BITS Pilani
Pilani Campus

innovate achieve lead

CS/IS F214
Logic in Computer Science

MODULE: PROPOSITIONAL LOGIC: SYNTAX

Languages and Syntax

LANGUAGES AND GRAMMARS

Languages

- A language is usually characterized by three aspects:
  - *form*, *meaning*, and *usage*.
- The study of these aspects are referred to as:
  - ***syntax***, ***semantics***, and ***pragmatics*** respectively.

syntax

study of **form**: i.e. *how sentences in the language are formed*

semantics

study of **meaning**: i.e. *how sentences in the language are interpreted*

pragmatics

study of **usage**: i.e.

- *how sentences are used in context(s)* and
- *how such usage varies from context to context*

Languages

Such studies of language are common in the context of natural languages

Syntax – e.g.

The *subject* appears before the *verb* in ***statements*** and after the first *verb* in ***questions.***

Pragmatics – e.g.

- *monologue* (reading and writing) vs. *dialogue* (speaking and listening)
- American English vs. British English:

It's five after eight	It's five past eight
I'll see you over the weekend	I'll see you at the weekend
I've got the answer	I've gotten the answer
I just ate	I've just eaten

Semantics – e.g.

to fasten = *to hold (something) and to stay (it)*

- fasten upon binding: ***lash***
- fasten upon entering: ***latch***
- fasten upon grounding: ***stake***
- fasten upon linking: ***chain***
- fasten upon surrounding: ***clasp***

[from the *Cambridge Encyclopedia of the English Language*]

Languages

Such studies of language are also common in the context of *artificial languages* : e.g.

- programming languages (*such as* **C**)
- Logics (*such as* **Propositional Logic**)
- Markup Languages (*such as* **HTML**)

Syntax – e.g.

The *conditional* statement

- starts with the keyword ***if***
- followed by a *conditional* expression
- and then by two branch statements separated by the keyword ***else***.

Semantics – e.g.

Arithmetic operations on ***int*** values are performed modulo $\mathbf{2^n}$ where **n** is ***sizeof int***

Pragmatics – e.g.

sizeof int is usually the size of the machine word

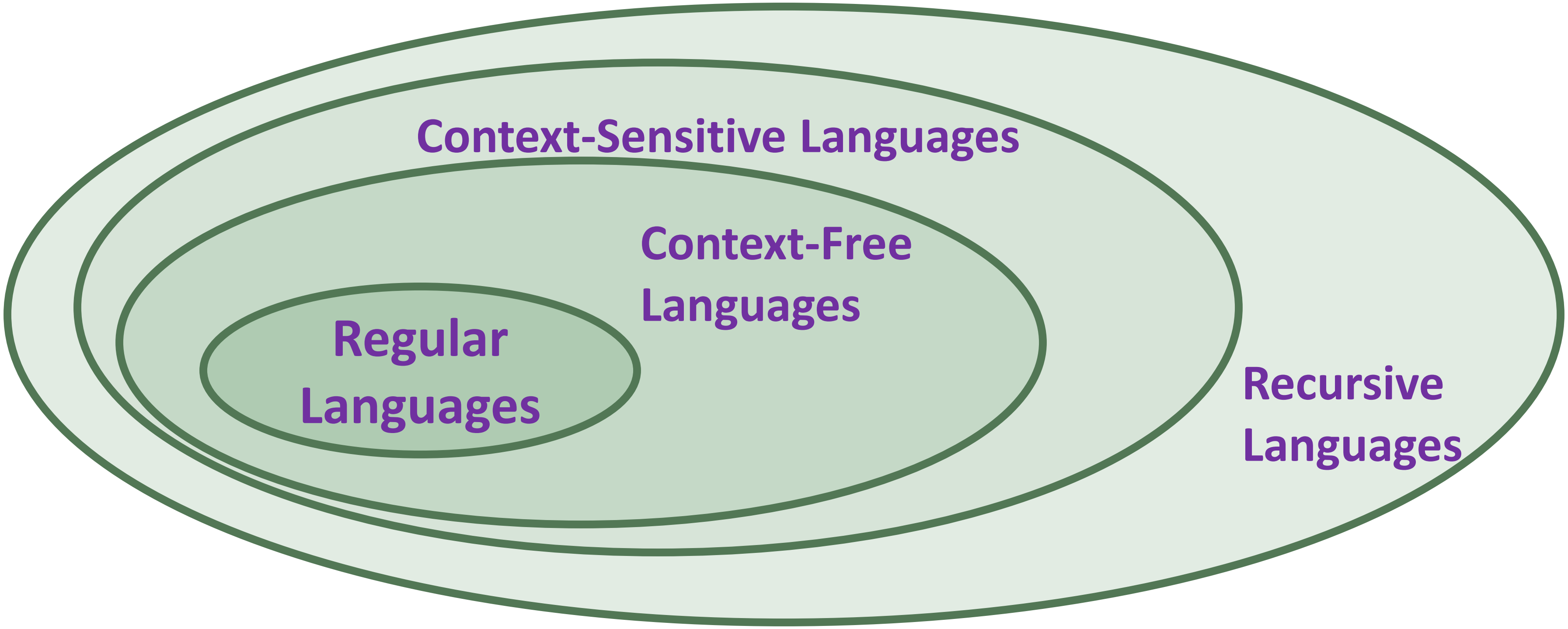
Grammars – Panini

- A (significant) part of syntax of a language is its ***grammar***
  - a grammar defines the *structure of sentences* in a language.

- The oldest and the most comprehensive formal grammar for any language was specified by **Panini** (for ***Sanskrit***) around 4th century B.C.
  - Panini invented – what are now referred to as – ***term re-write rules*** and ***context-free grammars***, as a notation for specification.

Grammars – Chomsky

- Later, other linguists, notably **Chomsky** (in mid 20th century) defined an inclusion *hierarchy of languages* (*and grammars* for specifying such languages).



Grammars vs. Machines

- *Recursive grammars* are equivalent to *recursive functions*
    - [RECALL: *Recursive functions* are equivalent to *Turing machines*.]
- i.e.
    - for any language defined by a *recursive grammar*:
        - there is a *Turing machine* that can decide whether a string belongs to the language or not.
    - and for any *Turing machine*:
        - there is a grammar that can be used to compute what the *Turing machine* computes.

Equivalences between grammars and machines

Grammars	Machines
Recursive	Turing Machines
Context-Sensitive	TMs with a *linear-bounded* tape*
Context-Free	TMs with a stack (*instead of a tape*)
Regular	TMs with a *read-only* tape

*Linear-bounded tape refers to the length of the tape being *bounded by a linear function* of the length of the input.

A Turing Machine with a read-only tape is referred to as a ***Finite State Machine***

Grammars and Rules

- Grammars are usually specified as rules for *generating* or *verifying* sentences belonging to the language (*that is defined by the grammar*):
  - these rules are also referred to as “***rewrite rules***” and are of the form:
    - <sequence of symbols>$_1$ ---> <sequence of symbols>$_2$
  - where
    - this rule states that *sequence$_1$* can be replaced by *sequence$_2$* (or vice versa)
    - and by repeated such replacements any string in the language can be **generated** (or **verified** – *to belong in a language*)

Context Free Grammars

- ***Context Free Grammars*** are special cases of such grammars where *sequence$_1$* is a singleton – i.e. rules are of the form:
    - symbol$_1$ ---> <sequence of symbols>$_2$
  - and the same replacement model is applicable.

  [*We will elaborate on this more!*]

- Today (and for the last several decades) **Context Free Grammars** are used to specify *artificial languages*
  - in particular, *programming languages*

- In this course, we will use them to specify **Logics as Languages**.

CONTEXT FREE GRAMMARS

Context Free Grammars - Usage

- The term “context free” is used to indicate that it is not “context specific”:
  - e.g. the rule that
    - *a variable (or a function or a type) must be defined before its use*

  in a program (in languages such as C, C++, Java) is **context specific**
- But most of the syntax of a programming language can and is captured using Context Free Grammars.

Context Free Grammars - Example

- Consider arithmetic expressions using only *numbers*, *addition*, and *multiplication* :
  - the syntax of these expressions can be defined by the following rules
    1. A ***number*** is an ***expression***
    2. Two expressions with a '+' in-fixed is an ***expression***
    3. Two expressions with a '*' in-fixed is an ***expression***

(and nothing else is an expression)

Context Free Grammars – Example – Arithmetic Expressions

- The definition of expressions from the previous slide can be captured in notation as (**CFG-Gr-Expr**):
  1. **E --> *num*** // a number is an expression
  2. **E --> E + E** // addition of two expressions is an expression
  3. **E --> E * E** // and so is multiplication of two expressions

  - "nothing else is an expression" is implied and not stated;
  - and ***num*** is used to denote numbers that are defined externally
    - i.e. we treat numbers as atomic entities.

Context Free Grammars – Example – Arithmetic Expressions

- **CFG-Gr-Expr**:
  1. **E --> *num***
  2. **E --> E + E**
  3. **E --> E * E**
- Often these rules are also stated in the following abbreviated form (*when the left symbol is common*)

  E --> num | E + E | E * E

  // an expression is a number or the

  // addition/multiplication of two expressions

[Comment on notation: **|** is used to denote alternatives.]

Context Free Grammars and Sentences

- Sentences in a language defined by a context free grammar
  - can be ***generated* from the grammar** or
  - can be ***verified* to be valid** per the grammar.

Generating Sentences

- Example :

Generate the expression 5 * 3 + 4 + 7 from the rules of **CFG-Gr-Expr.**

Generation Step	Rule
E	Start Symbol
E * E	Rule 3
5 * E	Rule 1
5 * E + E	Rule 2
5 * 3 + E	Rule 1
5 * 3 + E + E	Rule 2
5 * 3 + 4 + E	Rule 1
5 * 3 + 4 + 7	Rule 1

1. **E --> *num***
2. **E --> E + E**
3. **E --> E * E**

Verifying Sentences

- Example :

Verify that the expression 5 * 3 + 4 + 7 is valid per the rules of **CFG-Gr-Expr**.

Verification Step	Rule
5 * 3 + 4 + 7	Sentence
E * 3 + 4 + 7	Rule 1
E * E + 4 + 7	Rule 1
E + 4 + 7	Rule 3
E + E + 7	Rule 1
E + 7	Rule 2
E + E	Rule 1
E	Rule 2

1. **E --> *num***
2. **E --> E + E**
3. **E --> E * E**

The process of verification is referred to as ***parsing*** of the sentence.

PARSING AND PARSE TREES - EXAMPLE

CFGs – Parsing and Tree Representation of Sentences

Parsing order can be used to capture the sentence as a ***tree***:

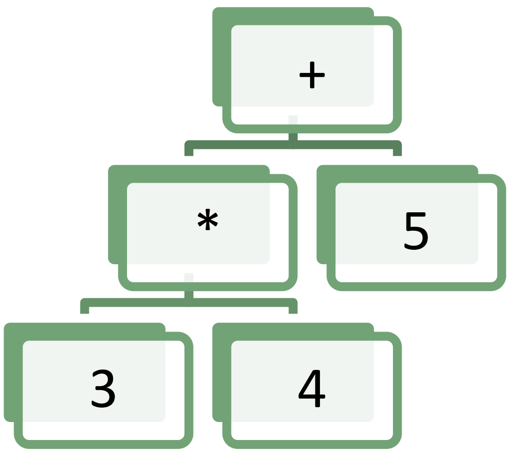
For instance, consider the expression: 3 * 4 + 5

1. **E --> *num***
2. **E --> E + E**
3. **E --> E * E**

Parsing:

Verification Step	Rule
3 * 4 + 5	
E * 4 + 5	Rule 1
E * E + 5	Rule 1
E + 5	Rule 3
E + E	Rule 1
E	Rule 2

This expression can then be captured as this tree:

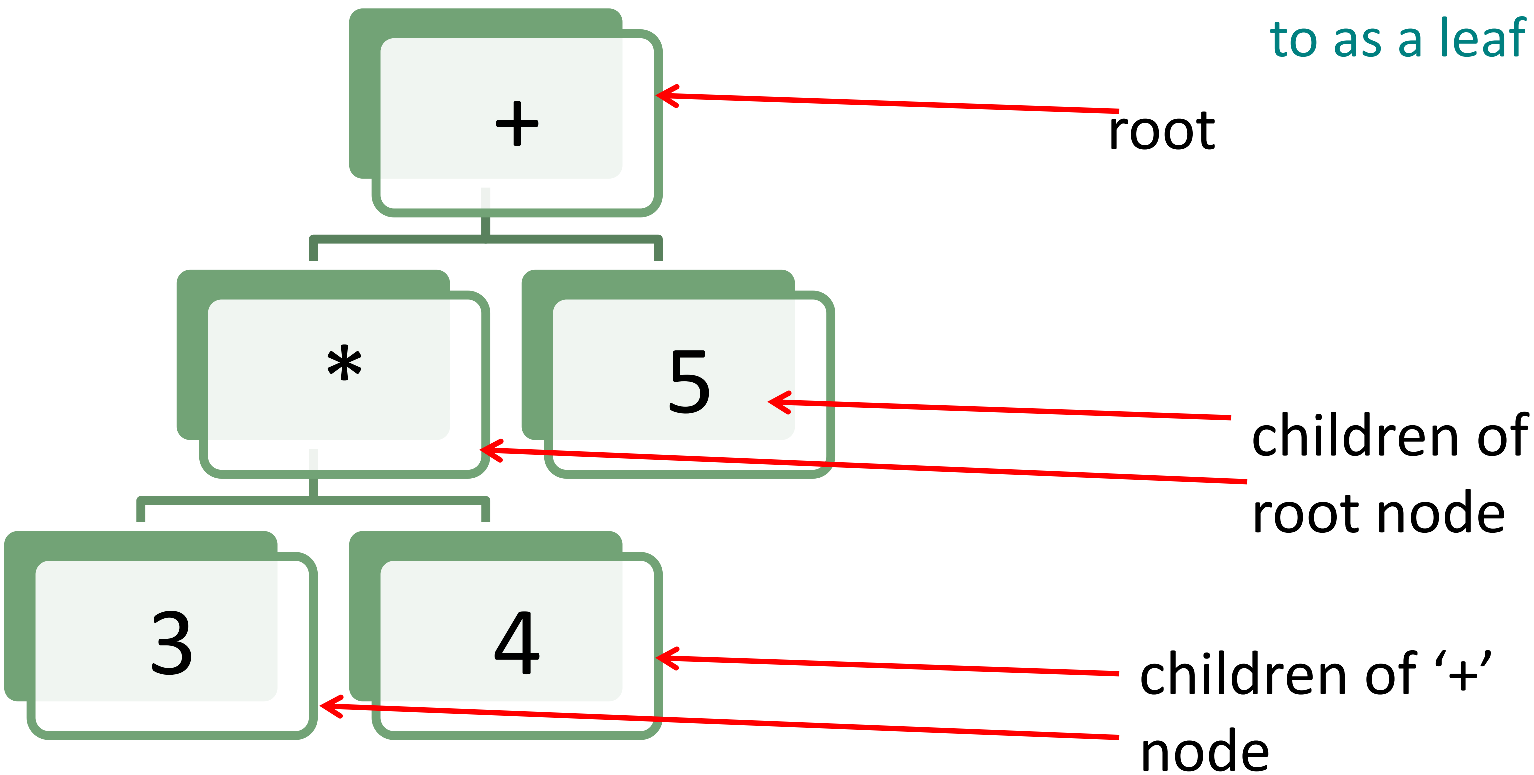


This is referred to as a ***parse tree***

Tree - Definition

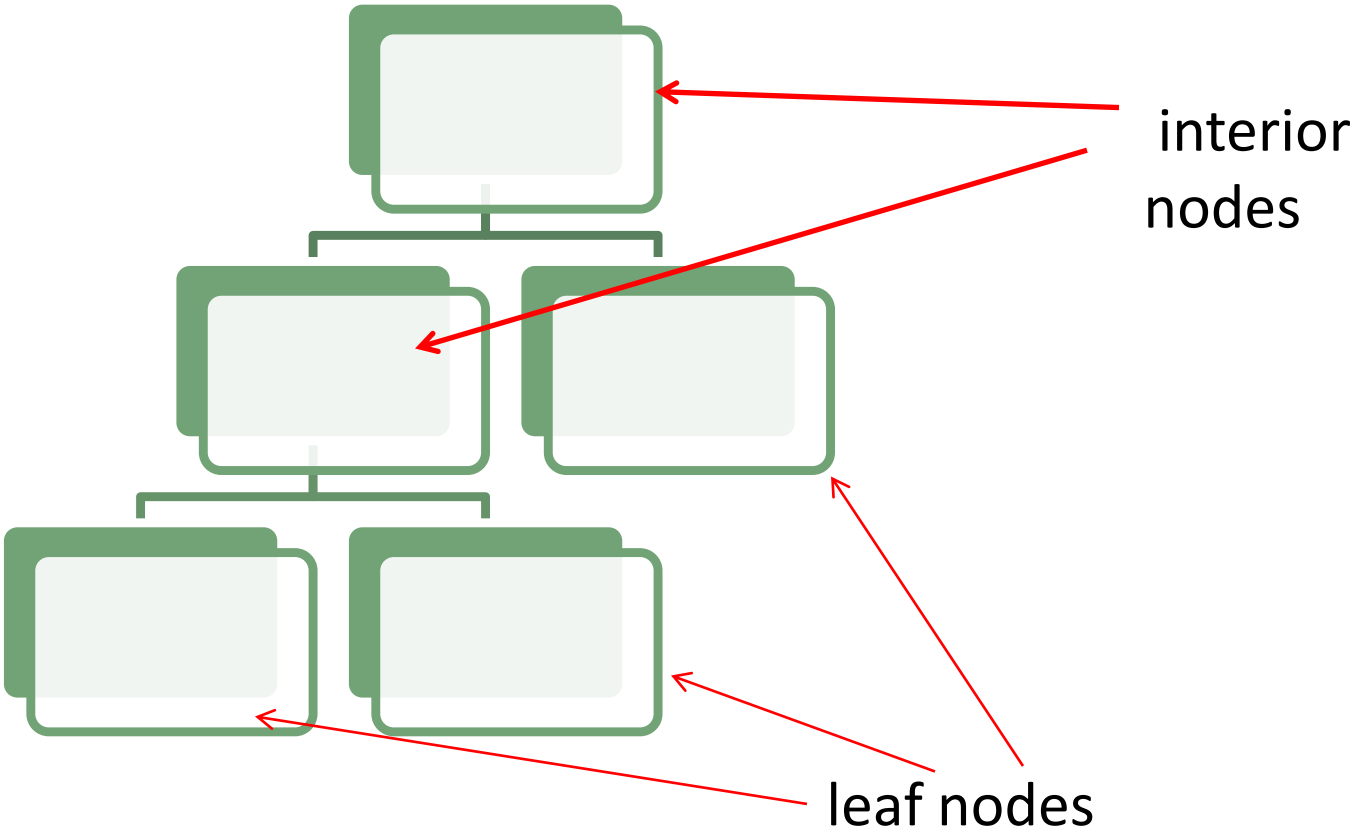
A (non-empty) ***tree*** is usually made of a ***root node*** and zero or more ***sub-trees*** (referred to as ***children***).

A node without children is referred to as a leaf node



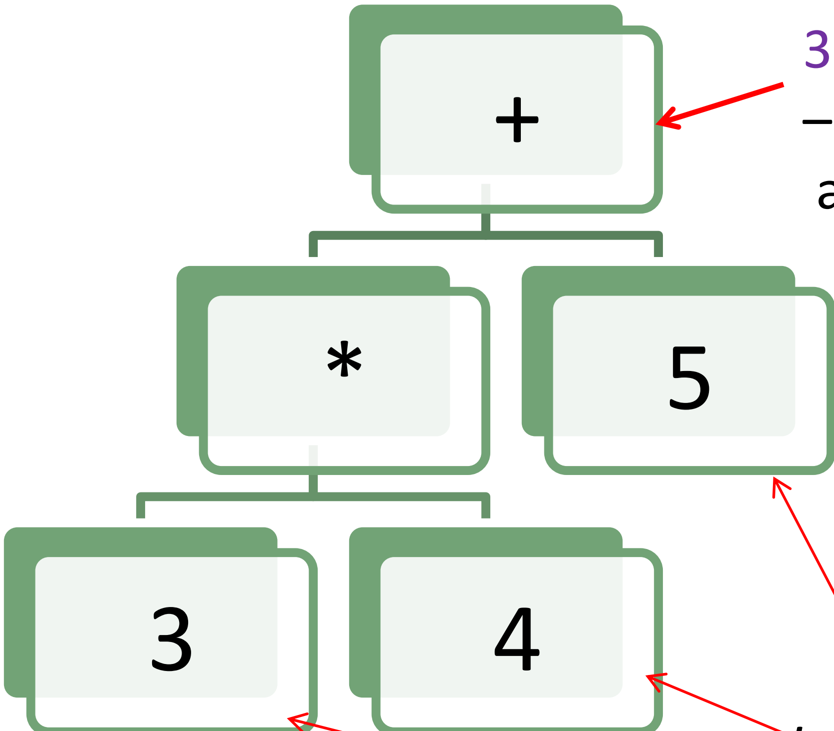
Tree – Root, Leaves, and Interior Nodes

A node without children is referred to as a ***leaf node***. Nodes that are not leaves are ***interior nodes***.



Parse Tree - Definition

A ***parse tree*** represents the structure of a sentence according to a given grammar as derived by a specific parsing.



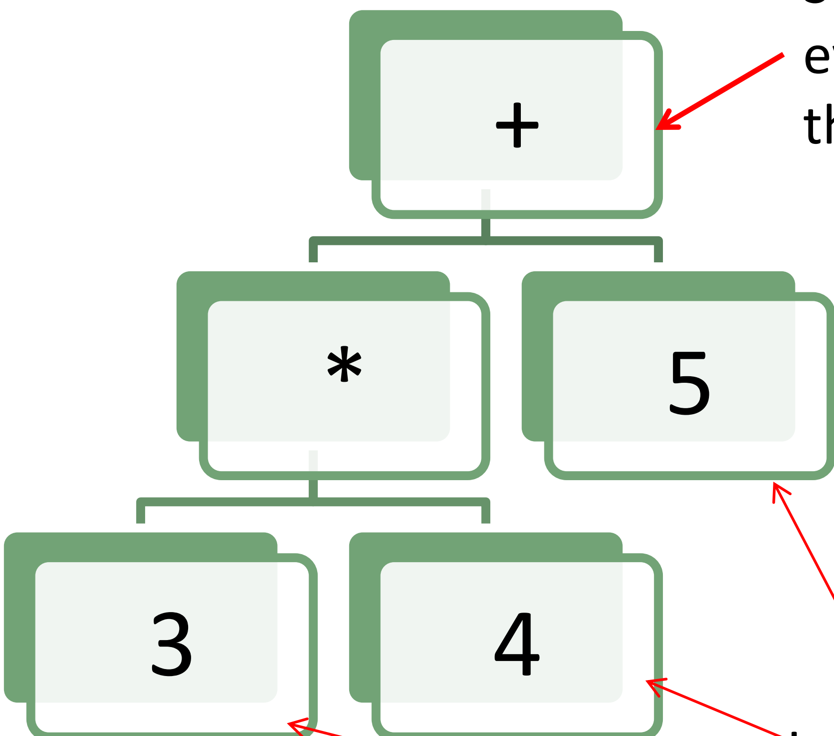
This denotes that the expression 3 * 4 + 5 is being treated – i.e. *it has been parsed* – as the **addition of 3*4 and 5**

This implies that **+** is the top-most operation i.e. *it will be performed (or evaluated) last*

leaf nodes hold atomic entities i.e. entities without a structure

Parse Tree – Use

A *parse tree captures the order of evaluation* (in case of computable sentences).



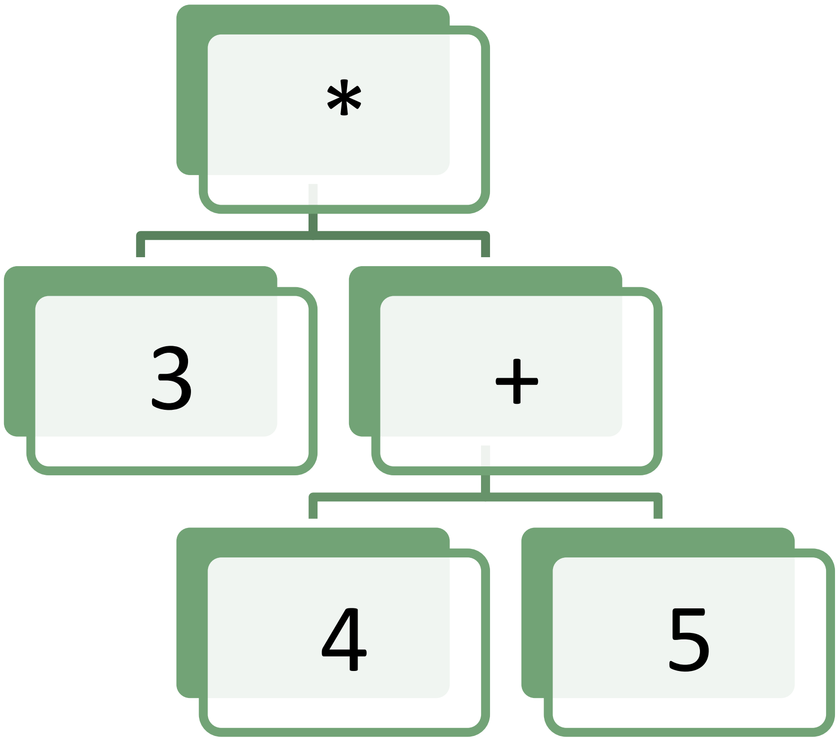
Sub-expressions **3*4** and **5** will be evaluated before **+** can be applied on the results (of evaluating them.)

Note that 3*4 and 5 can be evaluated in any order between them.

leaf nodes hold values (i.e. *no further evaluation is needed*).

Multiple Parse Trees

This is a different parse tree for the same expression: 3 * 4 + 5



This parse tree captures this parsing:

Verification Step	Rule
3 * 4 + 5	
E * 4 + 5	Rule 1
E * E + 5	Rule 1
E * E + E	Rule 1
E * E	Rule 2
E	Rule 3

It can be observed that in this case **+** *must be evaluated before **