

Syntax (II)

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.

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

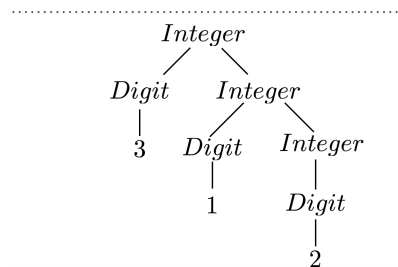
$$\begin{aligned} Integer &\Rightarrow DigitInteger \\ &\Rightarrow 3Integer \\ &\Rightarrow 3DigitInteger \\ &\Rightarrow 31Integer \\ &\Rightarrow 31Digit \\ &\Rightarrow 312 \end{aligned}$$

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

$$Integer \rightarrow Digit | DigitInteger$$

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



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.

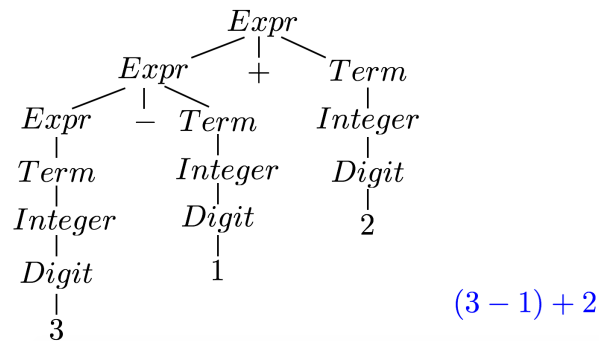
- **Example 1:** Draw a parse tree for $3 - 1 + 2$ using the following rules

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

$$Term \rightarrow Integer$$

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

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



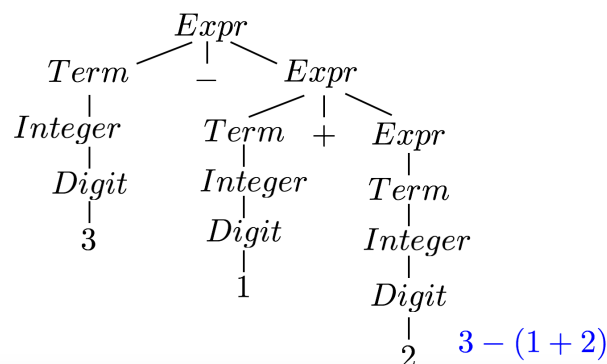
- **Example 2:** Draw a parse tree for $3 - 1 + 2$ using the following rules

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

$$Term \rightarrow Integer$$

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

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



- In the first $3 - 1 + 2$ example, the result is interpreted as $(3 - 1) + 2$, as an operator inside an operand's subtree binds more tightly than the operator combining those subtrees. 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.

Flexible & Ambiguous Grammars

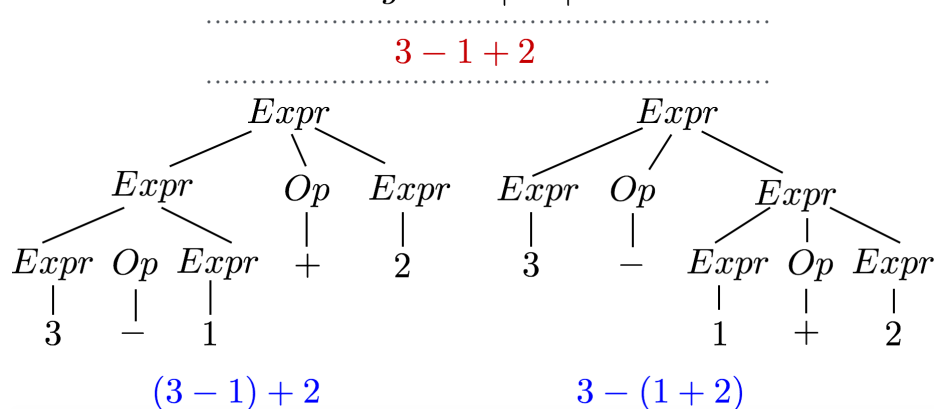
- Let's work on one more example.
- **Example:** Build a parse tree for **3 - 1 + 2** based on the following rules

$$Expr \rightarrow Expr \ Op \ Expr | (Expr) | Integer$$

$$Op \rightarrow + | - | * | /$$

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

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



- We can generate two distinct parse tree using above rules.
- We call a grammar **ambiguous grammar** if its language contains at least one string that has two or more distinct parse trees.
- Ambiguous grammars are not good as they may confuse the compilers. Compilers can generate two different results for the same expression.
- However, sometimes, we may want to use an ambiguous grammar to simplify the number of rules requires. In this case, ambiguities in grammars are generally resolved using additional rules.
 - For example, if we have a table of precedence and a default left-to-right ordering of operators of equal precedence, then we can resolve any ambiguities that arise.
- **Dangling else** is a common ambiguity in language syntax.
 - When an if statement is contained inside an if statement, which if statement does a subsequent else belong to?
 - Consider the following code snippet

```

if (x < 0)
    if (y < 0)
        y = y + 1;
    else
        y = 0;

```

- Without inserting curly braces, it seems like that the else branch could match either if condition. Different languages address this ambiguity differently.
- Solution of C
 - C addresses by including a description in the nature language in its documentation that an else clause is associated with the textually nearest if statement in any ambiguous case.

```

#include <stdio.h>

int main (int args, char *argv[]) {
    int a = -1;
    int b = 1;

    if (a < 0)
        if (b < 0)
            printf("here\n");
        else
            printf("there\n");
}

```

- So, the output of the follow code is “there.”
- Solution of Java
 - Clearly defined in grammar to address the ambiguity.
 - It is not permitted that an if statement without an else clause as the single statement after an if.
 - The following code snippet, for example, will not do what the tabbing implies. The actual

```

public class Ambiguity {
    public static void main (String args[]) {
        int a = -1;
        int b = 1;

        if (a < 0)
            if (b < 0)
                System.out.println("here");
        else
            System.out.println("there");
    }
}

```

output is “there”, since Java consider the else branch belongs to the second if statement.

- Solution of Python

- Require nested if statements to be indented. The actual output is "".

```
a = -1
b = 1

if (a < 0):
    if (b < 0):
        print "here\n"
    else:
        print "there\n"
```