Unit - II

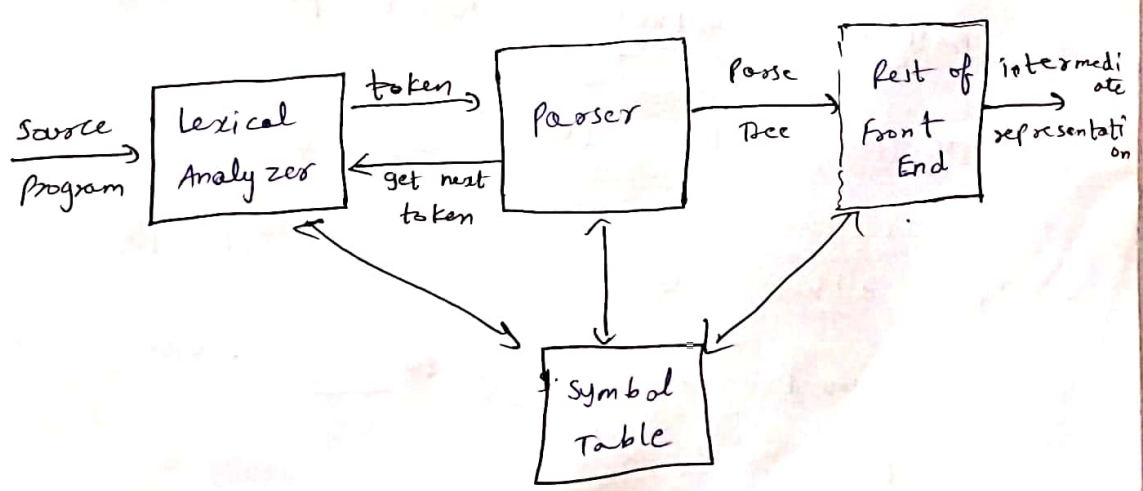
Syntax Analysis

1) Introduction

1.1 Role of Parser

→ The parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source Language.

→ Parser is expected to report any syntax errors in an intelligible fashion and to recover from commonly occuring errors to continue processing the remainder of the program.



→ There are three types of parsers for grammars: universal

Top-down

Bottom-up

→ universal parsing methods such as Cocke-Younger-Kasami algorithm and Earley's Algorithm can parse any grammar.

→ These are inefficient to use in production compilers.

→ The methods used in Compilers can be classified as either Top-down (or) Bottom-up

→ Top down methods build parse trees from the top(root) to the bottom (Leaves)

→ Bottom-up methods build parse trees from Leaves (Bottom) to the root (top)

→ In either case, input to the parser is scanned from left to right, one symbol at a time.

1.2 Representative Grami

→ Constructs that begin with keywords like while or int, are easy to parse, because the keyword guides the choice of the grammar production that must be applied to match the input.

→ Because of Associativity and precedence the expressions are difficult to parse.

Eg:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

→ The above expression grammar belongs (Left Recursive) to class of LR grammars that are suitable for Bottom up Parsing

→ This grammar can be adapted to handle additional operators and additional levels of precedence.

→ The above grammar is Left recursive

So it cannot be used for Top-down parsing.

Non Left Recursive grammar

$$E \longrightarrow T\,E'$$

$$E' \longrightarrow +\,T\,E' \mid \epsilon$$

$$T \longrightarrow F\,T'$$

$$T' \longrightarrow *\,F\,T' \mid \epsilon$$

$$F \longrightarrow (\,E) \mid id$$

→ Non left Recursive grammar can be used for Top-down parsing.

$$E \longrightarrow E + E \mid E * E \mid (E) \mid id$$

→ The above grammar treats * and + alike, so it is useful for handling ambiguities during parsing.

1.3 Syntax Error Handling

→ A compiler is expected to assist the programmer in locating and tracking

down errors.

→ Planning the Error Handling right from the start can both simplify the structure of compiler and improve its handling of errors

→ Common programming errors

(i) Lexical errors include mispelling of identifiers, keywords (or) operators.

(ii) Syntatic errors include misplaced semicolon (or) extra (or) missing braces ("{" or "}")

(iii) Semantic errors include type mismatches between operators and operands. (Eg: undeclared variable)

(iv) Logical errors can be anything from incorrect reasoning on the part of programmer to the use in a C program "=" instead of "==".

→ The precision of parsing methods allows syntatic errors to be detected very efficiently.

→ Parsing Methods such as LL and LR methods detect an error as soon as

possible, that is, when the stream of tokens from the lexical analyzer cannot be passed further according to grammar for the Language.

→ The error handling in a parser has goals

(i) Report the presence of errors clearly and accurately.

(ii) Recover from each error quickly enough to detect subsequent errors.

(iii) Add minimal overhead to the processing of correct programs.

1-4 Error-Recovery strategies

(i) Panic-Mode Recovery

→ On Discovering an error, the parser discards input symbols one at a time until one of designated set of synchronizing tokens is found.

→ Synchronizing Tokens are usually delimeters such as semicolon (;) or }, whose

to be in source program is clear and unambiguous.

→ While panic mode correction often skips a considerable amount of input without checking for additional errors.

→ Advantage (will not go into infinite loop)

(ii) Phrase-Level Recovery

→ On Discovering an error, a parser may perform local correction on remaining input, that is, it may replace a prefix of remaining input by some string that allows the parser to continue.

→ A typical local correction is to replace a comma, by a semicolon, delete a extra semicolon, or insert a missing semicolon.

→ It can be used in several error-repairing compilers, as it can correct any input string.

→ Drawback Difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

(iii) Error Productions

→ By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate erroneous constructs.

→ A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing.

(iv) Global Correction

→ The aim is to make as few changes possible in processing an incorrect input string.

→ There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction.

Eg:- Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such

that the no of insertions, deletions and changes of tokens required to transform x into y is as small as possible.

→ These methods are costly in terms of time and space

1.2 Context Free Grammars

→ Grammars are used to describe the syntax of programming language constructs like expressions and statements.

Eg:-

stmt → if (expr) stmt else stmt

The formal Definition of a Context-Free Grammar

→ CFG consists of Terminals, Non-Terminals, a start symbol and productions.

(i) Terminals are the basic symbols from which strings are formed. The term "Token name" is the synonym for "Terminal"

Eg:- if, else, "(", "y" are Terminals in the above grammar

(ii) NonTerminals are syntactic variables that denote set of strings.

Eg: In the above eg stmt & expr are non Terminals

→ The set of strings denoted by NonTerminals help define the Language generated by grammar.

(iii) In a grammar, one Non-Terminal is distinguished as start symbol, and set of strings it denotes is the Language generated by grammar.

(iv) The productions of a grammar specify the manner in which the terminals and Non-Terminals can be combined to form strings.

Each production consists of

(a) A NonTerminal called the head (or) Left side of the production; this production defines some of strings denoted by head.

(b) The symbol →

(c) A body or Right side consisting of zero or more Terminals and NonTerminals The components of the body describe one way in which strings of the Non-Terminal at the head can be constructed.

Grammar for simple arithematic Expressions

Eg:- expr → expr + term

expr → expr − term

expr → term

term → term * factor

term → term / factor

term → factor

factor → (expression)

factor → id

Terminals: id, +, −, *, /, (,)

Notational Conventions

(1) These symbols are Terminals:

(a) Lowercase letters (a, b, c)

(b) operator symbols (+, * ---)

(c) Punctuation symbols ("(", ";" ....)

(d) digits 0, 1, ... 9

(e) Bold face strings (if or id)

(2) These symbols are Non Terminals:

(a) Uppercase letters (A, B, C)

(b) The letter "S" (start symbol)

(c) Lower case italic names (expr or stmt)

(d) Uppercase symbols (F, T, E) can be used to represent non-terminals for constructs

(3) uppercase letters (X, Y, Z) represent Grammar symbols (either NT or T)

(4) Lowercase letters in alphabets (u, v, ... z) represents strings of Terminals

(5) Lowercase greek letters (α, β, γ) represent strings of grammar symbols. ($A \to \alpha$) A is head & α is Body

(6) A set of productions $A \to \alpha_1$; $A \to \alpha_2$ ..... $A \to \alpha_K$ with common head may be written as $A \to \alpha_1 | \alpha_2 | \cdots | \alpha_K$

(7) Unless stated otherwise, the head of first production is start symbol.

Eg:

$E \to E + T \mid E - T \mid T$

$T \to T * F \mid T / F \mid F$

$F \to (E) \mid id$

E - Start symbol

F & T - Non Terminals

id, +, -, *, /, (,) - Terminals

Derivations

→ The construction of a parse Tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules

→ Beginning with start symbol, each rewriting step

replaces a Non-Terminal by body of one its productions.

→ The Derivational view corresponds to the top-down construction of the Parse-Tree, but precision afforded by derivations will be especially helpful when bottom-up parsing is discussed

→ Bottom-up Parsing is related to class of derivations known as "Rightmost" derivations, in which rightmost NT is rewritten at each step.//

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

$E \Rightarrow -E$ (E derives -E)

$E \Rightarrow -E \rightarrow -(E) \Rightarrow -(id)$

→ we call such a sequence of replacements a derivation of $-(id)$ from E.

$\alpha \overset{*}{\Rightarrow} \alpha$ for any string α, and

If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$

$\overset{*}{\Rightarrow}$ (derives in zero or more derivations)

$\overset{+}{\Rightarrow}$ (derives in one or more steps)

If $S \overset{*}{\Rightarrow} \alpha$, where S is start symbol of grammar G,

α is Sentential form of G;

→ Sentential form may contain both terminals and nonTerminals and may be empty.

→ A sentence of G is a sentential form with no NonTerminals.

→ The Language generated by a Grammar is its set of sentences.

→ A string of terminals w is in L(G), the Language generated by G, if and only if w is a sentence of G (or $S \overset{*}{\Rightarrow} w$).

→ A Language generated by a CFG is called Context free Language.

→ If two grammars (CFG) generate same Language, the grammars are said to equal.

Eg: string $-(id+id)$ is sentence of G because there is a derivation;

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

$E \overset{*}{\Rightarrow} -(id+id)$

(or)

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(-E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

→ Each N.T is replaced by the same body in the two derivations, but the order of replacements is different.

(1) In Left most derivations, the Left most Non-Terminal in each sentential is always chosen.

If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced $\alpha \underset{lm}{\Rightarrow} \beta$.

(2) In Right most derivations, the rightmost Nonterminal is always choosen, $\alpha \underset{rm}{\Rightarrow} \beta$

→ Right most Derivations are also called Canonical derivations.

Parse Trees and Derivations

→ A parse Tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace Non Terminals.

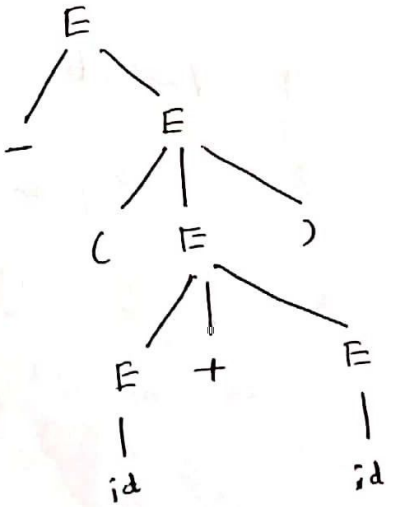
→ Each interior node of a parse Tree represents the application of a production.

→ The interior node is labeled with the nonterminal 'A' in the head of production.

→ The children of the node are labeled, from Left-to-right, by symbols in the body of production by which this 'A' was replaced during the derivation.

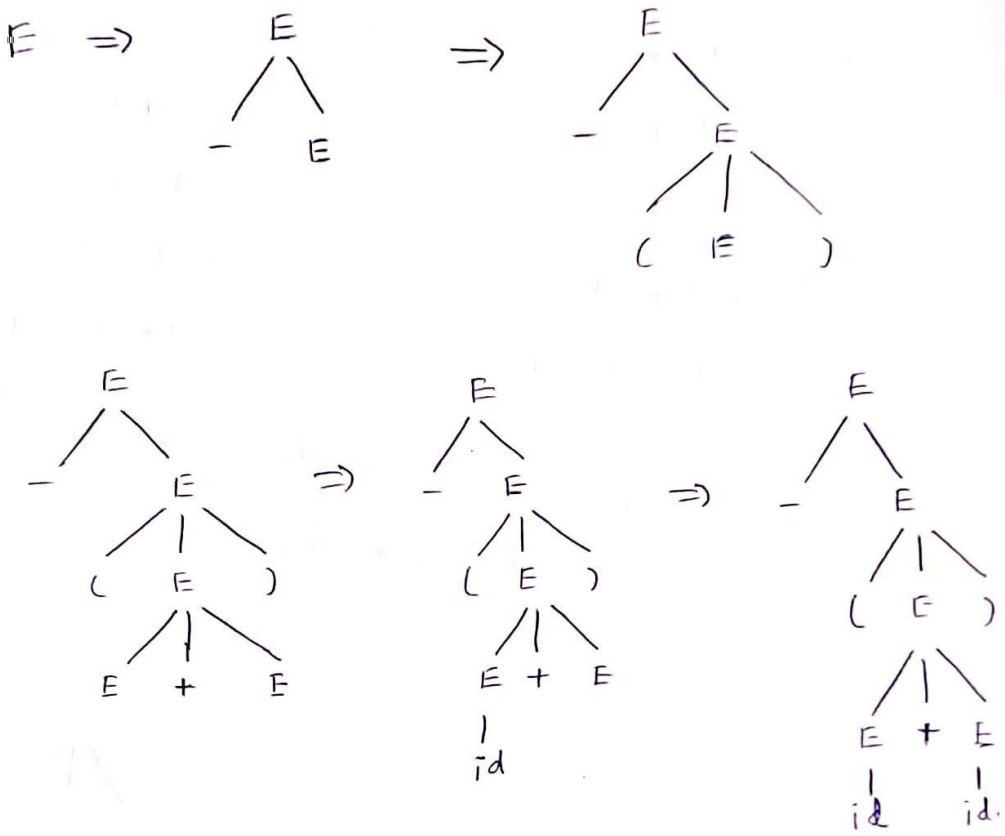
→ The leaves of a parse Tree are labeled by nonterminals (or) Terminals and read from left to right, constitute a sentential form, called yeild (or) Frontier of the Tree.

Eg:



Parse Tree for -(id+id).

Eg: sequence of parse trees for derivations.



Ambiguity

→ A grammar produces more than one parse Tree for some sentence is said to Ambiguous.

→ An Ambiguous Grammar is one that produces more than one leftmost derivation (or) more than one rightmost derivation for the same sentence.

Eg:- sentence $id + id * id$ has Two Left most derivations

$E \Rightarrow E + E$

$\Rightarrow id + E$

$\Rightarrow id + E * E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$

$E \Rightarrow E * E$

$\Rightarrow E + E * E$

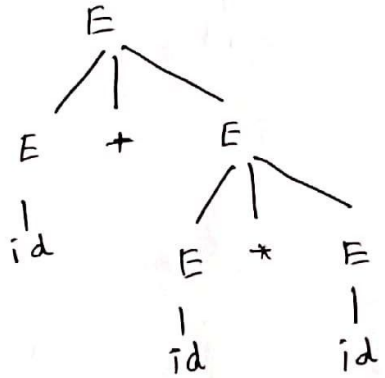
$\Rightarrow id + E * E$

$\Rightarrow id + id * E$

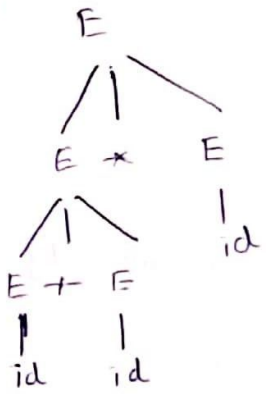
$\Rightarrow id + id * id$

→ For most parsers, it is desirable that the grammar be made unambiguous.

Eg: parse trees



(a)



(b)

Two parse Trees for $id + id * id$

Verifying Language generated by Grammar

→ A Grammar(G) generates a Language L has two parts:

(i) show that every string generated by 'G' is in 'L'

(ii) That every string in L can indeed be generated by G.

Eg: $S \rightarrow (S)\,S \mid \epsilon$

The above Grammar generates all strings of balanced parenthesis, and only such strings.

$$S \underset{lm}{\Longrightarrow} (S)\,S \overset{*}{\underset{lm}{\Longrightarrow}} (x)\,S \overset{*}{\underset{lm}{\Longrightarrow}} (x)\,y$$

→ $(x)\,y$ must be balanced.

Context-Free Grammars vs Regular Expressions

→ Grammars are more powerful notation than regular Expressions.

→ Every construct that can be described by a Regular Expression can be described by a Grammar but not vice versa.

Eg:- R.E = $(a|b)^* abb$ and Grammar

$A_0 \rightarrow a A_0 \mid b A_0 \mid a A_1$

$A_1 \rightarrow b A_2$

$A_2 \rightarrow b A_3$

$A_3 \rightarrow \epsilon$

→ The grammar was constructed from NFA

1) For each state i of NFA, create a nonterminal A_i

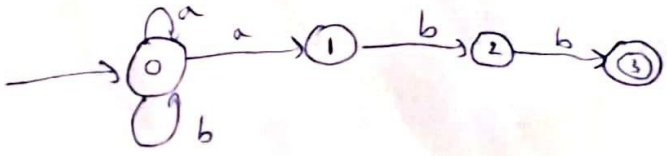
2) If state i has a transition to state j on input a, add the production $A_i \rightarrow a A_j$

If state i goes to state j on input ε, add the production $A_i \rightarrow A_j$

3) If i is an Accepting state, add $A_i \rightarrow \epsilon$

4) If i is the start state, make A_i be start symbol of grammar.

NFA for $(a|b)^* abb$



writing a Grammar

→ several Transformations that could be applied to get grammar more suitable for Parsing

→ Techniques are (i) Elimination of Ambiguity

(ii) Left-Recursion Elimination

(iii) Left-factoring.

→ The above techniques are useful for rewriting the grammars so they become suitable for top-down parsing.

Lexical vs Syntactic Analysis

(1) seperating the syntactic structure of a Language into lexical and Non-Lexical parts provide a convenient way of modularizing the front end of compiler into two manageable sized components.

(2) The Lexical Rules of a Language are quietly simple and to describe them we donot need a notation as powerful as grammars

(3) Regular Expressions generally provide a more concise and easier-to-understand notation for tokens than Grammars.

(4) More efficient Lexical Analyzers can be constructed automatically from the Regular Expressions than from Arbitary Grammars.

→ Regular Expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords and whitespaces

→ Grammars, are most useful for describing Nested structures such as balanced parenthesis, matching begin-end's, corresponding if then else and so on

Eliminating Ambiguity

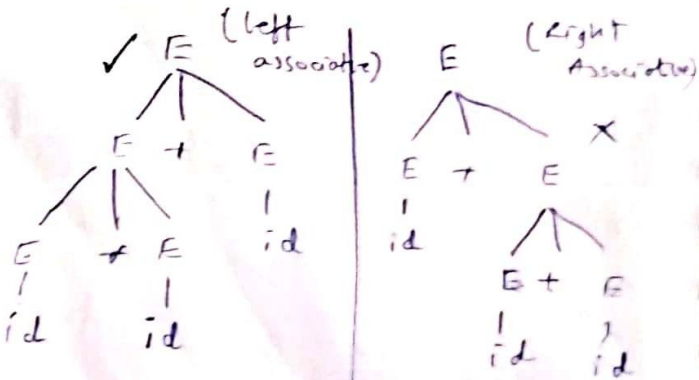
Eg:- $E \rightarrow E + E$

$E \rightarrow E * E$

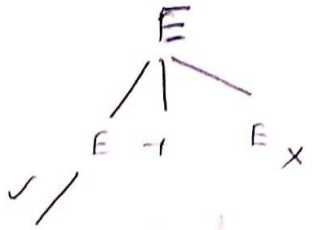
$E \rightarrow id$

Not following Associativity (Left or Right)

id + id + id



Solution Restrict to Left Associativity using Left Recursion



If grammar is → Left Recursion

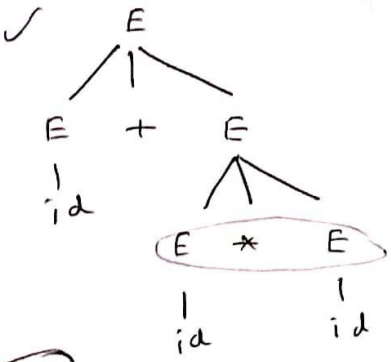
so Grammar will be without Ambiguity

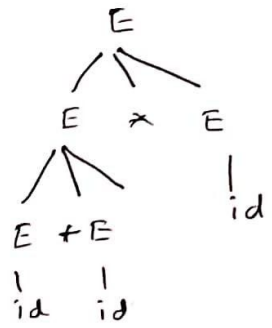
$E \rightarrow E + id \mid id$

than operator is Left Associativity.

Not Following (precedence)

id + id * id





precedence solution

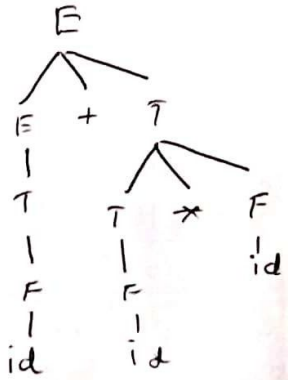
Solution use Levels (Higher levels '+' and '*' in Lower levels)

Final Grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$



*** * → Low level
+ → high level

→ ** Highest precedence should be at low level

Eg:

```
bExp → bExp or bExp
     | bExp and bExp
     | not bExp
     | True
     | True
```

Unambiguous grammar

```
E → E or F | F
F → F and G | G
G → not G | True | False
```

Ambiguous

Eg:

```
R → R + R
  | R R
  | R*
  | a
  | b
  | c
```

unambiguous

```
E → E + T | T
T → T F | F
F → F* | a | b | c
```

Eg:

```
A → A $ B | B
B → B # C | C
C → C @ D | D
D → d
```

$, #, @ → Left associative

@ -> # -> $ (Precedence)

Eg: E → E * F | F + E | F

F → F - F | id

* → left associative

+ → Right associative

- -> (* = +)

Left Recursion

→ A Grammar is Left Recursion if it has a Non-Terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.

Direct Left Recursion

$$A \rightarrow A a$$

Indirect Left Recursion

$$S \rightarrow A a \qquad S \overset{+}{\Rightarrow} S b$$
$$A \rightarrow S b$$

(Why?)

→ Top-down parsing methods cannot handle Left-Recursive grammars, so Transformation is needed to eliminate Left Recursion.

(How?)

→ $A \rightarrow A\alpha \mid \beta$ → Left Recursive Grammar

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

→ Non-Left Recursive productions

→ If there are Multiple A productions

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \cdots \mid \beta_n$$

where no β_i begins with A.

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$

Non Recursive Grammar

Advantage

→ We are able to generate the same Language even after removing Left Recursion.

Disadvantage

→ The above procedure only eliminates Direct Left Recursion (Immediate) but not indirect Left Recursion (Not Immediate)

Eg:- $E \rightarrow E + T \mid T$ (L.R) → $E \rightarrow E\underbrace{+T}_{\alpha} \mid \underbrace{T}_{\beta}$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$E \rightarrow TE'$

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

$T \rightarrow T\underbrace{*F}_{\alpha} \mid \underbrace{F}_{\beta}$ (L.R)

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$ (NO L.R)

Final Grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Eg:- $S \rightarrow S\underbrace{0S1S}_{\alpha} \mid \underbrace{01}_{\beta}$ (L·R)

Not L·R

$S \rightarrow 01S'$

$S' \rightarrow 0S1SS' \mid \epsilon$

Eg:- $L \rightarrow L\underbrace{,S}_{\alpha} \mid \underbrace{S}_{\beta}$ (L·R)

Not L·R

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \epsilon$

Eg:- (L·R)

$S \rightarrow SX$

$S \rightarrow Ssb$

$S \rightarrow Xs$

$S \rightarrow a$

$S \rightarrow S\underbrace{X}_{\alpha_1} \mid S\underbrace{sb}_{\alpha_2} \mid \underbrace{Xs}_{\beta_1} \mid \underbrace{a}_{\beta_2}$

Not L·R

$S \rightarrow XSS' \mid aS'$

$S' \rightarrow XS' \mid SbS' \mid \epsilon$

Eg:- $A \rightarrow A\underbrace{A}_{\alpha_1} \mid \underbrace{Ab}_{\alpha_2}$ (L·R)

Not L·R

$A' \rightarrow AA' \mid bA' \mid \epsilon$

Left factoring (Eliminating Non-Determinism)

→ Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or Top-down parsing.

→ when the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of input has been seen that we can make right choice.

Grammar

$A \to \alpha\beta_1 | \alpha\beta_2 | \ldots\ldots \alpha\beta_n | \gamma$, where γ represents all alternatives that donot begin with α

$A \to \alpha A' | \gamma$

$A' \to \beta_1 | \beta_2 | \ldots\ldots | \beta_n$

Eg:-

$S \to iEtS$

$| iEtSeS$

$| a$

$E \to b$

After Left Factoring

$S \to iEtSS' | a$

$S' \to \epsilon | eS$

$E \to b$

Eg:-

→ Eliminating #Left factoring will not eliminate the Ambiguity

Eg:

$S \rightarrow aSSbS$
$\quad | aSaSb$
$\quad | aabb$
$\quad | b$

$S \rightarrow aS' | b$

$S' \rightarrow SSbS | SaSb | bb$

After left factoring,

$S \rightarrow aS' | b$

$S' \rightarrow SS'' | bb$

$S'' \rightarrow SbS | aSb$

Eg.

$S \rightarrow bSSaaS$
$\quad | bSSaSb$
$\quad | bSb$
$\quad | a$

$S \rightarrow bSS' | a$

$S' \rightarrow SaSb | b$
$\quad | SaaS$

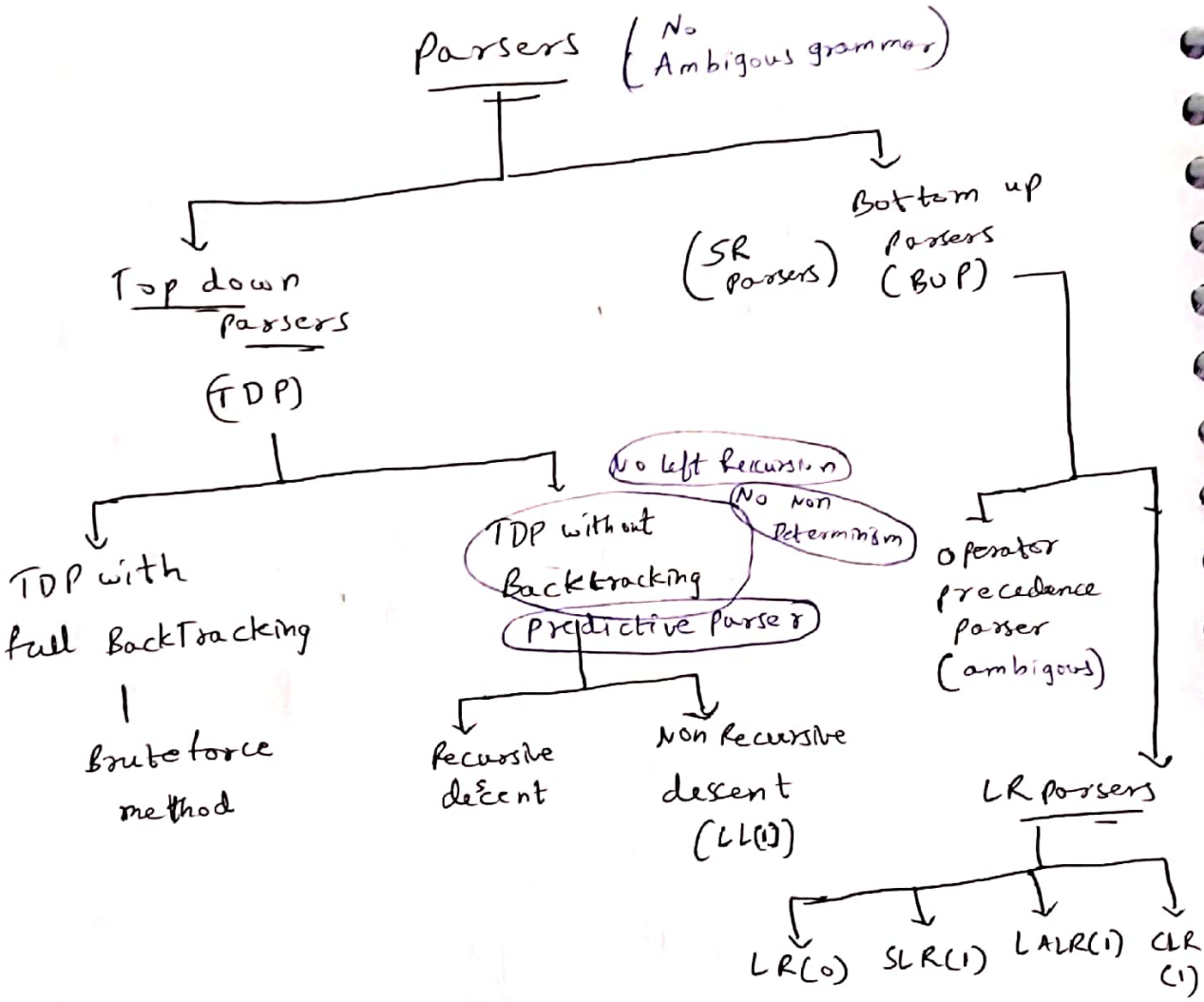
After left factoring

$S \rightarrow bSS' | a$

$S' \rightarrow SaS'' | b$

$S'' \rightarrow aS | Sb$

Types of parsers



Eg:- $S \longrightarrow aABe$

$A \longrightarrow Abc \mid b$

$B \longrightarrow d$

ω abbcde

Leftmost derivation

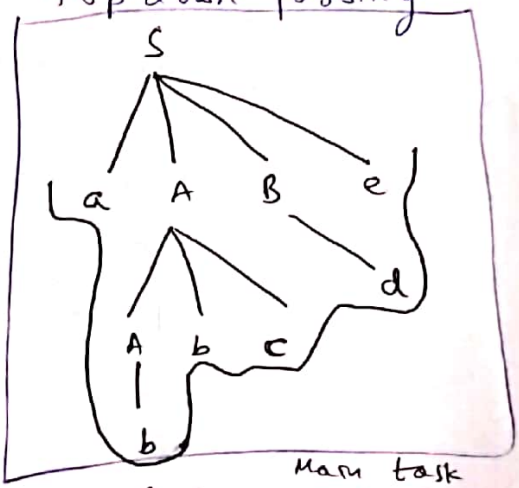
$S \Rightarrow a\,\textcircled{A}Be$

$\Rightarrow a\,\textcircled{A}bcBe$

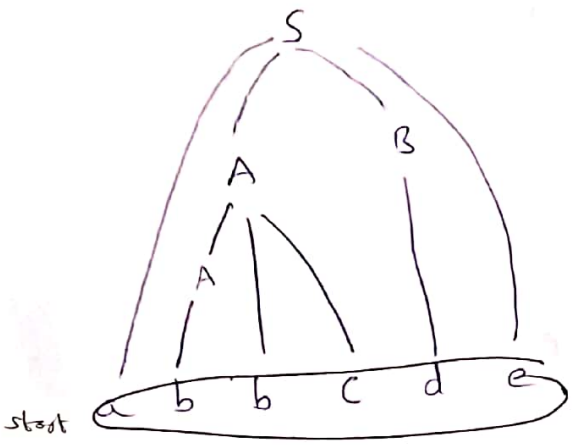
$\Rightarrow abbc\,\textcircled{B}e$

$\Rightarrow abbcde$

Top down parsing



Main task (what production to use)



Reverse of Right most derivation

Bottom up parsers
Main Task
(when to Reduce)

$S \Rightarrow aABe$
$\Rightarrow aAde$
$\Rightarrow aAbcde$
$\Rightarrow abbcde$

Top Down Parsing

→ Top-Down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of parse Tree in preorder.

→ Top-Down Parsing can be viewed as finding the leftmost derivation for the input string.

→ At each step of top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say A.

→ once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with input string.

→ Top Down Parser is constructed for the grammar if it is free from Ambiguity & left Recursion.

Recursive Descent parser

→ Parse is constructed from Top

→ Input is read from Left to Right.

→ The input is recursively parsed for preparing a parse Tree with or without Backtracking

Backtracking

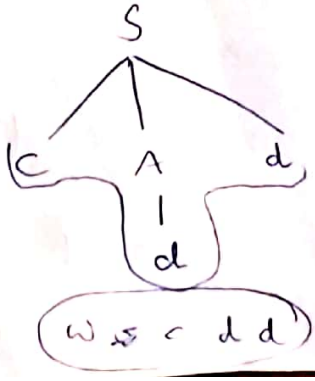
→ The Parse Tree is started from root node & input string is matched against the production rules for replacing them.

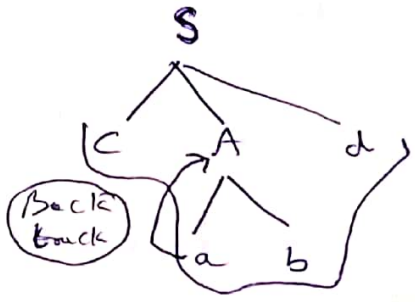
Eg:

$S \longrightarrow cAd$

$A \longrightarrow ab \mid d$

$w = cdd$





$w = cabd$

Limitations

→ If the given grammar has more no. of alternatives then the cost of Back-Tracking is high.

LL Parser

→ It accepts LL grammar

→ It is denoted by LL(K)

- L: Scans i/p from Left to right
- L: Left most derivation
- K: no. of look aheads, K = 1

∴ LL(K) = LL(1)

→ A grammar "G" is LL(1) if there are two distinct productions

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

(i) For no Terminal 'a' α and β derive string beginning with a.

(ii) Atmost one of α and β can derive Empty string.

(iii) If $\beta \overset{*}{\Rightarrow} \in$ then α doesnot derive any string beginning with a terminal in Follow(A)

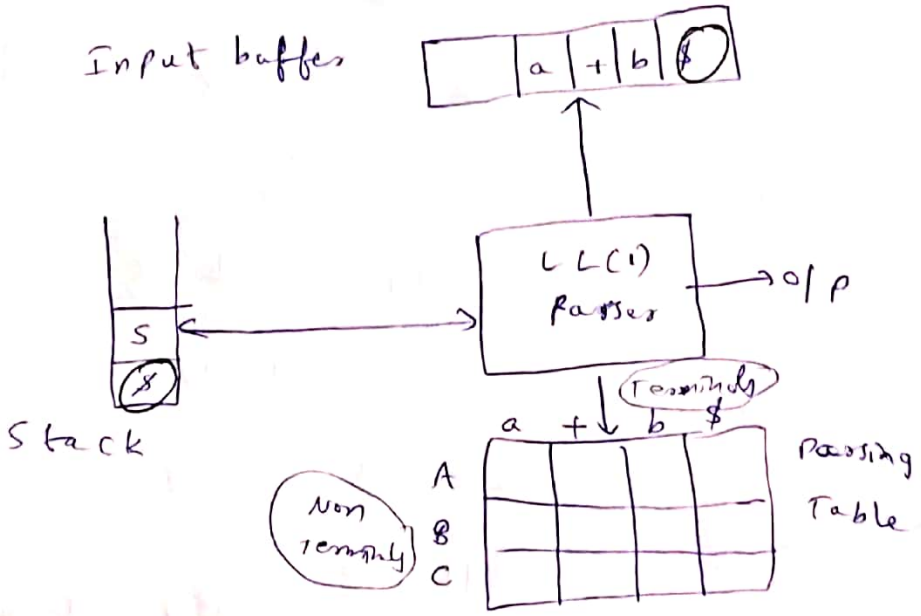
→ LL(1) uses data structure

(i) Input buffer

(ii) stack

(iii) parsing Table

Structure of LL(1)



Construction of predictive LL(1)

(1) FIRST() / Leading()

FOLLOW() / Trailing()

(2) Predictive parsing table by using FIRST & FOLLOW functions

(3) Parse the input string with the help of the table.

Eg:- $S \rightarrow aABCD$

$A \rightarrow b$

$B \rightarrow c$

$C \rightarrow d$

$D \rightarrow e$

First(S) = a
First(A) = b
First(B) = c
First(C) = d
First(D) = e

To compute FIRST(X), apply following rules until no more terminals (or) ϵ can be added to any FIRST set.

(i) If X is terminal, then FIRST(X) = {X}

(ii) If X is nonterminal, and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1) ..... FIRST(Y_{i-1}); that is, $Y_1 \ldots Y_{i-1} \Rightarrow^{*} \epsilon$. If ϵ is in FIRST(Y_j) for $j = 1, 2 \ldots k$, then add ϵ to FIRST(X).

(iii) If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

IRST &

the
tab 6.

→ To Compute FOLLOW(A) for all nonterminals A, apply following rules until nothing can be added to any FOLLOW set.

(i) Place $ in FOLLOW(S), where S is the start symbol. ($ is the right end marker in input)

(ii) If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).

(iii) If there is a production $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Eg:

Grammar

$S \rightarrow ABCDE$

$A \rightarrow a \mid \epsilon$

$B \rightarrow b \mid \epsilon$

$C \rightarrow c$

$D \rightarrow d \mid \epsilon$

$E \rightarrow e \mid \epsilon$

	First	Follow
S	{a, b, c}	{$}
A	{a, ε}	{b, c}
B	{b, ε}	{c}
C	{c}	{d, e, $}
D	{d, ε}	{e, $}
E	{e, ε}	{$}

Suppose $A \rightarrow BC$ Follow(C) = Follow(A)

Eg:

Grammar

Eg:

Eg:

Eg:

nals A,
added

the
in input)
everything
(B)-
$A \to \alpha B \beta$
in

Eg:

Grammar

$S \to Bb \mid Cd$

$B \to aB \mid \epsilon$

$C \to cC \mid \epsilon$

	First	Follow
S	{a, b, c, d}	{$}
B	{a, ε}	{b}
C	{c, ε}	{d}

Eg:

$E \to TE'$

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

$T \to FT'$

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

$F \to id \mid (E)$

	First	Follow
E	{id, (}	{$,)}
E'	{+, ε}	{$,)}
T	{id, (}	{+, $,)}
T'	{*, ε}	{+, $,)}
F	{id, (}	{*, +, $,)}

Eg:

$S \to AB \mid \epsilon bB \mid Ba$

$A \to da \mid BC$

$B \to g \mid \epsilon$

$C \to h \mid \epsilon$

	First	Follow
S	{d, g, h, b, ε}	{$}
A	{d, g, h, ε}	{h, g, $}
B	{g, ε}	{a, $, h, g}
C	{h, ε}	{h, g, $, b}

Eg:

$S \to aABb$

$A \to c \mid \epsilon$

$B \to d \mid \epsilon$

	First	Follow
S	{a}	{$}
A	{c, ε}	{d, b}
B	{d, ε}	{b}

Eg:

$S \to aBDh$

$B \to cC$

$C \to bC \mid \epsilon$

$D \to EF$

$E \to g \mid \epsilon$

$F \to f \mid \epsilon$

	First	Follow
S	{a}	{$}
B	{c}	{g, f, h}
C	{b, ε}	{g, f, h}
D	{g, f, ε}	{h}
E	{g, ε}	{f, h}
F	{f, ε}	{h}

Parsing Table

	id	+	×	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

Note:- ① The productions should be placed in First of Right hand side in the Table.

② If production is (like $A \to \epsilon$) then it should be placed in Follow of (A) in the Table.

Eg:- $S \to (S) \mid \epsilon$

Parsing Table

	(	)	$
S	$S \to (S)$	$S \to \epsilon$	$S \to \epsilon$

Eg:- $S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

	First	Follow
S	{a}	{$}
A	{ε}	{a, b}
B	{ε}	{b, a}

Matche

id
id
id
id+
id+
id+
id+id
id+i
id+id*
id+id*
id+id+id

	a	b	$
S	$S \to AaAb$	$S \to BbBa$	
A	$A \to \epsilon$	$A \to \epsilon$	
B	$B \to \epsilon$	$B \to \epsilon$	

Parsing Table

stack implementation by using

Input string = id + id * id $

Matched	STACK	Input	Action
	E $	id+id*id$	
	T E' $	id+id*id$	output E → TE'
	F T' E' $	id+id*id$	output T → F T'
	id T' E' $	id+id*id$	output F → id
id	T' E' $	+id*id$	match id
id	E' $	+id*id$	output T' → ϵ
id	+ T E' $	+id*id$	output E' → + T E'
id+	T E' $	id*id$	match +
id+	F T' E' $	id*id$	output T → F T'
id+	id T' E' $	id*id$	output F → id
id+id	T' E' $	*id$	match id
id+id	* F T' E' $	*id$	output T' → * F T'
id+id*	F T' E' $	id$	match *
id+id*	id T' E' $	id$	output F → id
id+id*id	T' E' $	$	match id

id+id*id	E'$	$	output T'→ε
id+id*id	$	$	output E'→ε

Error Recovery in LL(1) [Predictive Parsing]

→ An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol. (or) when NonTerminal A is on top of the stack, a is the next input symbol, and M[A,a] is error (i.e the parsing Table entry is empty).

Panic Mode

→ It is based on the idea of skipping symbols on the input until a token in a selected set of synchronization token appears.

→ The sets should be chosen so that the Parser recovers quickly from errors that are likely to occur in practice.

→ Som

(i) A

in F

Nonte

(ii) It

as sy

(iii) If

the sy

it m

to A

in th

(iv) If

string,

used a

This

that

recov

(v) If

matche

issue

insert

→ Some heuristics are as follow

(i) As a starting point, Place all symbols in FOLLOW(A) into the synchronizing set for Nonterminal A.

(ii) It is not enough to use FOLLOW(A) as synchronizing set for A.

(iii) If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

(iv) If a nonTerminal can generate that empty string, then the production deriving ε can be used as a default.
This approach reduces the number of nonterminals that have to be considered during error recovery.

(v) If a terminal on top of stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing.

→ The parsing Table is to be ch

(i) If parser looks up entry M[A
that it is blank, then the input sym
skipped.

(ii) If the entry is "synch", then the
on top of the stack is popped
attempt to resume parsing.

(iii) If token on top of the stack d
the input symbol, then we pop the tok
stack.

Parsing Table updated

	id	+	*	(
E	$E \to TE'$			$E \to TE'$
E'		$E' \to +TE'$		
T	$T \to FT'$	synch		$T \to FT'$
T'		$T' \to \epsilon$	$T' \to *FT'$	
F	$F \to id$	synch	synch	$F \to (E)$

Synchronizing tokens added to
Ta

input string =) id * + id

STACK	INPUT	REMARK
E $	) id * + id $	error, skip)
E $	id * + id $	E → T E'
T E' $	id * + id $	
F T' E' $	id * + id $	
id T' E' $	id * + id $	
T' E' $	* + id $	
* F T' E' $	* + id $	
F T' E' $	+ id $	error, M[F, +] = synch
T' E' $	+ id $	F has been popped
E' $	+ id $	
+ T E' $	+ id $	
T E' $	id $	
F T' E' $	id $	
id T' E' $	id $	
T' E' $	$	
E' $	$	
$	$	

Reduction

Phrase - Level - Recovery

→ It is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines.

→ These routines may change, insert (or) delete symbols on input and issue appropriate error messages.

→ The steps carried out by the parser might then not correspond to the derivation of any word in Language at all.

→ we must ensure that there is no possibility of infinite loop's.

→ checking that any recovery action eventually results in an input symbol being consumed is a good way to protect agianst such loops.

Bottom up parsing

→ In bottom up parsing method, the input string is taken first and we try to reduce this string with the help of grammar and try to obtain the start symbol.

→ The process of parsing stops(halts) successfully as soon as we reach to start symbol.

→ The parse tree is constructed from bottom to up that is from leaves to root.

→ In this process, the input symbols are placed at the leaf nodes after successful parsing.

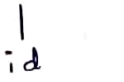
→ The bottom up parse tree is created starting from leaves, the leaf nodes together are reduced further to internal nodes, these internal nodes are further reduced and eventually a root node is obtained.

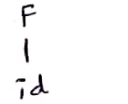
→ In this process, the parser tries to identify RHS of production rule and replace it by corresponding LHS. This is called Reduction.

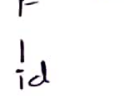
→ The task of bottom up parsing is to find the productions that can be used for reduction.

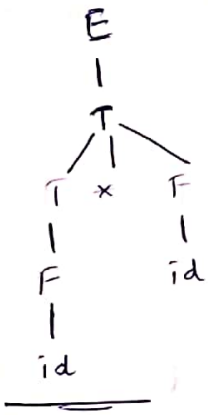
Eg:

id * id









Reductions

→ Bottom-up parsing can be thought of as process of "reducing" a string w to the start symbol of the grammar.

→ At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

→ The key decisions during bottom-up passing are about when to reduce and about what production to apply, as parse proceeds.

Handle pruning

→ Bottom-up parsing during a left-to-right scan of the input constructs a right most derivation in reverse.

→ "Handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation

Right Sentential form	Handle	Reducing Production
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$E \rightarrow T * F$

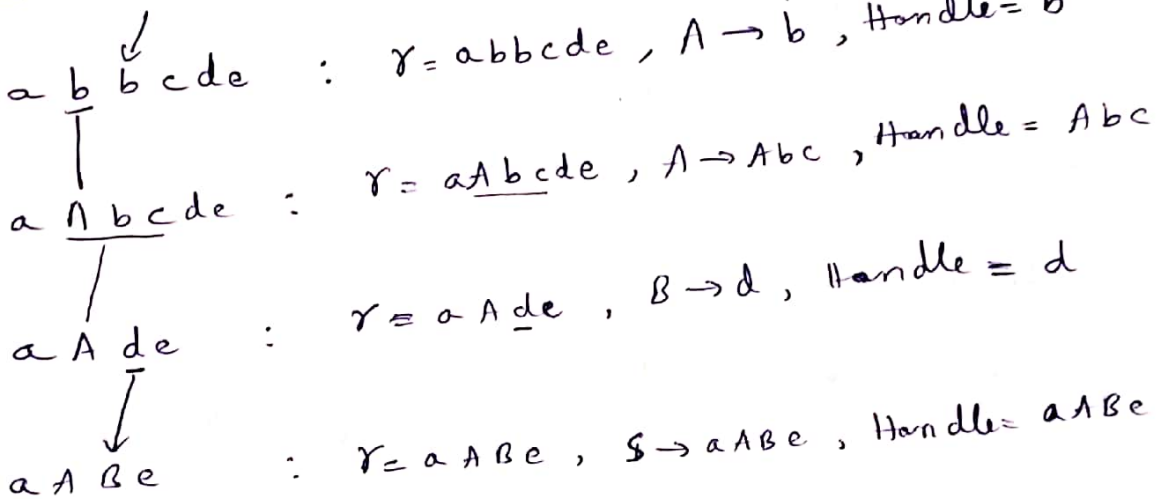
→ A handle of a right sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right

sentential form in a rightmost derivation of γ.

Eg:

$$S \xrightarrow[rm]{} aA\underline{B}e \xrightarrow[rm]{} a\underline{A}de \xrightarrow[rm]{} a\underline{A}\underline{b}cde \rightarrow a\underline{b}bcde$$

Right sentential form



Grammar

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

PRUNING THE HANDLE

→ Removing the children of left Hand side non Terminal from the Parse Tree is called Handle pruning.

→ A Rightmost derivation in Reverse can be obtained by Handle Pruning

Steps to Follow :

→ start with a string of terminals 'w' that is to be parsed.

→ Let $w = \gamma_n$, where γ_n is the n^{th} right sentential form of an unknown RMD.

→ To reconstruct the RMD in reverse, locate handle β_n in γ_n; Replace β_n by (γ_{n-1}) to get $(n-1)^{th}$ RSF of some $A_n \to \beta_n$

γ_{n-1}. Repeat

$$S \underset{rm}{\Rightarrow} \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

Eg: Grammar

$$E \to E+E \mid E*E \mid id$$

Right Sentitial Form	Handle	Reducing Production
$id_1 + id_2 * id_3$	id_1	$E \to id$
$E + id_2 * id_3$	id_2	$E \to id$
$E + E * id_3$	id_3	$E \to id$
$E + E * E$	$E+E$	$E \to E+E$
$E * E$	$E*E$	$E \to E*E$
(E)		

Handle pruning

Start symbol

Shift-Reducing Parser (SR)

→ It is a form of Bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.

→ The Handle always appears at the top of stack just before it is identified as Handle

→ we use '$' to mark the bottom of the stack and also the right end of the input.

→ Initially stack is empty, and string w is on input, as follows:

STACK	INPUT
$	w$

→ During left-to-right scan of the input string, the parser shifts zero (or) more input symbols onto top of the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.

→ It then reduces β to the head of the appropriate production.

→ The parser repeats this cycle until it has

detected an error (or) until the stack contains the start symbol and input is empty;

STACK	INPUT
$ S	$

→ upon Entering this configuration, the parser halts and announces successful completion of parsing.

→ Primary operations are shift and reduce

→ There are four possible Actions

(i) shift
(ii) reduce
(iii) Accept
(iv) error

Shift – shift the next input symbol onto the top of the stack.

Reduce – The right end of the string to be reduced must be at the top of the stack. Locate the left end of string within the stack and decide with what nonterminal to replace the string.

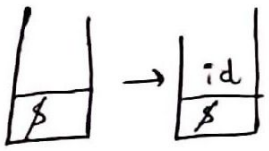
Accept – Announce successful completion of parsing.

Error → Discover a syntax error and call an error recovery routine.

$E \to E+E \mid E*E \mid id$

Eg:

STACK CONTENT	INPUT	ACTION
$	$id_1 + id_2 * id_3 \$$	shift
$\$ id_1$	$+ id_2 * id_3 \$$	Reduce by $E \to id$
$\$ E$	$+ id_2 * id_3 \$$	shift
$\$ E +$	$id_2 * id_3 \$$	shift
$\$ E + id_2$	$* id_3 \$$	Reduce by $E \to id$
$\$ E + E$	$* id_3 \$$	shift
$\$ E + E *$	$id_3 \$$	shift
$\$ E + E * id_3$	$\$$	Reduce by $E \to id$
$\$ E + E * E$	$\$$	Reduce $E \to E*E$
$\$ E + E$	$\$$	Reduce $E \to E+E$
$\$ E$ (start symbol)	$\$$	Accept.



Bottom →

Top Down Parsing

Recursive Descent parser

→ A parser that uses collection of recursive procedures for parsing the given input string is called Recursive Descent (RD) Parser.

→ In this grammar CFG is used to build the recursive routines.

→ The RHS of the production rule is directly converted to a program.

→ For each non terminal a seperate procedure is written and body of procedure (code) is RHS of the corresponding non Terminal

Basic steps for Construction of RD Parser

The RHS of the rule is directly converted into program code symbol by symbol.

(i) If the input symbol is nonTerminal then a call to procedure corresponding to the nonterminal is made.

(ii) If the input symbol is terminal then it is matched with the lookahead from input. The lookahead pointer has to be advanced on matching of the input symbol.

(iii) If the production rule has many alternatives then all these alternates has to be combined into a single body of procedure.

(iv) The parser should be activated by a procedure corresponding to the start symbol.

$E \rightarrow num\ T$

$T \rightarrow *\ num\ T \mid \epsilon$

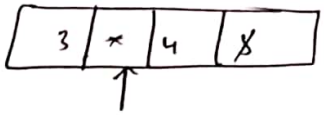
```
Procedure E
{
 if lookahead = num then
    {
    match (num);
    T;                    /call to procedure T/
    }
 else
    error;
 if lookahead = $
    {
    declare sucess; /Return on success/
    }
```

```
        else
          error;
}/End of procedure E/

Procedure T
{
if lookahead = '*'
   {
     match ('*');
     if lookahead = 'num'
      {
       match (num);
        T;
      }
      else
       error;
   }
   else NULL
}
procedure match (token t)
{
 if lookahead = t
 lookahead = next_token;     /lookahead pointer is advanced.)
  else
   error;
}
 procedure error
  {
   print ("Error");
```

E → num T

(a)

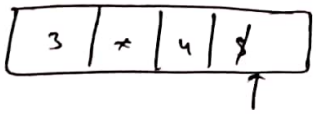


T → * num T

(b)

T → * num T

(c)



Declare success

(d)

→ The parser will be activated by calling procedure E. since the first input character (3) is matching with num the procedure match(num) will be invoked and then the lookahead will point to next token. And a call to procedure T is given.

→ A match with '*' is found hence

lookahead = next_token

→ Now '4' is matching with num hence again

the procedure for match(num) is fulfilled.

→ Then procedure for T is invoked and T is replaced by ε.

→ As look ahead pointer points to $ we quit by reporting success.

(Introduction to LR Parsing : Simple LR)

Why use stack for SR-parsers

→ Any handle will always appear on the top of the stack and the parser need not search within the stack at any time.

Conflicts in SR-parsing

→ Two decisions decide a successful SR-parsing

(i) Locate the substring to reduce

(ii) which production to choose when multiple productions with the selected substring on RHS exist.

→ SR parser may reach a configuration in which knowing the contents of stack and input buffer, still the parser cannot decide.

⇒ (a) Whether to perform a shift or a Reduce operations (shift Reduce conflict)

(b) ~~whether~~ which out of the several possible reductions to make (Reduce-Reduce conflict)

Eg:- $S \rightarrow$ if E then S
| if E then S else S
| other

STACK

$ if E then S
valid Handle

INPUT

else -- $
should not be reduced

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid x$

Eg:

STACK	Input	Action
$	x * x $	shift
$ x	* x $	Reduce by $F \rightarrow x$
$ F	* x $	Reduce by $T \rightarrow F$
$ T	* x $	shift
$ T *	x $	shift
$ T * x	$	Reduce by $F \rightarrow x$
$ T * F	$	Reduce by $T \rightarrow T*F$
$ T	$	Reduce by $E \rightarrow T$
$ E	$	Accept

Eg:-

STACK	INPUT	ACTION
$	x * x $	shift
$ x	* x $	Reduce F → x
$ F	* x $	Redue F → T
$ T	* x $	Reduce E → T
$ E	* x $	shift
$ E *	x $	shift
$ E * x	$	Reduce F → x
$ E * F	$	Redue T → F
$ E * T	$	Reduce E → T
$ E * F	$	Error

* $ T * F R
$ T * T Error

Operator Grammar

→ shift Reduce Parsers can be built successfully using / for 2 Main classes of grammar.

- LR Grammar
- Operator Grammar.

Properties of Operator Grammar

1) No production in the Grammar has ϵ on (Right Hand Side)

2) No 2 Non Terminals appear together on RHS of any production.

Eg:- $E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \wedge$

→ voilates Rule 2 → Not O.P.G

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \wedge E$

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

→ O.P.G

Eg:- $S \rightarrow SAS \mid a$ → Non O.P.G

$A \rightarrow bSb \mid b$

$S \rightarrow SbSbS \mid SbS \mid a$ → O.P.G

Operator precedence parsing

→ we define 3 precedence relation operations between pairs of Terminals.

$p <\cdot q$	p gives more precedence than q
$p \doteq q$	p has same precendece as q
$p \cdot> q$	p takes precedence over q.

Use → They help us in selection of handles

How to Assign Relations

→ using Associativity and precendence

→ For all terminals (including $) we design an operator precedence table.

Grammar

Eg: $E \rightarrow E+E \mid E*E \mid id$ I/P

T.OS \	id	+	*	$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	ACC

id .> +
id .> *
id .> $

$ <. id
$ <. +
$ <. *

Symbol on T.OS

Symbol I/P stn

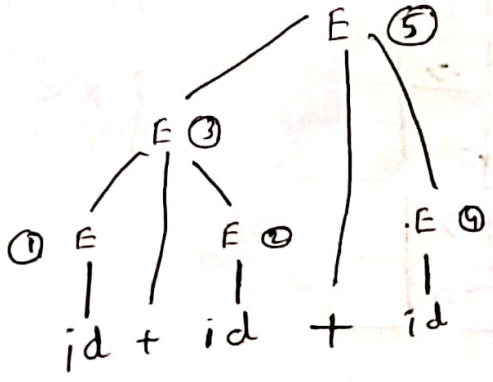
If I/p string symbol .> Tos : PUSH I/P symbol

else I/p string symbol <. Tos : Pop and Reduce Tos

else error

Eg:

I/P string	stack	
id + id * id $	$	
+ id * id $	$ id	Reduce ①
+ id * id $	[redacted] $	
id + id $	$ +	
+ id $	$ + id	Reduce ②
+ id $	$ +	Reduce ③
+ id $	$	
id $	$ +	
$	$ + id	Reduce ④
$	$ +	Reduce ⑤
$	$	Accept



** → Operator precedence parser is the only parser that will handle Ambigous grammar

Advantage of operator precedence parsing

→ Simplicity of the method

→ Error detecting capability of the parser

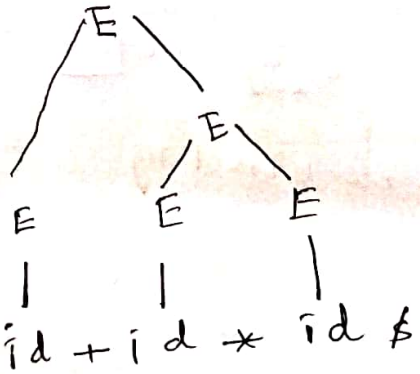
→ Ability to parser Ambigous Grammar (single parse Tree)

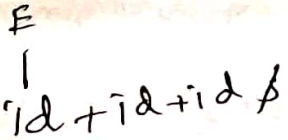
Disadvantage of operator Precedence parsing.

→ only a small class of Grammars can be parsed using operator precedence technique.

→ Difficult to handle operators which may have different roles (unary & binary plus/minus) in different situat

→ As the no of terminals increase the size of precendenc table also increases (solution operator precendence function)





Introduction to LR Parsing

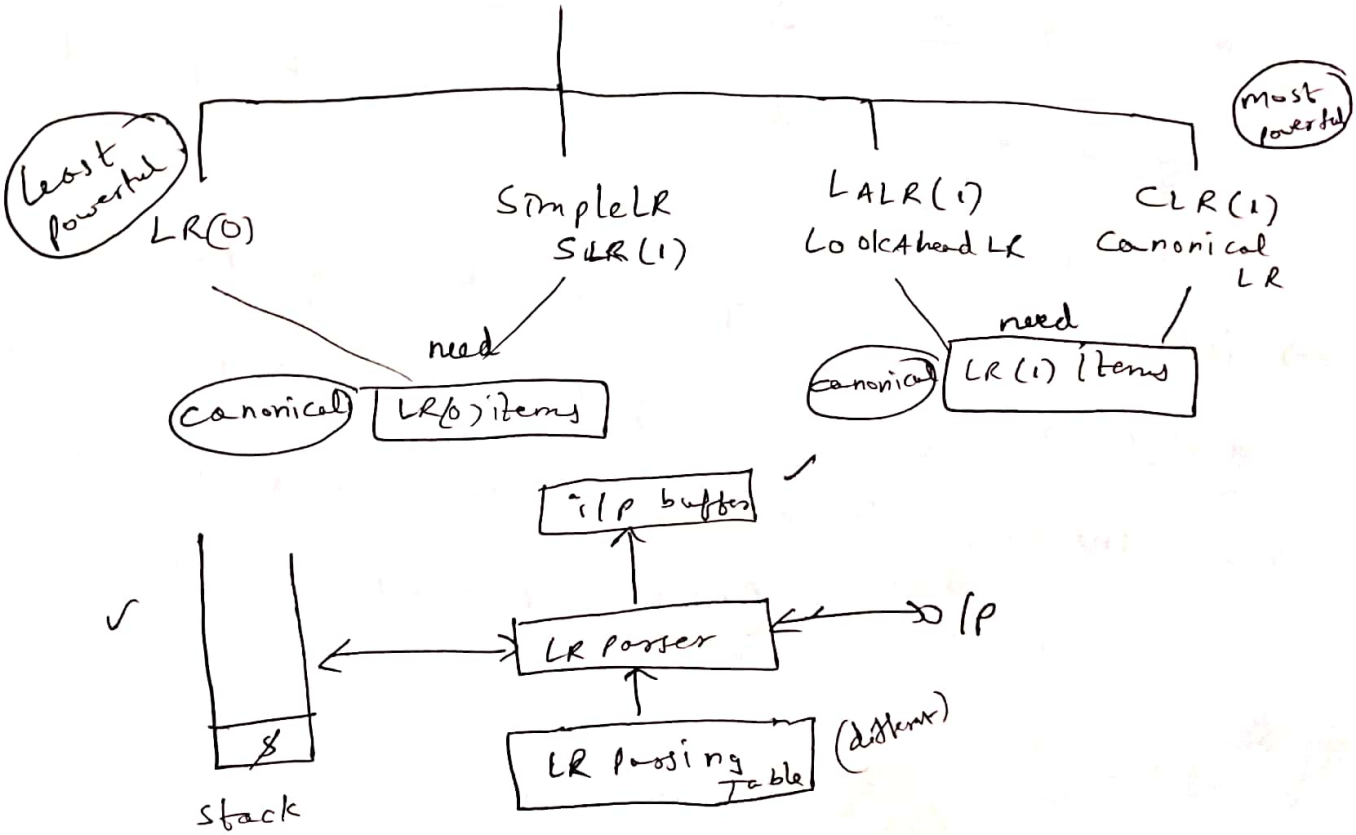
→ Most prevalent type of bottom-up parser today is based on concept called LR(k) parsing.

k - no of i/p symbols of lookahead used in making parsing decisions.

L - left to right scanning of input

R - for construction rightmost derivation in reverse.

LR parser



Benefits of using LR(k) parsing

→ Most generic Non-Backtracking shift Reduce Parsing Technique.

→ These parsers can recognize all programming Languages for which context-free Grammers can be written.

→ They are capable of detecting syntactic error as soon as possible in scanning of input.

Types of LR parsers

(i) Simple LR parser (SLR)

(a) easy to implement

(b) Least powerful

(c) May fail to work on some grammars accepted by CLR, LALR.

(ii) Canonical LR Parser (CLR)

(a) Most Powerful LR parser

(b) Most expensive

(iii) Lookahead LR Parser (LALR)

(a) Intermediate to SLR and CLR interms of Power and Cost.

Eg:- $S \rightarrow AA$

$A \rightarrow aA \mid b$

→ Add one more production

$S' \rightarrow S$

$S \rightarrow AA$ (1)

$A \rightarrow aA$ (2) $\mid b$ (3)

→ Augumented grammar

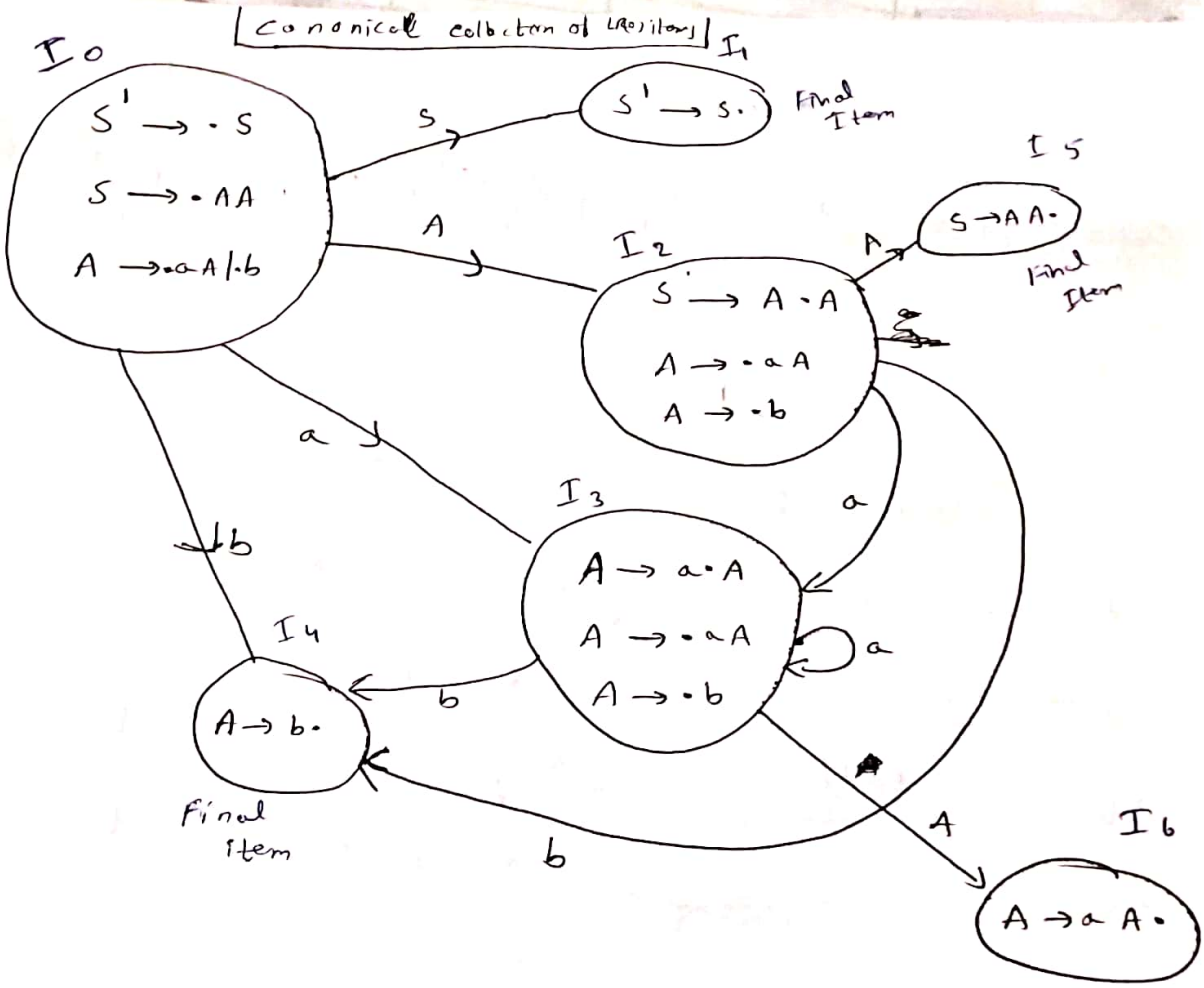
Various steps involved in Parsing

→ For the given I/P string write context free grammar
→ Check the ambiguity of the grammar
→ Add Augment production in the given grammar
→ Create canonical collection of LR(0) items
→ Draw a Data flow diagram (DFA)
→ Construct LR(0) parsing Table

Create canonical collection of LR(0) items

→ An LR(0) item is a production G with dot at some position on the right hand side of production
→ LR(0) items is useful to indicate that how much of the i/p has been scanned up to a given point in the process of parsing
→ In LR(0), we place the reduce node in entire row.

Canonical collection of LR(0) items

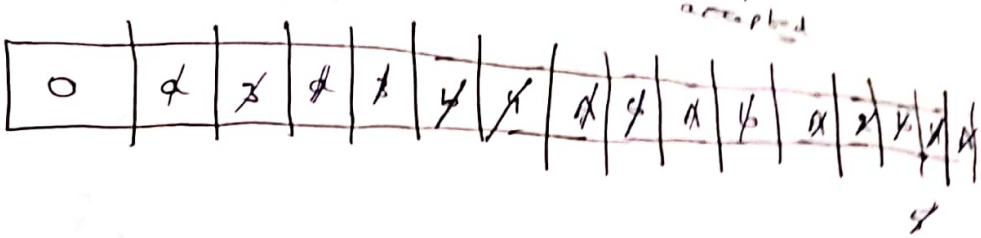


LR(0) parsing Table

	Action			Goto	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

Eg:- a a b b $

String is accepted



0	S	1

accept

LR(0) Table

→ If a state is going to another state on a terminal, it is correspond to a shift move.

→ If a state is going to some other state on a variable it is correspond to a goto move

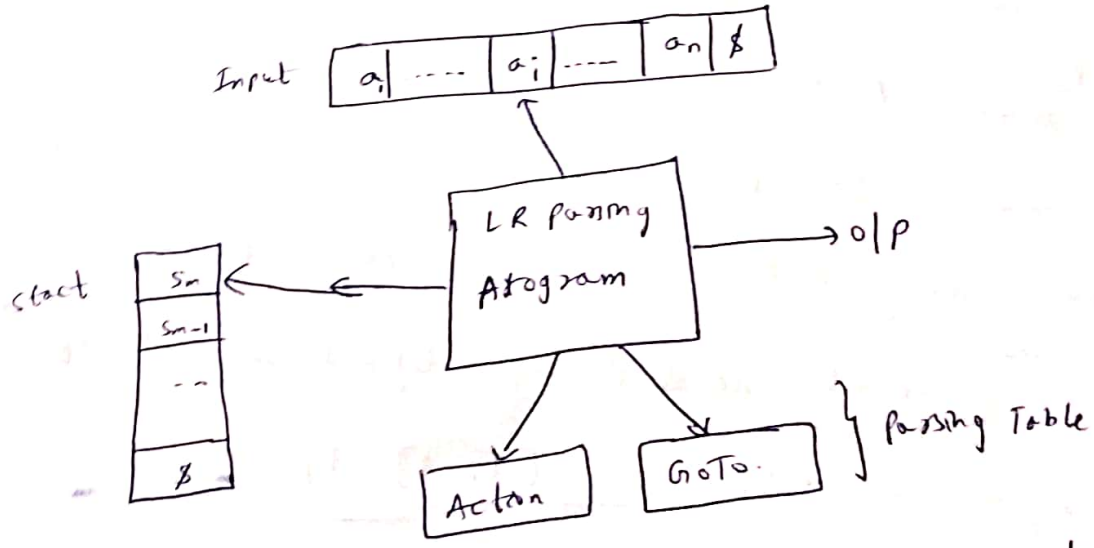
→ If a state contains the final item in a particular row then write reduce node completed.

Parsing I/P string

Steps / Line	Stacktr	Parsing stack (Symbols)	Input	Action
(1)	0	$0	a a b b $	shift a3
(2)		$ 0 a 3	a b b $	shift a3
(3)		$ 0 a 3 a 3	b b $	shift b4
(4)		$ 0 a 3 a 3 b 4	b $	reduce r3 (A→b)
(5)		$ 0 a 3 a 3 A 6	b $	reduce r2 (A→aA)
(6)		$ 0 a 3 A 6	b $	reduce r2 (A→aA)

(7)	$ 0 A 2	b $	shift by
(8)	$ 0 A 2 b 4	$	reduce $r_3 (A \to b)$
(9)	$ 0 A 2 A 5	$	reduce $r_1 (S \to AA)$
(10)	$ 0 S 1	$	Accept

Components of LR Parser



→ Parsing Table changes from 1 LR parser to another.

Structure of LR Parsing Table

→ The Parsing Table consists of two parts

(a) Parsing Action Function (ACTION)

(b) goto Function (GOTO)

→ The Action function takes as Arguments a state i and a terminal a (or $ the input end marker)

Action [i, a] can have four forms

(i) shift j, where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.

(ii) Reduce $A \to \beta$. The action of the parser effectively reduces β on the top of stack to head A.

(iii) Accept. The parser accepts the input and finishes parsing.

(iv) Error. The parser discovers an error in its input and takes some corrective action.

→ We extend GOTO Function, defined on sets of items, to states: if GOTO[I_i, A] = I_j, then GOTO also maps a state i and nonterminal A to state j.

Behavior of LR parser

→ The next move of the parser from the configuration is determined by reading a_i, the current i/p symbol and s_m the state on top of stack and then consulting the entry ACTION[s_m, a_i] in parsing action Table

The configurations resulting after each of the four types of move are as follows:

(i) If ACTION $[s_m, a_i]$ = shift s, the parser executes shift move, it shifts the next state s onto the stack, entering the configuration.

$$(s_0 s_1 \cdots s_m, a_{i+1} \cdots a_n \$)$$

(ii) If ACTION $[s_m, a_i]$ = reduce $A \to B$, then parser executes reduce move, entering the configuration

$$(s_0 s_1 \cdots s_{m-r} s, a_i a_{i+1} \cdots a_n \$)$$

$r \to$ length of B

$s = GOTO[s_{m-r}, A]$

(iii) If ACTION $[s_m, a_i]$ = accept, parsing is completed

(iv) If ACTION $[s_m, a_i]$ = error, parser has discovered an error and calls an error recovery routine

Constructing SLR Parsing Table

Input :- An augmented Grammar G'.

Output : SLR parsing Table functions ACTION & GOTO for G'.

Method :-

(1) Construct $C = \{I_0, I_1, \ldots I_n\}$, collection of sets of LR(0) items for G'.

(2) state i is constructed from I_i. The parsing actions for state i are determined as follows.

(i) If $[A \to \alpha . a\beta]$ is in I_i and GOTO$(I_i, a) = I_j$, then set ACTION$[i, a]$ to shift j. (a must be terminal)

(ii) If $[A \to \alpha .]$ is in I_i, then set ACTION $[i, a]$ to "reduce $A \to \alpha$" for all a in FOLLOW(A)

(A may not be S')

(iii) If $[S' \to S.]$ is in I_i, then set ACTION$[i, \$]$ to "Accept"

SLR(1) parsing Table

→ only change in Reduce moves

→ Reduce moves are placed in Follow(A)

If production is "A → α" not in entire row

	Action			Goto	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

$I_4 \rightarrow$ (3) $A \rightarrow b.$ → reduce move should be placed in Follow(A) = { a, b, $ }

$I_5 \rightarrow$ (1) $S \rightarrow AA.$ → reduce move should be placed in Follow(S) = { $ }

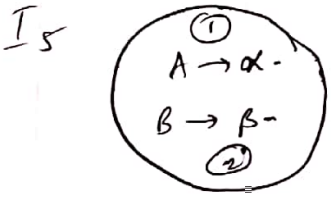
$I_6 \rightarrow$ $A \rightarrow aA.$ (2) → reduce move should be placed in Follow(A) = { a, b, $ }

In LR(0) Parser

→ we may get s/r conflict when

	a	b
5	s_6/r_1	r_1

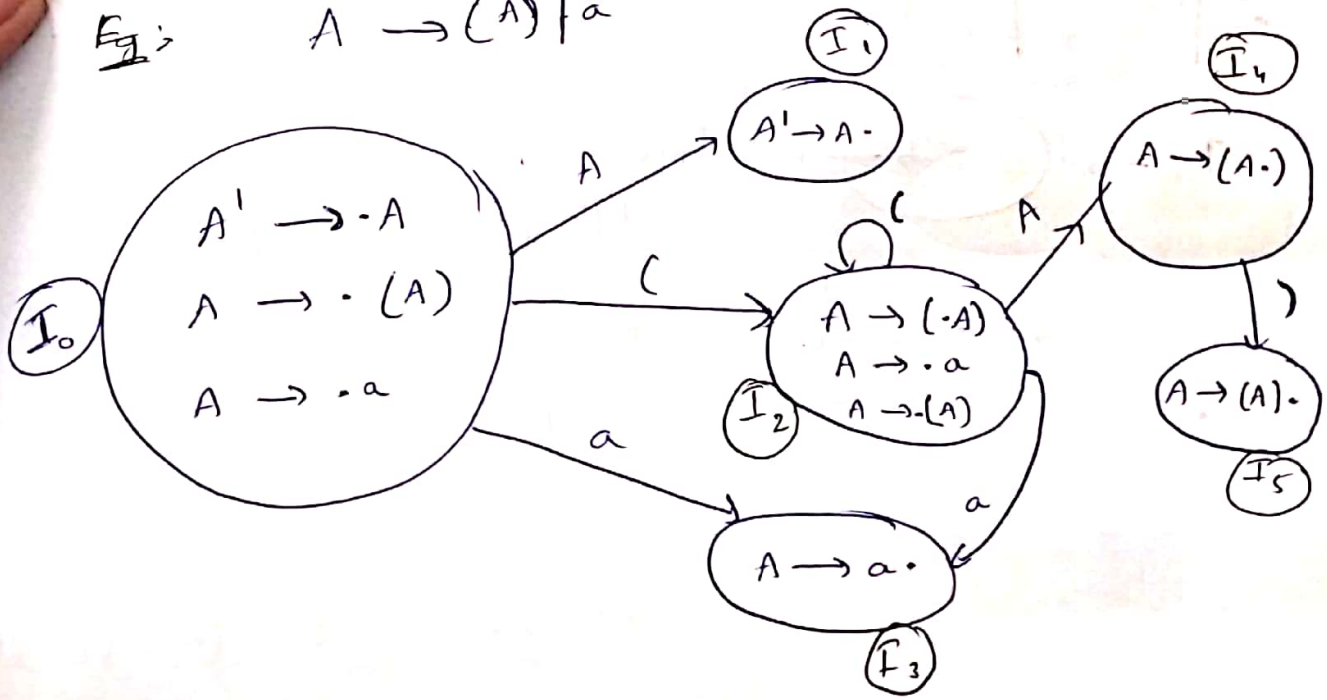
→ we may also get r/r conflict when



	a	b	c
5	r_1/r_2	r_1/r_2	r_2/r_1

[SLR(1) parsing Table]

Eg: $A \to (A) \overset{1}{|} a \overset{2}{}$



	Action				Goto
	(	)	a	$	A
0	S_2		S_3		1
1				Accept	
2	S_2		S_3		4
3		r_2		r_2	
4		S_5			
5		r_1		r_1	

I_3 and I_5 final Items

$I_3 \rightarrow$ place r_2 in $(A \rightarrow a)$ Follow(A) = {), $ }

$I_5 \rightarrow$ place r_1 in $(A \rightarrow (A))$ Follow(A) = {), $ }

Step	Parsing stack	Input	Action
(1)	$0	(a)$	shift 2
(2)	$0(2	a)$	shift 3
(3)	$0(2a3	)$	reduce 2 ($A \rightarrow a$)
(4)	$0(2A4	)$	shift 5
(5)	$0(2A4)5	$	reduce 1 ($A \rightarrow (A)$)
(6)	$0A1	$	Accept

Grammar

$E \rightarrow E+T$ 1

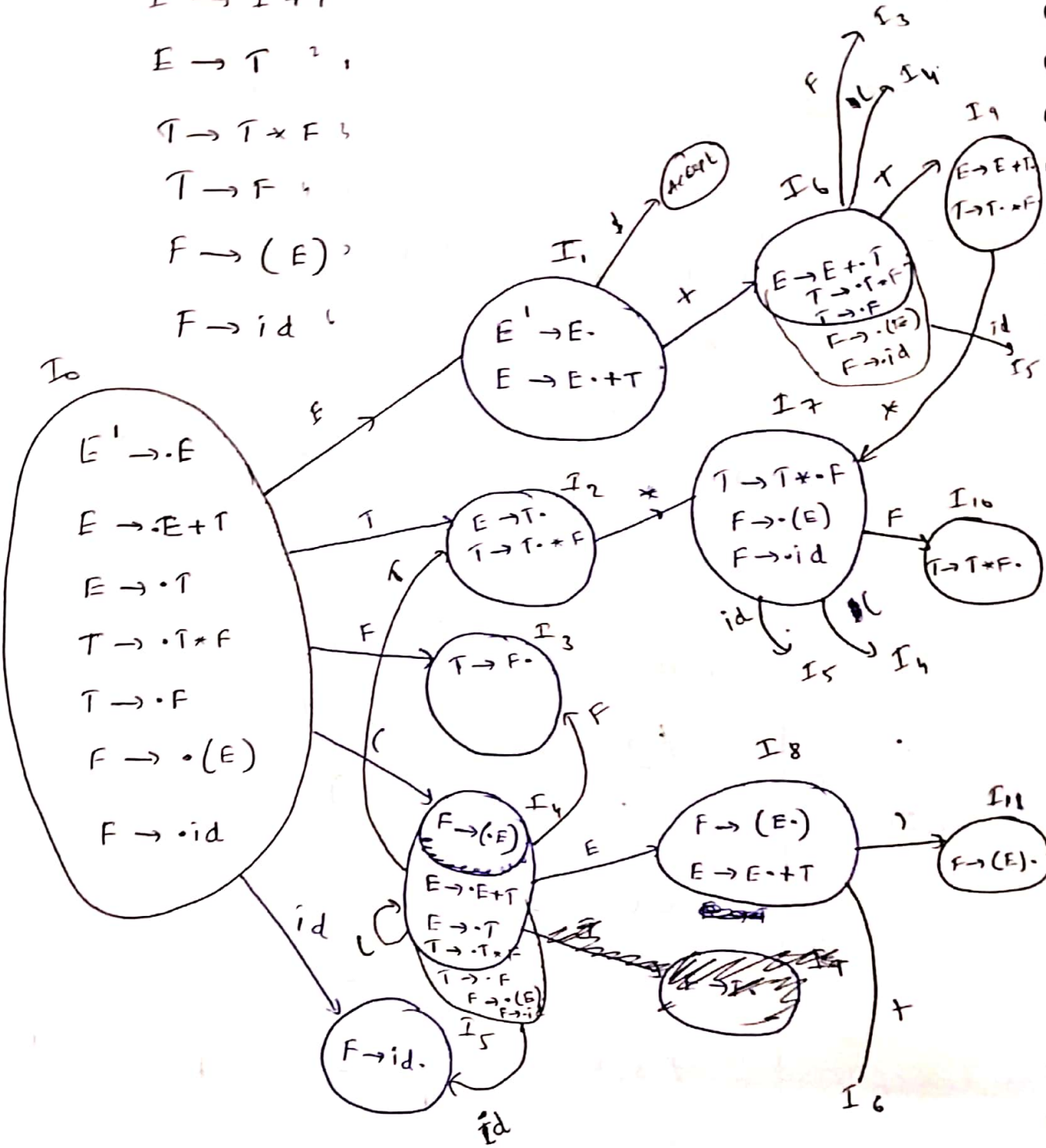
$E \rightarrow T$ 2

$T \rightarrow T*F$ 3

$T \rightarrow F$ 4

$F \rightarrow (E)$ 5

$F \rightarrow id$ 6



Follow (E) = { +,), $ }

Follow (T) = { *, +,), $ }

Follow (F) = { *, +,), $ }

Parsing Table

	Action						Goto		
	*	+	id	(	)	$	E	T	F
0			s_5	s_4			1	2	3
1		s_6				Accept			
2	s_7	r_2			r_2	r_2			
3	r_4	r_4			r_4	r_4			
4			s_5	s_4			8	2	3
5	r_6	r_6			r_6	r_6			
6			s_5	s_4				9	3
7			s_5	s_4					10
8		s_6			s_{11}				
9	s_7	r_1			r_1	r_1			
10	r_3	r_3			r_3	r_3			
11	r_5	r_5			r_5	r_5			

Eg:

S.No	Parsing stack	I/P string	Action
1	$0	id*id+id$	shift 5
2	$0id5	*id+id$	reduce 6 (E→id)
3	$0F3	*id+id$	reduce 4 (T→F)
4	$0T2	*id+id$	shift 7
5	$0T2*7	id+id$	shift 5
6	$0T2*7id5	+id$	Reduce 6 (F→id)

7)	$0T2*7F10	+id$
8)	$0T2	+id$
9)	$0E1	+id$
10)	$0E1+6	id$
11)	$0E1+6id5	$
12)	$0E1+6F3	$
13)	$0E1+6T9	$
14)	$0E1	$

CLR(1) and LALR(1) [LR(0) items]

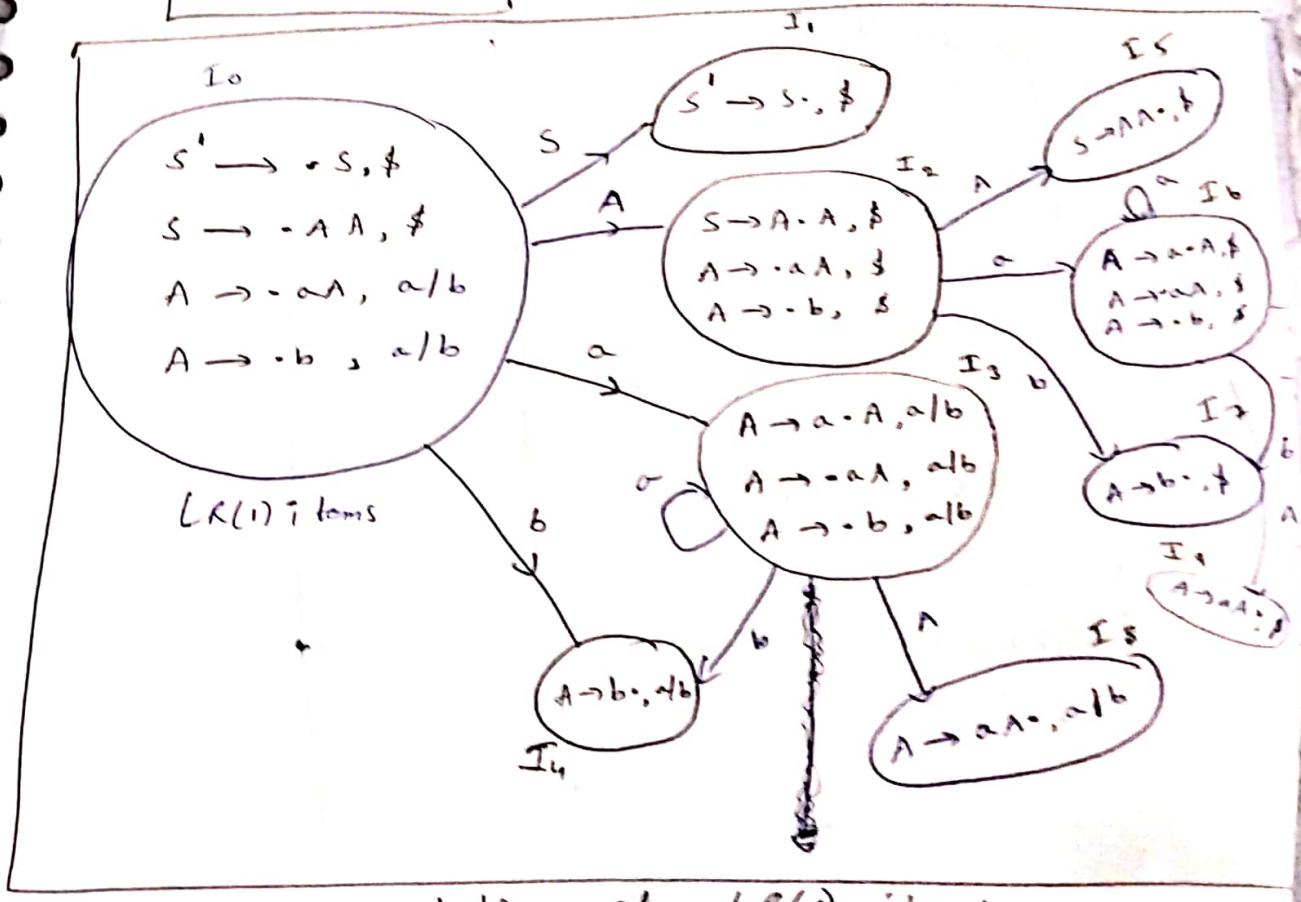
→ LR(1) item = LR(0) item + Look ahead

Grammar

$S \rightarrow AA$
$A \rightarrow aA/b$

Rule

$A \rightarrow \alpha \cdot B\beta,\ a/b$
$B \rightarrow \cdot\gamma,\ \text{First}(\beta a/b)$



Canonical Collection of LR(1) items

Parsing Table | CLR(1)

	Action			Goto	
	a	b	$	A	S
0	s_3	s_4		2	1
1			Accept		
2	s_6	s_7		5	
3	s_3	s_4		8	
4	r_3	r_3			
5			r_1		
6	s_6	s_7		9	
7			r_3		
8	r_2	r_2			
9			r_2		

No. of states in

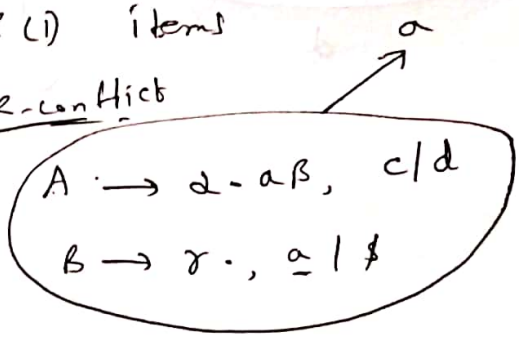
$$CLR(1) \geq LR(0) = SLR(1) = LALR(1)$$

LALR Parsing Table

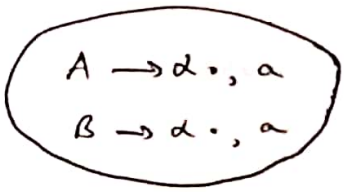
	Action			Goto	
	a	b	$	S	A
0	s_{36}	s_{47}			2
1			Accept		
2	s_{36}	s_{47}			5
36	s_{36}	s_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

LR(1) items

SR-conflict



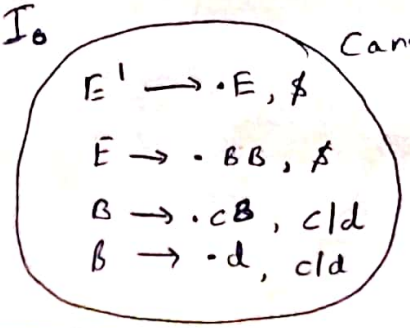
RR conflict

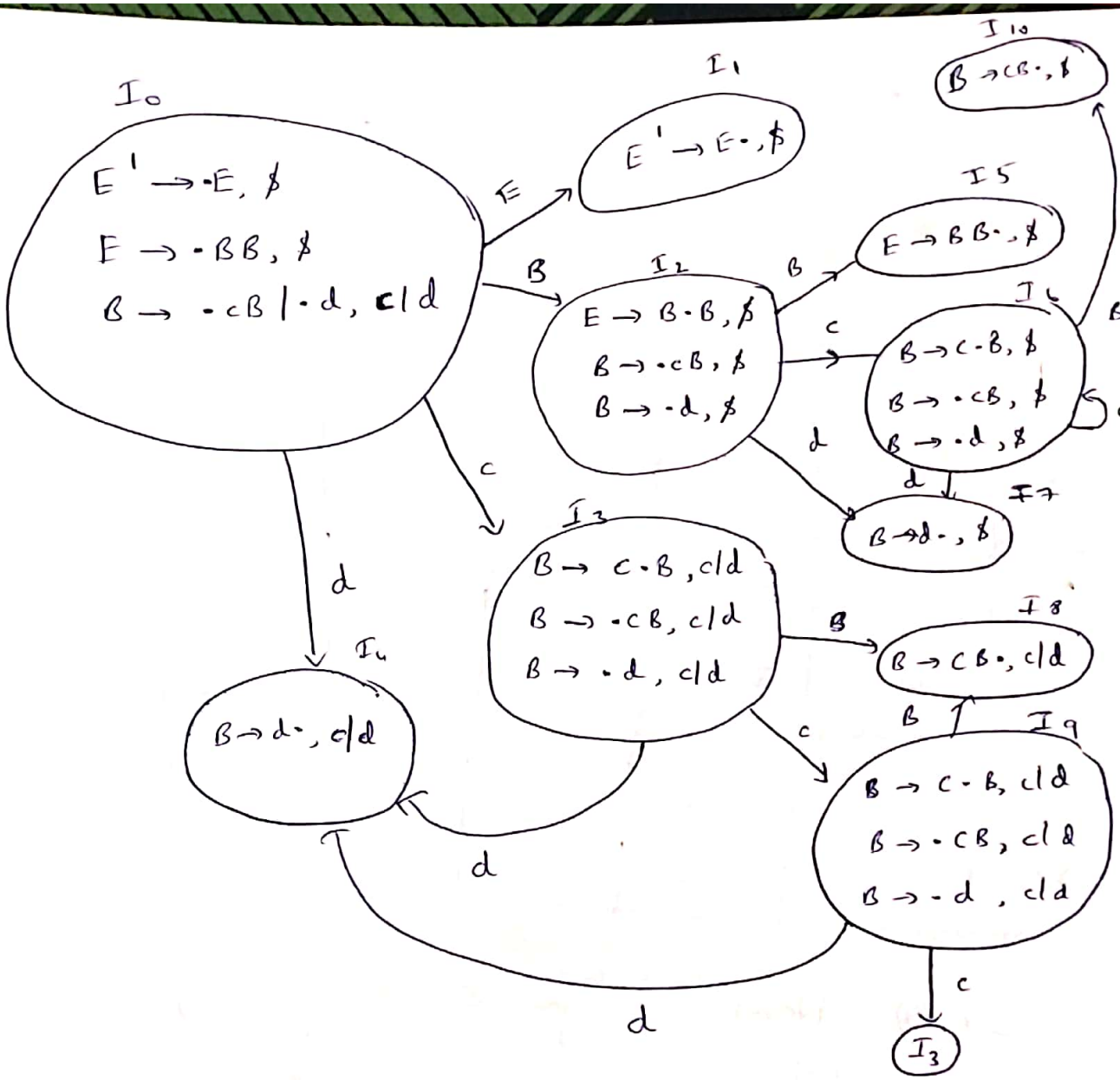


Eg:- E → B B

B → c B / d

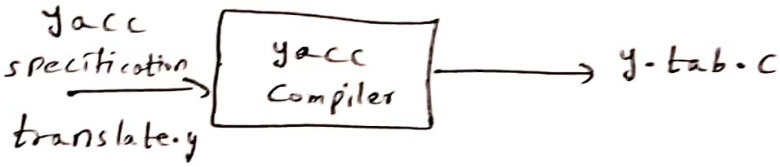
I_0 Canonical LR(1) items

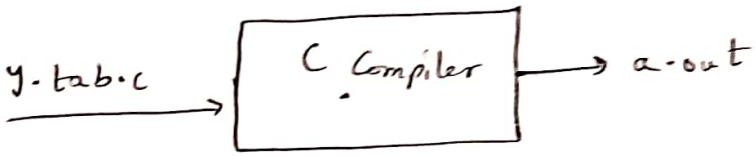


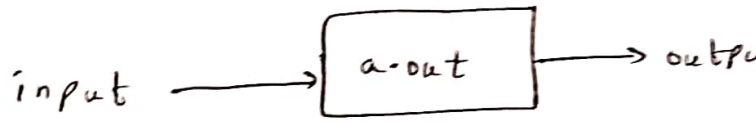


Parser Generators

→ Yet Another Compiler - Compiler (YACC)







→ Unix Command `yacc translate.y`

→ The above command translates the file translate.y into a C program called y.tab.c using LALR method.

→ y.tab.c is a representation of an LALR parser written in C,

→ By Compiling y.tab.c along with ly library that contains the LR parsing program using command

`cc y.tab.c -ly`

→ The yacc has three parts

declarations

%%

Translation rules

%%

supporting C routines

→ The Declarations Part

→ There are two sections in the declarations Part of a YACC program (Both are optional).

→ In the first section, we put ordinary C declarations, delimeted by %{ and %}.

→ The Translation Rules Part

→ In the part of the YACC specification after the first %% pair, we put the translation rules.

→ Each Rule consists of a grammar production and associated semantic action.

<head> ⟶ <body>1 | <body>2 | ..... | <body>n

would be written in YACC as

<head> : <body1> { <semantic action1> }
| <body2> { <semantic action2> }
⋮
| <bodyn> { <semantic actionn> }
;

→ A YACC semantic action is a sequence of 'C' statements. In a semantic action, the symbol $$ refers to the attribute value associated with the Nonterminal of the head, while $i refers to value associated with i^{th} grammar symbol (Terminal or Nonterminal) of the body.

Eg:- $E \rightarrow E + T \mid T$

↓

```
expr : expr '+' term { $$ = $1 + $3; }
     | term
     ;
```

→ The 'term' is the third grammar symbol of the body, '+' is second symbol.

→ The semantic Action associated with the first production adds value of expr and term of body and assign result to expr of head.

```
$$ = $1 + $3;
```

```
$$ = $1;
```
→ default action.

The Supporting C-Routines Part

→ The third part of a YACC specification consists of supporting C-routines.

→ A lexical analyzer by name yylex() must be provided.

→ The lexical analyzer yylex() produces token consisting of a token name and its associated attribute value.

→ The attribute value associated with a token is communicated to the parser through a YACC-defined variable yylval.

YACC specification of a simple Desk calculator

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line : expr '\n'   { printf("%d\n", $1); }
     ;
expr : expr '+' term { $$ = $1 + $3; }
     | term
     ;
```

```
term : term '*' factor { $$ = $1 * $3; }
     | factor
     ;
factor : '(' expr ')'    { $$ = $2; }
       | DIGIT
       ;
%%

yylex()
{
  int c;
  c = getchar();
  if (isdigit(c))
  {
    yylval = c - '0';
    return DIGIT;
    }
  return c;
  }
```

→ YACC will ~~set~~ resolve all parsing action conflicts using two following rules:

(i) A reduce/reduce conflict is resolved by choosing the conflicating production listed first in the YACC specification.

(ii) A shift/reduce conflict is resolved in favour of shift.

Eg:-

%. left '+' '-' [left Associative]

+ & - same precedence

→ YACC resolves shift/reduce conflict by attaching a precedence & associativity to each production involved in a conflict, as well as to each terminal involved in a conflict.

→ If it must choose between shifting input symbol a and reducing by production $A \rightarrow \alpha$, YACC reduces if the precedence of the production is greater than a, (or) if the precedences are the same and the associativity of production is left. otherwise shift action is performed