

# CONTEXT-FREE GRAMMARS (CFG)

## GROUP MEMBERS

1. Dennis Macharia
2. Dennis Wambua
3. Bettyrose Waithera
4. Lorraine Opiyo

# TABLE OF CONTENTS

01

CFGs

Macharia

02

Parse Trees

Wambua

03

Applications of  
CFGs

Waithera

04

Ambiguity in  
Grammars and  
Languages

Lorraine



# 01 Introduction to CFG's

t

# INTRODUCTION

A **grammar** is basically a **natural recursive notation** of a **language**.

**Context Free Grammars**(CFG's) are Language defining mechanisms for **Context Free Languages**.




# Definition of CFGs

- Context Free Grammars (CFGs) are basically described as follows
- $G = (V, T, P, S)$
- Where:
  - $V$  = Set of **Variables**
  - $T$  = **Terminals**
  - $P$  = Set of **Productions**
  - $S$  = **Start Symbol**



## Example 1

- Consider a situation where we only allow the letters a and b and the digits 0 and 1.
  - Every string must begin with an a or b which may be followed by any string in  $\{a, b, 0, 1\}^*$
  - We need two **variables** in this grammar.
    - The **start symbol** which is E, represents the language of the expressions.
    - The other **variable** I which represents identifiers.
- 

This is the language of the regular expression:  $(a+b)(a+b+0+1)^*$



We are going to use a set of productions that say essentially the same thing as this regular expression:

- x 1.  $E \rightarrow I$
- x x 2.  $E \rightarrow E + E$
- x 3.  $E \rightarrow E * E$
- 4.  $E \rightarrow (E)$
  
- 5.  $I \rightarrow a$
- 6.  $I \rightarrow b$
- 7.  $I \rightarrow Ia$
- 8.  $I \rightarrow Ib$
- 9.  $I \rightarrow I0$
- 10.  $I \rightarrow I1$

The grammar for the expressions is stated formally as:

$$G = (\{E, I\}, T, P, E)$$

Where **T** is the set of **terminal symbols** {+, \*, (,), a, b, 0, 1} and **P** is the set of **Productions**.

- Rule(1) is the *basis*
- Rule (2) through (4) describe the *inductive case* for expressions.
- Rules (5) through (10) describe identifiers I.
- The *basis* is the rules(5) and (6) which say that a and b are identifiers
- The rest of the four rules are the *inductive* cases.



# Derivations Using Grammar

	String Inferred	For Language of	Production Used	String(s) Used
i.	a	I	5	_____
ii	b	I	6	_____
iii	b0	I	9	ii
iv	b00	I	9	iii
v	a	E	1	i
vi	b00	E	1	iv
vii	a + b00	E	2	v, vi
viii	(a + b00)	E	4	vii
ix	a * (a + b00)	E	3	v, vii

x  
x x  
x

- **Basis:** for any string  $x$  of terminals and variables, we say that  $x \stackrel{*}{\Rightarrow} x$ . That is, any string derives itself.
- **Induction:** if  $x \stackrel{*}{\Rightarrow} y$  and  $y \Rightarrow z$ , then we say  $x \stackrel{*}{\Rightarrow} z$

# Leftmost and Rightmost Derivations

- Consider the string  **$a * (a + b00)$**

Leftmost:

$E \Rightarrow E * E \Rightarrow I * E \Rightarrow a * E \Rightarrow a * (E) \Rightarrow a * (E + E) \Rightarrow a * (I + E) \Rightarrow a * (a + E) \Rightarrow a * (a + I) \Rightarrow a * (a + I0) \Rightarrow a * (a + I00) \Rightarrow \mathbf{a * (a + b00)}$

Rightmost:

$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (E + I) \Rightarrow E * (E + I0) \Rightarrow E * (E + I00) \Rightarrow E * (E + b00) \Rightarrow E * (I + b00) \Rightarrow E * (a + b00) \Rightarrow I * (a + b00) \Rightarrow \mathbf{a * (a + b00)}$

# The Language of Grammar

## Palindrome example:

**Basis:** We use lengths 0 and 1 as the basis. If  $|w| = 0$  or  $|w| = 1$ , then  $w$  is  $\epsilon$ , 0 or 1. Since there are productions  $P \rightarrow \epsilon$ ,  $P \rightarrow 0$ ,  $P \rightarrow 1$ , we conclude that  $P \xrightarrow{*} w$  in any of these cases.

**Induction:** Suppose that  $|w| \geq 2$ . Since  $w = w^R$ , we must begin and end with the same symbol. ie,  $w = 0x0$  or  $1x1$ . Moreover,  $x$  must be a palindrome; ie,  $x = x^R$

# Sensial Forms

Consider the following:

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow \mathbf{E} * (\mathbf{I} + \mathbf{E})$$

This derivation is neither **left-sential** or **right-sential** since at the last step, the middle **E** is replaced.

As a leftmost example, consider **a \* E**:

$$E \Rightarrow E * E \Rightarrow \mathbf{I} * E \Rightarrow \mathbf{a} * E$$

Additionally, the derivation:

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow \mathbf{E} * (\mathbf{E} + \mathbf{E})$$

Shows that **E \* (E + E)** is a **right-sential** form.



02

# Parse Trees

t

# PARSE TREES / DERIVATION TREE



A parse tree is an ordered rooted tree that graphically represents the semantic information of strings derived from a context free grammar





## METHODS OF TREE DERIVATION

- Leftmost Tree Derivation
- Rightmost Tree Derivation



## Left Most Tree

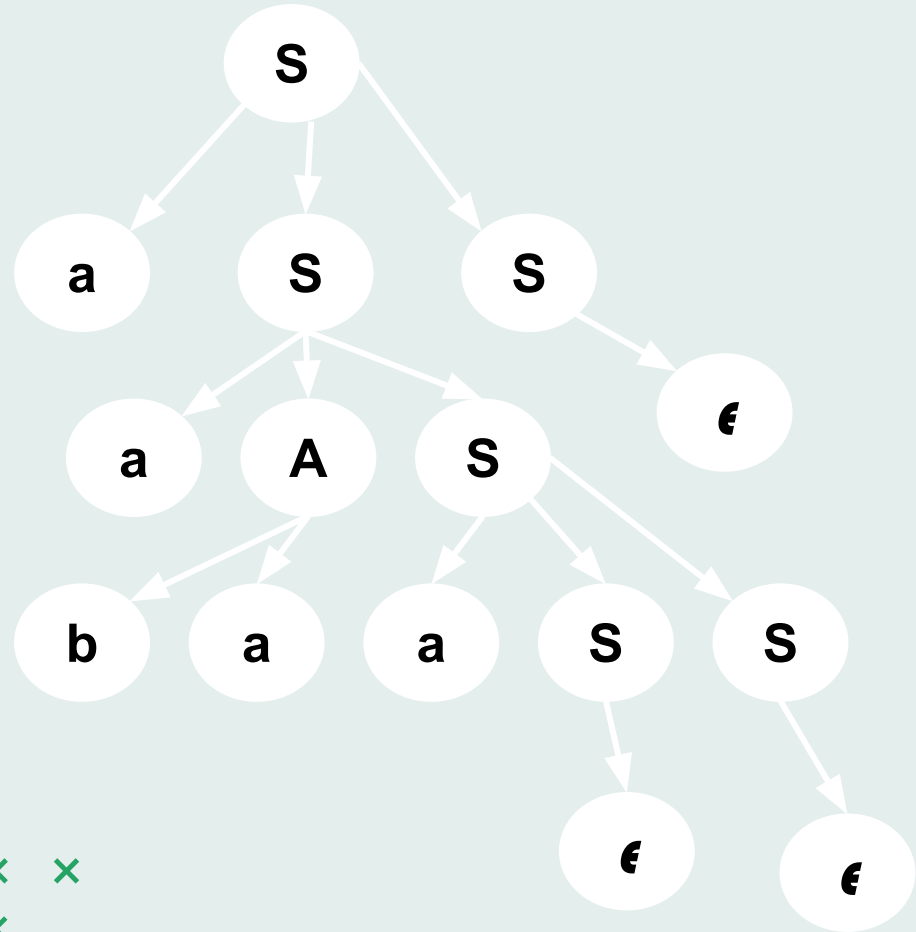
A left Derivation is obtained by applying production to the left most variable in each step.

For Example generating the string aabaa from the grammar

$S \rightarrow aAS \mid aSS \mid E, A \rightarrow SbA \mid ba$



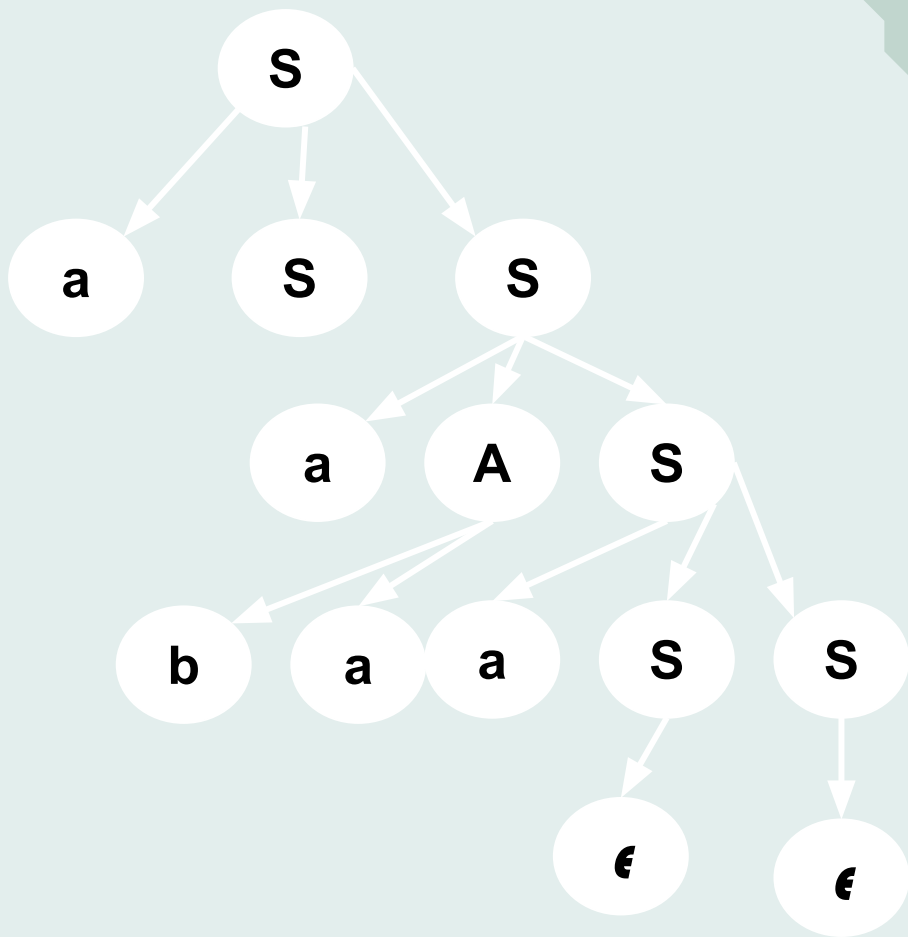
1. Since we want to obtain the string aabb S will be the root vertex.
2. On the first level from the root level we choose the leftmost derivation since we are doing the leftmost derivation we'll use the first S from a to get aAS
3. At the second level we will go with A to get ba and S to get aSS
4. Afterwards we close all S's with  $\epsilon$
5. Finally we obtain the string aabb



# RIGHT MOST DERIVATION

A right derivation tree is obtained by applying the production to the right most variable in each step





x  
x x  
x

+



03

# Applications of CFGs

t

x x  
x x



# 1. Parses

Many aspects of a programming language have a structure that may be described by regular expressions. However, there are some important aspects of typical programming that cannot be represented by regular expressions alone.

For example: Consider the string `iee`. The first `e` is matched with `I` to its left. They are removed, leaving the string `e`. Since there are more `e`'s the next is considered. However, the `I` is left to its left so the test fails, `iee` is not in the language. This is valid since you cannot have more `else`'s than `if`'s in a program. We can conclude that in an `if-else` structure, the `else`'s must match to the `if`'s and the first `if` is unmatched.

×

×

×

## 2. The YACC Parse-Generator

The generation of a parser (function that creates parse trees from source programs) has been institutionalized in the YACC command that appears in all UNIX systems.

The input to YACC is a CFG, in a notation that differs only in details.

Associated with each production is an action, code to construct the node, which is performed whenever a node of the parse tree is created.

×

×

×

---

# 3. Markup Languages

Tags used in markup languages tell us something about the semantics of various strings within the document. HTML, a markup language, has two major functions which include creating links between documents and describing the format of the document. For example,

Things I hate

1. Moldy bread

The text is viewed as

`<p> The things I hate: </OL>`

`<L1>Moldy bread </OL>`



# Markup Languages

In the above example, we see unmatched tags `<P>` and `<L1>` which introduce paragraphs and list items respectively.

HTML encourages that these tags are matched by `</P>` and `</L1>` at the end, but it does not require these matching.

Therefore, the tags are left off to provide some complexity to the language grammar. This suggests that its structure and how a CFG could be used both to describe the legal HTML documents and guide the processing.

x

x x

x

# 4. XML and Document Type Definitions

All programming languages can be described by their own CFGs. In XML (eXtensible Markup Language), CFG's play a vital role especially when used to describe the “semantics” of the text.

- ✗ In order to make the tags in the XML language clear, standards in the form of DTD (Document-Type Definition) are developed.
- ✗ ✗ DTDs are CFGs with its own notation for describing the variables and productions. The language for describing a DTD is a CFG notation.

# XML and Document Type Definitions

For example, the form of a DTD is

```
<!DOCTYPE name-of-DTD [
```

List of element definitions

```
]>
```

An element definition has the form: `<!ELEMENT element-name (description of the element)>`

The allowed operators are :

1. `|` which stands for union
2. `,` which stands for concatenation
3. Three variants of the closure, `*` which stands for “zero or more occurrences of”, `+` which stands for “one or more occurrences of” and `?` which stands for “zero or one occurrence”



04

# Ambiguity in grammar and languages

t



When a grammar fails to provide structures, it is sometimes possible to redesign the grammar fails to provide unique structure,it is sometimes possible to redesign the grammar to make structure unique for each string in the language

Unfortunately, sometimes we cannot do so . That is, there are some CFL's that are "inherently ambiguous" every grammar for the language puts more than one structure on some strings in the language.

## Ambiguous Grammar

Example:

For instance, consider the sentential form  $E + E * E$ . It has two derivations from  $E$ :

$$1. E \Rightarrow E + E \Rightarrow E + E * E$$

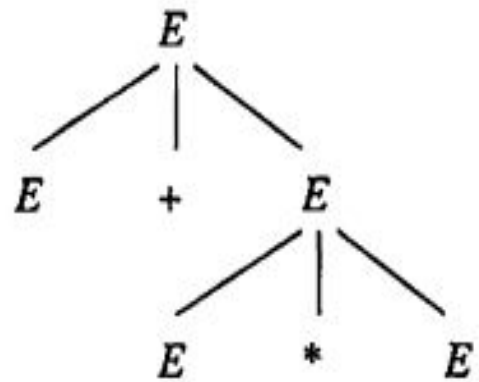
$$2. E \Rightarrow E * E \Rightarrow E + E * E$$

×

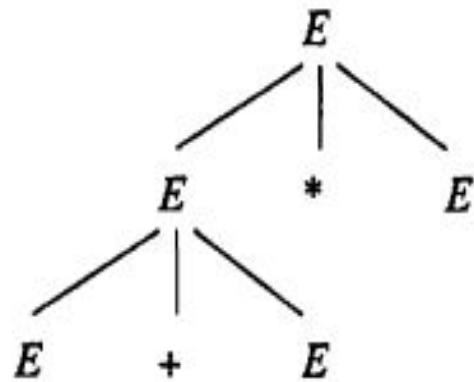
×

×

Notice that in derivation (1), the second  $E$  is replaced by  $E * E$ , while in derivation (2), the first  $E$  is replaced by  $E + E$ . The figures below shows the two parse trees, which we should note are distinct trees.



(a)



(b)

×  
× ×  
×

The difference between these two derivations is significant. As far as the structure of the expressions is concerned, derivation(1) says that the second and third expressions are multiplied, and the result is added to the first expression and multiplies the result by the third. In more concrete terms, the first derivation suggests that  $1 + 2 * 3$  should be grouped  $1 + ( 2 * 3 ) = 7$ , while the second derivation suggests the same expression should be grouped  $( 1 + 2 ) * 3 = 9$ .

Obviously, the first of these, and not the second, matches our notion of correct grouping of arithmetic expressions. Since the grammar gives two different structures to any string of the terminals that is derived by replacing the three  $E + E * E$  by identifiers, we see that this grammar is not a good one for providing unique structure.

To use this expression grammar in a compiler, we would have to modify it to provide only the correct groupings since it also gives strings incorrect groupings in arithmetic expressions.

There are two causes of ambiguity:

- 1) The precedence of operators is not respected.
- 2) A sequence of identical operators can group either from the left or from the right.

The solution to the problem of enforcing precedence is to introduce several different variables, each of which represents those expressions that share a level of “binding strength.” Specifically:

- 1) A factor is an expression that cannot be broken apart by any adjacent operator, either  $a^*$  or  $a+$ .
- 2) A term is an expression that cannot be broken by the  $+$  operator.
- 3) An expression will henceforth refer to any possible expression, including those that can be broken by either an adjacent  $*$  or an adjacent  $+$ .



### **Inherent Ambiguity**

A context free language  $L$  is said to be inherently ambiguous if all its grammars are ambiguous. If even one grammar for  $L$  is unambiguous, then  $L$  is an unambiguous language.



# THANK YOU!

Do you have any questions?

CREDITS: This presentation template was created  
by **Slidesgo**, including icons by **Flaticon**, and  
infographics & images by **Freepik**