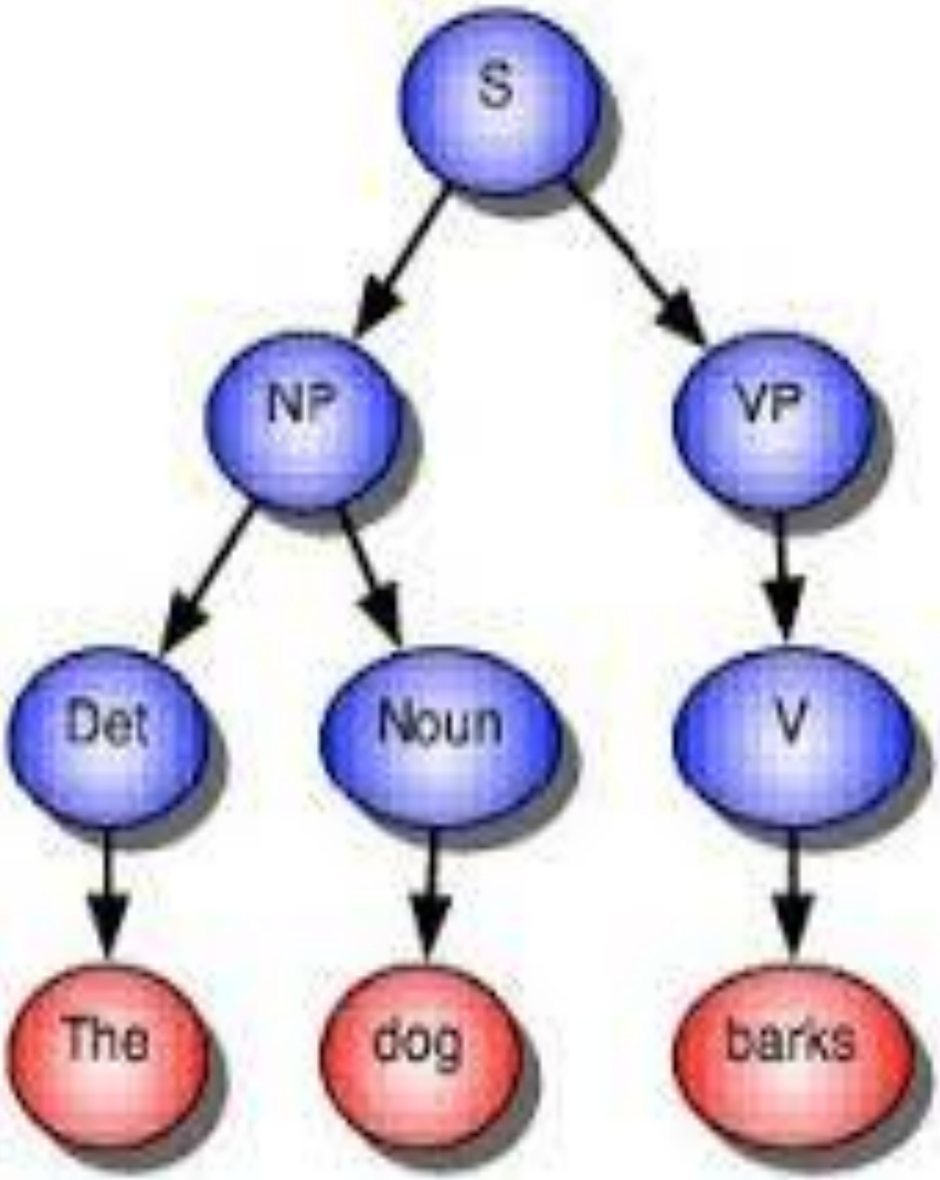
Context Free Grammar CFG



CFG and Parsing

Instructor : Dr. Hanaa Bayomi Ali
Mail : h.mobarz @ fci-cu.edu.eg

The path so far

- Originally, we treated language as a **sequence of words**
  n-gram language models
- Then, we introduced the notion of **syntactic properties of words**
  part-of-speech tags
- Now, we look at **syntactic relations** between words
  syntax trees

Introduction

- Finite Automata **accept** all regular languages and only regular languages

- Languages that require an ability to count are not regular.

- Many simple languages are non regular:

  -$\{a^n b^n : n = 0, 1, 2, \ldots\}$
  - the language of well-matched sequences of brackets. ((((())))),(()())

and there is no finite automata that accepts them.

- So we'd like some more powerful means of defining languages.

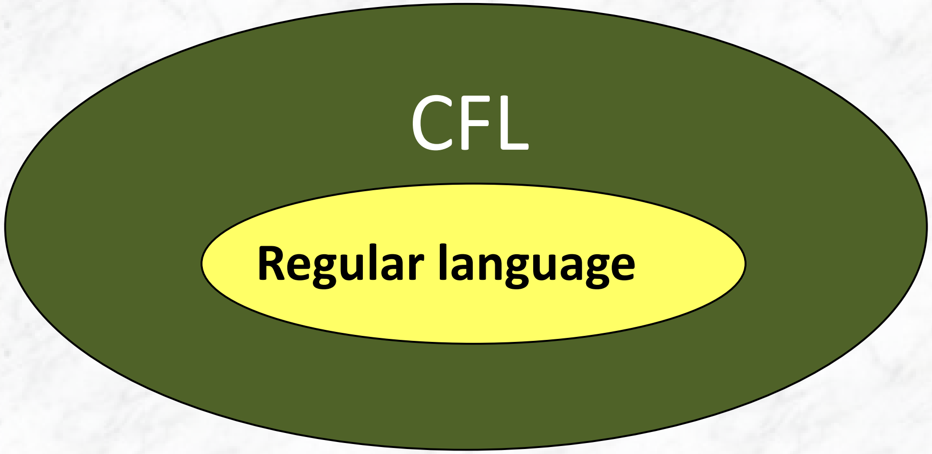
- **Generative grammars**. A language is defined by giving a set of rules capable of `generating' all the sentences of the language.

- The particular kind of generative grammars we'll consider are called **context-free grammars.**

Context-Free Grammars

- Languages that are **generated** by context-free grammars are **context-free languages**

- Context-free grammars are more expressive than finite automata: if a language L is **accepted** by a finite automata then L can be **generated** by a context-free grammar
    - Beware: The converse is NOT true



Context-Free Grammars

- A *context-free grammar* is a notation for describing languages.

- It is more powerful than finite automata or RE’s, but still cannot define all possible languages.

- Useful for nested structures, e.g., parentheses in programming languages.

Example: CFG for { 0^n1^n | n $\geq$ 1}

- Productions:

  S -> 01

  S -> 0S1

- Basis : 01 is in the language.
- Induction: if w is in the language, then so is 0w1.

CFG Formalism

- *Terminals* = symbols of the alphabet of the language being defined. (RHS)

S -> 01

S -> 0S1

CFG Formalism

- *Terminals* = symbols of the alphabet of the language being defined.(RHS)
- *Variables* = *nonterminals* = a finite set of other symbols, each of which represents a language.
- (mainly in LHS but may be appear in RHS)

S -> 01

S -> 0S1

CFG Formalism

- *Terminals* = symbols of the alphabet of the language being defined.(RHS)
- *Variables* = *nonterminals* = a finite set of other symbols, each of which represents a language.
- (mainly in LHS but may be appear in RHS)
- *Start symbol* = the variable whose language is the one being defined.

S -> 01

S -> 0S1

CFG Formalism

- *Terminals* = symbols of the alphabet of the language being defined.(RHS)
- *Variables* = *nonterminals* = a finite set of other symbols, each of which represents a language.
- (mainly in LHS but may be appear in RHS)

S -> 01

S -> 0S1

- *Start symbol* = the variable whose language is the one being defined.
- *A production* has the form

variable -> string of variables and terminals.

Example: Formal CFG

- Here is a formal CFG for $\{ 0^n 1^n \mid n \geq 1\}$.
- Terminals = {0, 1}.
- Variables = {S}.
- Start symbol = S.
- Productions =
  - S -> 01
  - S -> 0S1

Derivations – Intuition

We *derive* strings in the language of a CFG by starting with the **start symbol**, and repeatedly replacing some variable A by the right side of one of its productions.

That is, the "productions for A" are those that have A on the left side of the ->.

- **Definition.** v is **one-step derivable** from u, written $u \Rightarrow v$

- **Definition.** v is **derivable** from u, written $u \Rightarrow^* v$, if:

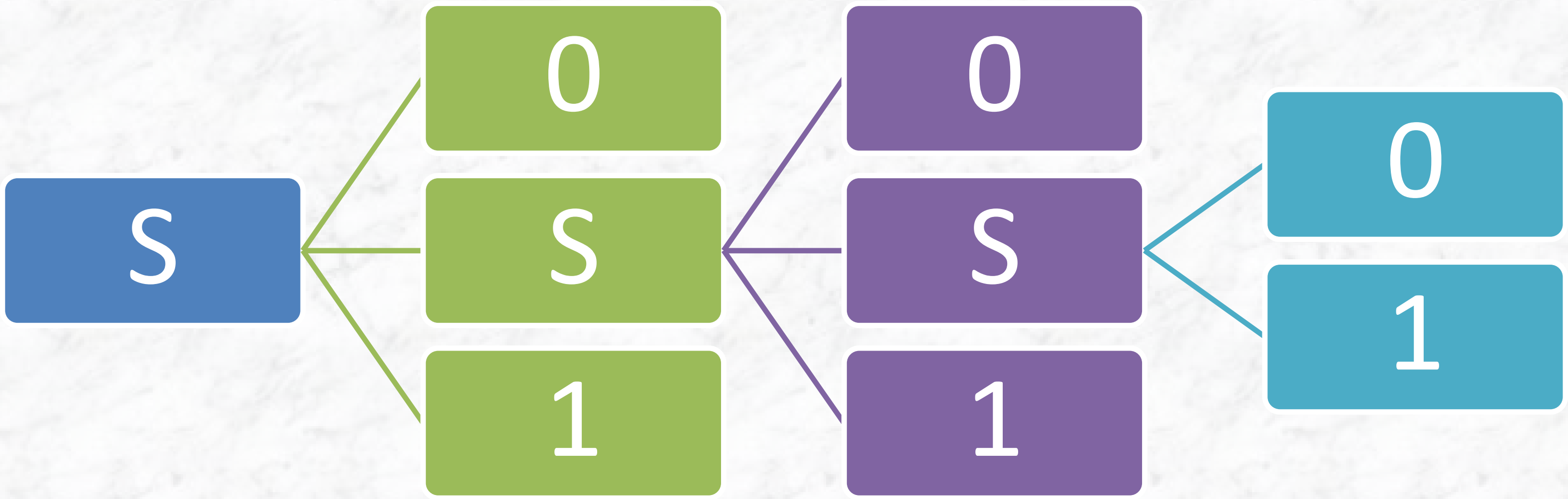
There is a chain of one-derivations of the form:

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow v$$

Example

Example: S -> 01; S -> 0S1.

S => 0S1 => 00S11 => 000111.



Chain of one-derivations

Context-free grammars: examples

Q1) all strings of balanced parentheses

((((((((()))))))))

$$P \rightarrow ()$$

$$P \rightarrow (P)$$

Q2) $\{a^m\ b^n\ c^{m+n} \mid m,n \geq 0\}$

Rewrite as $\{\mathbf{a}^m\ \mathbf{b}^n\ \mathbf{c}^n\ \mathbf{c}^m \mid m,n \geq 0\}$:

$S \rightarrow S' \mid \mathbf{a}\ S\ \mathbf{c}$

$S' \rightarrow \varepsilon \mid \mathbf{b}\ S'\ \mathbf{c}$

Context-free grammars: example3

Q3) Write CFG generates simple arithmetic expressions such as

6 + 7   5*(x + 3)   x *((z* 2) + y)   8   z

Exp → Var | Num | (Exp)
Exp → Exp + Exp
Exp → Exp * Exp
Exp → Exp / Exp
Var → x |y | z
Num → 0 9

Context-free grammars: example3

Exp → Var | Num | (Exp)
Exp → Exp + Exp
Exp → Exp * Exp
Exp → Exp / Exp
Var → x |y | z
Num → 0 9

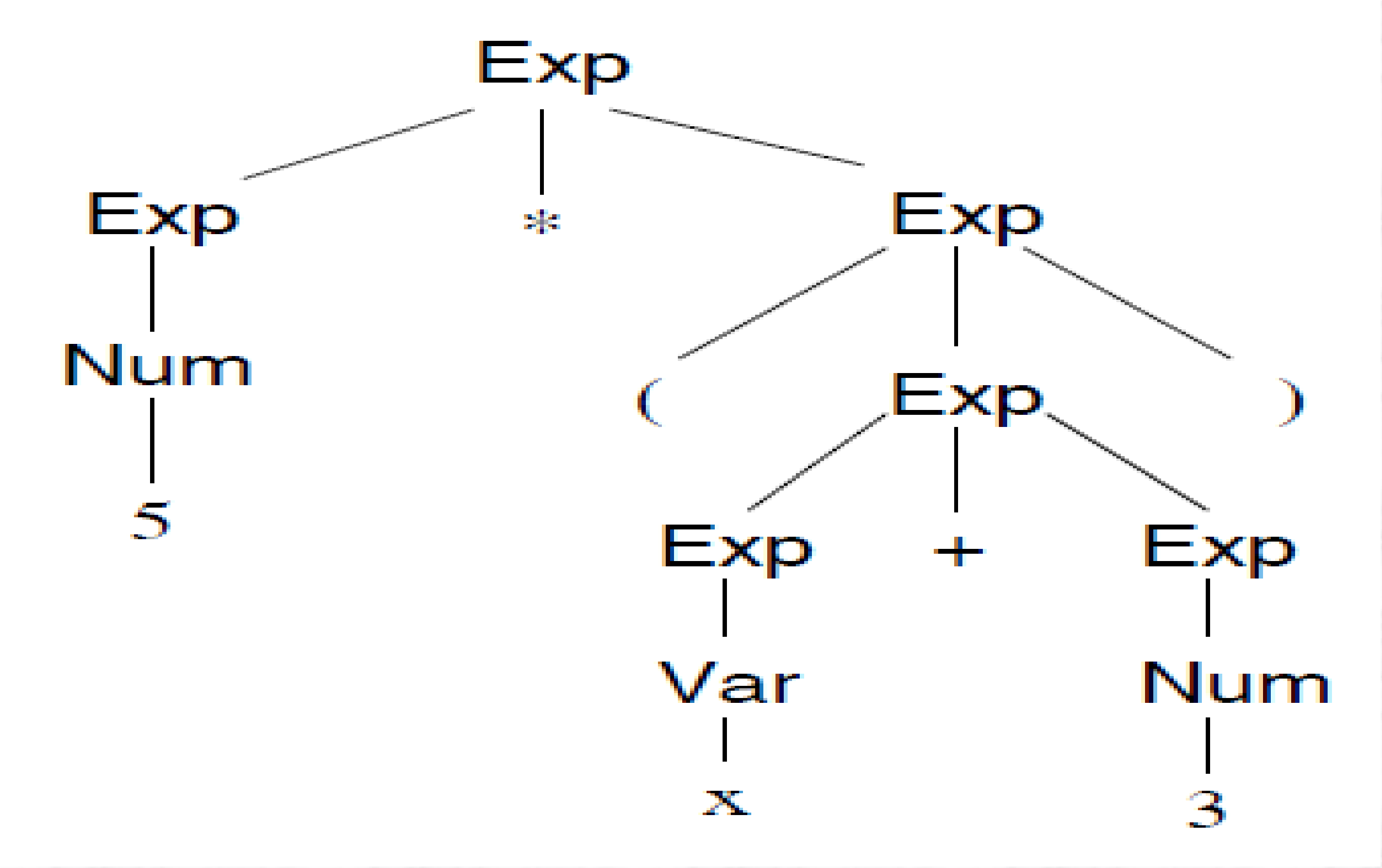
The symbols **+,*, /, (,), x, y, z, 0, ….., 9** are called terminals: these form the ultimate constituents of the phrases we generate.

The symbols **Exp, Var, Num** are called non-terminals: they name various kinds of `sub-phrases'.

We designate Exp the start symbol.

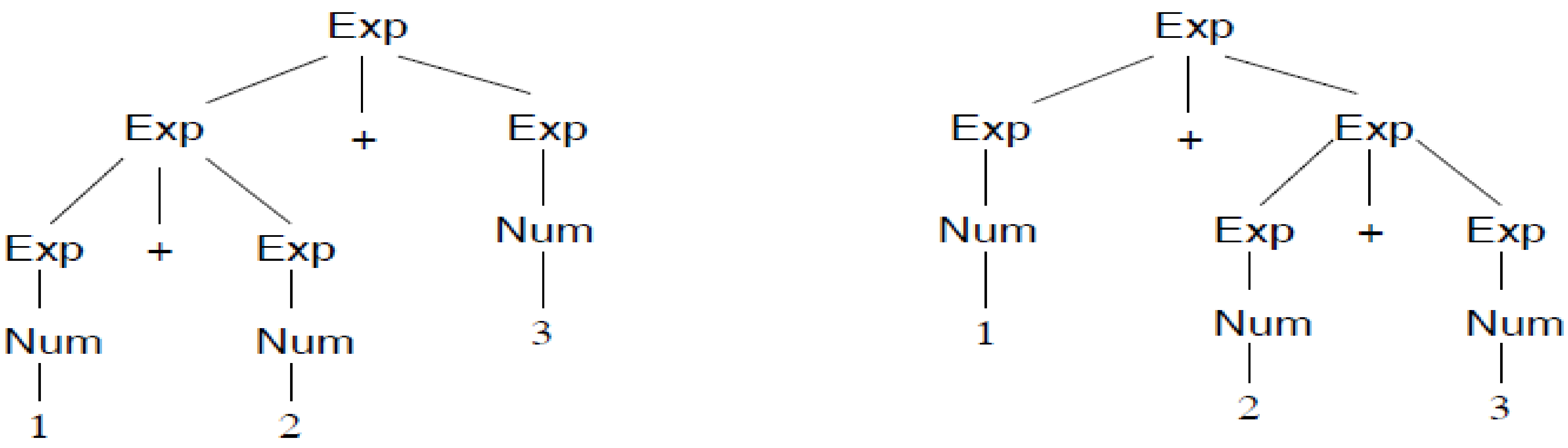
Syntax tree

We grow syntax trees by repeatedly **expanding non-terminal** symbols using these rules. E.g.: 5*(x +3)



The language defined by a grammar

- By choosing different rules to apply, we can generate infinitely many strings from this grammar.
- The **language** generated by the grammar is, by definition, the set of all strings of terminals that can be derived from the start symbol via such a syntax tree.
- Note that strings such as 1+2+3 may be generated by more than one tree (structural ambiguity):



Challenge question

How many possible syntax trees are there for the string below

1+2+3+4

Exp → Var | Num | (Exp)
Exp → Exp + Exp
Exp → Exp * Exp
Exp → Exp / Exp
Var → x |y | z
Num → 0 9

Derivations

As a more `machine-oriented' alternative to syntax trees, we can think in terms of derivations involving (mixed) strings of terminals and non-terminals. E.g.

Exp ->Exp * Exp
->Num * Exp
-> Num* (Exp)
-> Num * (Exp + Exp)
-> 5 * (Exp + Exp)
-> 5 * (Exp + Num)
-> 5 * (Var + Exp)
->5 * (x + Exp)
->5 * (x + 3)

Exp → Var | Num | (Exp)
Exp → Exp + Exp
Exp → Exp * Exp
Exp → Exp / Exp
Var → x |y | z
Num → 0 9

At each stage, we choose one non-terminal and expand it using a suitable rule. When there are only terminals left, we can stop!

Multiple derivations

Clearly, any derivation can be turned into a syntax tree.
However, even when there's only one syntax tree, there might be many derivations for it:

$$\begin{aligned} \text{Exp} &\Rightarrow \text{Exp} + \text{Exp} \\ &\Rightarrow \text{Num} + \text{Exp} \\ &\Rightarrow 1 + \text{Exp} \\ &\Rightarrow 1 + \text{Num} \\ &\Rightarrow 1 + 2 \end{aligned}$$

(. . . a leftmost derivation)

$$\begin{aligned} \text{Exp} &\Rightarrow \text{Exp} + \text{Exp} \\ &\Rightarrow \text{Exp} + \text{Num} \\ &\Rightarrow \text{Exp} + 2 \\ &\Rightarrow \text{Num} + 2 \\ &\Rightarrow 1 + 2 \end{aligned}$$

(. . . a rightmost derivation)

In the end, it's the syntax tree that matters | we don't normally care about the difference between various derivations for it.

However, derivations - especially leftmost and rightmost ones-will play a significant role when we consider parsing algorithms.