ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

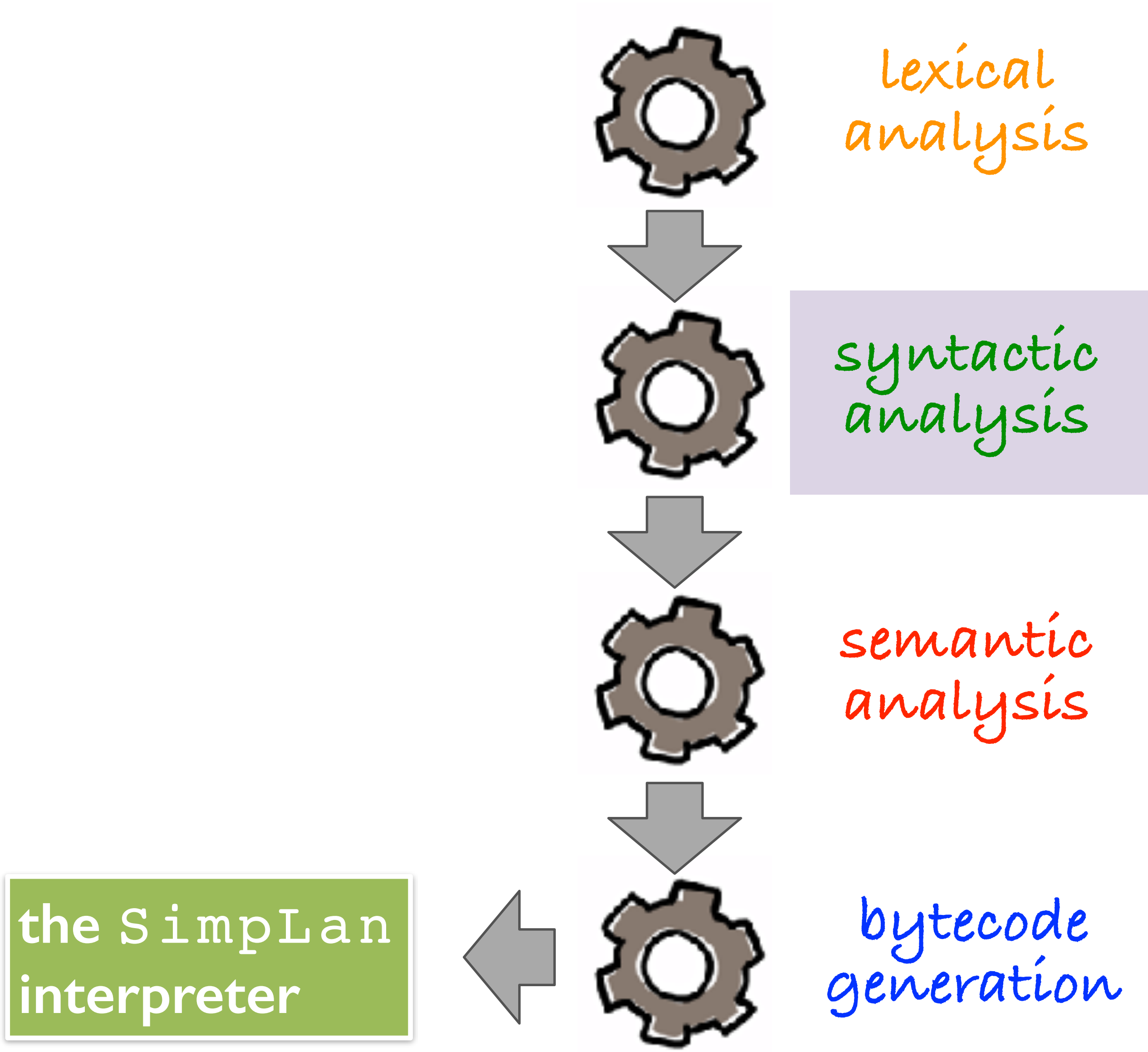
SYNTACTIC ANALYSIS

COSIMO LANEVE

cosimo.laneve@unibo.it

CORSO 72671 - COMPLEMENTI DI LINGUAGGI DI PROGRAMMAZIONE

THIS LECTURE



OUTLINE

* recaps about grammars
* parse trees and ambiguity
* design of a parser (preliminaries)

reference: Torben Morgensen: **Basics of Compiler Design**, chapter 3 (sections 1—5)

DERIVATIONS AND PARSE TREES

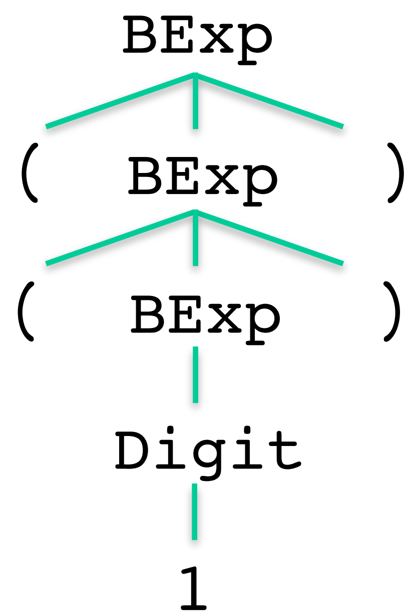
take the grammar

```
BExp → ( BExp )
BExp → Digit
Digit → 0 | 1 | . . . | 9
```

the derivation `BExp` $\Longrightarrow$ `( BExp )` $\Longrightarrow$ `( ( BExp ) )` $\Longrightarrow$ `( ( Digit ) )` $\Longrightarrow$ `( ( 1 ) )`

may be represented graphically by **trees** where

- the **root** is the initial symbol
- the **leaf** is a terminal or ε
- every **internal node** is a non-terminal
- the **edges node-descendant** represent a production



these trees are called **parse trees** = syntax trees

PARSE TREES AND AMBIGUITY

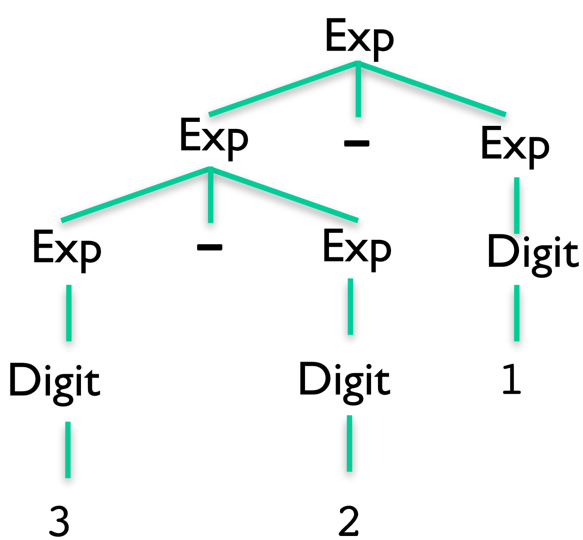
```
Exp → Exp - Exp
Exp → Digit
Digit → 0 | 1 | . . . | 9
```

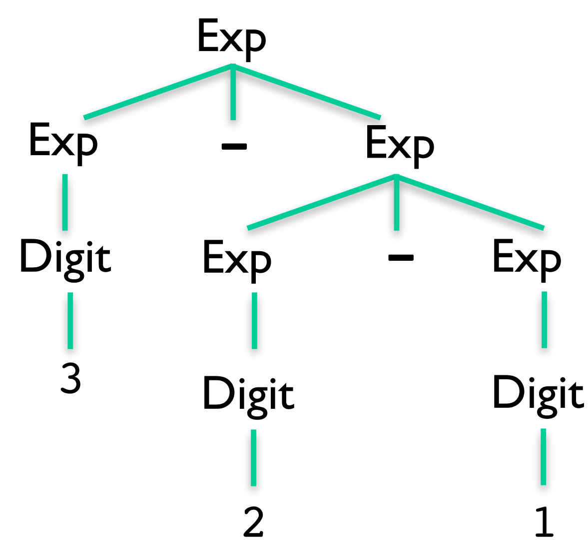
the two leftmost derivations

```
Exp ⟹ Exp - Exp
    ⟹ Exp - Exp - Exp
    ⟹ Digit - Exp - Exp
    ⟹ 3 - Exp - Exp
    ⟹ 3 - Digit - Exp
    ⟹ 3 - 2 - Exp
    ⟹ 3 - 2 - Digit
    ⟹ 3 - 2 -1
```

```
Exp ⟹ Exp - Exp
    ⟹ Digit - Exp
    ⟹ 3 - Exp
    ⟹ 3 - Exp - Exp
    ⟹ 3 - Digit - Exp
    ⟹ 3 - 2 - Exp
    ⟹ 3 - 2 - Digit
    ⟹ 3 - 2 - 1
```

correspond to the **two** parse trees



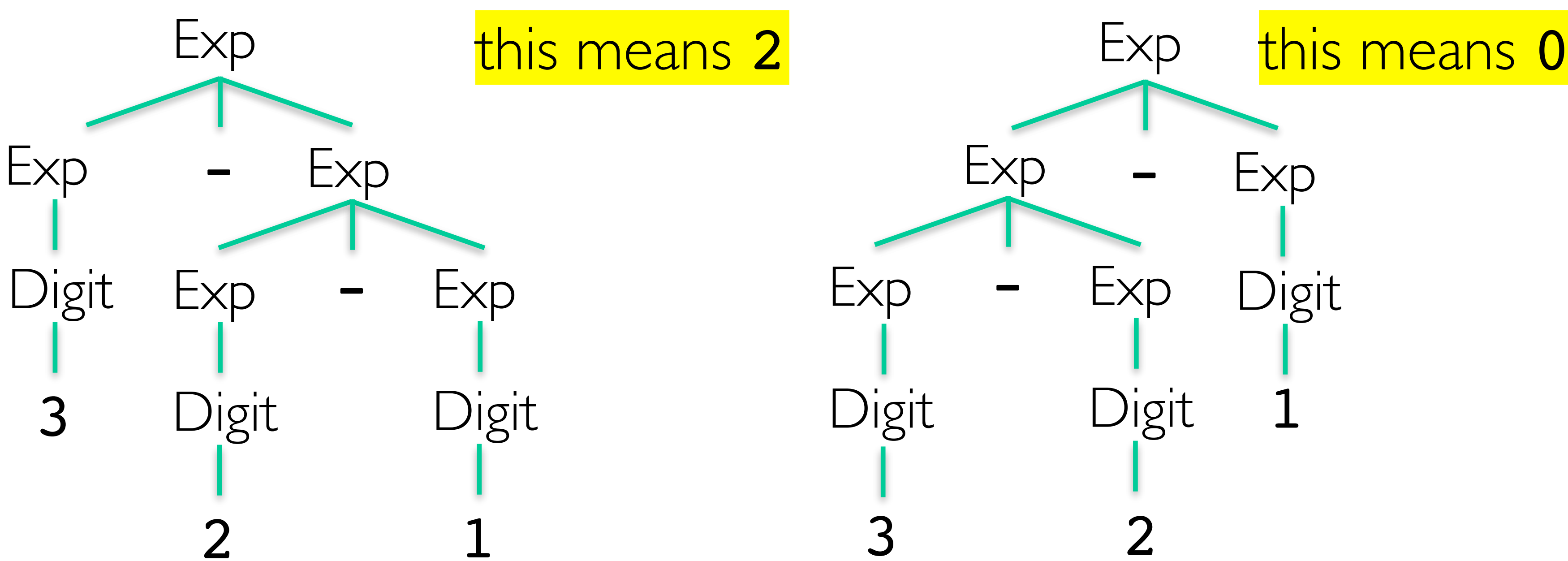


LEFTMOST DERIVATIONS, PARSE TREES AND AMBIGUITY

Definition: ambiguous grammar

let $\mathcal{G}$ be a grammar, if a string in $\mathcal{L}(\mathcal{G})$ has **several leftmost derivations** (or several rightmost derivations) or is **represented by different parse trees**, then $\mathcal{G}$ is **ambiguous**

note: ambiguity means **different semantics** of the same sentence



ambiguity is problematic and must be solved! (see below)

PARSING

once sequences of characters have been recognized in tokens, then one needs to **analyze the syntactic structure** of the sentences/programs to **check whether they belong or not to the language**

parsing = takes in input token sequences and returns **abstract syntax tree** (AST)

example: `if (x == y) z = 1; else z = 2;`

corresponds to the token sequence (lexer's output)

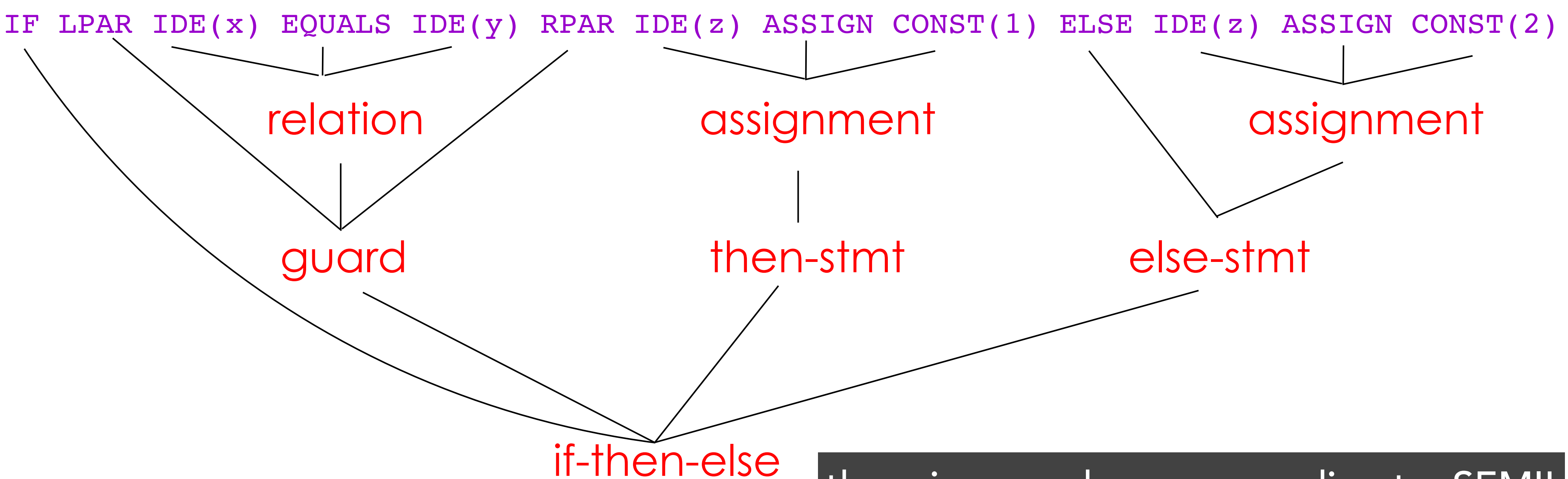
```
IF LPAR IDE(x) EQUALS IDE(y) RPAR IDE(z) ASSIGN
CONST(1) SEMI ELSE IDE(z) ASSIGN CONST(2) SEMI
```

EXAMPLE OF PARSE TREE

the parse tree of

```
if (x == y) z = 1; else z = 2;
```

is



there is no node corresponding to SEMI!

compare it with the previous abstract syntax tree!

PARSE TREES VS. ABSTRACT SYNTAX TREES

parse trees

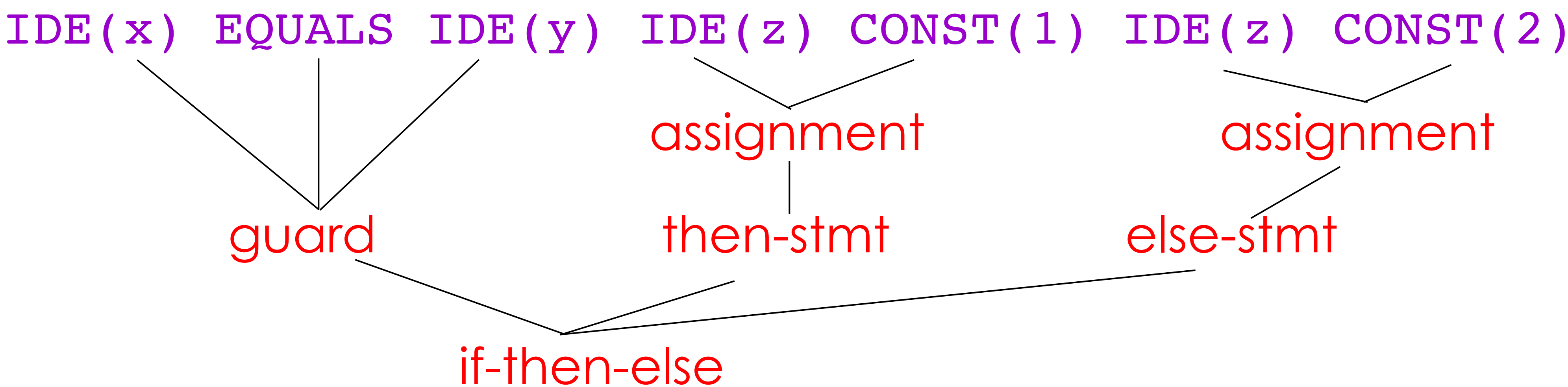
* **have all the tokens**, included those that the parser uses for detecting
  - nesting of sub-expressions (such as parentheses)
  - punctuation marks (semicolons, colons, etc.)
* technically, the parse trees show up all the concrete syntax
* the parse trees are almost never built explicitly — **they are too-much verbose**; they are used during the computations of the parsers

abstract syntax tree (AST)

* remove partial results of the parsing, erasing useless tokens, flattening the tree by removing internal nodes, etc.
* technically, the AST show up an "abstract" version of the syntax

PARSING

the **parser** returns the **abstract syntax tree**



in the abstract syntax tree several tokens are removed!

DESIGN OF A PARSER

it can be done "**by hand**", of course

* ok for small languages
* very hard for real programming languages

or, as for the lexer, it is possible **to use an automatic parser generator**

* you need to specify the syntactic structure of the language (the productions)
* and the generator output the parser

as for the lexer, **we start with a parser done "by hand"** (thus you can understand why it is better to use a parser generator)

FIRST EXAMPLE: THE BEXP GRAMMAR

```
bexp → ( bexp )
bexp → NUM
NUM → (0 | 1 | . . . | 9)+
```

question (before describing the parser): **why a (simple) DFA cannot recognise this language**?

PARSER CODE PRELIMINARIES

- let `TOKEN` be an enumerated data-type that define the possible tokens
  - `LPAR`, `RPAR`, `NUM`
- let `in[]` be a (global) array whose elements are of type `TOKEN` and that represent the sequence of tokens returned by the lexer
- let `next` be a (global) integer that represents the index of the token sequence

THE PARSER CODE DONE "BY HAND"

```
bexp → (bexp)
bexp → NUM
NUM → (0 | 1 | . . . | 9)+
```

```
public void ParseBexp() {
   next = next+1 ;
   TOKEN nextToken = in[next];
   if (nextToken == NUM) return() ;
   else if (nextToken == LPAR){
            ParseBexp();
            next = next+1 ;
            if (in[next] == RPAR) return() ;
            else System.out.print("syntax error") ;
   } else System.out.print("syntax error") ;
}
```

`nextToken` is useless !

WHERE IS BUILT THE PARSE TREE?

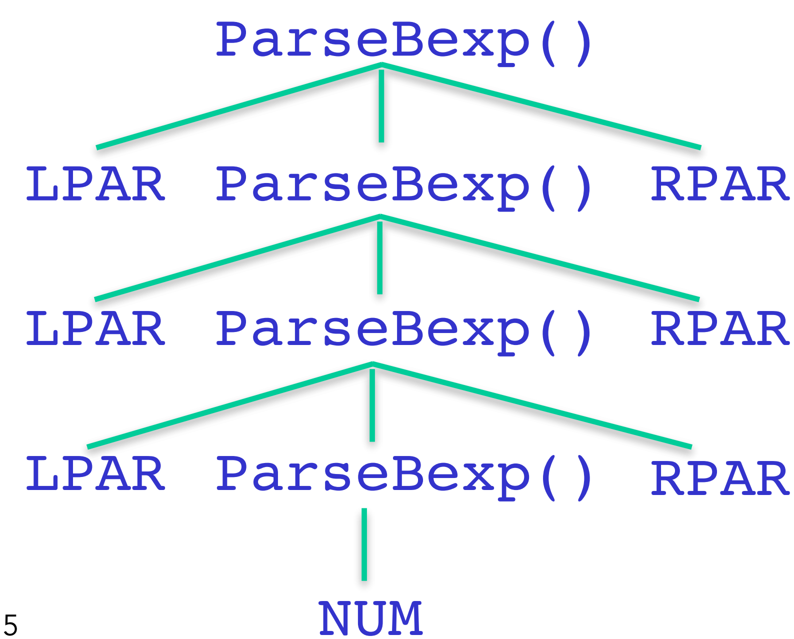
in the previous method: **`NOWHERE`**!

however it is possible to extend the method **`ParseBexp`** in order to build the parse tree following the invocations

example: with input `(((1)))` the lexer returns

`LPAR LPAR LPAR NUM RPAR RPAR RPAR`

and the (extended) parser builds



SECOND EXAMPLE: THE LANGUAGE EXP

```
exp → exp - exp
exp → NUM
NUM → (0 | 1 | . . . | 9)+
```

let **TOKEN** be an enumerated data-type that defines the possible tokens (as before)

- we have tokens **MINUS**, **NUM**

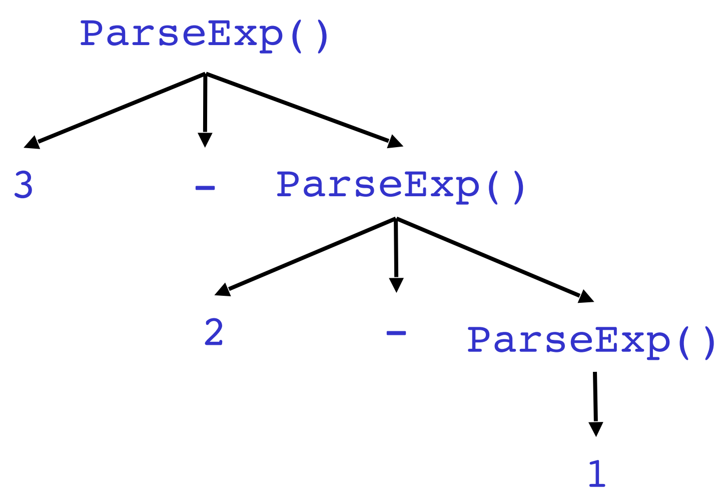
```
public void ParseExp(){
        next = next+1; TOKEN nextToken = in[next];
        if (nextToken==NUM) {
              if (in[next+1]==MINUS)  {
                  next = next+1; ParseExp();
              } else return();
        } else System.out.print("syntax error");
}
```

SUBTRACTION EXPRESSIONS CONTINUED

remarks:

* a more complex language
  * hence, harder to see how the parser works (and if it works correctly at all)

* the parse tree is actually not really what we want
  * consider input `3-2-1`
  * what's undesirable about this parse tree's structure?



WE NEED A CLEAN SYNTACTIC DESCRIPTION

just like with the scanner, writing the parser by hand is painful and error-prone

- consider adding **+**, *****, / to the last example!

let's separate the **what** and the **how**

- **what**: the syntactic structure — described with a context-free grammar
- **how**: the parser — which reads the grammar as input and produces the parse tree

THE WHAT: CONTEXT-FREE GRAMMARS

idea: we can describe the syntactic structure by using context-free grammars!

programming language constructs have **recursive structure**

* this is the reason why our hand-written parser had this structure, too

example: an expression is either:

- a number, or
- a variable, or
- an expression + expression, or
- an expression - expression, or
- an (expression), or
- …

simple arithmetic expressions:

```
exp  →  NUM | ID | ( exp )
        | exp - exp | exp + exp
```

THE HOW: USE DERIVATIONS FOR PARSING?

a program (a string of tokens) has **no syntax error** if it can be derived from the grammar

* so far you only know how to derive some (any) string
* you do not know how to check whether a given string is derivable or not

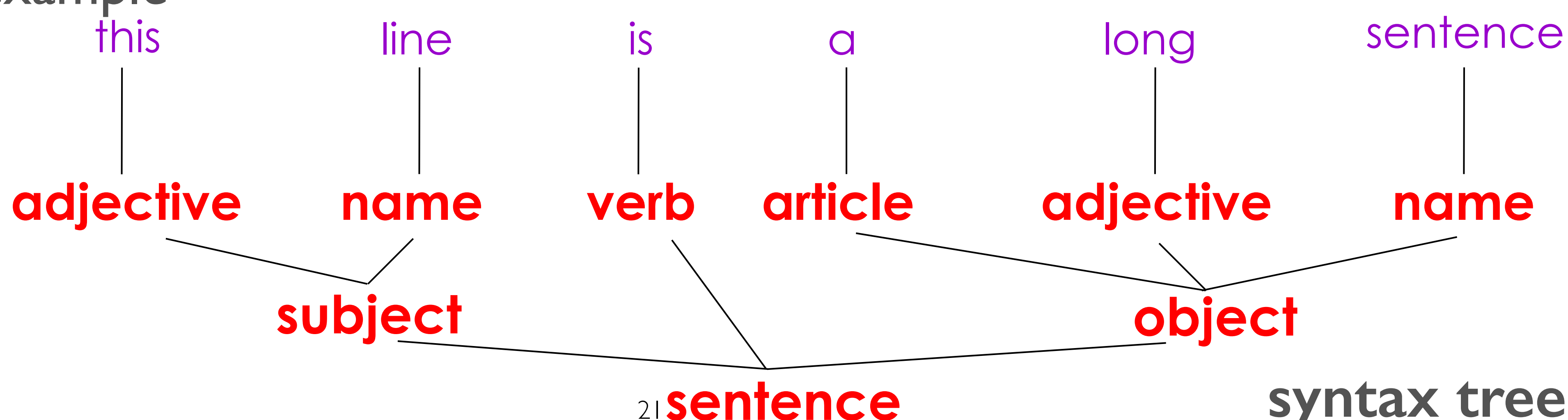
how to do parsing?

PARSING

once the sequence of characters have been recognized as sequence of tokens, one needs to analyze the syntactic structure of sentences/programs to check whether they belong to the language or not

parsing = takes in input sequences of tokens and returns abstract syntax trees (AST)

example



syntax tree

COMPARISON WITH LEXICAL ANALYSIS

Phase	Input	Output
Lexer	sequence of characters	sequence of tokens
Parser	sequence of tokens	AST, built from parse tree

NEXT LECTURE