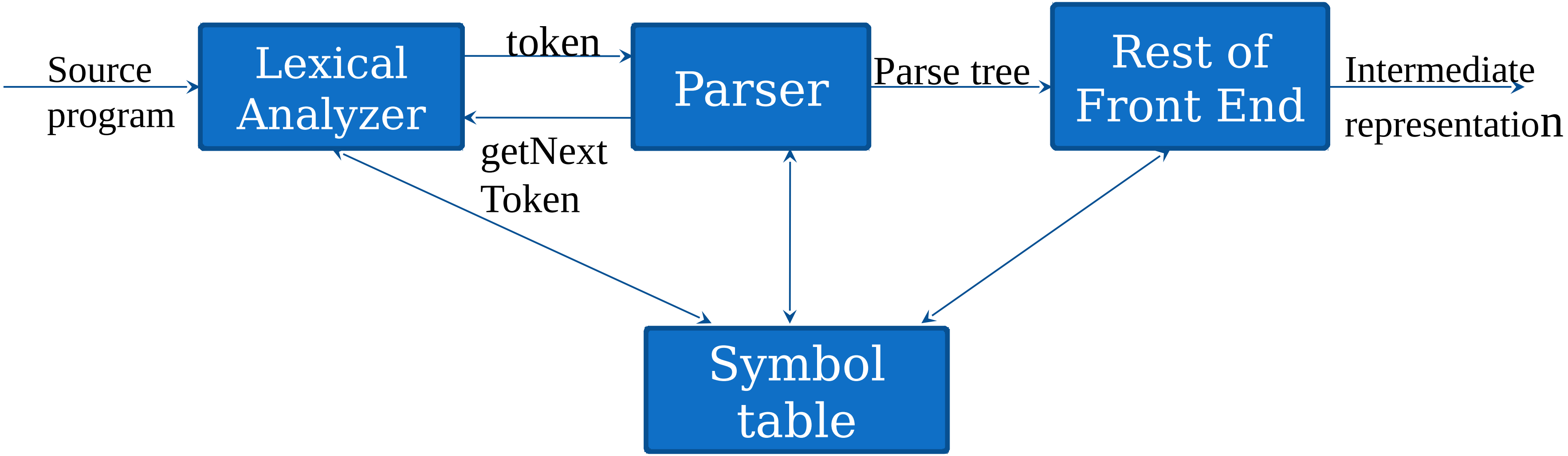
COMPILER DESIGN

UNIT 1

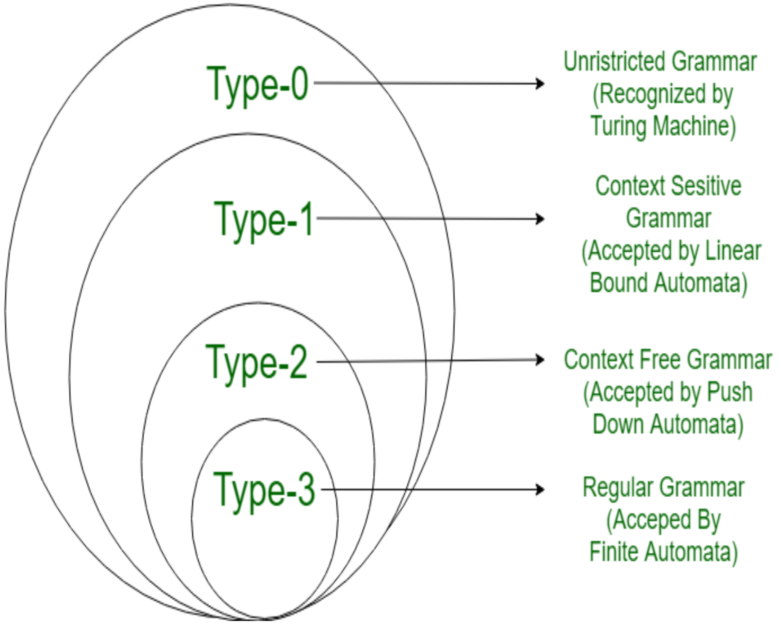
PART 4

The role of parser



According to [Chomsky hierarchy](), grammars are divided of 4 types:

- Type 0 - unrestricted grammar.
- Type 1 - context sensitive grammar.
- Type 2 - context free grammar.
- Type 3 - Regular Grammar.



Type 0: Unrestricted Grammar

Type-0 grammars include all formal grammars.

Type 0 grammar language are recognized by turing machine.

These languages are also known as the Recursively Enumerable languages

For example,

1) Sab –> ba

   A –> S.

2) S → ACaB

   Bc → acB

   CB → DB

   aD → Db

Grammar Production in the form of

$\boldsymbol{\alpha} \longrightarrow \boldsymbol{\beta}$

Where

$\boldsymbol{\alpha}$ is (V + T)* V (V + T)*

V : Variables(Non terminal)

T : Terminals

(α is a string of terminals and nonterminals with at least one non-terminal)

$\boldsymbol{\beta}$ is (V + T)*

(β is a string of terminals and non-terminals)

In type 0 there must be at least one variable on Left side of production.

Type - 1 Grammar

Type-1 grammars generate the context-sensitive languages. The language generated by the grammar are recognized by the [Linear Bound Automata]

For Example,

1) S –> AB

   AB –> abc

   B –> b

2) AB $\rightarrow$ AbBc

   A $\rightarrow$ bcA

   B $\rightarrow$ b

In Type 1

I. First of all Type 1 grammar should be Type 0.

II. Grammar Production in the form of

$\alpha \rightarrow \beta$

$|\alpha| <= |\beta|$

i.e count of symbol in α is less than or equal to β

Type - 2 Grammar

Type-2 grammars generate the context-free languages. The language generated by the grammar is recognized by a Pushdown automata.

For example,
S –> AB
A –> a
B –> bB
B -> c

Type-2 grammars generate the context-free languages.
In Type 2,
1. First of all it should be Type 1.
2. Left hand side of production can have only one variable.

$|\alpha| = 1.$

There is no restriction on β
.

Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form

$X \rightarrow a$ or $X \rightarrow aY$

where $X, Y \in N$ (Non terminal)

and $a \in T$ (Terminal)

The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule.

Example

$X \rightarrow \varepsilon$
$X \rightarrow a \mid aY$
$Y \rightarrow b$

A-> aB
A-> ε

WHAT ARE CONTEXT FREE GRAMMARS?

- “Context Free Grammar are notation for specifying concrete syntax”

- The languages generated by context free grammars are known as the context free languages

- Inherently recursive structure of a programming language are defined by a context free grammar.

FORMAL DEFINITION OF CFG

CFG stands for context-free grammar. It is is a formal grammar which is used to generate all possible patterns of strings in a given formal language. Context-free grammar G can be defined by four tuples as:

G = (V, T, P, S)

Where,

G is the grammar, which consists of a set of the production rule. It is used to generate the string of a language.

T is the final set of a terminal symbol. It is denoted by lower case letters.

V is the final set of a non-terminal symbol. It is denoted by capital letters.

P is a set of production rules, which is used for replacing non-terminals symbols(on the left side of the production) in a string with other terminal or non-terminal symbols(on the right side of the production).

S is the start symbol which is used to derive the string. We can derive the string by repeatedly replacing a non-terminal by the right-hand side of the production until all non-terminal have been replaced by terminal symbols.

EXAMPLE

Construct a CFG for a language L = {wcwR | where w € (a, b)*}.

Solution:

The string that can be generated for a given language is

L {c, aca, aacaa, bcb, abcba, bacab, abbcbba, ...}

The grammar could be:

1. S → aSa rule 1
2. S → bSb rule 2
3. S → c rule 3

Now if we want to derive a string "abbcbba", we can start with start symbols.

1. S → aSa
2. S → abSba from rule 2
3. S → abbSbba from rule 2
4. S → abbcbba from rule 3

Thus any of this kind of string can be derived from the given production rules.

EXAMPLE

Construct a CFG for the language L = $a^n b^{2n}$ where n>=1.

Solution:

The string that can be generated for a given language is {abb, aabbbb, aaabbbbbb....}.

The grammar could be:

1. S → aSbb | abb

Now if we want to derive a string "aabbbb", we can start with start symbols.

1. S → aSbb
2. S → aabbbb

S → 0S1

S → 01

Construct the CFG for the language having any number of a's over the set ∑= {a}.

L= {a,aa,aaa,aa….}

S-> aS

S->epsilon

WHAT IS BNF?

- It stands for Backus-Naur Form
- It is a formal, mathematical way to specify context-free grammars
-

```
<number> ::= <digit> | <number> <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

“::=”means “is defined as” (somevariants use “:=”instead)

“|” means “or”

- Angle brackets mean a Non Terminal
- Symbols without angle brackets are Terminals

Example 123=> 12 3=> 1 2 3

BNF for Expressions

<expression> ::= <expression> + <term>
| <expression> - <term>
| <term>
<term> ::= <term> *<factor>
| <term> /<factor>
| <factor>
<factor> ::= (<expression>)
| <variable>
| <number>

1+2*3
a+b*c
1+2+3
1*2*3
1*(2+3*5)

E-> E+T |E-T| T
T-> T*F | T/F| F
F-> (E) | V| D
V->a |b| z
D-> 0|1 9

There is the production for any grammar as follows:
S → aSa
S → bSb
S → c

In BNF, we can represent above grammar as follows:

S → aSa| bSb| c

1*2*3

EBNF is a few simple extensions to BNF which make expressing grammars more convenient

- “*****” The Kleene Star): means 0 or more occurrences
- “**+**” (The Kleene Cross): means 1 or more occurrences
- “[…]”:means 0 or 1 occurrences
- ()Use of parentheses for grouping
- {}* used to show arbitary sequence
- If you have a rule such as:
-

```
<real no> ::= <integer part>.<fraction part> | .<fraction part>
```

You can replace it with:

```
<real no> ::= [<integer part>].<fraction part>
```

- If you have a rule such as:

```
<signed integer> ::= + <integer> | - <integer>
<integer> ::= <digit>
              | <integer> <digit>
```

You can replace it with:

```
<signed integer> ::=[+|-]<digit>{ <digit>}*

<signed integer> ::=[+|-]{ <digit>}+
```

- If you have a rule such as:

```
<exp> ::= <term>
          | <exp> + <term>
          | <exp> - <term>
```

You can replace it with:

```
<exp> ::= <term>[(+|-)<term>]
```

PARSE TREE

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It is the graphical representation of symbol that can be terminals or non-terminals.
- Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

The parse tree follows these points

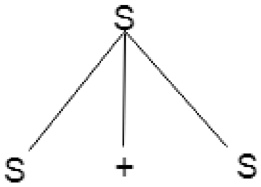
- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

Pragya Gaur

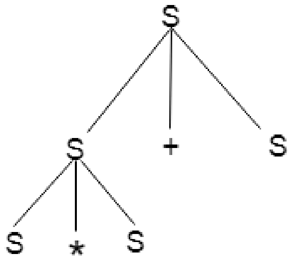
Production rules:

S→S + S | S * S
S→ a| b| c
w= a * b + c

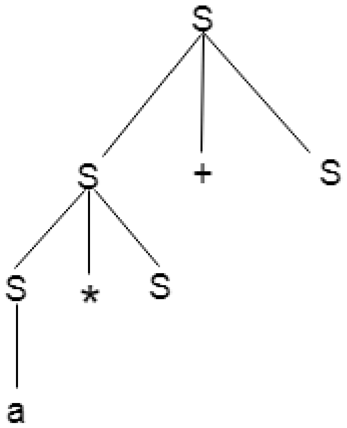
Step 1:



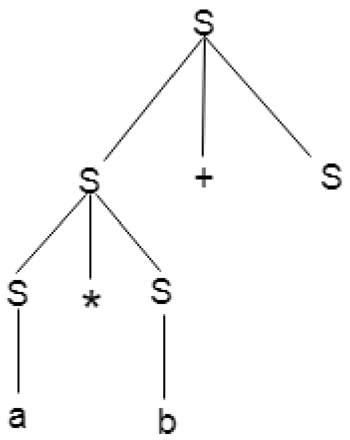
Step 2:



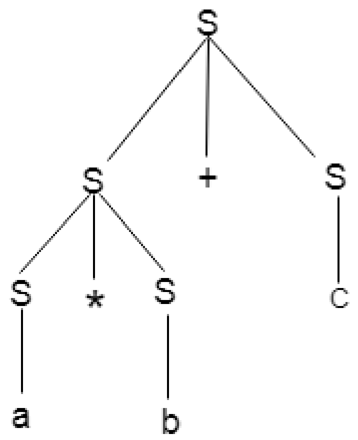
Step 3:



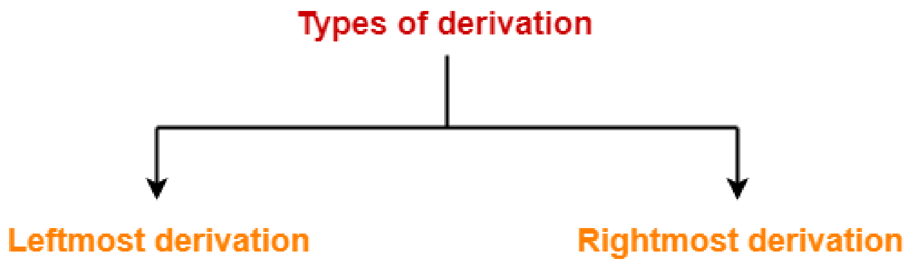
Step 4:



Step 5:



LEFTMOST, RIGHTMOST DERIVATIONS



- A **left-most derivation** of a sentential form is one in which rules transforming the left-most nonterminal are always applied
- A **right-most derivation** of a sentential form is one in which rules transforming the right-most nonterminal are always applied

Consider the following grammar-

S → aB / bA

S → aS / bAA / a

B → bS / aBB / b

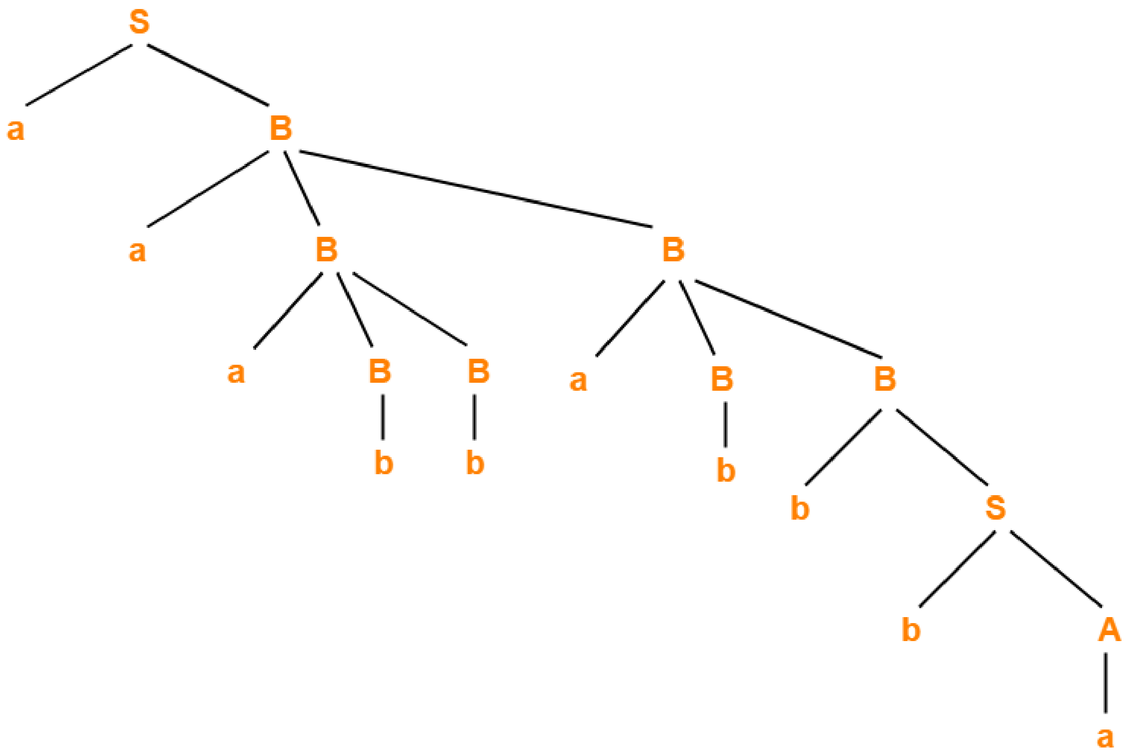
(Unambiguous Grammar)

Let us consider a string w = aaabbabbba

Now, let us derive the string w using leftmost derivation.

Leftmost Derivation-

S	→ aB → aaBB	(Using B → aBB)
	→ aaaBBB	(Using B → aBB)
	→ aaabBB	(Using B → b)
	→ aaabbB	(Using B → b)
	→ aaabbaBB	(Using B → aBB)
	→ aaabbabB	(Using B → b)
	→ aaabbabbS	(Using B → bS)
	→ aaabbabbbA	(Using S → bA)
	→ aaabbabbba	(Using A → a)



Leftmost Derivation Tree

Consider the following grammar-

S → aB / bA

S → aS / bAA / a

B → bS / aBB / b

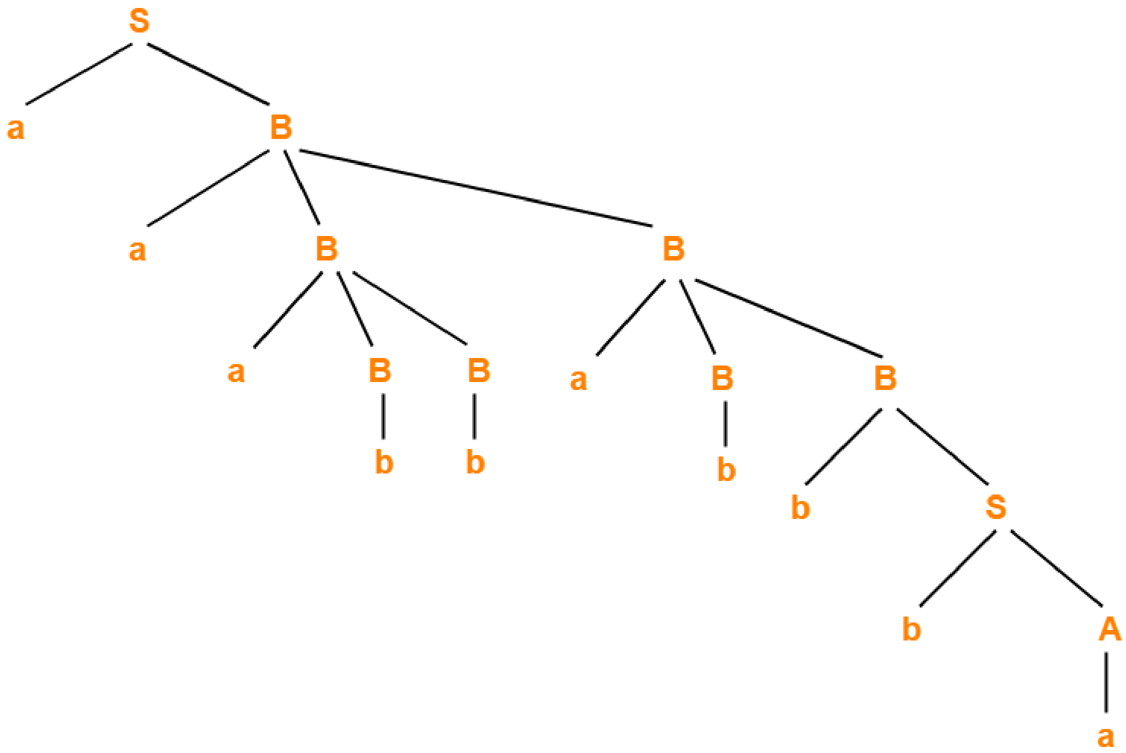
(Unambiguous Grammar)

Let us consider a string w = aaabbabbba

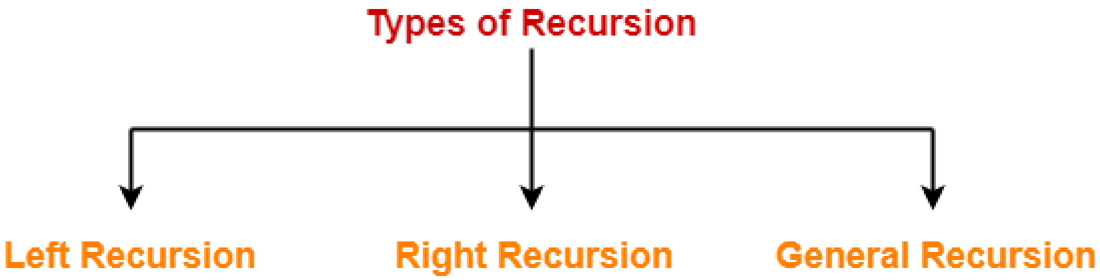
Now, let us derive the string w using rightmost derivation.

Rightmost Derivation-

S	→ aB	
	→ aaBB	(Using B → aBB)
	→ aaBaBB	(Using B → aBB)
	→ aaBaBbS	(Using B → bS)
	→ aaBaBbbA	(Using S → bA)
	→ aaBaBbba	(Using A → a)
	→ aaBabbba	(Using B → b)
	→ aaaBBabbba	(Using B → aBB)
	→ aaaBbabbba	(Using B → b)
	→ aaabbabbba	(Using B → b)



Rightmost Derivation Tree



1. Left Recursion
2. Right Recursion
3. General Recursion

General Recursion-

- The recursion which is neither left recursion nor right recursion is called as general recursion.

Example-

$$S \rightarrow aSb \mid \in$$

Right Recursion-

- A production of grammar is said to have **right recursion** if the rightmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having right recursion is called as Right Recursive Grammar.

Example-

$$S \rightarrow aS \ / \in$$

(**Right Recursive Grammar**)

Left Recursion-

- A production of grammar is said to have **left recursion** if the leftmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having left recursion is called as Left Recursive Grammar.

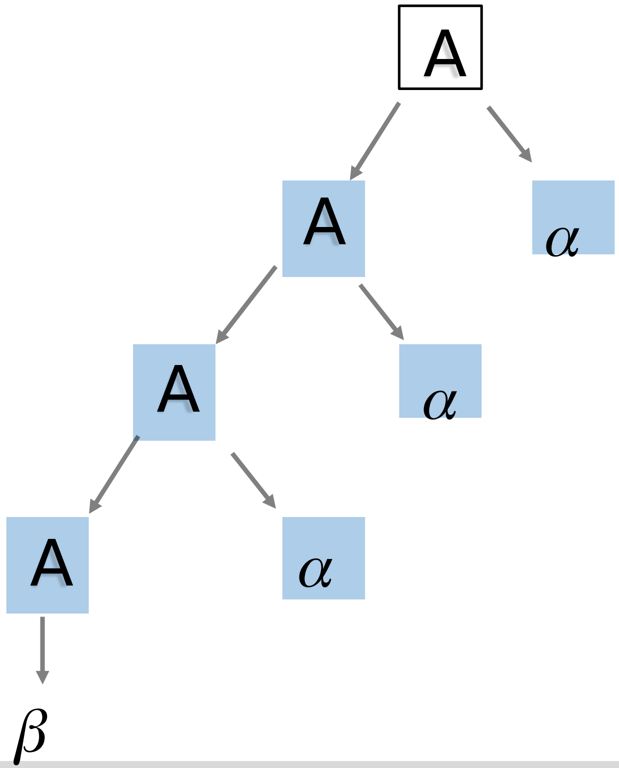
Example-

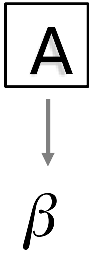
$$S \rightarrow Sa \;/\; \in$$

(**Left Recursive Grammar**)

LEFT RECURSION

$A \to A\alpha / \beta$





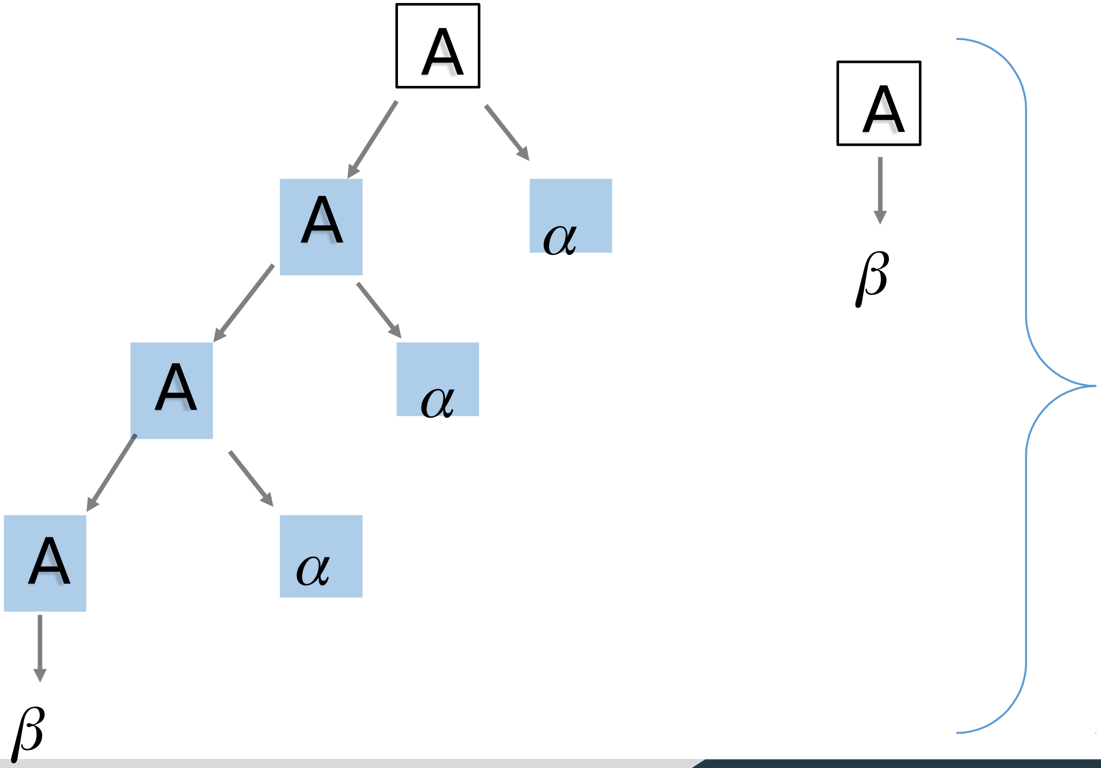
Top down parser doesn’t allow in Left Recursion bcz of it’s infinite problem.

So… we need to eliminate this problem.

Infinite problem

ELIMINATION OF LEFT RECURSION

$A \rightarrow A\alpha / \beta$



$\beta\ \alpha^*$

Pragya Gaur

ELIMINATION OF LEFT RECURSION

$$A \rightarrow A\alpha | \beta$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' | \epsilon$$

ELIMINATION OF LEFT RECURSION

$$A \rightarrow \beta\, \alpha^*$$

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' | \epsilon$$

$$A \rightarrow A\alpha 1 \mid A\ \alpha 2 | \beta 1 \mid \beta 2$$

$$A \rightarrow \beta 1 A' \mid \beta 2 A'$$

$$A' \rightarrow \alpha 1 A' | \alpha 2 A' | \epsilon$$

EXAMPLE : IMMEDIATE LEFT RECURSION

$$\underbrace{E}_{A} \rightarrow \underbrace{E}_{A}\ \underbrace{+T}_{\alpha}\ |\ \underbrace{T}_{\beta}$$

$$E \rightarrow TE'$$

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

$$A \rightarrow A\alpha \mid \beta$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

Pragya Gaur

EXAMPLE

$\underbrace{T}_{A} \rightarrow \underbrace{T}_{A}\underbrace{* F}_{\alpha} \mid \underbrace{F}_{\beta}$

$T \rightarrow FT'$

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

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

Pragya Gaur

A → Abc | A B | d | fB

A $A\alpha1$ $A\alpha2$ $\beta1$ $\beta2$

B → g

UPDATED GRAMMAR

A → d A' | f BA'

A' → bc A' | BA' |∈

B → g

$A \rightarrow A\alpha1|\alpha2|\beta1|\beta2$

$A \rightarrow \beta1A' \mid \beta2A'$

$A' \rightarrow \alpha1A' | \alpha2A' | \in$

$A \rightarrow Abc \mid A B \quad \mid d$

$A \quad A\alpha 1 \; A\alpha 2 \quad \beta 1$

$B \rightarrow g$

UPDATED GRAMMAR

$A \rightarrow d A'$

$A' \rightarrow bc A' \mid BA' \mid \in$

$B \rightarrow g$

$T \rightarrow T*F \mid F$

$T \rightarrow T * F | F$

$T \rightarrow F\ T'$

$T' \rightarrow *F\ T' | \in$

$S \rightarrow A$

$A \rightarrow Ad / Ae / aB / ac$

$B \rightarrow bBc / f$

$T \to T * F$

$T \to T / F$

$T \to a$

T →T*F | T/F | a

A A α1 A α 2

β

T → a T'

T' → *F T' | /FT' |

$\in$

$A \to A\alpha | \beta$

$A \to \beta A'$

$A' \to \alpha 1 A' | \alpha 2 A' | \in$

INDIRECT LEFT RECURSION

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

A $\qquad \alpha \qquad \beta$

A

Case 1: order of Non Terminal- S ,A
Case 2: order of Non Terminal- A, S

1. $E \rightarrow E + T \mid T$

   $T \rightarrow T \times F \mid F$

   $F \rightarrow id$

2. $S \rightarrow (L) \mid a$

   $L \rightarrow L , S \mid S$

1. $A \rightarrow Ba / Aa / c$

$B \rightarrow Bb / Ab / d$

1. $E \rightarrow TE'$

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

   $T \rightarrow FT'$

   $T' \rightarrow xFT' \mid \in$

   $F \rightarrow id$

EXAMPLE: INDIRECT LEFT RECURSION

CASE 1:

S →Aa | b

A →Ac | Sd| f

For S

- There is no immediate left recursion
- Don't enter in the inner loop

For A

- Replace A→Sd with S →Aa | b
  A→Ac | Aad| bd |f

$A \rightarrow A\alpha | \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' | \epsilon$

Pragya Gaur

EXAMPLE

Remove immediate left recursion from A

$$\underbrace{A}_{A} \rightarrow \underbrace{A}_{A}\ \underbrace{c}_{\alpha 1} \mid \underbrace{A}_{A}\ \underbrace{ad}_{\alpha 2} \mid \underbrace{bd}_{\beta 1} \mid \underbrace{f}_{\beta 2}$$

$$A \rightarrow bdA' \mid fA'$$

$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

$$A \rightarrow A\alpha 1 \mid A\alpha 2 \mid \beta$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha 1 A' \mid \alpha 1 A' \mid \epsilon$$

$$S \rightarrow Aa \mid b$$

$$A \rightarrow bdA' \mid fA'$$

$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

EXAMPLE: INDIRECT LEFT RECURSION

CASE 2:

S →Aa | b

$$\underbrace{A}_{A} \rightarrow \underbrace{A}_{A}\underbrace{c}_{\alpha} \mid \underbrace{Sd \mid f}_{\beta}$$

Remove immediate left recursion from A

$A \rightarrow SdA' \mid fA'$

$A' \rightarrow cA' \mid \epsilon$

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

EXAMPLE

For S : replace A by A→ SdA′ | fA′ in S →Aa | b

$$S \rightarrow Aa \mid b$$

$$S \rightarrow SdA'a \mid fA'a \mid b$$

$$\underbrace{S}_{A} \rightarrow \underbrace{SdA'}_{A}\underbrace{a}_{\alpha}\mid \underbrace{fA'a \mid b}_{\beta}$$

$$S \rightarrow fA'aS' \mid bS'$$

$$S' \rightarrow dA'aS' \mid \in$$

$$A \rightarrow A\alpha / \beta$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' / \in$$

$$A \rightarrow SdA' \mid fA'$$

$$A' \rightarrow cA' \mid \in$$

$$S \rightarrow fA'aS' \mid bS'$$

$$S' \rightarrow dA'aS' \mid \in$$

Pragya Gaur

LEFT FACTORING

- In left factoring it is not clear which of two alternative productions to use to expand a nonterminal A.

  i.e. if $A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$

- W e don't know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$
- To remove left factoring for this grammar replace all A productions containing α as prefix by

  $A \longrightarrow \alpha A'$

  $A' \longrightarrow \beta_1 \mid \beta_2$

Pragya Gaur

$S \rightarrow abc \mid ab$

```
A --> Br
B --> Cd

C --> At
```

YACC(yet another compiler compiler)

- YACC generates C code for syntax analyzer or parser.
- YACC uses grammar rules that allow it to analyze token from LEX and create syntax tree.
- YACc issues a warning message whenever a conflict occurs.
- YACC takes a default action when there is a conflict-

Conflict can be:

- shift reduce conflict
- Reduce reduce conflict

Structure of YACC

Definations

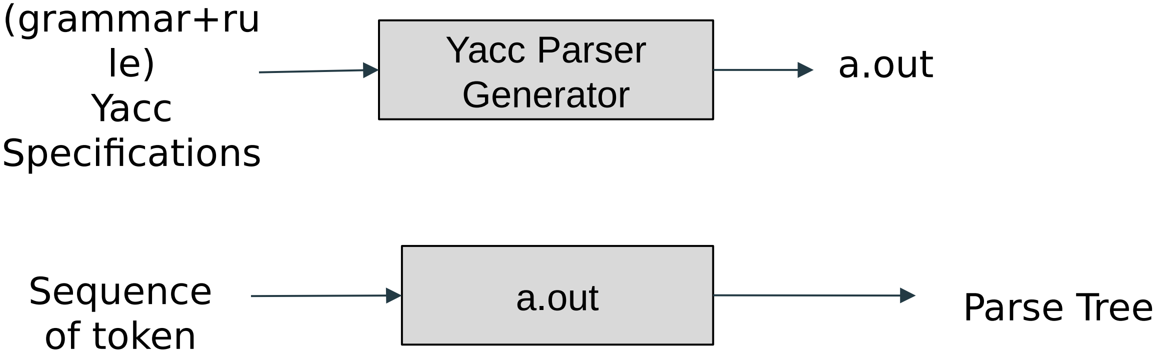
%%

Rules

%%

Subroutines (supporting C routines)

YACC(yet another compiler compiler)



YACC(yet another compiler compiler)