Topics in Context-Free Grammar CFG's

HUSSEIN S. AL-SHEAKH

Outline

- Context-Free Grammar
- Ambiguous Grammars
- LL(1) Grammars
- Eliminating Useless Variables
- Removing Epsilon
- Nullable Symbols

Context-Free Grammar (CFG)

- Context-free grammars are powerful enough to describe the syntax of most programming languages; in fact, the syntax of most programming languages is specified using context-free grammars.
- In linguistics and computer science, a context-free grammar (CFG) is a formal grammar in which every production rule is of the form

$$V \rightarrow w$$

- Where V is a "non-terminal symbol" and w is a "string" consisting of terminals and/or non-terminals.
- The term "context-free" expresses the fact that the non-terminal V can always be replaced by w, regardless of the context in which it occurs.

Definition: Context-Free Grammars

- Definition **3.1.1** **(A. Sudkamp book – Language and Machine 2ed Ed.)**

A context-free grammar is a quadruple (V, **Z,** P, S) where:

V is a finite set of variables.

E (the alphabet) is a finite set of terminal symbols.

P is a finite set of rules (A→x).

Where x is string of variables and terminals

S is a distinguished element of V called the start symbol.

The sets V and **E** are assumed to be disjoint.

Definition: Context-Free Languages

A language L is context-free

IF AND ONLY IF

there is a grammar G with $L=L(G)$.

Example

- A context-free grammar G :

$$S \to aSb$$
$$S \to \in$$

A derivation:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

$$L(G) = \{a^n b^n : n \geq 0\}$$

(((( ))))

Derivation Order

$$1.\ S \rightarrow AB \qquad 2.\ A \rightarrow aaA \qquad 4.\ B \rightarrow Bb$$

$$3.\ A \rightarrow \in \qquad 5.\ B \rightarrow \in$$

Leftmost derivation:

$$S \overset{1}{\Rightarrow} AB \overset{2}{\Rightarrow} aaAB \overset{3}{\Rightarrow} aaB \overset{4}{\Rightarrow} aaBb \overset{5}{\Rightarrow} aab$$

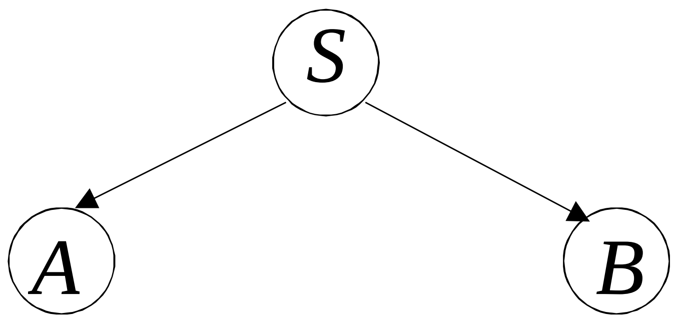
Rightmost derivation:

$$S \overset{1}{\Rightarrow} AB \overset{4}{\Rightarrow} ABb \overset{5}{\Rightarrow} Ab \overset{2}{\Rightarrow} aaAb \overset{3}{\Rightarrow} aab$$

Derivation Trees

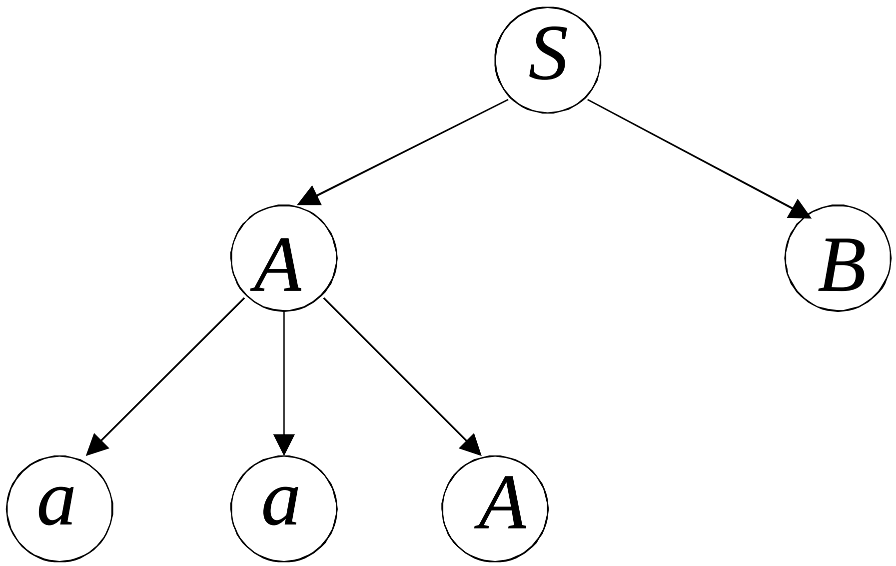
$S \to AB$ $\qquad$ $A \to aaA \mid \in$ $\qquad$ $B \to Bb \mid \in$

$$S \Rightarrow AB$$



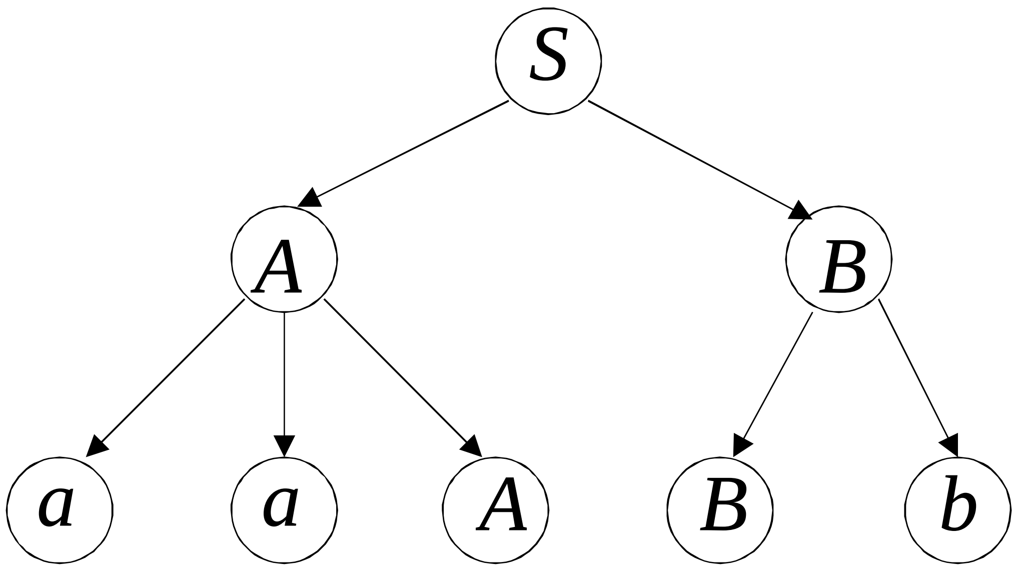
$$S \to AB \qquad A \to aaA \mid \in \qquad B \to Bb \mid \in$$

$$S \Rightarrow AB \Rightarrow aaAB$$



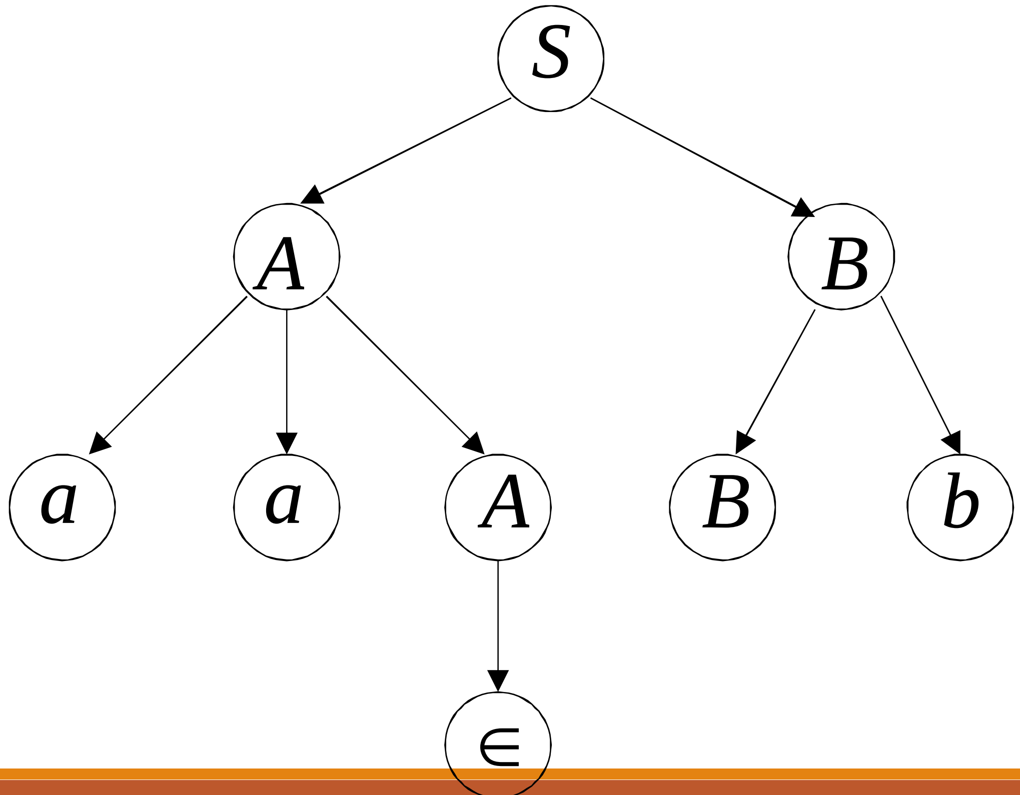
$$S \to AB \qquad A \to aaA \mid \in \qquad B \to Bb \mid \in$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb$$



$$S \to AB \qquad A \to aaA \mid \in \qquad B \to Bb \mid \in$$

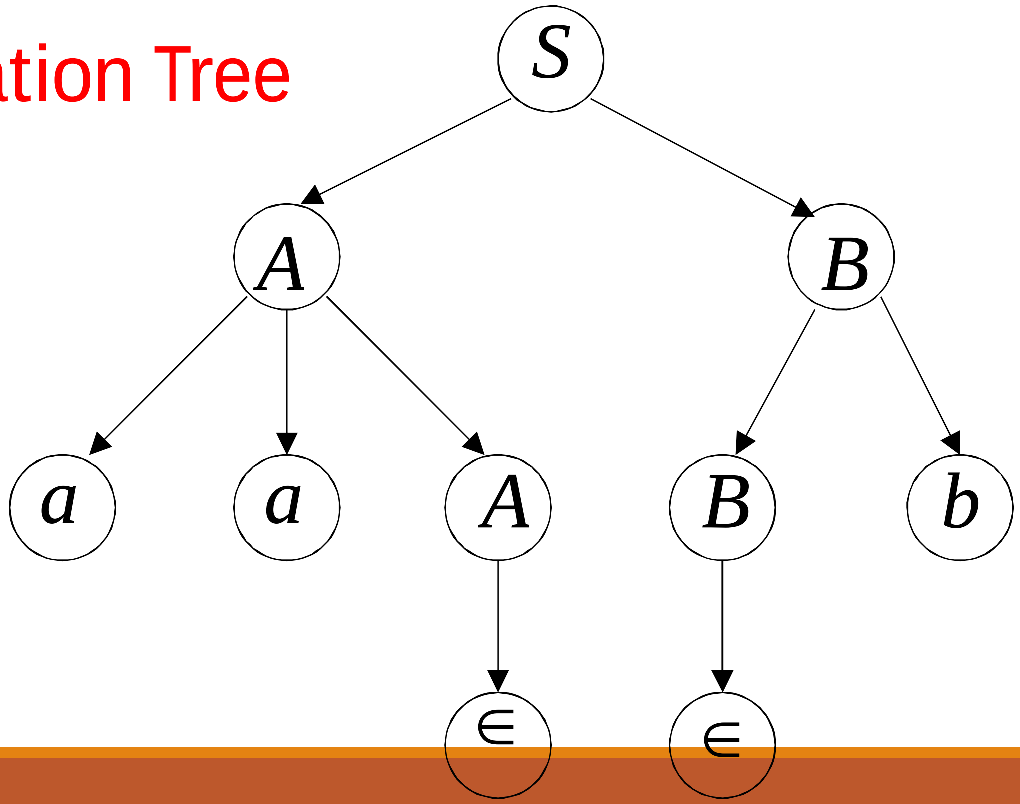
$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb$$



$S \to AB$ $\quad A \to aaA | \in$ $\quad B \to Bb | \in$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

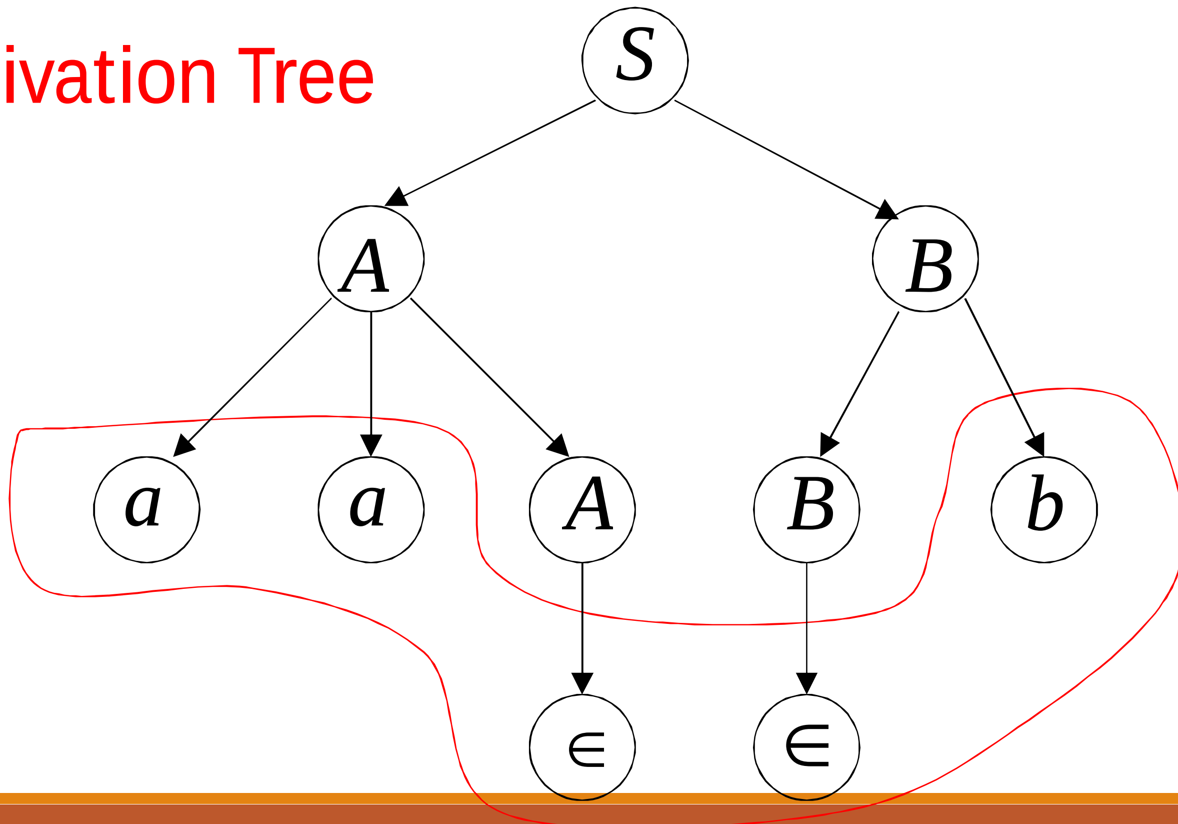
Derivation Tree



$$S \to AB \qquad A \to aaA \mid \in \qquad B \to Bb \mid \in$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

Derivation Tree



yield

$$aa \in \in b$$

$$= aab$$

Ambiguous Grammars

- problem: compilers use parse trees to interpret the meaning of parsed expressions.
- Assigns a unique parse tree to each string in the language is important in many application.
- A CFG is *ambiguous* if there is in its language that has at least two different parse trees (yield of two or more parse trees).
- Two different leftmost / rightmost derivations should produce different parse trees.
- Definition:

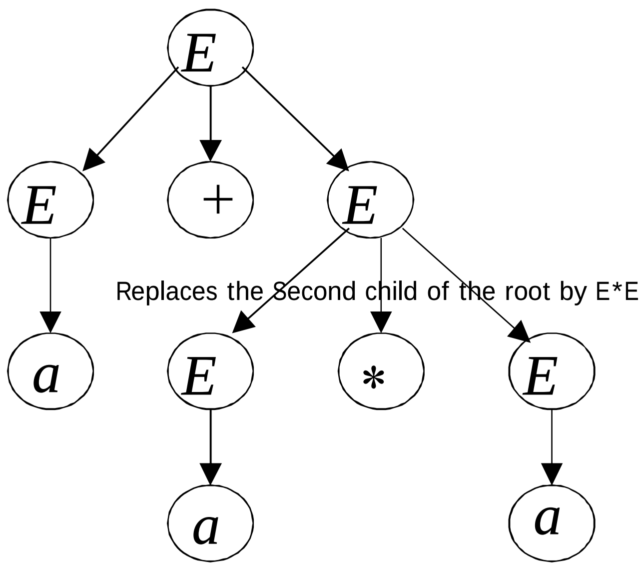
A context-free grammar G is **ambiguous** if some string $w \in L(G)$ has: two or more leftmost/rightmost derivation trees.

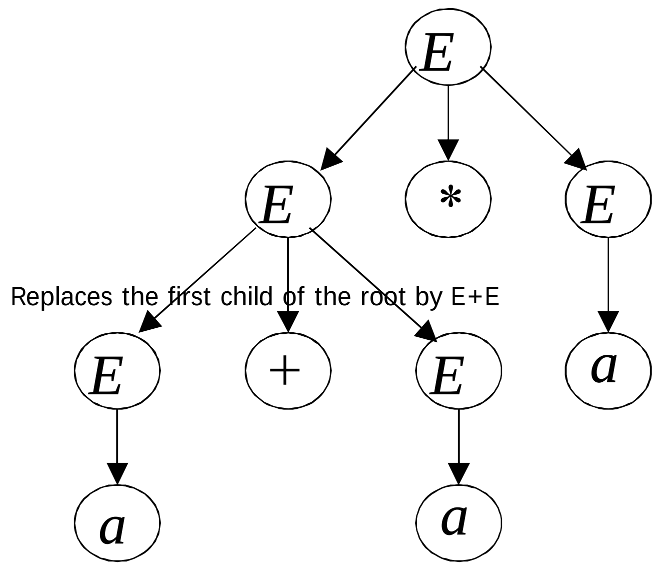
Q1) give a definition and example of ambiguous Grammars?

The grammar $E \rightarrow E + E \mid E * E \mid (E) \mid a$

is **ambiguous:**

string $a + a * a$ has two derivation trees





The grammar $E \to E+E \mid E*E \mid (E) \mid a$

is **ambiguous:**

string $a+a*a$ has two derivation trees

$$E \Rightarrow E+E \Rightarrow a+E \Rightarrow a+E*E$$

$$\Rightarrow a+a*E \Rightarrow a+a*a$$

$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow a+E*E$$

$$\Rightarrow a+a*E \Rightarrow a+a*a$$

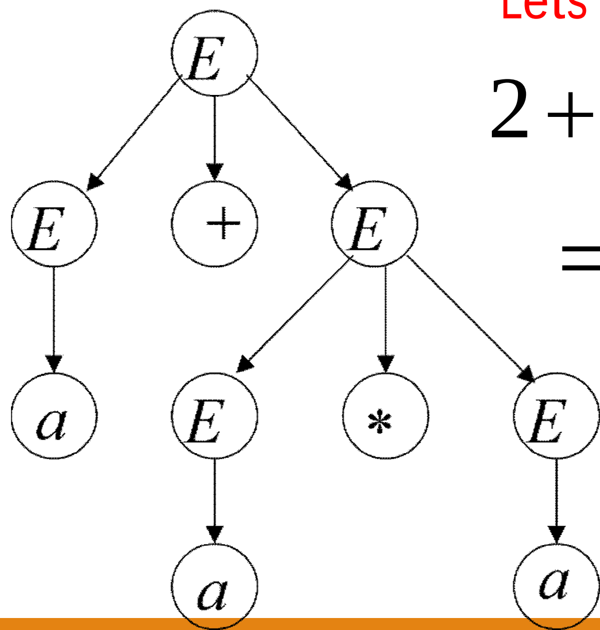
Why do we care about ambiguity?

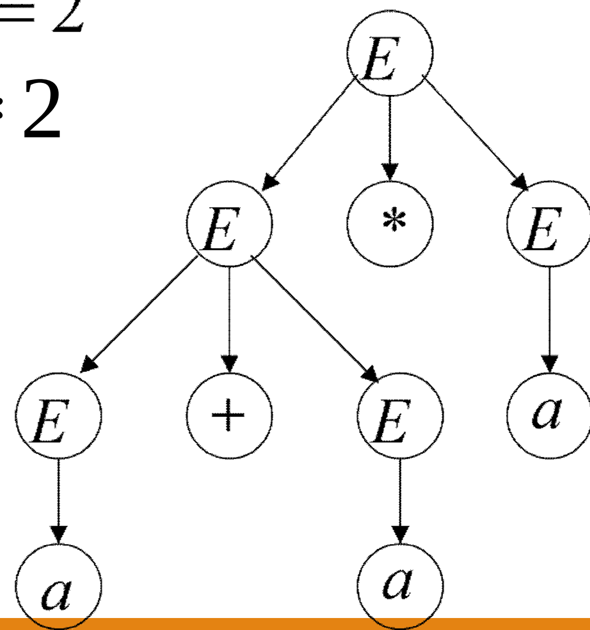
$$a + a * a$$

Lets $a = 2$

$$2 + 2 * 2$$

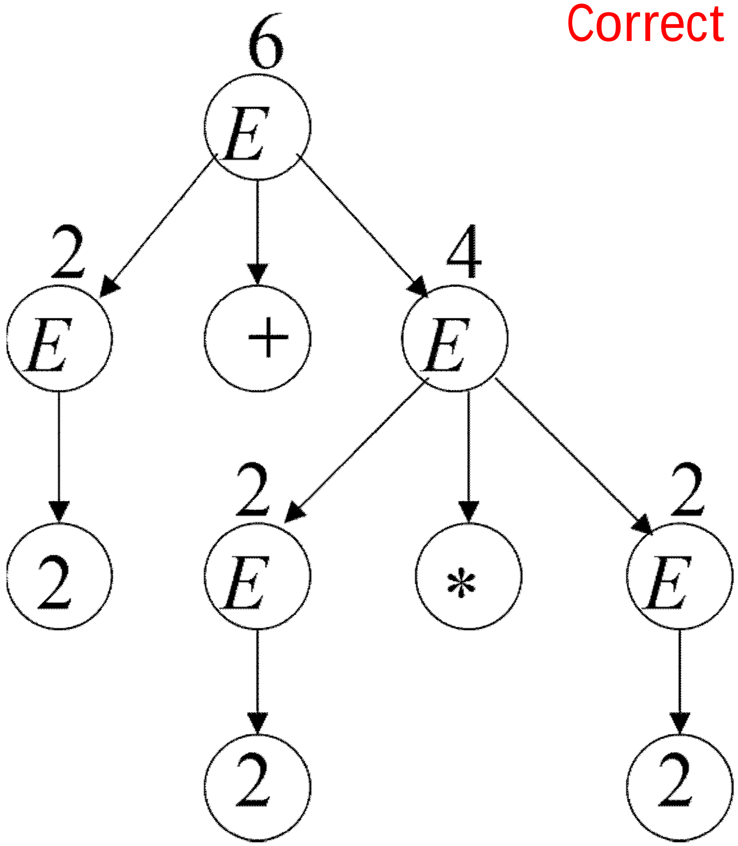
$$= ?$$



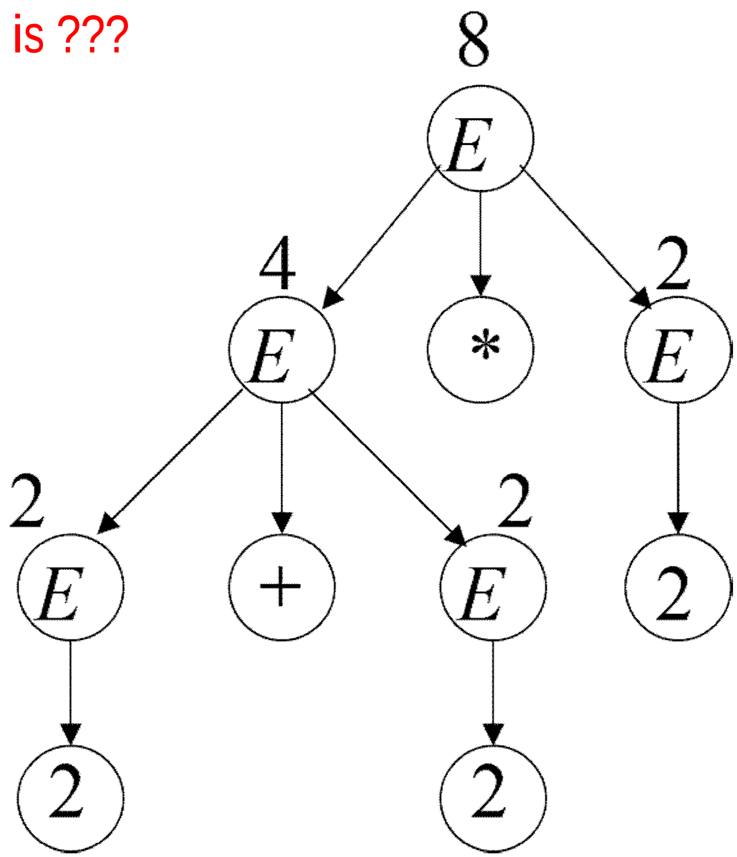


$$2 + 2 * 2 = 6$$

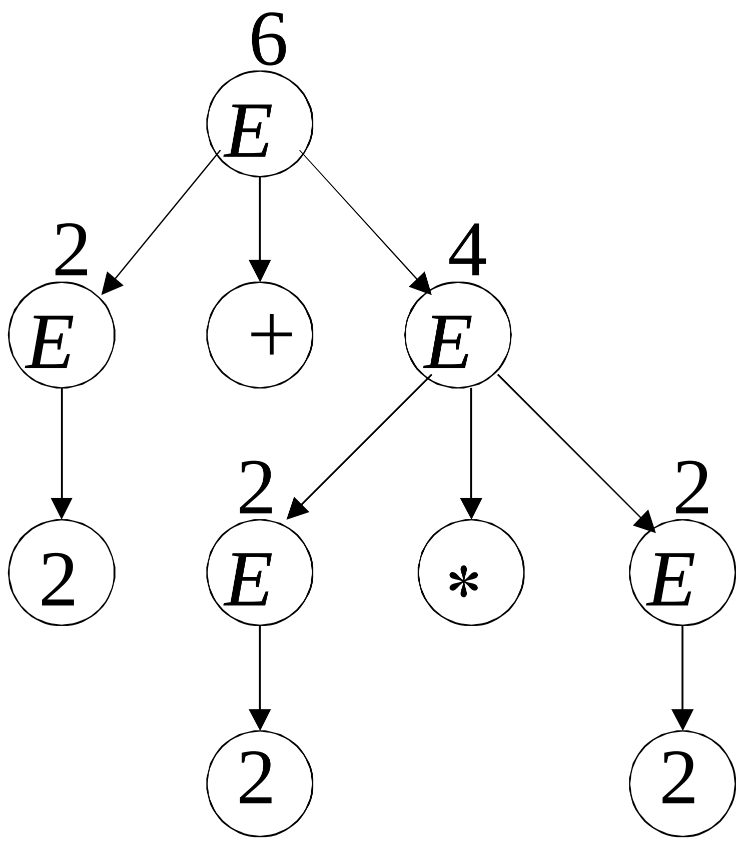
$$2 + 2 * 2 = 8$$



Correct result is ???



Correct result = $2 + 2 * 2 = 6$



Therefore,

Ambiguity is **bad** for programming languages

We need to remove ambiguity

Fix the ambiguous grammar:

$$E \to E + E \quad | \; E * E \; | \; (E) \; | \; a$$

New non-ambiguous grammar:

$$E \to E + T$$

$$E \to T$$

$$T \to T * F$$

$$T \to F$$

$$F \to (E)$$

$$F \to a$$

$$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F$$

$$\Rightarrow a+F*F \Rightarrow a+a*F \Rightarrow a+a*a$$

$$E \rightarrow E+T$$

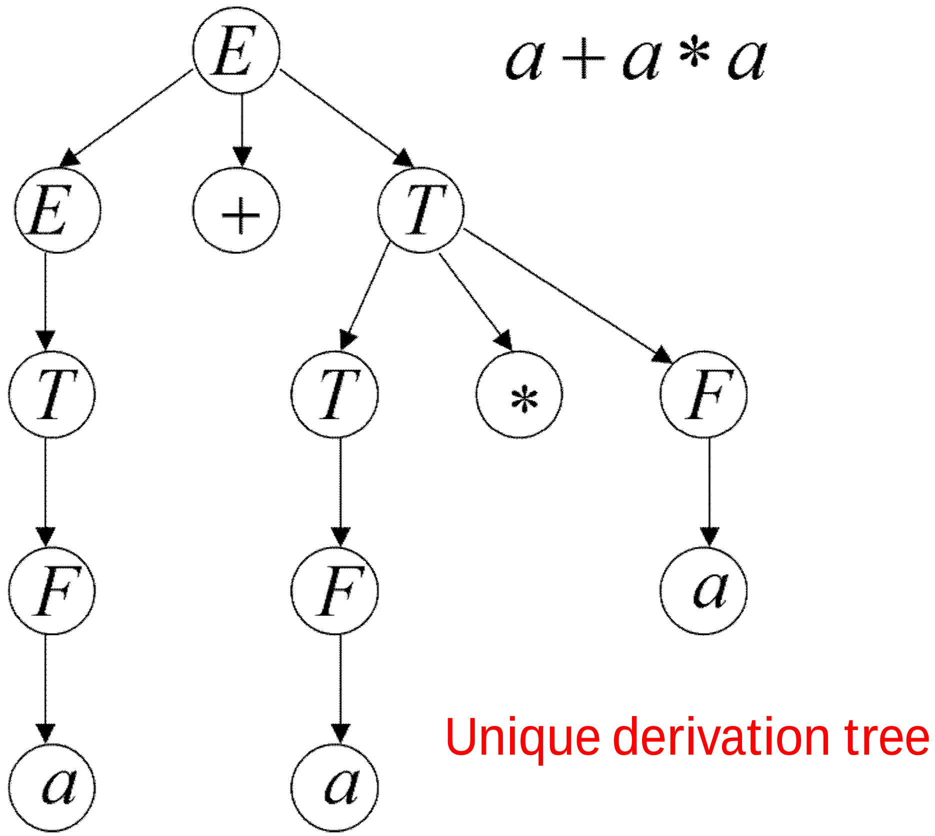
$$E \rightarrow T$$

$$T \rightarrow T*F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow a$$



The grammar G : $E \to E + T$

$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to a$

IS non-ambiguous:

Every string $w \in L(G)$ has a unique derivation tree

LL(1) Grammars

"Leftmost derivation, Left-to-right scan, 1 symbol lookahead."

- First L: scans input from left to right.
- Second L: produces a leftmost derivation.
- 1: uses one input symbol of lookahead at each step to make a parsing decision.
- A grammar whose parsing table has no multiply-defined entries is a LL(1) grammar.
- No ambiguous or left-recursive grammar can be LL(1)

Definition 16.1.1 from (A. Sudkamp book – Language and Machine 2ed Ed.)

- Let **G** = (V, **E,** P, ***S)*** be a context-free grammar and A $\in$ V.
  - The lookahead set of the variable A, LA(A), is defined by

$$LA(A) = \{ x \mid S \Rightarrow^* uAv \Rightarrow^*: ux \in \Sigma^* \}$$

  - For each rule A → *w* in P, the lookahead set of the rule A → *w* is defined by

$$LA(A \rightarrow w) = \{x \mid wv \Rightarrow^* x \text{ where } x \in \Sigma^* \text{ and } S \Rightarrow^* uAv\}$$

- LA(A) consists of all terminal strings derivable from strings Av, where uAv is a left sentential form of the grammar.
- LA(A → *w)* is the subset of LA(A) in which the subderivations Av →*x are initiated with the rule A → *w.*

Example **16.1.1**

From (A. Sudkamp book – Language and Machine 2ed Ed.)

- The lookahead sets are constructed for the variables and the rules of the grammar

$$G1: \quad S \rightarrow Aabd \mid cAbcd$$

$$A \rightarrow a \mid b \mid \in$$

- LA(S) consists of all terminal strings derivable from S.

$$LA(S) = \{aabd, babd, abd, cabcd, cbbcd, cbcd\}$$

$$LA(S \rightarrow Aabd) = \{aabd, babd, abd\}$$

$$LA(S \rightarrow cAbcd) = \{cabcd, cbbcd, cbcd\}$$

- Knowledge of the first symbol of the lookahead string is sufficient to select the appropriate S rule.

Lookahead Example Cont.

- We must consider derivations from all the left sentential forms of G1 that contain A, to construct the lookahead set for the variable A.
- There are only two such sentential forms:

Aabd and *cAbcd*

- The lookahead sets consist of terminal strings derivable from *Aabd* and *Abcd are:*

$$LA(A \rightarrow a) = \{aabd, abcd\}$$

$$LA(A \rightarrow b) = \{babd, bbcd\}$$

$$LA(A \rightarrow \in) = \{abd, bcd\}$$

- The substring ab can be obtained by applying A→ a to *Abcd* and by applying A→ ∈ to *Aabd.*

Length-Three Lookahead

- Looking ahead three symbols (length-three) in the input string provides sufficient information to discriminate between these rules.
- A top-down parser with a three-symbol lookahead can deterministically construct derivations in the grammar G1.
- The length-three lookahead sets for the rules of the grammar G1

G1: LA3(S → *Aabd)* = *{aab , bab , abd}*

LA3(S → *cAbcd)* = *{cab , cbb , cbc}*

LA3(A →*a)* = *{aab , abc }*

LA3(A → *b)* = *{bab , bbc }*

LA(A → ∈) = *{abd, bcd}*

- Since there is no string in common in the length three lookahead sets of the S rules or the A rules, a three symbol lookahead is sufficient to determine the appropriate rule of G1.

Example 16.1.4

From (A. Sudkamp book – Language and Machine 2ed Ed.)

- The language $\{a^i abc^i \mid i > 0\}$ is generated by each of the grammars G1, G2, and G3. The minimal length lookahead sets necessary for discriminating between alternative productions are given for these grammars.

Rule		**Lookahead Set**	
G_1:	$S \rightarrow aSc$	$\{aaa\}$	Three symbol lookahead is required to determine the appropriate rule
	$S \rightarrow aabc$	$\{aab\}$	
G_2:	$S \rightarrow aA$		
	$A \rightarrow Sc$	$\{aa\}$	S → *aSc* and S→*aabc* *using* (left factoring) technique to reduces the length of the lookahead needed to select the rules.
	$A \rightarrow abc$	$\{ab\}$	
G_3:	$S \rightarrow aaAc$		
	$A \rightarrow aAc$	$\{a\}$	The recursive A rule generates an a while the nonrecursive rule terminates the derivation by generating a *b*.
	$A \rightarrow b$	$\{b\}$	

Q2: Give an example to show the deference between lookahead sets?

LL(1) Grammar Example

- Construct the parse table for the following LL(1) grammar.

$E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow (E)$
$E \rightarrow id$

- This grammar is left-recursive, ambiguous and requires left-factoring. It needs to be modified before we build a predictive parser for it:

Remove ambiguity:

$E \rightarrow E+T$
$T \rightarrow T*F$
$F \rightarrow (E)$
$F \rightarrow id$

Remove left recursion:

$E \rightarrow TE'$
$E' \rightarrow +TE' | \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' | \varepsilon$
$F \rightarrow (E)$
$F \rightarrow id$

Compute FIRST(X) as follows:

- if X is a terminal, then FIRST(X)={X}
- if $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X)
- if X is a non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_n$ is a production, add FIRST(Y_i) to FIRST(X) if the preceding Y_js contain ε in their FIRSTs

Compute FOLLOW as follows:

- FOLLOW(S) contains EOF
- For productions $A \rightarrow \alpha B \beta$, everything in FIRST($\beta$) except ε goes into FOLLOW(B)
- For productions $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε, FOLLOW(B) contains everything that is in FOLLOW(A)

Building a parser

The grammar:

$E \to TE'$
$E' \to \mathbf{+}TE' | \varepsilon$
$T \to FT'$
$T' \to \mathbf{*}FT' | \varepsilon$
$F \to \mathbf{(}E\mathbf{)}$
$F \to \mathbf{id}$

FIRST(E) = {**(**, **id**}
FIRST(T) = {**(**, **id**}
FIRST(F) = {**(**, **id**}
FIRST(E') ={**+**, ε}
FIRST(T') ={*, ε}
FOLLOW(E) = {**$**, **)**}
FOLLOW(E') = {**$**, **)**}
FOLLOW(T) = {**+**, **$**, **)**}
FOLLOW(T') = {**+**, **$**, **)**}
FOLLOW(F) = {*, **+**, **$**, **)**}

(first = first terminal after arc, if not, non-terminal derivation)
(follow= first (next), if null → follow (non-terminal)

Parsing table

	+	*****	**(**	**)**	**id**	**$**
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'	$E' \rightarrow \mathbf{+}TE'$			$E' \rightarrow \varepsilon$		$E' \rightarrow \varepsilon$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow \varepsilon$	$T' \rightarrow \mathbf{*}FT'$		$T' \rightarrow \varepsilon$		$T' \rightarrow \varepsilon$
F			$F \rightarrow \mathbf{(}E\mathbf{)}$		$F \rightarrow \mathbf{id}$	
+	match					
*		match				
(			match			
)				match		
id					match	
$						accept

Eliminating Useless Variables

- Context-Free grammars can be badly designed, some variables that play no role in the derivation of any terminal string.
- A symbol X is useful for Grammar G = {V, T, P, S}, if there is some derivation of the form S ==>* a X b ==>* w , where w ϵ T*.
- X ϵ V or X ϵ T.
- The sentential form of a X b might be the first or last derivation.
- If X is not useful, then X is useless .

Q3) give example of useless symbols.

Characteristics of useful symbols

1. X is generating if $X \Longrightarrow^* w$ for some terminal string w. Every terminal is generating since w can be that terminal itself, which is derived by 0 steps.

2. X is reachable if there is a derivation

   $S \Longrightarrow^* aXb$ for some a and b.

- A symbol which is useful is surely to be both generating and reachable.

Removing All Useless Variables

Step 1: Remove Nullable Variables

Step 2: Remove Unit-Productions

Step 3: Remove Useless Variables

Nullable Variables

Theorem

If L is a CFL, then L-{ε} has a CFG with no ε-productions.

Basis: If there is a production A → ε, then A is nullable.

$$\in -\text{production}: \quad A \to \in$$

Induction: If there is a production A→ α, and all symbols of α are nullable, then A is nullable.

$$A \Rightarrow \ldots \Rightarrow \in$$

Removing Nullable Variables

Example Grammar:

$$S \to aMb$$
$$M \to aMb$$
$$M \to \in$$

Nullable variable

$$S \to aMb$$

$$M \to aMb$$

$$\cancel{M \to \in}$$

$$M \to \in$$

Final Grammar

$$S \to aMb$$

$$S \to ab$$

$$M \to aMb$$

$$M \to ab$$

Unit-Productions

Unit Production: $A \to B$

(a single variable in both sides)

Removing Unit Productions

Observation:

$$A \rightarrow A$$

Is removed immediately

Example Grammar:

$$S \rightarrow aA$$

$$A \rightarrow a$$

$$A \rightarrow B$$

$$B \rightarrow A$$

$$B \rightarrow bb$$

$$S \to aA$$

$$A \to a$$

$$\cancel{A \to B}$$

$$B \to A$$

$$B \to bb$$

Substitute

$$A \to B$$

$$S \to aA \mid aB$$

$$A \to a$$

$$B \to A \mid B$$

$$B \to bb$$

$$S \to aA \mid aB$$

$$A \to a$$

$$B \to A \mid \cancel{B}$$

$$B \to bb$$

Remove

$$B \to B$$

$$S \to aA \mid aB$$

$$A \to a$$

$$B \to A$$

$$B \to bb$$

$$S \to aA \mid aB$$

$$A \to a$$

$$\cancel{B \to A}$$

$$B \to bb$$

Substitute

$$B \to A$$

$$S \to aA \mid aB \mid aA$$

$$A \to a$$

$$B \to bb$$

Remove repeated productions

$$S \rightarrow aA \mid aB \mid \cancel{aA}$$

$$A \rightarrow a$$

$$B \rightarrow bb$$

Final grammar

$$S \rightarrow aA \mid aB$$

$$A \rightarrow a$$

$$B \rightarrow bb$$

Useless Productions

$$S \to aSb$$

$$S \to \in$$

$$S \to A$$

$$A \to aA$$ Useless Production

Some derivations never terminate...

$$S \Rightarrow A \Rightarrow aA \Rightarrow aaA \Rightarrow \ldots \Rightarrow aa\ldots aA \Rightarrow \ldots$$

In general:

IF

$$S \Rightarrow \ldots \Rightarrow xAy \Rightarrow \ldots \Rightarrow w$$

contains only terminals

$$w \in L(G)$$

then variable A is useful

otherwise, variable A is useless

A production $A \to x$ is useless
if any of its variables is useless

Variables		Productions
	$S \to aSb$	
	$S \to \in$	
	$S \to A$	useless
useless	$A \to aA$	useless
useless	$B \to C$	useless
useless	$C \to D$	useless

Removing Useless Productions

Example Grammar:

$$S \to aS \mid A \mid C$$

$$A \to a$$

$$B \to aa$$

$$C \to aCb$$

First: find all variables that can produce strings with only terminals

$$S \to aS \mid A \mid C$$

$$A \to a$$

$$B \to aa$$

$$C \to aCb$$

Round 1: $\{A, B\}$

$$S \to A$$

Round 2: $\{A, B, S\}$

Keep only the variables
that produce terminal symbols: $\{A, B, S\}$

(the rest variables are useless)

$$S \rightarrow aS \mid A \mid \cancel{C}$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

$$\cancel{C \rightarrow aCb}$$

$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

Remove useless productions

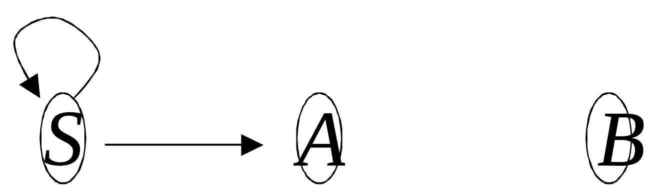
Second: Find all variables reachable from S

Use a Dependency Graph

$$S \to aS \mid A$$

$$A \to a$$

$$B \to aa$$



not reachable

Keep only the variables reachable from S

(the rest variables are useless)

$$S \to aS \mid A$$

$$A \to a$$

$$\cancel{B \to aa}$$

Final Grammar

$$S \to aS \mid A$$

$$A \to a$$

Q5) give example of remove useless productions

References

Elaine A. Rich (2008) Automata, Computability, and Complexity: Theory and Applications, Pearson Prentice Hall.

T. A. Sudkamp, *Languages and machines: an introduction to the theory of computer science*. Reading, MA: Addison Wesley, 1994.

Thank You