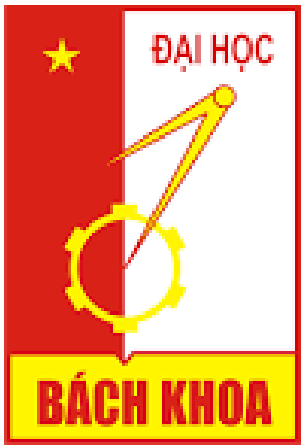
Nature of Programming Languages

Topic 4. Language syntax and semantics

Han, Nguyen Dinh (han.nguyendinh@hust.edu.vn)



Faculty of Mathematics and Informatics
Hanoi University of Science and Technology

Outline

1. Language processors
2. Lexical structure of programming languages
3. Language grammars
4. Parsing techniques and tools
5. Case study: Developing a parser for TinyAda

1

Language processors

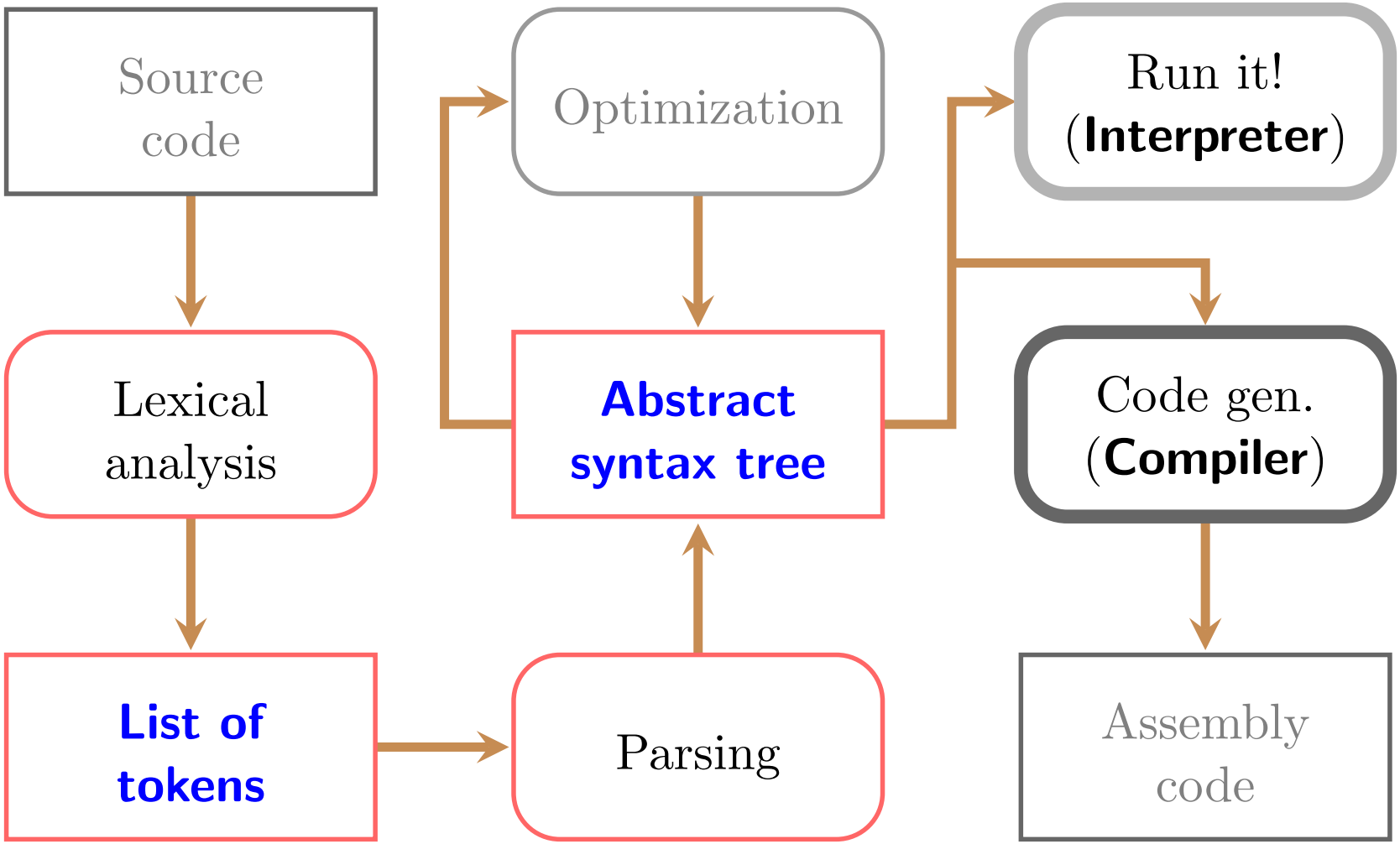
Language processors

We recall that a language is a set of sentences, which in turn are sequences of *terminal symbols*. Every language has syntax and semantics

For a programming language, its syntax or *structure* influences how programs are written (i.e. how expressions, commands, declarations, etc., are put together to *form* programs), and its semantics is concerned with the *meaning* of programs

However, before a program can be run, it first must be translated into a form in which it can be executed by a computer. The system for processing programs is called *a language processor*. Examples of language processors are compilers, interpreters, and auxiliary tools like syntax-directed editors

Interpreter and compiler structure



2

Lexical structure of programming languages

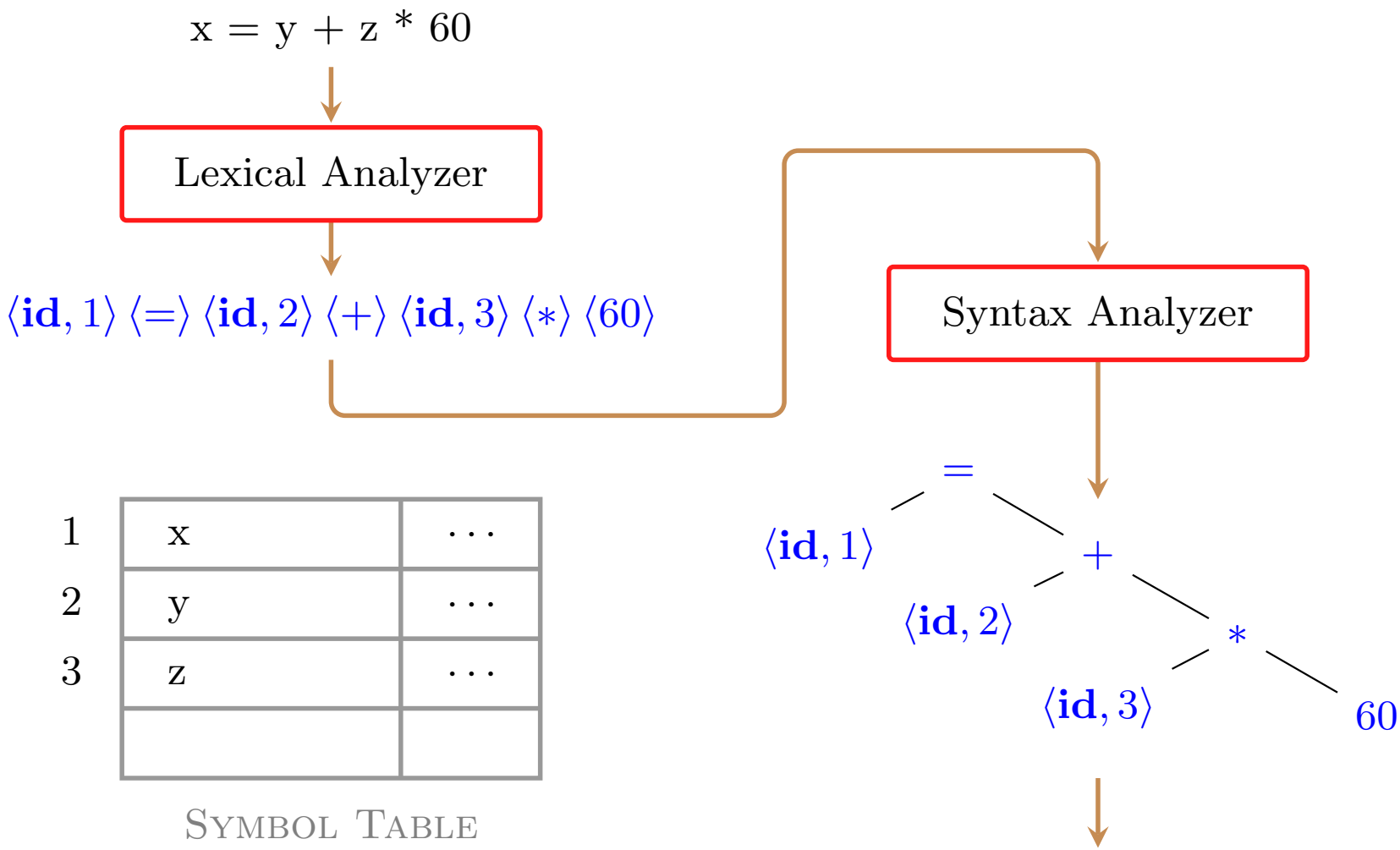
Lexical structure of programming languages

The lexical structure of a programming language is the structure of its words or **tokens**, where each token is represented by a string called **lexemes** denoting a logical entity in the programming language (e.g. a *number*, *constant*, *identifier*, *literal*, *keywords* like if, then, else, etc.)

Typically, during its *lexical analysis* phase, a translator collects sequences of characters from the input program and forms them into tokens. The translator then processes these tokens in its *parsing* phase to determine the program's syntactic structure. The language processors responsible for these two phases are so called Lexical Analyzer and Syntax Analyzer, respectively

Example: The string x = y + z * 60 is translated as follows

Translation of an assignment statement



Specification of tokens

The format of a program can affect the way tokens are recognized. However, most modern languages are *free format* as the format has no effect on their program structure

Tokens in a programming language are often described in English, but they can also be described formally by **regular expressions**, which are descriptions of *patterns* of characters

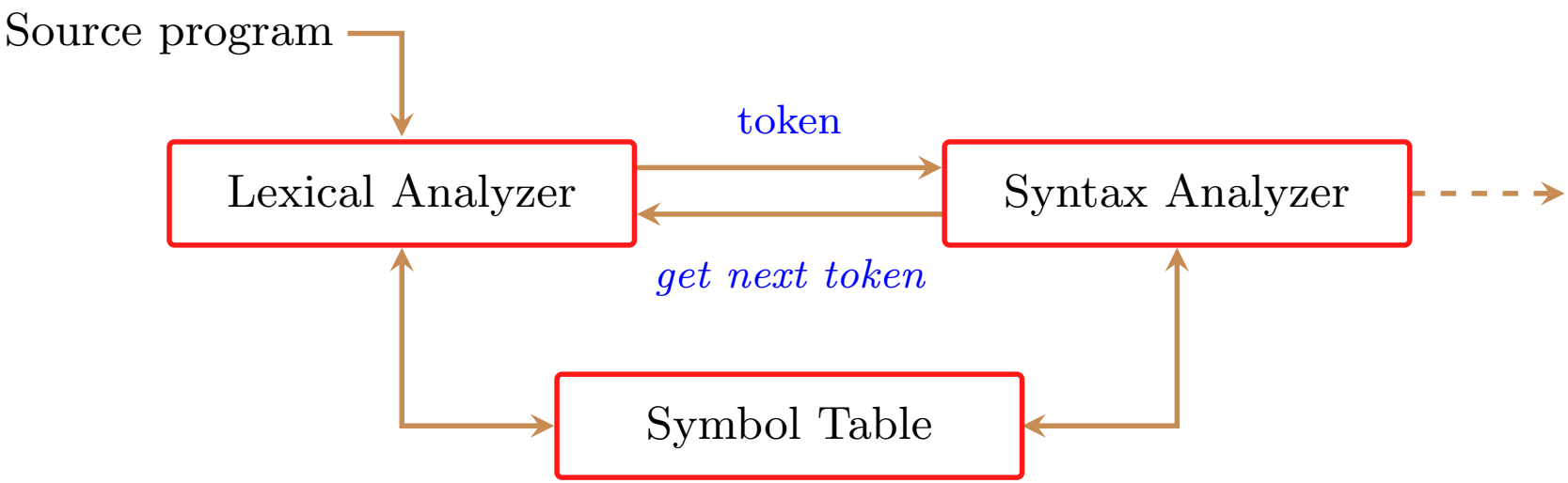
Regular expressions have three basic operations: *concatenation*, *repetition*, and *choice* or *selection*. Repetition is indicated by the use of an asterisk after the item to be repeated, choice is indicated by a vertical bar between the items from which the selection is to be made, and concatenation is given simply by sequencing the items without an explicit operation. Parentheses are also often included to allow for the grouping of operations

Specification of tokens

For example, $(a|b) * c$ is a regular expression indicating 0 or more repetitions of either the characters a or b (choice), followed by a single character c (concatenation); strings that match this regular expression include $ababaac$, aac, $babbc$, and just c itself, but not $aaaab$ or bca

Regular expression notation is often extended by additional operations and special characters to make them easier to write. For example, $[0-9]+$ is a regular expression for simple integer *constants* consisting of one or more digits (characters between 0 and 9). Also, the regular expression $[0-9]+(.[0-9]+)?$ describes simple unsigned floating-point *literals* consisting of one or more digits followed by an optional fractional part consisting of a decimal point, followed again by one or more digits

Recognition of tokens



Exercise: Try to run C codes of a scanner for simple integer arithmetic expressions given in Figure 6.1, page 207 of the textbook. Note that no order for the tokens is specified by the scanner. Defining and recognizing an appropriate order is the subject of syntax and parsing

3

Language grammars

Language grammars

It is important to note that, the parsing phase of a language translator can be automated if the language syntax is expressed in Backus-Naur Forms (BNF) using *context-free grammars*

Indeed, a context-free grammar consists of a series of *grammar rules*. These rules, in turn, consist of a left-hand side that is a single phrase structure name, followed by the metasymbol "$\longrightarrow$", followed by a right-hand side consisting of a sequence of items that can be symbols or other phrase structure names. For example, to describe that simple sentences consist of a noun phrase and a verb phrase followed by a period, we use the following grammar rule:

$$\textit{sentence} \longrightarrow \textit{noun-phrase verb-phrase}.$$

Language grammars

A typical simple example of the use of a context-free grammar in a programming language is the description of simple integer arithmetic expressions with addition and multiplication as given below

$$\textit{expr} \longrightarrow \textit{expr} + \textit{expr} \,|\, \textit{expr} * \textit{expr} \,|\, (\textit{expr}) \,|\, \textit{number}$$
$$\textit{number} \longrightarrow \textit{number}\ \textit{digit} \,|\, \textit{digit}$$
$$\textit{digit} \longrightarrow 0 \,|\, 1 \,|\, 2 \,|\, 3 \,|\, 4 \,|\, 5 \,|\, 6 \,|\, 7 \,|\, 8 \,|\, 9$$

In a grammar rule, the names for phrase structures (e.g. *expr*, *number* and *digit*) are called **nonterminals**, since they are broken down into further phrase structures. The token symbols (e.g. +, 0, 1, etc.) are also called **terminals**, since they cannot be broken down

Language grammars

Now we can construct a legal expression according to the foregoing grammar as follows: We start with the symbol *expr* (the *start symbol* for the grammar) and proceed to replace left-hand sides by choices of right-hand sides in the foregoing rules. This process is called a **derivation** in the language. For example, below we have a derivation for the number 234:

$$\begin{aligned} \textit{number} &\Rightarrow \textit{number digit} \Rightarrow \textit{number digit digit} \\ &\Rightarrow \textit{digit digit digit} \\ &\Rightarrow 2\ \textit{digit digit} \Rightarrow 23\ \textit{digit} \Rightarrow 234 \end{aligned}$$

Thus, grammar rules are also called **productions**, since they "produce" the strings of the language using derivations. Productions are in **BNF** if they are as given using only the metasymbols "$\longrightarrow$" and "|". Sometimes parentheses are also allowed

4

Parsing techniques and tools

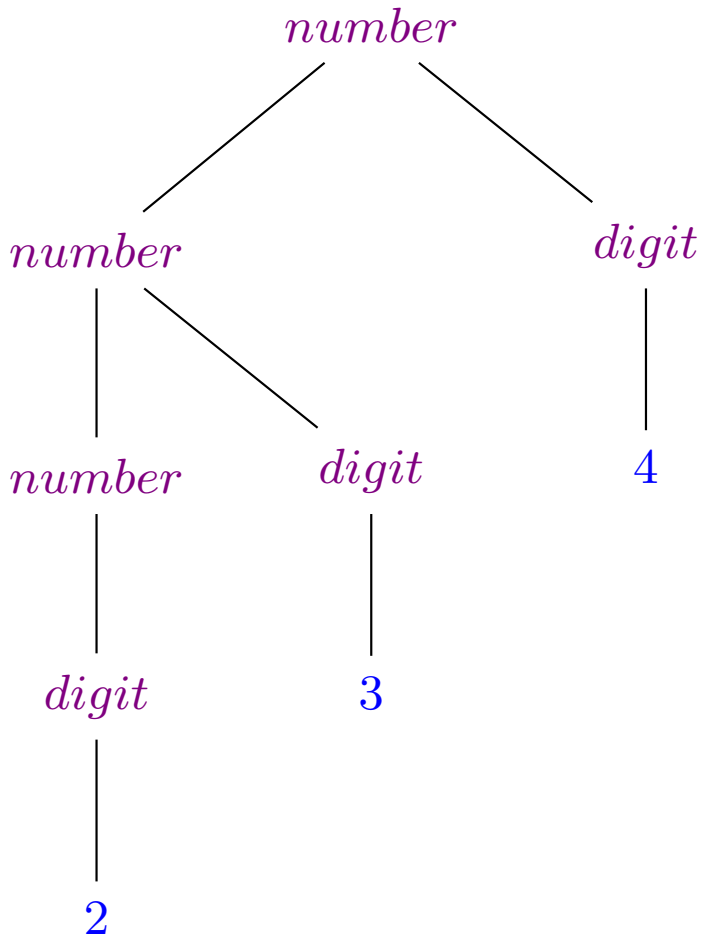
Parsing techniques and tools

Syntax establishes structure, not meaning, but the meaning of a sentence (or program) is related to its syntax. For example, in the grammar for expressions, when we write:

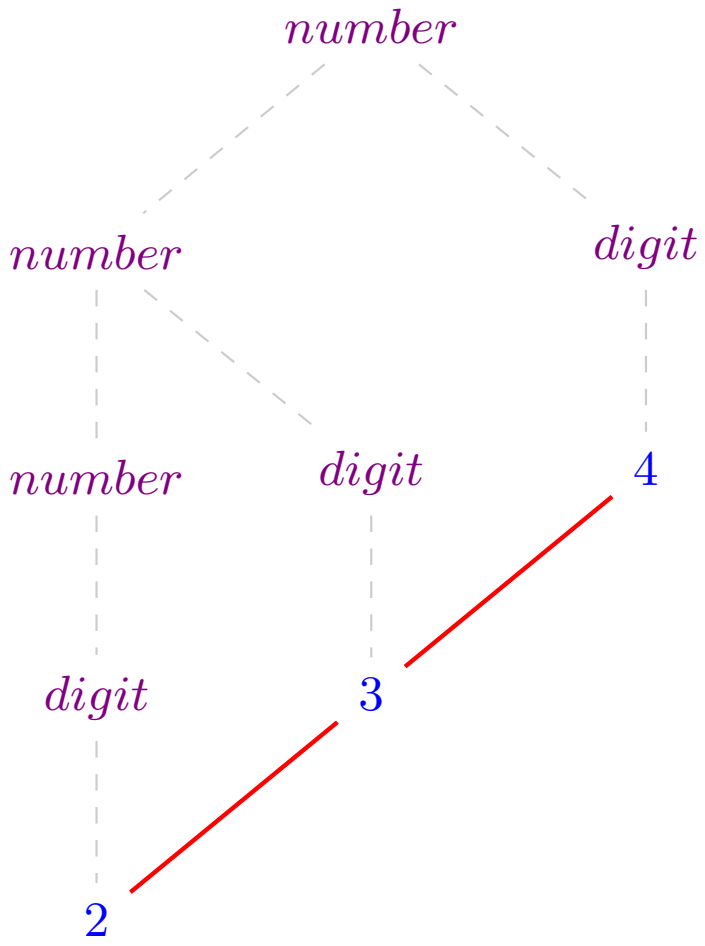
$$expr \longrightarrow expr + expr$$

we expect to add the values of the two right-hand expressions to get the value of the left-hand expression. This process of associating the semantics of a construct to its syntactic structure is called **syntax-directed semantics**. We must, therefore, construct the syntax so that it reflects the semantics we will eventually attach to it as much as possible. Syntax-directed semantics could just as easily have been called *semantics-directed syntax*

To make use of the syntactic structure of a program to determine its semantics, we must have a way of expressing this structure as determined by a derivation. A standard method for doing this is with **a parse tree**, which is a graphical depiction of the replacement process in a derivation. For example, here we have the parse tree for the number 234



A parse tree is labeled by non-terminals at interior nodes and terminals at leaves. Thus, all the terminals and nonterminals in a derivation are included in the parse tree. However, not all the terminals and nonterminals are necessary. For example, the structure of the number 234 can be completely determined from the condensed parse tree or *abstract syntax tree* (in red) shown in the right hand side figure



Parsing techniques and tools

The simplest form of a parser is a **recognizer** - a program that accepts or rejects strings, based on whether they are legal strings in the language. More general parsers build parse trees and carry out other operations

Given a grammar in one of the forms we have discussed, how does it correspond to the actions of a parser? One method of parsing attempts to match an input with the right-hand sides of the grammar rules. The parsers using this method are bottom-up parsers. In the other major parsing method, top-down parsing, nonterminals are expanded to match incoming tokens and directly construct a derivation. Both top-down and bottom-up parsing can be automated; that is, a program, called a **parser generator**, can be written that will automatically translate a BNF description into a parser

5

Case study:

Developing a parser for TinyAda

Thank you very much for your attention!