Theory of Automata
Context Free Grammars

Week 7

- Languages (concepts/Algorithms/Pseudocode)
  - Regular
  - Non-regular
    - Context Free Languages

Regular vs Context Free

Context Free CFL is a bigger set, and regular RL is a subset.

CFL: bigger set, have more languages, more power,

e.g. palindrome, $a^n b^n$

- CFL cover that RL do not cover and also cover what RL cover.
- Since RL is a subset of CFL, so any language that is part of RL is always part of CFL
- E.g BiggerSet = {1,2,3,4}
- Subset = {3}

- RL:
  - FA/RE/TG/
- CFL:
  - CFG/PDA
  - Context Free Grammar / Pushdown Automata

Contents

Context Free Grammars

- Three fundamental areas covered in the book are
  1. **Theory of Automata**
  2. **Theory of Formal Languages**
  3. **Theory of Turing Machines**
- We have completed the first area.
- We begin exploring the second area in this chapter.

Syntax as a Method for Defining Languages

- In Chapter 3 we recursively defined the set of valid arithmetic expressions as follows:

  Rule 1: Any number is in the set AE.

  Rule 2: If x and y are in AE, then so are

  (x), -(x), (x + y), (x - y), (x * y), (x/y), (x ** y)

where ** is our notation for exponentiation

- Note that we use parentheses around every component factor to avoid ambiguity expressions such as 3 + 4 - 5 and 8/4/2.
- There is a different way for defining the set AE: **using a set of substitution rules similar to the grammatical rules**.

Defining AE by substitution rules

- Start → AE

  AE → (AE + AE)

  AE → (AE - AE)

  AE → (AE * AE)

  AE → (AE/AE)

  AE → (AE ** AE)

  AE → (AE)

  AE → -(AE)

  AE → d

Example

- We will show that ((3 + 4) * (6 + 7)) is in AE

Start $\rightarrow$ AE $\rightarrow$ (AE * AE)

$\rightarrow$((AE + AE) * (AE + AE))

$\rightarrow$((3 + 4) * (6 + 7))

Definition of Terms

- A word that cannot be replaced by anything is called **terminal**.
  - In the above example, the terminals are the phrase AnyNumber, and the symbols + - * / ** ( )
- A word that must be replaced by other things is called **non-terminal**.
  - The non-terminals are Start and AE.
- The sequence of applications of the rules that produces the finished string of terminals from the starting symbol is called a **derivation** or a **generation** of the word.

- The grammatical rules are referred to as **productions**.

Symbolism for Generative Grammars

Definition:

- A **context-free grammar (CFG)** is a collection of three things:

1. An alphabet ∑ of letters called **terminals** from which we are going to make strings that will be the words of a language.

2. A set of symbols called **non-terminals**, one of which is the symbol S, standing for “start here”.

3. A finite set of **productions** of the form:

One non-terminal → finite string of terminals and/or non-terminals

where the strings of terminals and non-terminals can consist of only terminals, or of only non-terminals, or of any mixture of terminals and non-terminals, or even the empty string. We require that at least one production that has the non-terminal *S* as its left side.

Definition:

- The **language generated by a CFG** is the set of all strings of terminals that can be produced from the start symbol S using the productions as substitutions.
- A language generated by a CFG is called a **context-free language (CFL)**.

Notes:

- The language generated by a CFG is also called the **language defined by the CFG**, or the **language derived from the CFG**, or the **language produced by the CFG**.

- We insist that **non-terminals be designated by capital letters**, whereas **terminals are designated by lowercase letters and special symbols**.

Example

- Let the only terminal be *a* and the productions be

  Prod1 S → aS

  Prod2 S → Λ
- If we apply Prod 1 six times and then apply Prod 2, we generate the following:

  S ➔ aS ➔ aaS ➔ aaaS ➔ aaaaS

  ➔ aaaaaS ➔ aaaaaaS ➔ aaaaaaΛ = aaaaaa

- If we apply Prod2 without Prod1, we find that Λ is in the language generated by this CFG.
- Hence, this CFL is exactly a*.
- Note: the symbol "→" means "can be replaced by", whereas the symbol "➔" means "can develop to".

- Let the terminals be a and b, the only non-terminal be S, and the productions be

  Prod1 S → aS

  Prod2 S → bS

  Prod3 S → Λ
- The word *ab* can be generated by the derivation

  S ➔ aS ➔ abS ➔ abΛ = ab
- The word baab can be generated by

  S ➔ bS ➔ baS ➔ baaS ➔ baabS ➔ baabΛ = baab
- Clearly, the language generated by the above CFG is

  (a + b)*.

Example

- Let the terminals be a and b, the the non-terminal be S and X, and the productions be

  Prod 1 $S \rightarrow XXXaXaXX$

  Prod 2 $X \rightarrow aX$

  Prod 3 $X \rightarrow bX$

  Prod 4 $X \rightarrow \Lambda$

- We already know from the previous example that the last three productions will generate any possible strings of a's and b's from the non-terminal X. Hence, the words generated from S have the form

  anything aa anything

- Hence, the language produced by this CFG is

$$(a + b)^*aa(a + b)^*$$

  which is the language of all words with a double a in them somewhere.
- For example, the word *baabb* can be generated by

  S ➔ XaaX ➔ bXaaX ➔ baaX ➔ baaX

  ➔ baabX ➔ baabbX ➔ baabbΛ = baabb

Example

- Consider the CFG:

  $S \rightarrow aSb$

  $S \rightarrow \Lambda$

- It is easy to verify that the language generated by this CFG is the **non-regular** language $\{a^n b^n\}$.
- For example, the word a^4b^4 is derived by

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$

$\Rightarrow aaaaSbbbb \Rightarrow aaaa\Lambda bbbb = aaaabbbb$

Derivation and some Symbols

- If v and w are strings of terminals and non-terminals

$v \Rightarrow^{n} w$ » denotes the derivation of w from v of length *n* steps

$v \Rightarrow^{+} w$

» derivation of w from v in one or more steps

$v \Rightarrow_{G}^{*} w$

» derivation of w from v in zero or more steps of application of rules of grammar G.

Sentential Form

- A string w ε(n U ∑)* is a sentential form of G if there is a derivation

$$v \Rightarrow^* w$$

- A string w is a sentence of G if there is a derivation in G

$$v \Rightarrow^* w$$

- The language of G, denoted by L(G) is the set

$$\left\{ w \in \sum^* \ \ | \, S^* \overset{*}{\Rightarrow} w \right\}$$

Example

- It is not difficult to show that the following CFG generates the **non-regular** language $\{a^n b a^n\}$:

  S → aSa

  S → b

- Can you show that the CFG below generates the language PALINDROME, another **non-regular** language?

  S → aSa

  S → bSb

  S → a

  S → b

  S → Λ

Disjunction Symbol |

- Let us introduce the symbol | to mean disjunction (or).
- We use this symbol to combine all the productions that have the same left side.
- For example, the CFG

  Prod 1 $S \rightarrow XaaX$

  Prod 2 $X \rightarrow aX$

  Prod 3 $X \rightarrow bX$

  Prod 4 $X \rightarrow \Lambda$

can be written more compactly as

Prod 1 $S \rightarrow XaaX$

Prod 2 $X \rightarrow aX|bX|\Lambda$

a* terminal – a, b, non-terminal for S, X, A

- S -> aS | Δ
- aaaa
- S -> aS
- S->aS
- S -> aS
- S -> aS
- S -> Δ
-

- S -> Sa | Δ
- aaaa
- S -> Sa
- S->Sa
- S->Sa
- S->Sa
- S -> Δ

Language: ab*

- CFG:

  S -> Sb | a

  abbb

  S -> Sb

  S - > Sb

  S -> Sb

  S -> a

- b*a
- CFG:

  S -> bS | a

- abbb
- bbba

Language: b*ab*

- CFG: S -> XaX

  X -> bX | Δ

  bbbab

S->XaX

X->bX

X->bX

X->bX

- X->bX
- X->Δ
-

Plaindrome; not a regular, No FA, no TG, no RE, *yes CFG*

- Odd Palindrome:
- { a, b, aaa, aba, bab, bbb, aaaaa, aabaa, ababa, abbba, baaab, bbabb, babab, bbbbb,
- aaaaaaa, aaabaaa,…}

- S -> aSa | bSb | a |b
- aaba a abaa
- aabababaa

Even Palindrom

- {Δ, aa, bb, aaaa, abba, baab, bbbb, aaaaaa, aabbaa, abaaba, …}

- S -> aSa |bSb |Δ

- abaaba
- aabbaSabbaa

- abaaba
- abaaba

a^nb^n

- {Δ, ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb, aaaaaabbbbbb,….}
- S -> aSb | Δ
- aaaaaabbbbbb
- aaaaSbbbb
- XaX

(a+b)*

- S -> aS | bS | Δ
- S -> Sa | Sb | Δ
- S -> Sa | bS | Δ

{Δ, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, …}

- abbb

- S -> aS
- S -> bS
- S->bS
- S->bS
- S -
  >

Example

$S \rightarrow aSa \mid aBa$

$B \rightarrow bB \mid b$

- First production builds equal number of a's on both sides and recursion is terminated by $S \rightarrow aBa$
- Recursion of $B \rightarrow bB$ may add any number of b's and terminates with $B \rightarrow b$

- $L(G) = \{a^n b^m a^n \; n>0, m>0\}$

example

$L(G) = \{a^n b^m c^m d^{2n} \mid n>0, m>0\}$

- Consider relationship between leading a's and trailing d's.

$S \rightarrow aSdd$

In the middle equal number of b's and c's

- $S \rightarrow A$
- $A \rightarrow bAc$
- This middle recursion terminates by $A \rightarrow bc$.

- Grammar will be

$S \rightarrow aSdd \mid aAdd$

$A \rightarrow bAc \mid bc$

- $a^n b^n c^m d^m e^p f^p g^q h^q$ | n>0, m>0
- S -> XYZW
- X -> aXb |ab
- Y -> cYd | cd
- Z -> eZf |ef
- W -> gWh |gh

- $a^n b^m e^p g^q h^q f^p c^m d^n$ | n>0, m>0
- S -> /aSd | aXd
- X -> bXc |bYc
- Y -> eYf |ef

Example

Consider another CFG

$S \rightarrow aSb \mid aSbb \mid \Lambda$

- Language defined is

$L(G) = \{a^n b^m \mid 0 \leq n \leq m \leq 2n\}$

Example

- A grammar that generates the language consisting of even-length string over {a, b}
  S → aO | bO | Λ
  O → aS | bS

- S and O work as counters i.e. when an S is in a sentential form that marks even number of terminals have been generated

- Presence of O in a sentential form indicates that an odd number of terminals have been generated.

- The strategy can be generalized, say for string of length exactly divisible by 3 we need three counters to mark 0, 1, 2

  S → aP | bP | Λ
  P → aQ | bQ
  Q → aS | bS

Even-Even

- $\Sigma = \{a,b\}$

Productions:

- $S \to SS$
- $S \to XS$
- $S \to \Lambda$
- $S \to YSY$
- $X \to aa$
- $X \to bb$
- $Y \to ab$
- $Y \to ba$

Devise a grammar that generates strings with even number of a's and even number of b's

Remarks

- We have seen that some regular languages can be generated by CFGs, and some non-regular languages can also be generated by CFGs.
- In Chapter 13, we will show that ALL regular languages can be generated by CFGs.
- In Chapter 16, we will see that there is some non-regular language that cannot be generated by any CFG.
- Thus, the set of languages generated by CFGs is properly **larger** than the set of regular languages, but properly **smaller** than the set of all possible languages.

Trees

- Consider the following CFG:

$$S \to AA$$

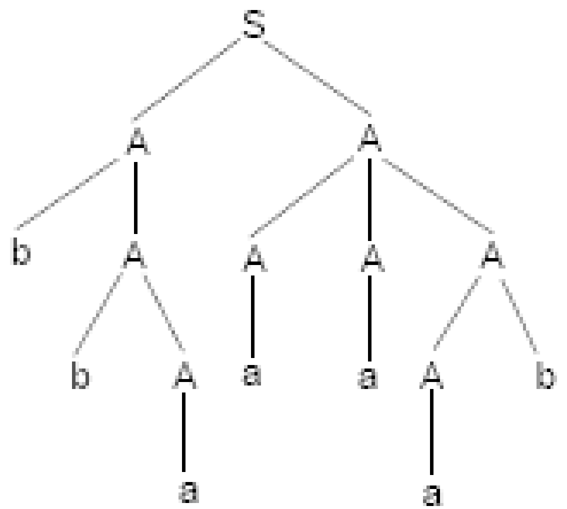
$$A \to AAA \mid bA \mid Ab \mid a$$

- The derivation of the word *bbaaaab* is as follows:

S ➔ AA ➔ bAAAA ➔ bbAaaAb ➔ bbaaaab

- We can use a tree diagram to show that derivation process:

We start with the symbol S**. Every time we use a production to replace a non-terminal by a string, we draw downward lines from the non-terminal to EACH character in the string.**



- Reading from left to right produces the word bbaaaab.
- Tree diagrams are also called **syntax trees**, **parse trees**, **generation trees**, **production trees**, or **derivation trees.**

Lukasiewicz Notation - Example

- Also called the polish prefix notation.
- A parenthesis free notation
- Consider the following CFG for a simplified version of arithmetic expressions:

  $S \rightarrow S + S \mid S * S \mid \text{number}$

  where the only non-terminal is S, and the terminals are number together with the symbols +,* .

- Obviously, the expression 3 + 4 * 5 is a word in the language defined by this CFG; however, it is ambiguous since it is not clear whether it means (3 + 4) * 5 (which is 35), or 3 + (4 * 5) (which is 23).
- To avoid ambiguity, we often need to use parentheses, or adopt the convention of “hierarchy of operators” (i.e., * is to be executed before +).
- We now present a new notation that is unambiguous but does not rely on operator hierarchy or on the use of parentheses.

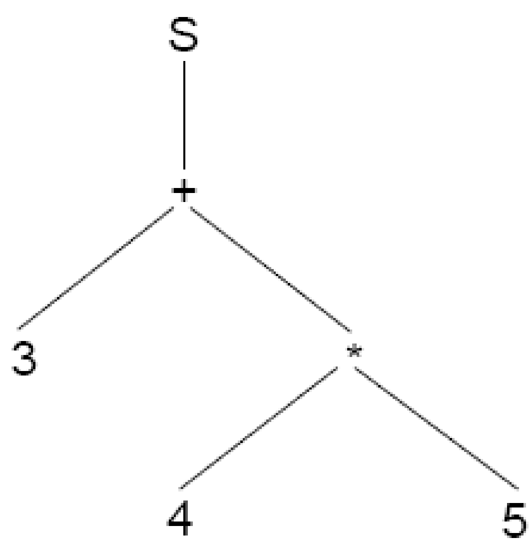
- Let us define a new CFG in which S, $+$, and $*$ are nonterminals and $\underline{\text{number}}$ is the only terminal. The productions are

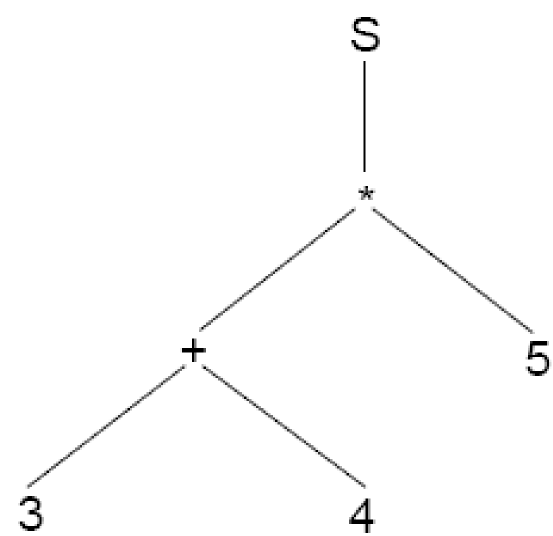
$$S \rightarrow * \mid + \mid \underline{\text{number}}$$

$$+ \rightarrow ++ \mid +* \mid +\,\underline{\text{number}} \mid *+ \mid ** \mid *\,\underline{\text{number}} \mid \underline{\text{number}}\,+ \mid \underline{\text{number}}\,* \mid \underline{\text{number}}\ \underline{\text{number}}$$

$$* \rightarrow ++ \mid +* \mid +\,\underline{\text{number}} \mid *+ \mid ** \mid *\,\underline{\text{number}} \mid \underline{\text{number}}\,+ \mid \underline{\text{number}}\,* \mid \underline{\text{number}}\ \underline{\text{number}}$$

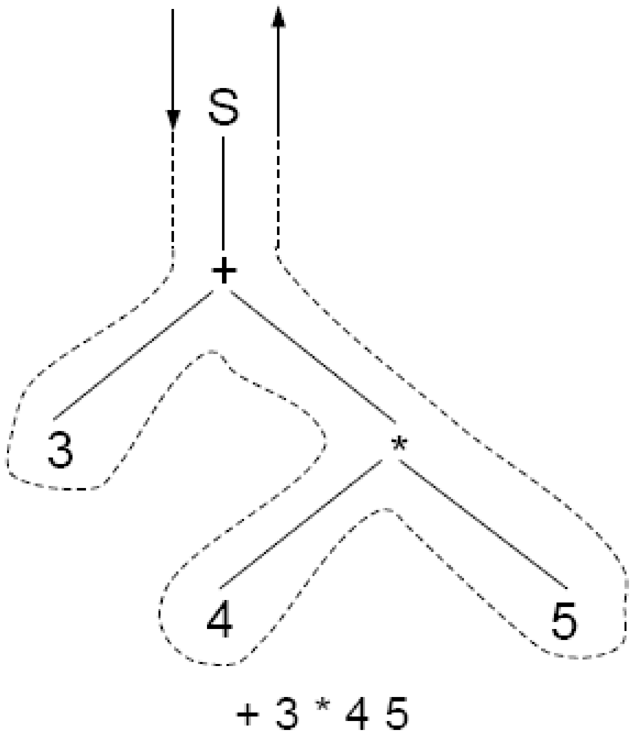
- Let us draw the derivation tree for the expression $3 + (4 * 5)$ and $(3 + 4) * 5$ respectively, using the new CFG above.

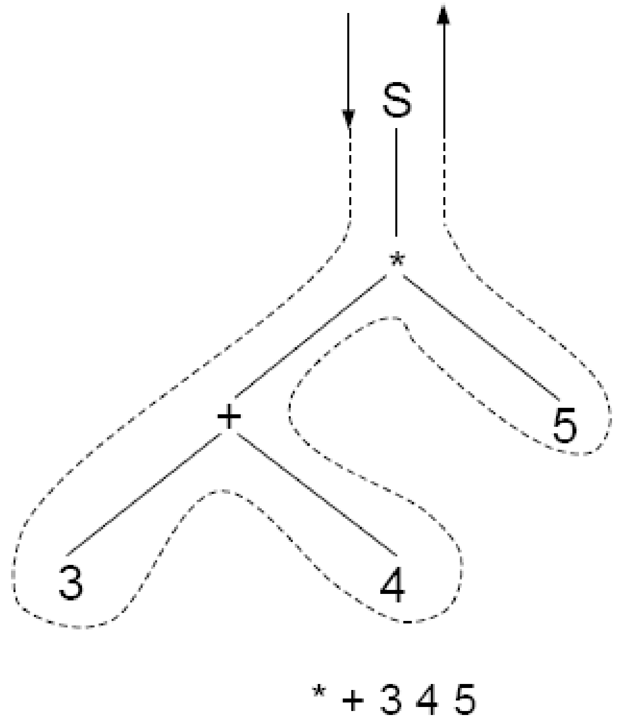




New Notation: Lukasiewicz notation

- We can now construct a **new notation** for arithmetic expressions:
  - We walk around the tree and write down symbols, **once each**, as we encounter them.
  - We begin on the left side of the start symbol S and head south.
  - As we walk around the tree, we always keep our left hand on the tree.





- Using the algorithm above, the first derivation tree is converted into the notation: + 3 * 4 5.
- The second derivation tree is converted into * + 3 4 5.

Example

- Consider the expression: + 3 * 4 5:

String	**First o-o-o**
$+\ 3\ *\ 4\ 5$	$*\ 4\ 5$
$\Downarrow$	
$+\ 3\ 20$	$+\ 3\ 20$
$\Downarrow$	
23	

- Consider the second expression: * + 3 4 5:

String	**First o-o-o**
$* + 3\ 4\ 5$	$+ 3\ 4$
$\Downarrow$	
$* 7\ 5$	$* 7\ 5$
$\Downarrow$	
35	

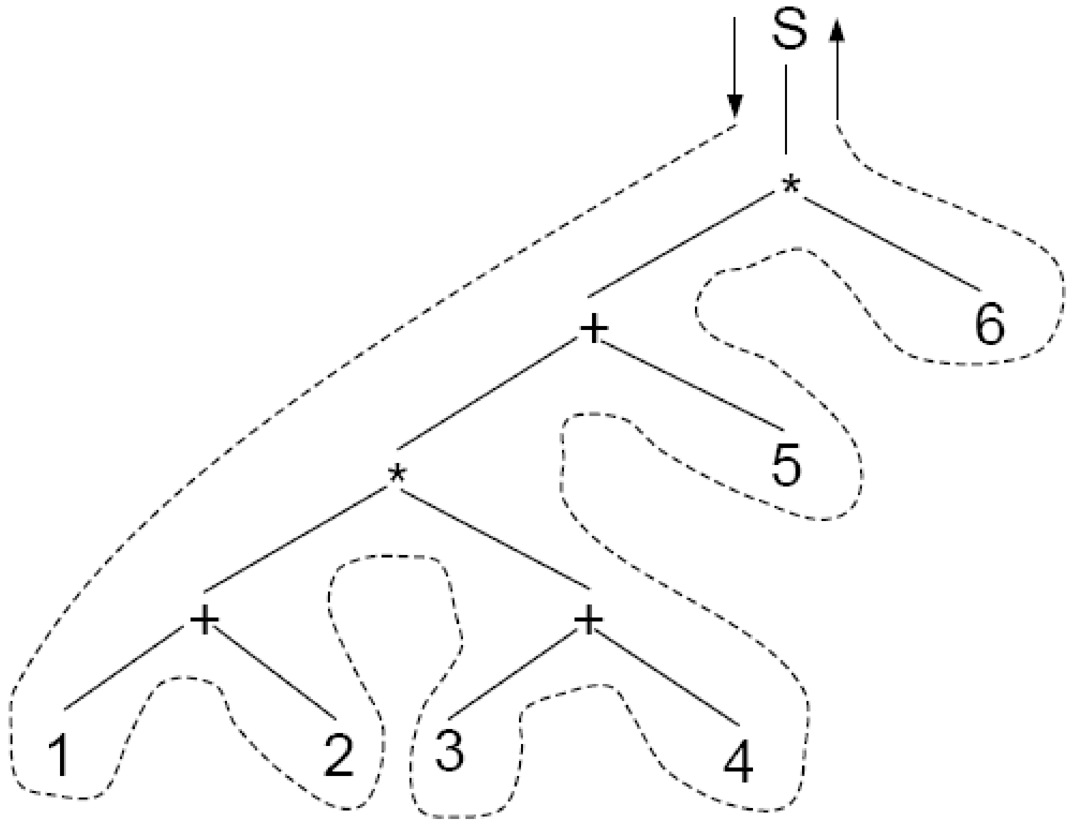
Example

- Convert the following arithmetic expression into operator prefix notation:

  ((1 + 2) * (3 + 4) + 5) * 6.

- This normal notation is called **operator infix notation**, with which we need parentheses to avoid ambiguity.

- Let's us draw the derivation tree:



- Reading around the tree gives the equivalent prefix notation expression:
- * + * + 1 2 + 3 4 5 6.

Evaluate the String

String	First o-o-o
$* + * + 1\ 2 + 3\ 4\ 5\ 6$	$+ 1\ 2$
$\Downarrow$	
$* + * 3 + 3\ 4\ 5\ 6$	$+ 3\ 4$
$\Downarrow$	
$* + * 3\ 7\ 5\ 6$	$* 3\ 7$
$\Downarrow$	
$* + 21\ 5\ 6$	$+ 21\ 5$
$\Downarrow$	
$* 26\ 6$	$* 26\ 6$
$\Downarrow$	
156	

- This operator prefix notation was invented by Lukasiewicz (1878 - 1956) and is often called Polish notation.

- There is a similar **operator postfix notation** (also called Polish notation), in which the operation symbols (+, -, ...) come after the operands. This can be derived by tracing around the tree of the other side, keeping our **right** hand on the tree and then reversing the resultant string.

- Both these methods of notation are useful for computer science: Compilers often convert infix to prefix and then to assembler code.

Ambiguity- example

- Consider the language generated by the following CFG:

$S \to AB$

$A \to a$

$B \to b$

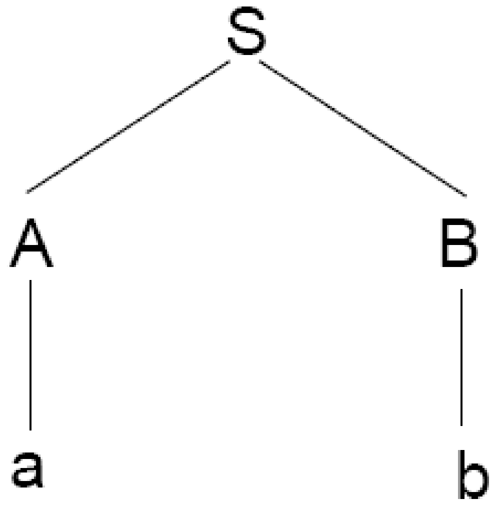
- There are two derivations of the word ab:

$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$

or

$S \Rightarrow AB \Rightarrow Ab \Rightarrow ab$

- However, These two derivations correspond to the same syntax tree:



- The word ab is therefore not ambiguous. In general, when all the possible derivation trees are the same for a given word, then the word is unambiguous.

Ambiguity - Definition

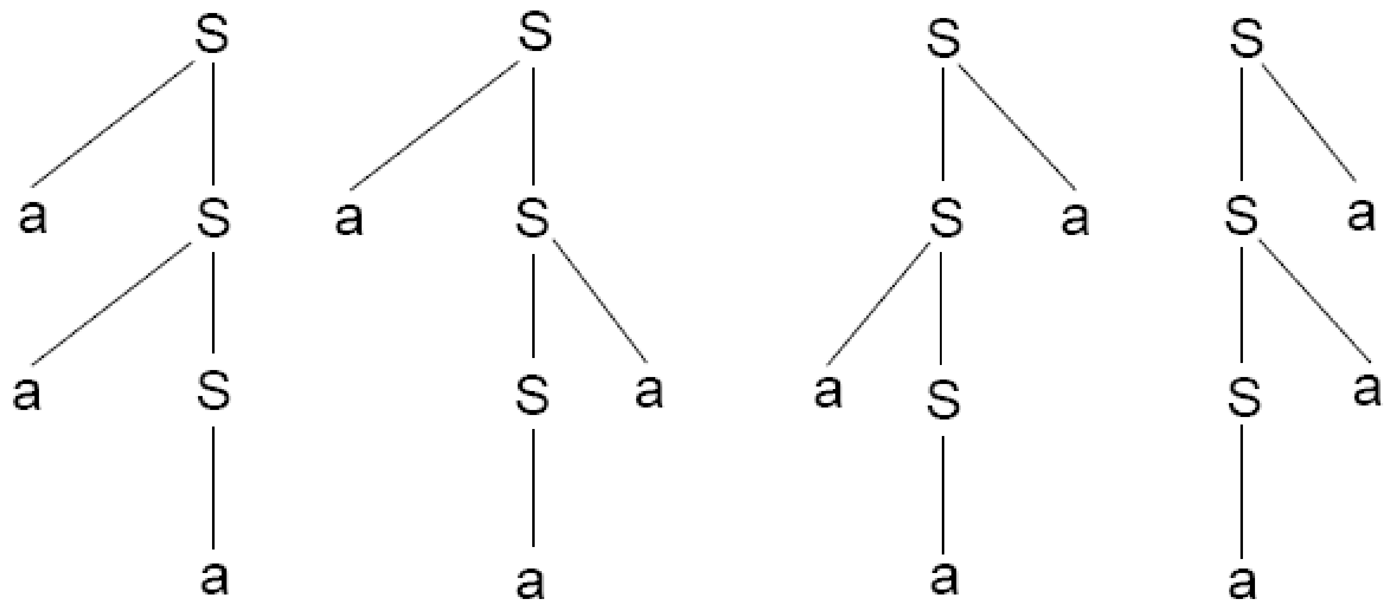
A CFG is called **ambiguous** if for at least one word in the language that it generates, there are two possible derivations of the word that correspond to different syntax trees. If a CFG is not ambiguous, it is called **unambiguous**.

Example

- The following CFG defines the language of all non-null strings of a's:

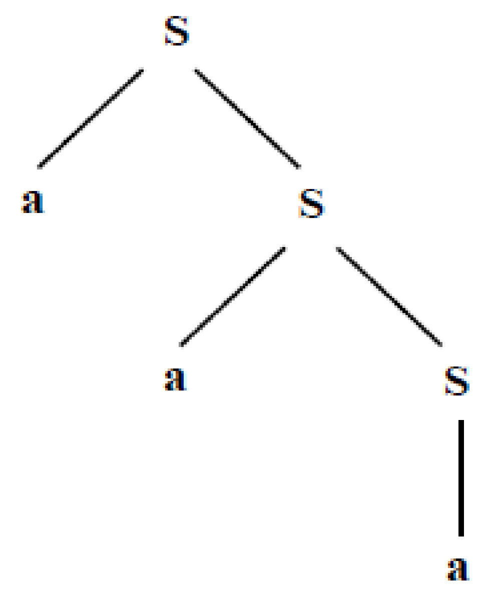
  $S \rightarrow aS \mid Sa \mid a$

- The word a^3 can be generated by 4 different trees:



Example

- the CFG, $S \to aS|a$ is not ambiguous as neither the word aaa nor any other word can be derived from more than one production trees. The derivation tree for aaa is as follows:



The Total Language Tree

- It is possible to depict the generation of all the words in the language of a CFG simultaneously in one big (possibly infinite) tree.

Definition:

- For a given CFG, we define a tree with the start symbol S as its root and whose nodes are working strings of terminals and non-terminals. The descendants of each node are all the possible results of applying every applicable production to the working string, one at a time. A string of all terminals is a terminal node in the tree. The resultant tree is called the **total language tree** of the CFG.

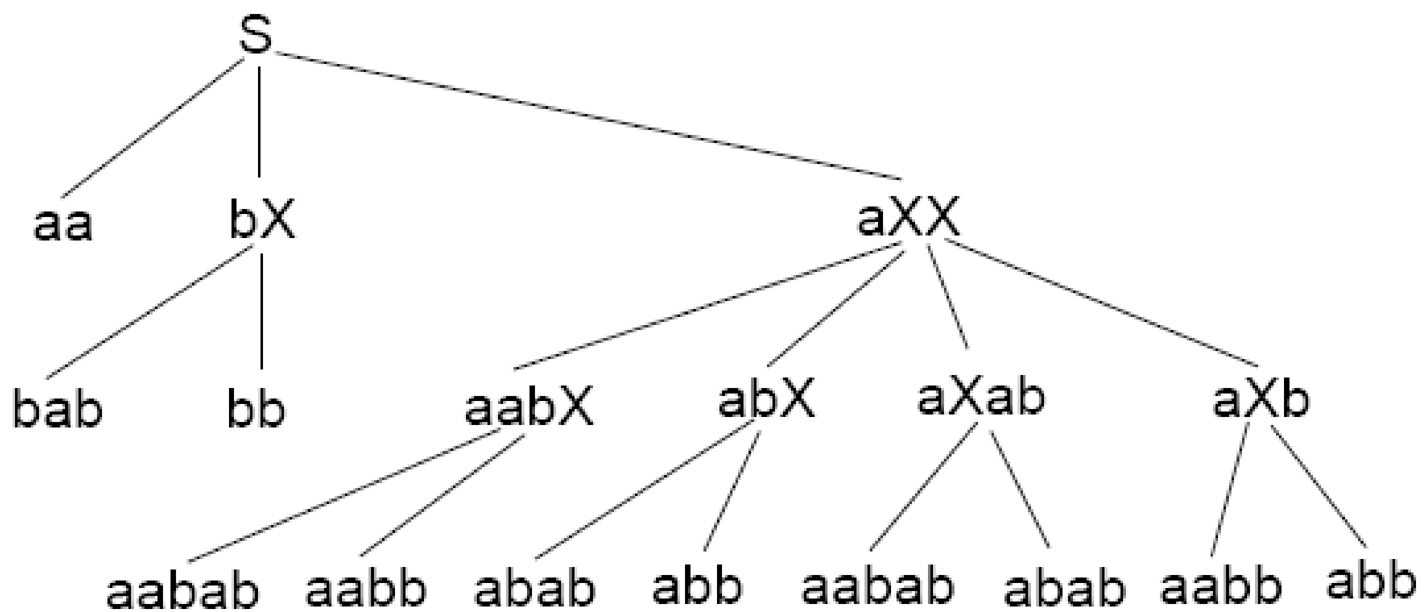
Example

- Consider the CFG:

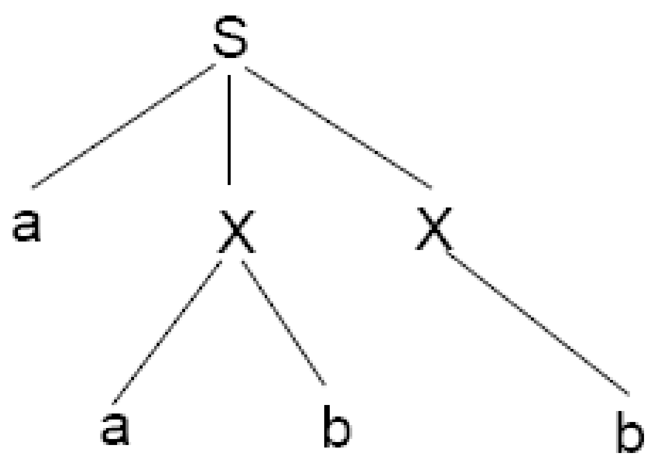
  $S \rightarrow aa \mid bX \mid aXX$

  $X \rightarrow ab \mid b$

- The total language tree is



- The above total language has only 7 different words.
- Four of its words (abb, aabb, abab, aabab) have two different derivations because they appear as terminal nodes in two different places.
- However, these words are NOT generated by two different derivation trees. Hence, the CFG is unambiguous. For example,



Example

- Consider the CFG:

$$S \to aSb \mid bS \mid a$$

- The language of this CFG is infinite, so is the total language tree: The tree may get arbitrary wide as well as infinitely long.
- Can you draw the beginning part of this total language tree?

Semi Word

- For a given CFG, semi-word is a string of terminals (may be none) concatenated with exactly one non-terminal (on the right).

- In general semi-word has the shape

(terminal) (terminal)….(terminal) (Non-Terminal)

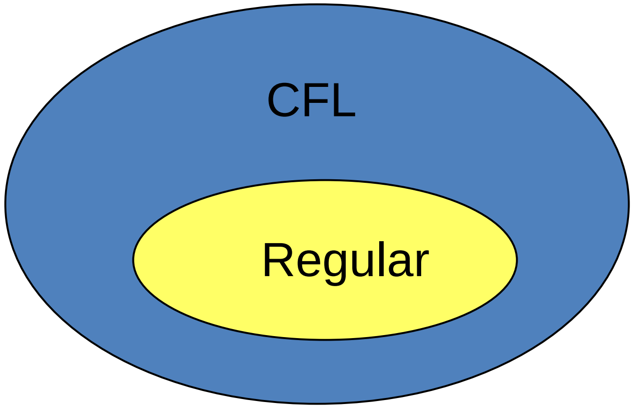
e.g. aaaX abcY bbY

A word is a string of terminals only (zero or more terminals)

Regular Grammar

Given an FA, there is a CFG that generates exactly the language accepted by the FA.

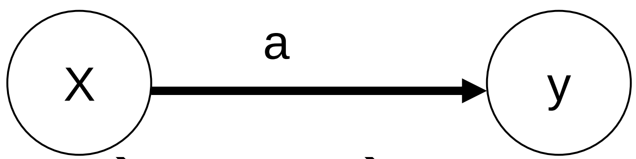
- In other words, all regular languages are CFLs

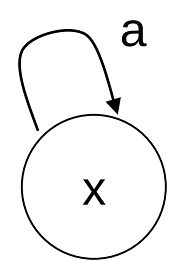


Creating a CFG from an FA

<u>Step-1</u> The Non-terminals in CFG will be all names of the states in the FA with the start state renamed S.

<u>Step-2</u> For every edge





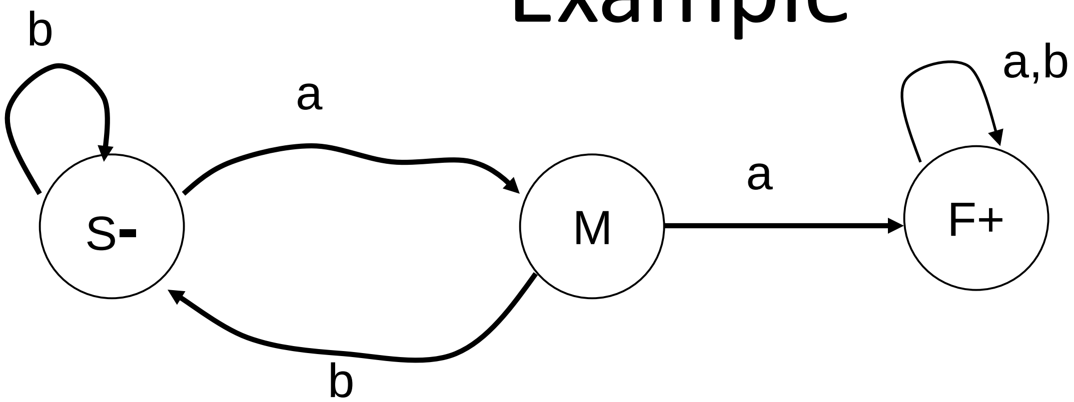
Create productions X→aY or X→aX

Do the same for b-edges

<u>Step-3</u> For every final-state X, create the production

X→Λ

Example



$S \rightarrow aM$

$S \rightarrow bS$

$M \rightarrow aF$

$M \rightarrow bS$

$F \rightarrow aF$

$F \rightarrow bF$

$F \rightarrow \Lambda$

Note: It is not necessary that each CFG has a corresponding FA. But each FA has an equivalent CFG.

Regular Grammar

Theorem 22:

If all the productions in a given CFG fit one of the two forms: Non-terminal → semiword

or Non-terminal → word

(Where the word may be a Λ or string of terminal), then the language generated by the CFG is Regular.

Proof:

For a CFG to be regular is by constructing a TG from the given CFG.

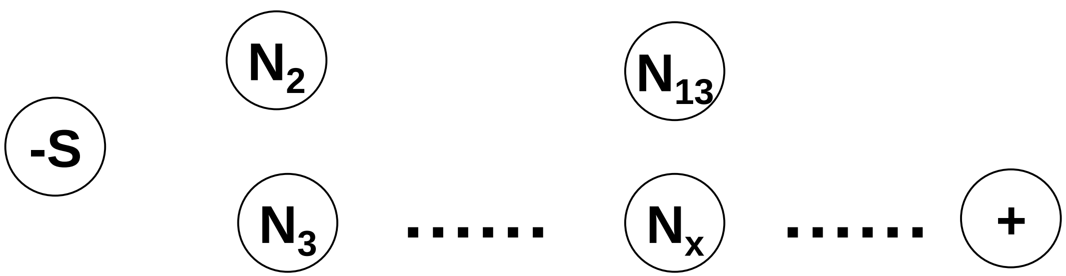
Proof contd.

- Let us consider a general CFG in this form

$N_1 \rightarrow w_1N_2$	$N_7 \rightarrow w_{10}$
$N_1 \rightarrow w_2N_3$	$N_{18} \rightarrow w_{23}$
$N_2 \rightarrow w_3N_4$	---------------
---------------	---------------

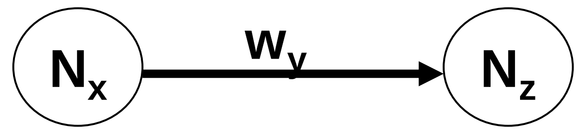
Where N's are non-terminal and w's are the string of terminal and part w_yN_z are semiwords.

Let N_1=S. Draw a small circle for each N and one extra circle labelled +, the circle for S we label (-)

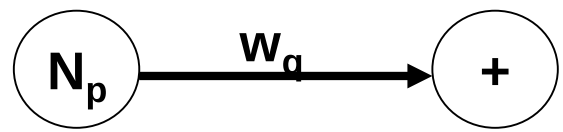


Proof contd.

- For each production of the form $N_x \rightarrow w_y N_z$, draw a directed edge from state N_x to N_z with label w_y.



- If Nx = Nz, the path is a loop
- For every production of the form $N_p \rightarrow w_q$, draw a directed edge from Np to + and label it with w_q even if $w_q = \Lambda$.



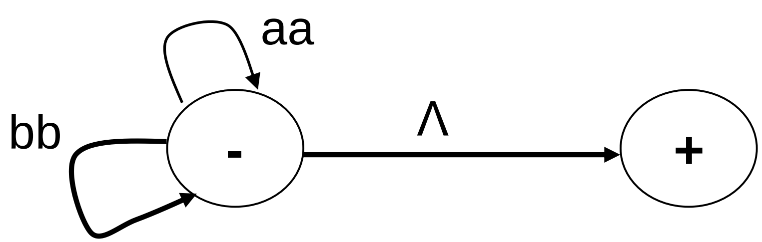
- Any path in TG form – to + corresponds to a word in the language of TG (by concatenating symbols) and simultaneously corresponds to sequence of productions on the CFG generating words.
- Conversely every production of the word in the CFG:

S ➔ wN ➔ wwN ➔ wwwN ➔ ….. ➔ wwwww

Corresponds to a path in this TG.

Example

- Consider the CFG S → aaS | bbS | Λ



- The regular expression is given by (aa + bb)*.
- Consider the CFG

S→aaS | bbS | abX | baX | Λ

X→ aaX | bbX | abS | baS

- Language accepted?

- EVEN-EVEN