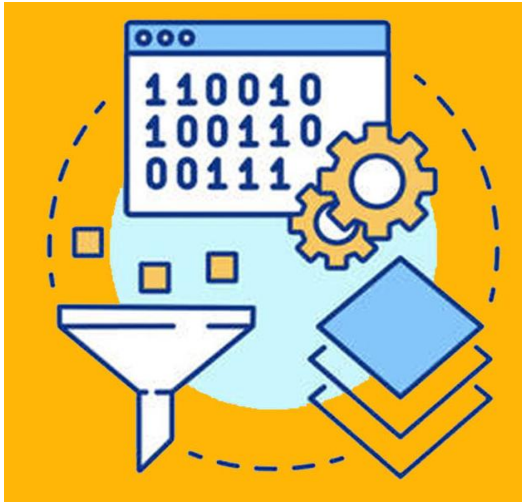
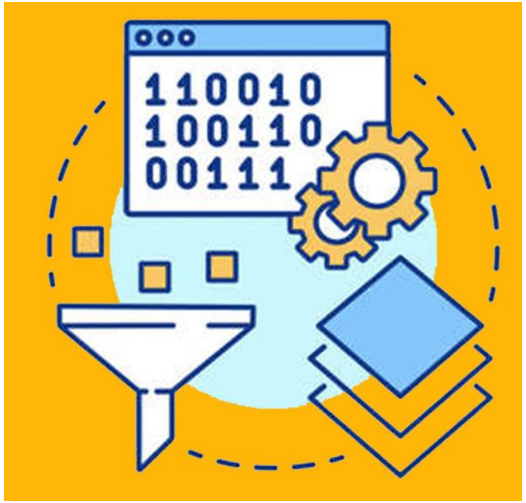
Compiler Design



Syntax Analysis



Prof. S. D. PADIYA| SSGMCE, Shegaon
padiyasagar@gmail.com

2021-2022 (Spring)

Syllabus: Unit 01

Syntax Analysis:

The role of the parser,

Review of context-free grammar for syntax analysis.

Top-down Parsing:

Recursive descent parsing,

Predictive parsers,

Transition diagrams for predictive parsers,

Non-recursive predictive parsing,

FIRST and FOLLOW,

Construction of predictive parsing tables, LL (1) grammars.

Non-recursive predictive parsing, Error recovery in predictive parsing.

THE ROLE OF THE PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

A parser is a compiler that is used to break the data into smaller elements coming from the lexical analysis phase.

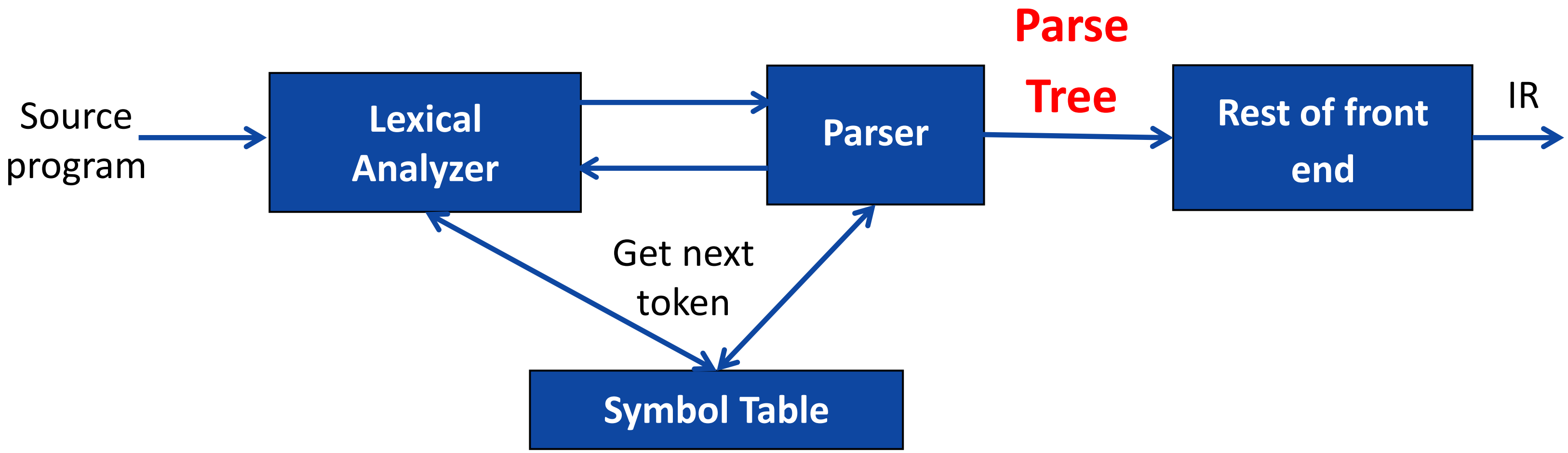
A parser takes input in the form of a sequence of tokens and produces output in the form of a parse tree.

THE ROLE OF THE PARSER

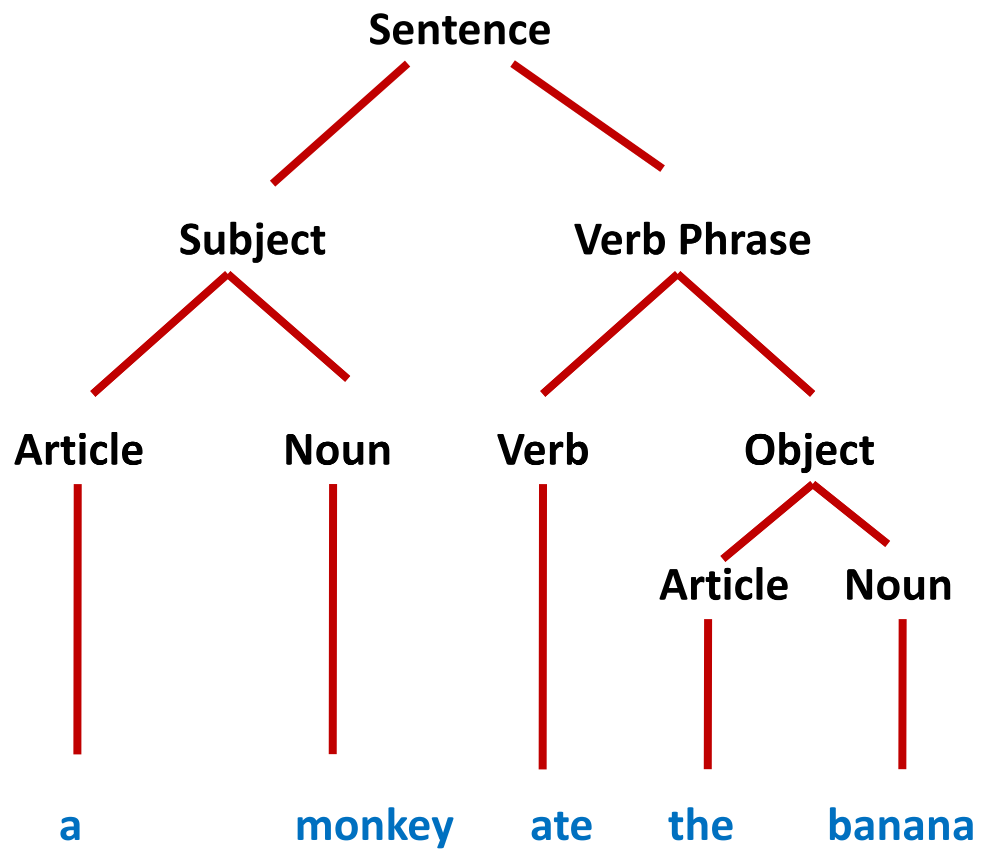
Parser obtains a string of tokens from the lexical analyzer and reports **syntax error** if any otherwise generates **syntax tree**.

There are two types of parser:

1. Top-down parser
2. Bottom-up parser



CONTEXT-FREE GRAMMAR



CONTEXT-FREE GRAMMAR

A grammar is defined by a four-tuple: $G = <V_N, V_T, S, \varnothing>$

Where **V_N is a set of non-terminals**

V_T is a set of terminals

S is starting symbol

$\varnothing$ is finite subset

- Nonterminal Symbols:
  - The name of the syntax category of a language, e.g., noun, verb, etc.
  - The It is written as a **single capital letter**, or as a **name enclosed between < … >,** e.g., A or <Noun>

<Noun Phrase> → <Article><Noun>
<Article> → a | the
<Noun> → monkey | banana

CONTEXT-FREE GRAMMAR

A grammar is defined by a four-tuple: $G = <V_N, V_T, S, \varnothing>$

Where V_N is a set of non-terminals

V_T is a set of terminals

S is starting symbol

$\varnothing$ is finite subset

- Terminal Symbols:
  - A symbol in the alphabet.
  - It is denoted by lower case letters and punctuation marks used in language.

<Noun Phrase> → <Article><Noun>

<Article> → a | the

<Noun> → monkey | banana

CONTEXT-FREE GRAMMAR

A grammar is defined by a four-tuple: $G = \langle V_N, V_T, S, \varnothing \rangle$

Where V_N is a set of non-terminals

V_T is a set of terminals

S is starting symbol

$\varnothing$ is finite subset

- Starting Symbol:
  - First nonterminal symbol of grammar is called the start symbol.

<Noun Phrase> → <Article><Noun>

<Article> → a | the

<Noun> → monkey | banana

CONTEXT-FREE GRAMMAR

A grammar is defined by a four-tuple: $G = \langle V_N, V_T, S, \varnothing \rangle$

Where V_N is a set of non-terminals

V_T is a set of terminals

S is starting symbol

$\varnothing$ is finite subset

- Production:

➥ A production, also called a rewriting rule, is a rule of grammar. It has the form of

A nonterminal symbol → String of terminal and nonterminal symbols

<Noun Phrase> → <Article><Noun>

<Article> → a | the

<Noun> → monkey | banana

CONTEXT-FREE GRAMMAR

Problem: Let G_2 = <{E, T, F}, {a, +, *, (,)}, S, ∅ >

Where ∅ consists of the productions

S → E + T

E → T

T → T * F

T → F

F → (E)

F → a

Obtain the expression **a * a + a**

Terminals, non terminals, start symbol, and productions for given grammar are:

Terminals:	**+, *, (,)**
Non terminals:	**S, E, T,**
Start symbol:	**S**
Productions:	**S → E + T**
	E → T
	T → T * F
	T → F
	F → (E)
	F → a

SDPadiya
SSGMCE

CONTEXT-FREE GRAMMAR

Problem: Let G_2 = <{E, T, F}, {a, +, *, (,)}, S, ∅ >

Where ∅ consists of the productions

S → E + T

E → T

T → T * F

T → F

F → (E)

F → a

Obtain the expression **a * a + a**

Solution: Therefore the generation or derivation of a sentence is:

S → E + T

S → **T** + T

S → **T * F** + T

S → **F** * F + T

S → **a** * F + T

S → a * **a** + T

S → a * a + **F**

S → a * a + **a**

SDPadiya
SSGMCE

DERIVATION & AMBIGUITY

DERIVATION

Derivation is used to find whether the string belongs to a given grammar or not.

Types of derivations are:

1. Leftmost derivation
2. Rightmost derivation

DERIVATION & AMBIGUITY

LEFTMOST DERIVATION

A derivation of a string W in a grammar G is a left most derivation if at every step the left most non terminal is replaced.

Grammar: S → S+S | S-S | S*S | S/S | a

Output string: a*a-a

S → **S** - S

S → **S** * S – S

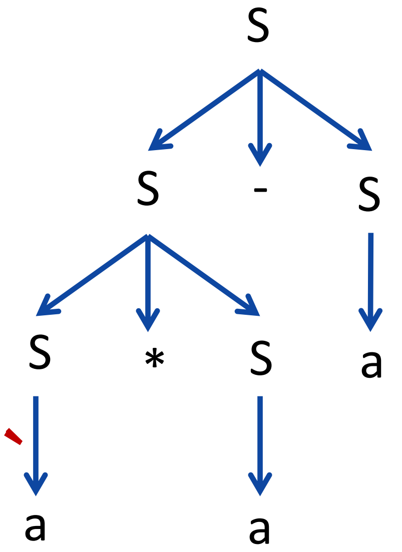
S → a * **S** – S

S → a * a – **S**

S → a * a – a

Leftmost Derivation

Parse tree represents the structure of derivation



Parse tree

DERIVATION & AMBIGUITY

RIGHTMOST DERIVATION

A derivation of a string W in a grammar G is a right most derivation if at every step the right most non terminal is replaced. It is also called canonical derivation.

Grammar: S → S+S | S-S | S*S | S/S | a

Output string: a*a-a

$S \rightarrow S * S$

$S \rightarrow S * S - S$

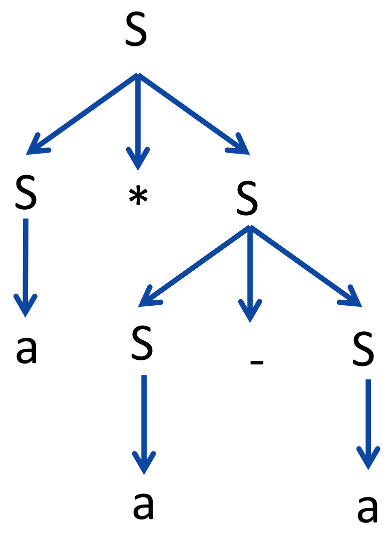
$S \rightarrow S * S - a$

$S \rightarrow S * a - a$

$S \rightarrow a * a - a$

Rightmost Derivation

Parse tree represents the structure of derivation



Parse tree

DERIVATION & AMBIGUITY

1. Perform leftmost derivation and draw parse tree.

$S \rightarrow A1B$

$A \rightarrow 0A \mid \epsilon$

$B \rightarrow 0B \mid 1B \mid \epsilon$

Output string: 1001

2. Perform leftmost derivation and draw parse tree.

$S \rightarrow 0S1 \mid 01$

Output string: 000111

3. Perform rightmost derivation and draw parse tree.

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

Output string: id + id * id

DERIVATION & AMBIGUITY

AMBIGUITY

Ambiguity, is a word, phrase, or statement which contains more than one meaning.

Chip

A long thin piece of potato

A small piece of silicon

SDPadiya
SSGMCE

DERIVATION & AMBIGUITY

AMBIGUITY

In formal language grammar, ambiguity would arise if identical string can occur on the RHS of two or more productions.

Grammar:

$$N1 \rightarrow \alpha$$

$$N2 \rightarrow \alpha$$

SDPadiya
SSGMCE

DERIVATION & AMBIGUITY

AMBIGUITY

Ambiguous grammar is one that produces more than one leftmost or more then one rightmost derivation for the same sentence.

Grammar: S→ S+S | S*S | (S) | a

S

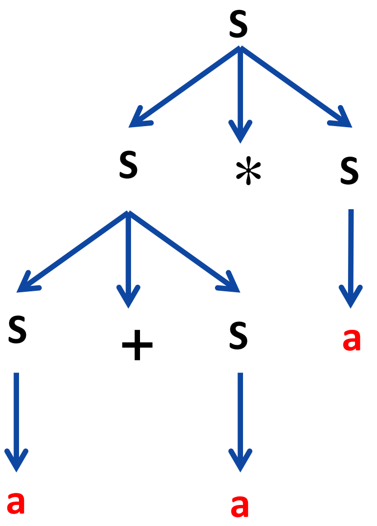
→S*S

→S+S*S

→a+S*S

→a+a*S

→a+a*a



Output string: a+a*a

S

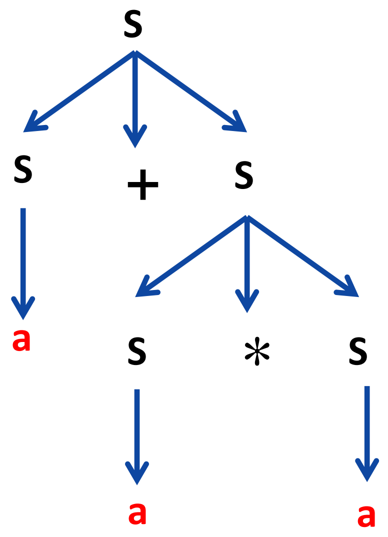
→S+S

→a+S

→a+S*S

→a+a*S

→a+a*a



Here, Two leftmost derivation for a+a*a is possible hence, above grammar is ambiguous.

SDPadiya
SSGMCE

COMPILER DESIGN

DERIVATION & AMBIGUITY

AMBIGUITY

Check Ambiguity in following grammars:

1) S→ aS | Sa | ϵ (output string: aaaa)

2) S→ aSbS | bSaS | ϵ (output string: abab)

3) S→ SS+ | SS* | a (output string: aa+a*)

4) <exp> → <exp> + <term> | <term>

<term> → <term> * <letter> | <letter>

<letter> → a|b|c|…|z (output string: a+b*c)

5) Prove that the CFG with productions: S → a | Sa | bSS | SSb | SbS is ambiguous

(output string: aaaabb)

SDPadiya
SSGMCE

LEFT RECURSION

- A grammar is said to be left recursive if it has a non terminal A such that there is a derivation $A \rightarrow A\alpha$ for some string α.

$$
\begin{aligned}
A &\rightarrow A\alpha \\
&\rightarrow A\alpha\alpha \\
&\rightarrow A\alpha\alpha\alpha \\
&\rightarrow A\alpha\alpha\alpha\alpha \\
&\rightarrow A\alpha\alpha\alpha\alpha\alpha \\
&\rightarrow A\alpha\alpha\alpha\alpha\alpha\alpha
\end{aligned}
$$

LEFT RECURSION

Algorithm to eliminate left recursion

1. Arrange the non terminals in some order $A_1, \dots, An$
2. for $i := 1\ to\ n$ **do begin**

    for $j := 1\ to\ i - 1$ **do begin**

        replace each production of the form $A_i \rightarrow Ai\gamma$

        by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$,

        where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all the current A_j

    productions;

    end

    eliminate the immediate left recursion among the A_i – productions

end

$$A \rightarrow A\alpha \mid \beta \quad \Longrightarrow \quad \begin{aligned} A &\rightarrow A' \\ A' &\rightarrow A' \mid \epsilon \end{aligned}$$

SDPadiya
SSGMCE

C O M P I L E R D E S I G N

LEFT RECURSION

$E \rightarrow E+T \mid T$

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow T*F \mid F$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$X \rightarrow X\%Y \mid Z$

$X \rightarrow ZX'$

$X' \rightarrow \%YX' \mid \varepsilon$

SDPadiya
SSGMCE

LEFT RECURSION

Exercise:

1) A→ Abd | Aa | a

   B→ Be | b

2) A→ AB | AC | a | b

3) S→ A | B

   A→ ABC | Acd | a | aa

   B→ Bee | b

4) Exp→ Exp+term | Exp-term | term

LEFT FACTORING

$$A \rightarrow \alpha\beta \mid \alpha\delta \longrightarrow \begin{array}{l} A \rightarrow A' \\ A' \rightarrow \quad \mid \end{array}$$

Thank You

Prof. S. D. Padiya
padiyasagar@gmail.com
SSGMCE, Shegaon