on a production



Implementing a Parser

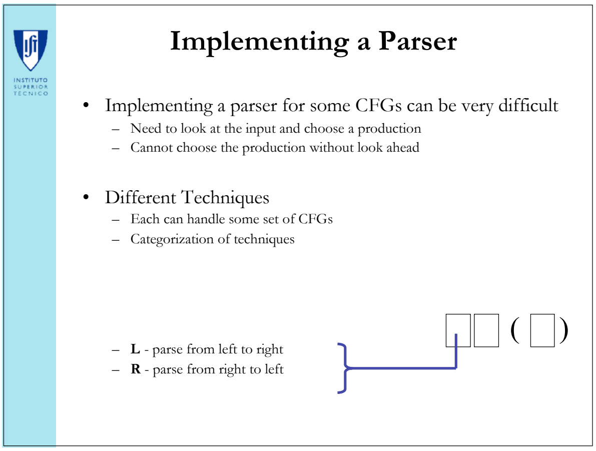
- Implementing a parser for some CFGs can be very difficult
  - Need to look at the input and choose a production
  - Cannot choose the production without look ahead
- Different Techniques
  - Each can handle some set of CFGs
  - Categorization of techniques



Implementing a Parser

- Implementing a parser for some CFGs can be very difficult
  - Need to look at the input and choose a production
  - Cannot choose the production without look ahead
- Different Techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

□□ (□)

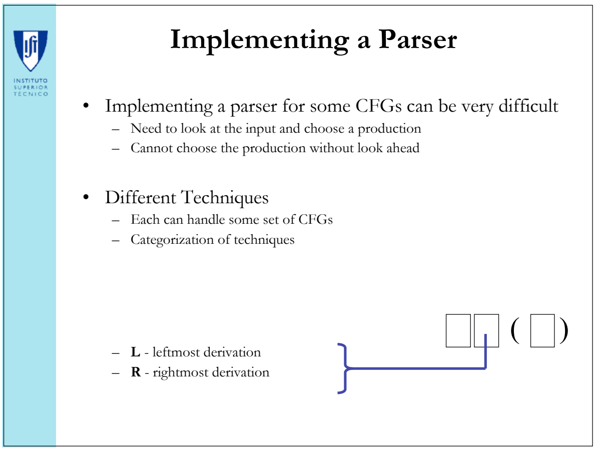


Implementing a Parser

- Implementing a parser for some CFGs can be very difficult
  - Need to look at the input and choose a production
  - Cannot choose the production without look ahead
- Different Techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

□□ (□)

- **L** - parse from left to right
- **R** - parse from right to left

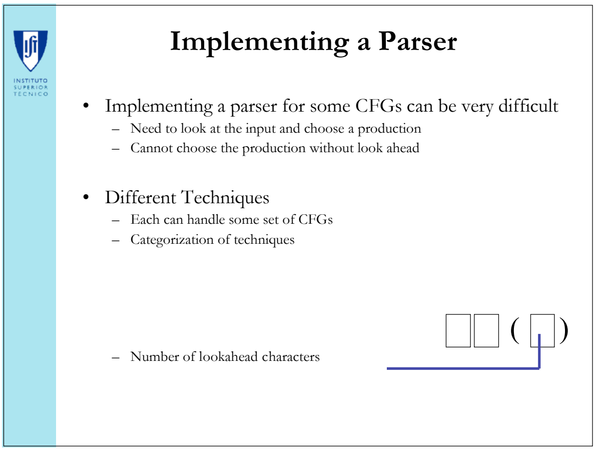


Implementing a Parser

- Implementing a parser for some CFGs can be very difficult
  - Need to look at the input and choose a production
  - Cannot choose the production without look ahead
- Different Techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

□□ (□)

  - **L** - leftmost derivation
  - **R** - rightmost derivation

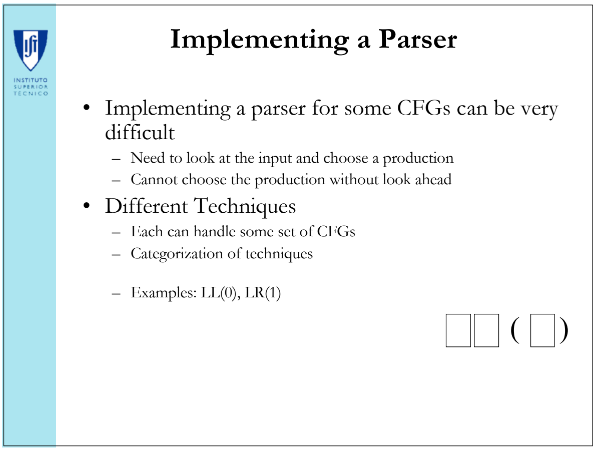


Implementing a Parser

- Implementing a parser for some CFGs can be very difficult
  - Need to look at the input and choose a production
  - Cannot choose the production without look ahead
- Different Techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

□□ (□)

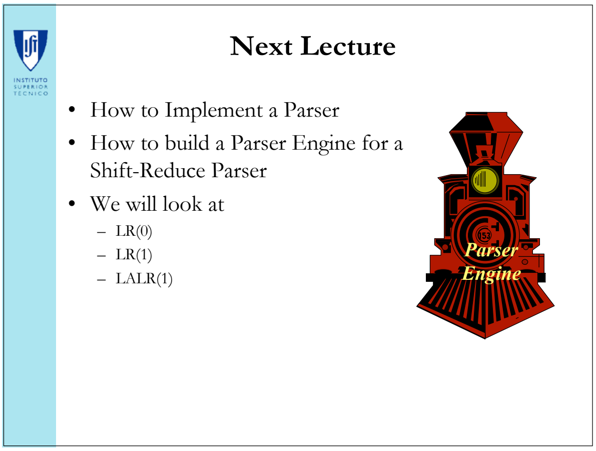
  - Number of lookahead characters



Implementing a Parser

- Implementing a parser for some CFGs can be very difficult
  - Need to look at the input and choose a production
  - Cannot choose the production without look ahead
- Different Techniques
  - Each can handle some set of CFGs
  - Categorization of techniques
  - Examples: LL(0), LR(1)

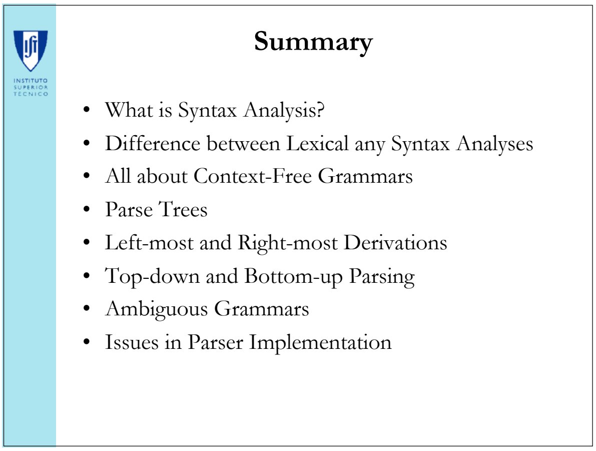
□□ (□)



Next Lecture

- How to Implement a Parser
- How to build a Parser Engine for a Shift-Reduce Parser
- We will look at
  - LR(0)
  - LR(1)
  - LALR(1)

Parser
Engine



Summary

- What is Syntax Analysis?
- Difference between Lexical any Syntax Analyses
- All about Context-Free Grammars
- Parse Trees
- Left-most and Right-most Derivations
- Top-down and Bottom-up Parsing
- Ambiguous Grammars
- Issues in Parser Implementation