

Syntax (II)

EBNF

- To **increase the clarity and brevity** of syntax descriptions, **Extended BNF (EBNF)** was introduced.
- Based on BNF, EBNF has a few more metasympols other than those from BNF:
 - Curly braces **{}**: include the enclosed symbols **0 or more times**
 - Parentheses **()**: include the enclosed symbols **1 or more times**
 - Square brackets **[]**: indicate an **optional sequence of symbols**
- **Example:** Describe the integer formally in EBNF

$$Integer \rightarrow Digit\{Digit\}$$

- The above production cannot avoid the situation where an integer with multiple digits starts with 0. How to rewrite the production to avoid the situation?

$$\begin{aligned} Digit &\rightarrow 0 \mid \dots \mid 9 \\ nonzeroDigit &\rightarrow 1 \mid \dots \mid 9 \\ Integer &\rightarrow Digit \mid nonzeroDigit(Digit) \end{aligned}$$

or

$$\begin{aligned} zero &\rightarrow 0 \\ nonzeroDigit &\rightarrow 1 \mid \dots \mid 9 \\ Integer &\rightarrow zero \mid nonzeroDigit\{[zero, nonzeroDigit]\} \end{aligned}$$

Derivation

- To **determine** if a string is **valid** according to a grammar.
- A derivation is **a series of replacements** defined by the productions.
- Leftmost Derivation steps
 - The derivation **begins** from the **start symbol**.
 - The production rules **iteratively replace nonterminal** symbols with **nonterminal and terminal** symbols **until** there are **no more nonterminal** symbols in the string.
 - In each iteration, apply a production rule to the leftmost symbol.
 - If the **terminal symbols match** the string, the string is **valid**.
- **Example:** Given the formal definition of integer, check whether 312 is a valid integer.

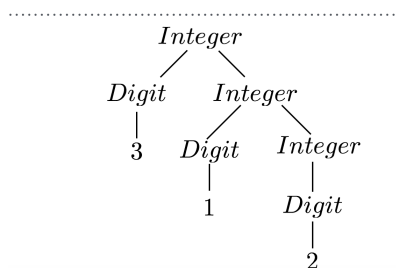
$$\begin{aligned} Integer &\rightarrow Digit \mid DigitInteger \\ Digit &\rightarrow 0 \mid \dots \mid 9 \end{aligned}$$

$$\begin{aligned}
 \text{Integer} &\Rightarrow \text{DigitInteger} \\
 &\Rightarrow 3\text{Integer} \\
 &\Rightarrow 3\text{DigitInteger} \\
 &\Rightarrow 31\text{Integer} \\
 &\Rightarrow 31\text{Digit} \\
 &\Rightarrow 312
 \end{aligned}$$

Parse Tree

- A [graphical form](#) of derivation.
- Each derivation step corresponds to a new subtree.
- **Example:** Draw a top-down parse tree for 312 by using the following rules

$$\text{Integer} \rightarrow \text{Digit} | \text{DigitInteger}$$

$$\text{Digit} \rightarrow 0 | \dots | 9$$


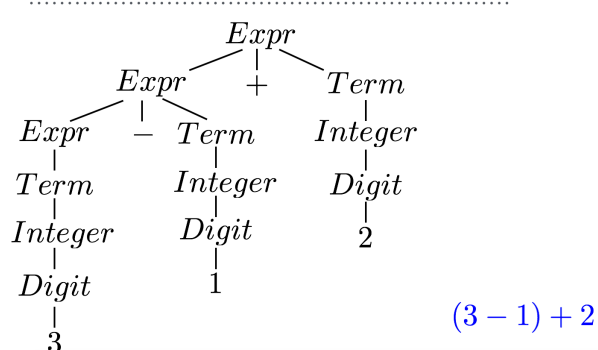
Operator Associativity

- The order of non-terminal symbols on the right hand side of productions is important.
- Different orders will generate parse trees with different associativities.
- **Example 1:** Draw a parse tree for $3 - 1 + 2$ using the following rules

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

$$\text{Term} \rightarrow \text{Integer}$$

$$\text{Integer} \rightarrow \text{Digit} \{ \text{Digit} \}$$

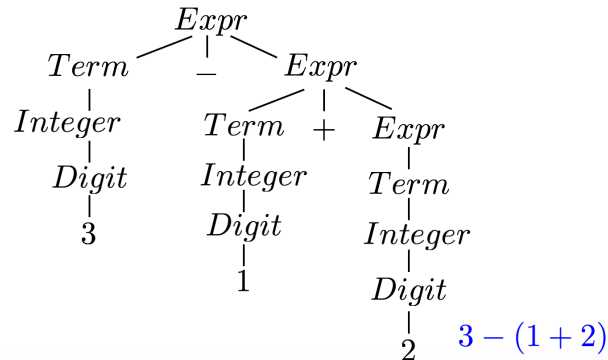
$$\text{Digit} \rightarrow 0 | \dots | 9$$


- **Example 2:** Draw a parse tree for $3 - 1 + 2$ using the following rules

$$Expr \rightarrow Term + Expr | Term - Expr | Term$$

$$Term \rightarrow Integer$$

$$Integer \rightarrow Digit \{ Digit \}$$

$$Digit \rightarrow 0 | \dots | 9$$


- In the first $3 - 1 + 2$ example, the result is interpreted as $(3 - 1) + 2$, as the deepest tree is evaluated first. In the second $3 - 1 + 3$ example, the result is interpreted as $3 - (1 + 2)$. They have different associativities, and the second one is not the associativity it should be.
- Maintaining the order of operations is essential in programming languages.
- In order to specify hierarchies of operations, grammars can become extremely complex.