Compiler Design (L.P)

9|12|10

Syllabus:
1. Lexical Analysis
2. Parsing
3. Syntax Directed Translation (SDT)
4. Intermediate code generation
5. Basics of code optimization

Translator:

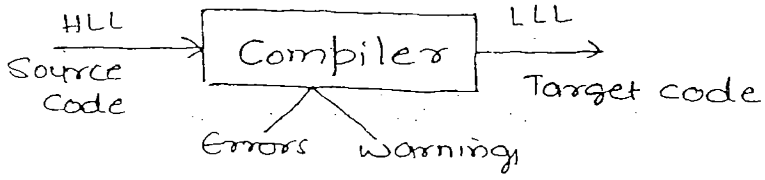
A software system which convert the source coc from one form of language to another form of language is called Translator.

Types of Transalators.

There are two type of Transalators
1. Compiler
2. Assembler.

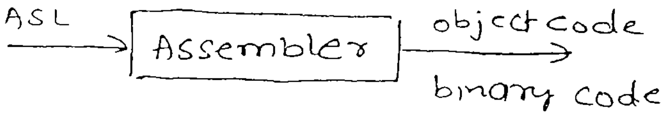
⇒ Compiler:

A transalator which convert the source cod from high level language to low level language is called compiler.

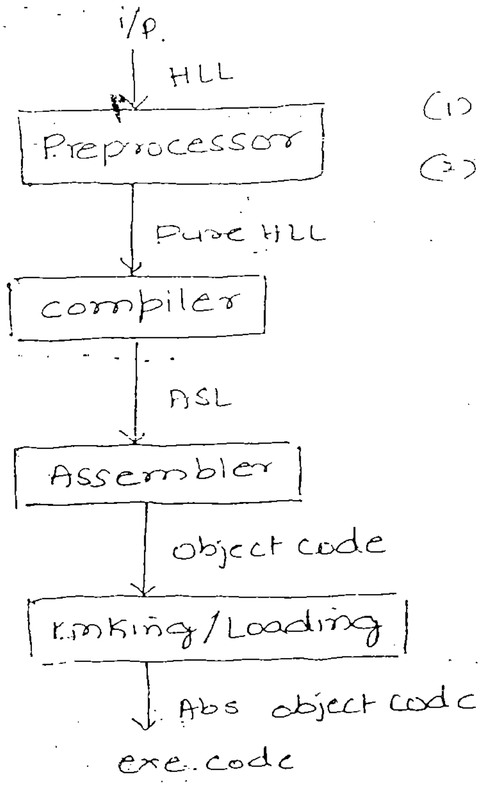


⇒ Assembler:

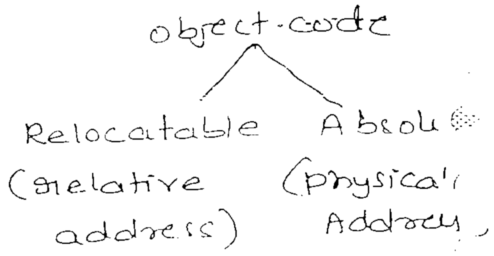
A Software S/m which converts assembly Code into object code or Binary code is called Assembler



Functions of language processing system (L.P.S):



(1) File Inclusion
(2) MACRO Evaluation



1) Preprocessor includes all the header files and also evaluates if any MACRO is included.
The preprocessor is also called as MACRO Evaluator
Preprocessing is optional i.e if any language which does not support #include and MACRO's preprocessing is not required.

2) Compiler take the preprocessor as input and converts into assembly code.

3) Assembler convert the assembly language into object code or (binary code) or (machine code)

4) Linking and Loading provides four functions.
   a. Allocation
   b. Relocation
   c. Linker
   d. Loader

Allocation : Getting the memory portions from operating system and storing the object data or object code.

Relocation:

mapping the relative address to the physical address and relocating the object code.

Linker:

Combines all executable object module to pre single executable file.

Loader:

Loading the executable file into permanent

Design of compiler:

I/p
↓
Lex. Analysis
↓
Syntax Analysis (Parsing)
↓
Symantic Analysis (SDT)
↓
Intermediate codegene
↓
code optimization
↓
Target code
↓ (ASL)
O/p

Phase:

Converting the source code from one form of representation to another form of representation is called Phase.

Lexical Analysis:

Lexical Analyser scan the source code an divide into tokens. i.e input is source code and o/p i stream of tokens.

Ex:

int	|x|	=	|20|	;
↓	↓	↓	↓	↓
kw	id	opr	const	opr

kw id opr const

2. Syntax Analysis :

Syntax analyzer verify the grammatical mistake of the source code. To verify the syntax of the source code the language must be defined by CFG. Syntax analyzer take the stream of tokens as i/p & generates the parse tree.

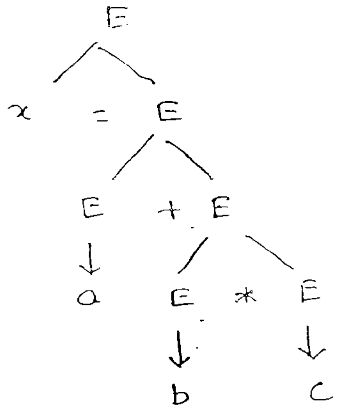
Ex: $x = a + b * c$

CFG

$E \rightarrow id$

$E \rightarrow E + E \mid E * E \mid id$

Parse Tree :



3. Symantic Analysis :

Symantic Analysis verify the meaning of each and every sentance by performing type check type

syntax analyzer just verifies whether the operator is operating on required no: of operands or not and does not look into working

ex : int x ;

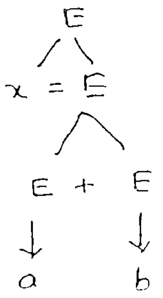
char a ;

Float b;

$x = a + b$ ⟨ No Syntax Error / Symantic Error

$E \rightarrow id = E$

$E \rightarrow E + E \mid id$



4: Intermediate code generation:

The source code is converted into interm representation to make code generation process simpl & easy & also to achieve the platform Independency.

Ex: $x = a + b * c$

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

} Intermediate code.

5. Code optimization.

Reduceing the numbers of instructions without affecting the outcome of the source program is called code optimization.

Optimization is of two type.

1. Machine independent optimization
2. Machine dependent optimization.

Machine independent optimization:

$x = a + b$; Load, ADD, move

$y = b + a$; L, A, m

$\Rightarrow$ $x = a + b$;

$y = x$;

6. Target code:

The optimizationed source code should converted into assembly code.

ex: $x = a + b * c$;

```
MUL   R1  R2 //b*c
ADD   R0  R2 //a+b*c
move  R2  x //x = a+b*c.
```

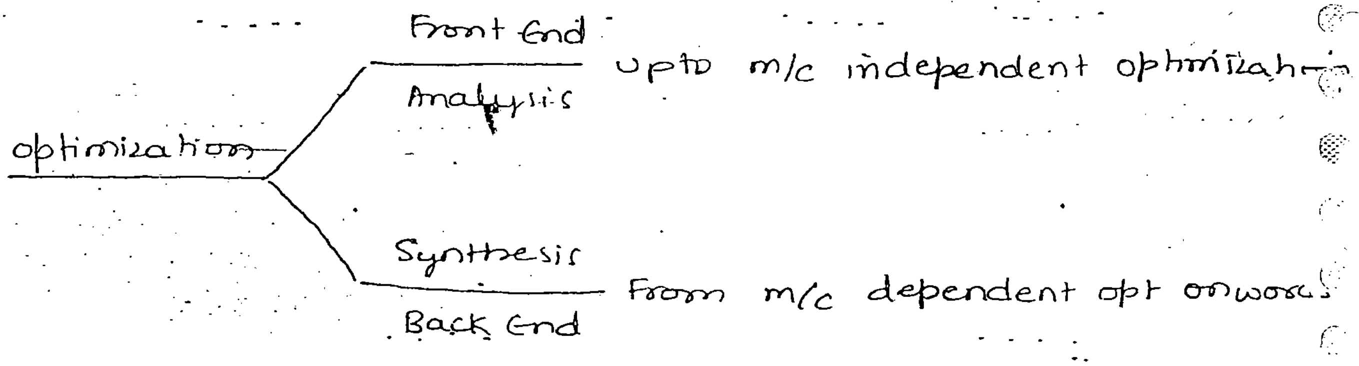
$R_0 \to a$

$R_1 \to b$

$R_2 \to c$.

Note:

Code optimization phase divide the compiler into two Parts ⓐ Front End (Analysis)

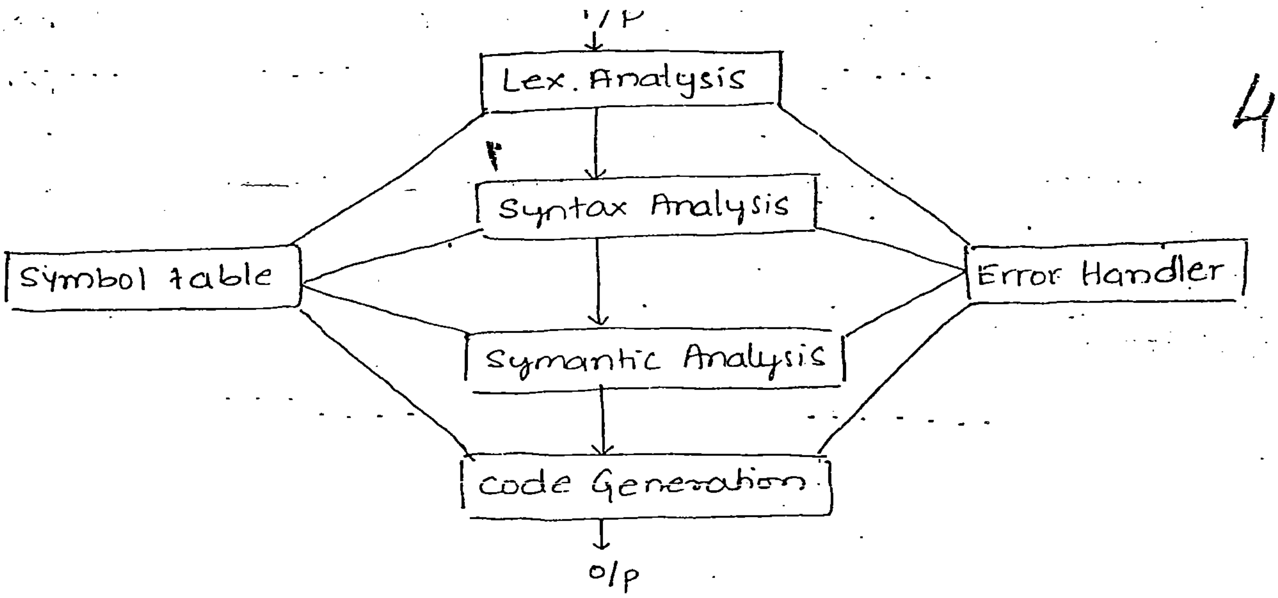


Interpreter:

Software system that convert the source code into executable code is an Interpreter.

Compiler	Interpreter.
a. I/p : HLL O/p : LLL i.e process is Translation	a. I/p : Source code O/p : result. Process is execution.
b. Scan the entire	b. scan the source code line by line.
c. Requires ~~less~~ more memory but time is less.	c. Requires less memory but time is more.
d. code optimization is possible	d. code optimization is not possible.
e. scope is static	e. scope is dynamic
f. FORTRAN is the first language	

Note:
Some languages are both compiler and Interpreter Orientation. Ex: Java.



Symbol table :

Symbol table ADT used by the complier to store the complete information of Source code. At any phase if any variable is encountered that wi [illegible] commuahng each and every phase of compiler.

Lexical Analysis is the first phase interact with Symbol table and symbol table is created by the Lexical Analysis.

compiler will provide the memory for symbol table Information of the identifier stored in the symbol :

1. Name
2. Type, size
3. Location
4. scope
5. other information like array size for arrays
6.

Symbol table can be implemented by anyone of the following data structure

1. Linear Table
2. List
3. Tree
4. Hash Table.

Operation on Symbol table

1. Insert
2. Look up
3. Modify
4. Delete.

Error Handler:

Err andler apart from collecting the errors from erent phases of and also responsible for continuing compiler even error occur.

compiler car andle three types of error.

① lexical Erro

ex: int = "Hello";

int i) =10;

(r = 20:

② Syntax Er

ex: 1. ir x =10.

2. f ;)

{

}

3. (x ;) if

{

}

③ Symanti Errors

ex: $\underset{\substack{\downarrow \\ \text{int}}}{x} = \underset{\substack{\downarrow \\ \text{char}}}{a} + \underset{\substack{\downarrow \\ \text{boolean}}}{b}$;

Apart from these three errors we may have Fatal errors which are generated by systems.

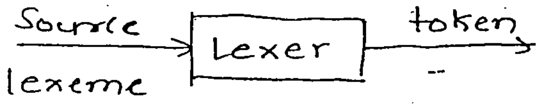
Phase(1) (Lexical Analysis):

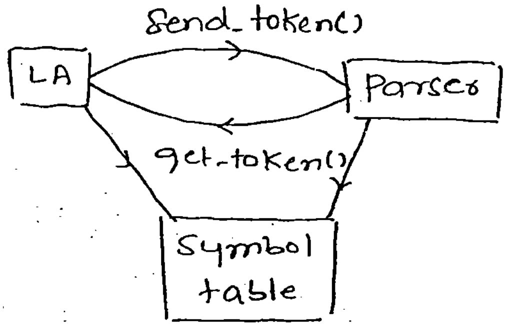
Lexical Analyzer is also called as Lexer. Lexer read the source code and divide the source code tokens. where token is a group of characters with logical meaning.

lexeme : is the actual representation of stream of Charac

Ex: 1. $PI = 3.1415$
↓
id

The lexer take the stream of lexeme and o/p strear of tokens.





Construction of lexical Analyzer.

1. Define the rule based on the i/p stream these rules are pattern recognizing tools.
2. Construct the Regular Expression
3. Convert the RE into Finite Automata.

They are two ways for the construct of lexer

① Hand code

② Lex tool.

Lexical Analyzer is also called as Scaner or tokenizer.

Find the no. of tokens in the following source code.

1. Printf | (| " | TOC " |) | ; |

1 2 3 4 5

No. of tokens = 5.

2. DO I = 15.5 ;

1 2 3 4 5

3. int Add (int x , int y)

1 2 3 4 5 6 7 8 9

{ return x + y ;

10 11 12 13 14 15

}

16

4. Printf (" i = %d , $i = %9 " , i , &i) ;

1 2 3 4 5 6 7 8 9 10

Secondary function of lexical Analyzer:

1 Remove the comment lines and whitespace characters

2. Correlating the Error messages by tracking the line no.

***

PARSING:

The processes of constructing the parse tree for a given input string is called Parsing.

To parse any i/p string the language must be defined by Context free grammer (CFG).

Grammer:

Finite set of rules that may generate infinite no. of sentances is called grammer.

(or)

Grammer is a 4 tuple G = (V, T, P, S)

V = set of all Non-terminals

T = set of all terminals

P = set of all productions

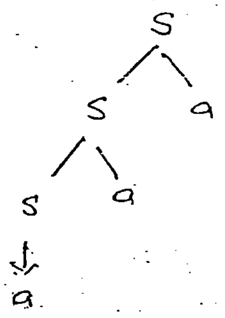
S - start symbol.

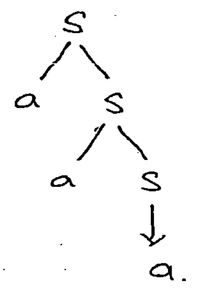
Ex: $S \to aS \mid Sa \mid a$

$w = aaa$

LMD: $S \to Sa$
$\to Saa$
$\to aaa$

RMD: $S \to aS$
$\to aaS$
$\to aaa$





Classification of the Grammer:

1. Based on the No of derivation Trees (classification - I)
2. Based on the No. of strings (classification - II)

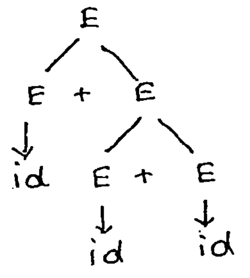
Classification - I

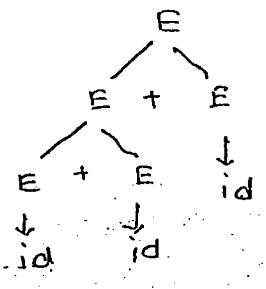
~~1. Ambigeous Grammer~~
2. UnAmbigeous Grammer.

Ambigeous Grammer: The Grammer 'G' is said to be Ambigeous if there exist more than one derivation tree for the given i/p string more than one LMDT or RMDT.

Ex: $E \to E + E / id$

$w = id + id + id$





⇒ The grammer 'G' is Ambigeous

NOTE.

The grammer whose productions are in the form

$A \to \alpha \quad A \in V, \alpha \in (V+T)^*$ is called CFG.

Ex: ① $E \to E+E \mid E*E \mid id$.

$\Sigma = \{+, *, id\}$

② $E \to E+E$

$\to (E)$

$\to E==E$

$\to id$

$T = \{+, ==, (,), id\}$

③ $S \to A \% B + C (D)$

$A \to id$

$B \to id$

$C \to id$

$D \to id$

$\Sigma = \{\%, +, (,), id\}$.

Derivation:

The process of deriving a string is called derivation and geometrical representation is called Derivation tree/parse tree/syntax Tree.

All the intermediate steps in the derivation is called sentantial form.

$E \to E+E$

$\to E+E*E$

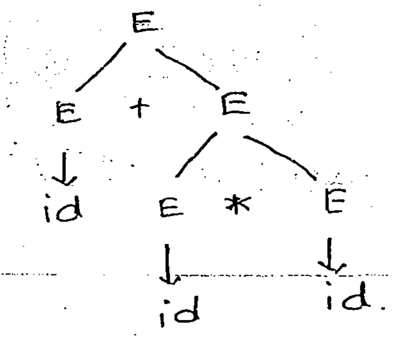
$\to id+E*E$

$\to id+id*E$

$\to id+id*id$

it is sentential form

Parse Tree



Types of derivation:

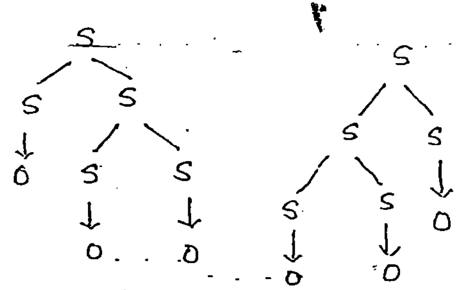
1. LMD (Left most derivation)
2. RMD (Right most derivation).

LMD: The process of deriving a string by expanding the left most non terminals is called LMD. and geometrical representation of LMD is called LMDT.

RMD: The process of deriving a string by expanding right most variables is called RMD and geometrical representation of RMD is called RMDT

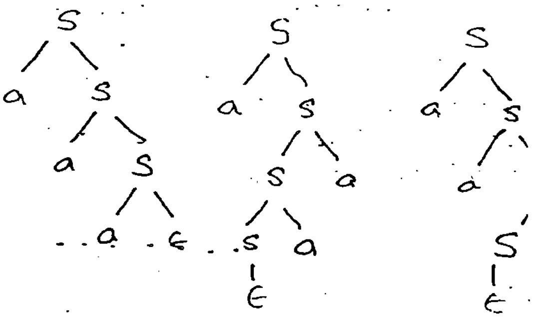
2. $S \to SS|0$

$w = 000$



3. $S \to aS|Sa|\epsilon$

$w = aaa$



No. of derivation = 6.

Note:

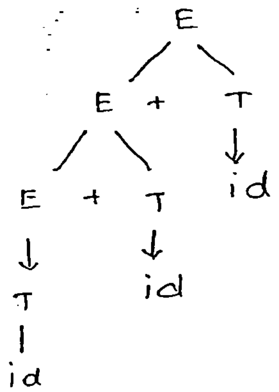
1. In ambigeous Grammer both LMD & RMD represent different parse tree LMDT ≠ RMDT
2. The ambiguity of CFG is Undecidable.
3. Ambiguity can be eliminated by rewriting the Gramm

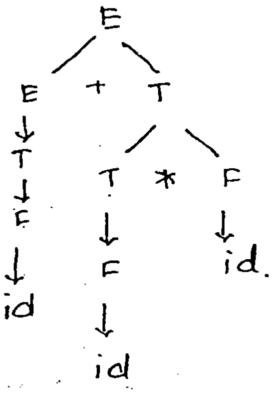
Ex: $E \to E+E/id$

$E \to E+T/T$
$T \to id$
$w = id+id+id$ } UAG.

2. $E \to E+E|E*E|id$ AG.

$E \to E+T/T$
$T \to T*F/F$
$F \to id$ } UAG.

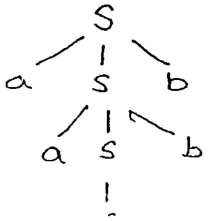




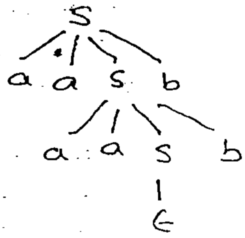
Unambigeous Grammer:

The grammer 'G' is said to be Unambigeous if ∃ unique derivation tree for every i/p string.

Ex: $S \to aSb|\epsilon$
$w = aabb$ UAG



② $S \to aaSb|\epsilon$
$w = aaabb$ UAG.



verfiying the following grammer is ambigeous or Unambi

1. $S \rightarrow A|a$
   $A \rightarrow a$ (A)

2. $S \rightarrow SaS|b$ (A)

3. $S \rightarrow SSS|a$ (A)

4. $S \rightarrow SaSb|SbSa|\epsilon$ (A)

5. $A \rightarrow A(A)|a$ (A)

6. $A \rightarrow AA|(A)|a$ (A)

7. $S \rightarrow aSbS|\epsilon$ (A)

8. $S \rightarrow SS|AB$
   $A \rightarrow Aa|a$
   $B \rightarrow Bb|b$ (A)

9. $S \rightarrow (L)|a$
   $L \rightarrow L,S|S$ (UA)

10. $S \rightarrow AA$
    $A \rightarrow aA$
    $A \rightarrow b$ (UA)

Classification II rule :

① Recursive Grammer:

② Non - Recursive Grammer.

Recursive Grammer

The grammer 'G' is said to be recursive if ∃ atleast one production which as same variable both at LHS and RHS.

Ex: ① $S \rightarrow Sa|b$ ② $S \rightarrow aS|b$ ③ $S \rightarrow aSb|\epsilon$

Non recursive Grammer

The grammer 'G' is said to be non recursive if No production contains same variable both at LHS & RHS

Ex: ① $S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

② $S \rightarrow A10B$
$A \rightarrow 01|10$
$B \rightarrow 10|C$
$C \rightarrow 0|1$

Note:

1. The grammer 'G' is recursive iff it generates infinite langu
2. The grammer 'G' is non-recursive iff it generates finite langua

Types of Recursion:

① Left Recursion

② Right Recursion.

Left Recursion:

The grammer is said to be left recursio if the left most variable of RHS is same as the vario at LHS.

Ex: 1. $S \rightarrow Sa \mid b$

2. $S \rightarrow AB$
$A \rightarrow Aa \mid b$
$B \rightarrow aB \mid a$

3. $E \rightarrow E+T \mid T$
$T \rightarrow id.$

Right Recursion:

The grammer is said to be right recurs if the right most variable of RHS is same as the variable at LHS.

Ex: 1. $S \rightarrow aS \mid b$

2. $S \rightarrow$

3. $E \rightarrow T+E \mid T$
$T \rightarrow id.$

Note:-

1. The grammer which is both left and Right recursi is Ambigeous.

ex: $S \rightarrow SS \mid AB$
$A \rightarrow Aa \mid a$
$B \rightarrow Bb \mid b$

---

1. The Grammer $S \rightarrow SS \mid 0$ is

a. No ambigeous [✓ b. Ambigeous] c. left recursion d. right recu

2. $A \rightarrow AA \mid (A) \mid a$

a. No ambigeous [✓ b. Ambigeous] c. left recursion d. right recu

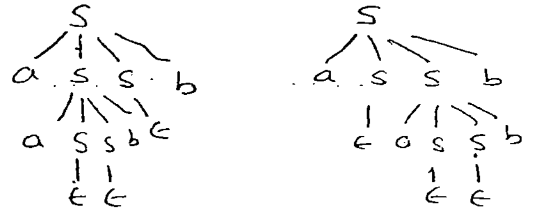
2. The recursion which neither left nor right is called General recursion.

ex: ① $A \rightarrow (A) \mid a$ ② $S \rightarrow aSb \mid bSa \mid \epsilon$ ③ $S \rightarrow aSSb \mid \epsilon$

It is of general recursion ~~the~~ but the grammer is ambiqeous

ex: $S \rightarrow \underline{a}SSb \mid \epsilon$

$w = aabb$



3. If the grammer is left recurlive then the Parser go to infinite loop. So to avoid the looping we need to conv the left recursive grammer into right recursive gram

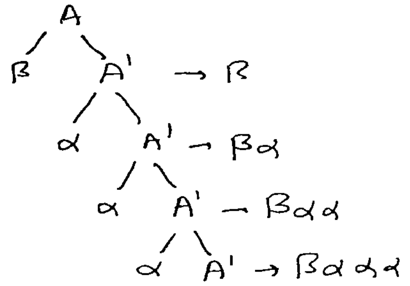
Conversion of left recursion into Right recursion.

LRG → RRG.

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

$\Rightarrow A \rightarrow \beta A'$

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



$\boxed{\beta\alpha^*}$

2. $A \rightarrow A\alpha \mid \beta_1 \mid \beta_2$

$\Rightarrow A \rightarrow \beta_1 A' \mid \beta_2 A'$

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

3. $A \rightarrow A\alpha \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_n$

$\Rightarrow A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$

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

4. $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \dots \mid A\alpha_n \mid \beta$

$\Rightarrow A \rightarrow \beta A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \epsilon$

5. $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \dots \mid \beta_n$

$\Rightarrow A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A'$

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

Remove the left recursion for the following Grammer

1. $A \rightarrow Aa \mid b$

$\Rightarrow A \rightarrow bA'$

$A' \rightarrow aA' \mid \epsilon$

2. $S \rightarrow S\underbrace{S}_{\alpha} \mid \underbrace{0}_{\beta}$

$S \rightarrow 0S'$

$S' \rightarrow SS' \mid \epsilon$

3. $S \rightarrow S\underbrace{SS}_{\alpha} \mid \underbrace{0}_{\beta}$

$S \rightarrow 0S'$

$S' \rightarrow SSS' \mid \epsilon$

4. $S \rightarrow S\underbrace{0S1}_{\alpha} \mid \underbrace{\epsilon}_{\beta}$

$S \rightarrow \epsilon S'$

$S' \rightarrow 0S1S' \mid \epsilon$

5. $S \rightarrow S\underbrace{aSb}_{\alpha_1} \mid S\underbrace{bSa}_{\alpha_2} \mid \underbrace{\epsilon}_{\beta}$

$S \rightarrow \epsilon S'$

$S' \rightarrow aSbS' \mid bSaS' \mid \epsilon$

6. $E \rightarrow E\underbrace{+E}_{\alpha_1} \mid E\underbrace{*E}_{\alpha_2} \mid$

$E \rightarrow idE'$

$E' \rightarrow +EE' \mid *EE' \mid$

7. $A \to A\underset{\alpha_1}{\underline{bd}} \mid A\underset{\alpha_2}{\underline{cB}} \mid \underset{\beta_1}{\underline{d}} \mid \underset{\beta_2}{\underline{f}} \mid \underset{\beta_3}{\underline{e}}$

$A \to dA' \mid fA' \mid eA'$

$A' \to bdA' \mid cBA' \mid \epsilon$

8. $E \to E\underset{\alpha}{\underline{+T}} / \underset{\beta}{\underline{T}}$ (LR)

$T \to T\underset{\alpha}{\underline{*F}} \mid \underset{\beta}{\underline{F}}$ (LR)

$F \to (E) \mid id$

Sol. $E \to TE'$

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

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

$F \to (E) \mid id.$

9. $A \to A(A) \mid a$

$A \to aA'$

$A' \to (A)A' \mid \epsilon$

10. $S \to (L) \mid a$

$L \to L\underset{\alpha}{\underline{,S}} \mid \underset{\beta}{\underline{S}}$

Sol

$S \to (L) \mid a$

$L \to SL'$

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

11. $S \to AA \mid a$

$A \to Ab \mid SAb \mid b$

Sol: $S \to AA \mid a$

$A \to A\underset{\alpha_1}{\underline{b}} \mid A\underset{\alpha_2}{\underline{AAb}} \mid \underset{\beta_1}{\underline{aAb}} \mid \underset{\beta}{\underline{b}}$

$\Rightarrow$ $S \to AA \mid a$

$A \to aAbA' \mid bA'$

$A' \to bA' \mid AAbA' \mid \epsilon$

Non Deterministic Grammer:

The grammer with common prefixes is called non deterministic grammer.

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$$

non determinisitic

Note:

1. The grammer with common prefixes requires backtracki
2. Backtracking is costly
3. To avoid the backtracking we need to convert the non-deterministic grammer into deterministic we need to perform left factoring

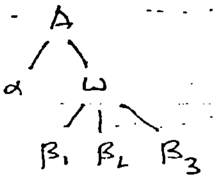
Left factoring:

The process of conversion of grammer with common prefixes into deterministic grammer is called left factoring

Ex: $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

$\to A \to \alpha\omega$

$\omega \to \beta_1 \mid \beta_2 \mid \beta_3$



Ex. ① $S \to \underline{a}Sb \mid \underline{a}bS \mid \underline{a}b$

$S \to aS'$

$S' \to Sb \mid bS \mid b$

2. $S \to abcd \mid abce \mid abf \mid ag$

(1) $S \to abcS' \mid abf \mid ag$

$S' \to d \mid e$

(2) $S \to abS'' \mid ag$

$S'' \to cS' \mid f$

$S' \to d \mid e$

(3) $S \to aS'''$

$S''' \to bS'' \mid g$

$S'' \to cS' \mid f$

$S' \to d \mid e$

(4) Dangling else problem - Defining the grammer for Conditional statement.

$S \to iEtS \mid iEtSES \mid \epsilon$

$E \to b$

Sol: $S \to iEtSS' \mid \epsilon$

$S' \to \epsilon \mid ES$

$E \to b.$

classification of parsers:

① Top Down parser (TDP)

② Bottom up parser (BUP)

Top Down parser (TDP):

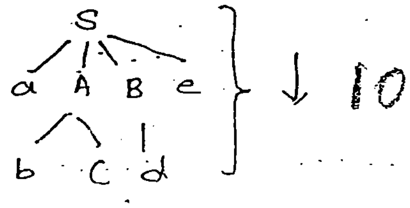
The process of construction of parse tree starting from root and proceed to children is called TDP. i.e starting from start symbol of the grammer and reaching the i/p string.

Ex: $S \rightarrow aABe$

$A \rightarrow bc$

$B \rightarrow d$

$w = abcde$



Note:

1. TDP internally uses lelf most derivation.
2. TDP constructed for the grammer if it is free from ambiguity and left recursion.
3. TDP used for the grammer with less complexity if complexity is more then the parsing mechanism be slow and hence performance is low.
4. Average Time complexity is $O(n^4)$

Classification of Top down parser

① with Back Tracking → BruteForce Technique

② without Back Tracking → predective parsers

① LL(1) or Non-recursive

② Recursive Descent parser.

BruteForce Technique:

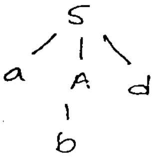
whenever a non terminal is expanding first time, go with the first alternative and compare with the input string if does not matches go with the second alternative and compare with the input string if does not matches go with the third and so on and continue with all the alternatives if atleast one alternative is matches the i/p string then i/p string is parsed succesfully otherwise the string can not be parsed.

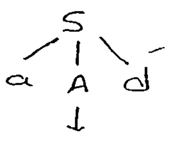
Ex: $S \rightarrow aAd \mid aB$

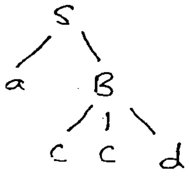
$A \rightarrow b \mid c$

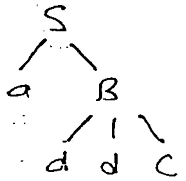
$B \rightarrow ccd \mid ddc$

$w = \overset{\uparrow}{a}ddc.$





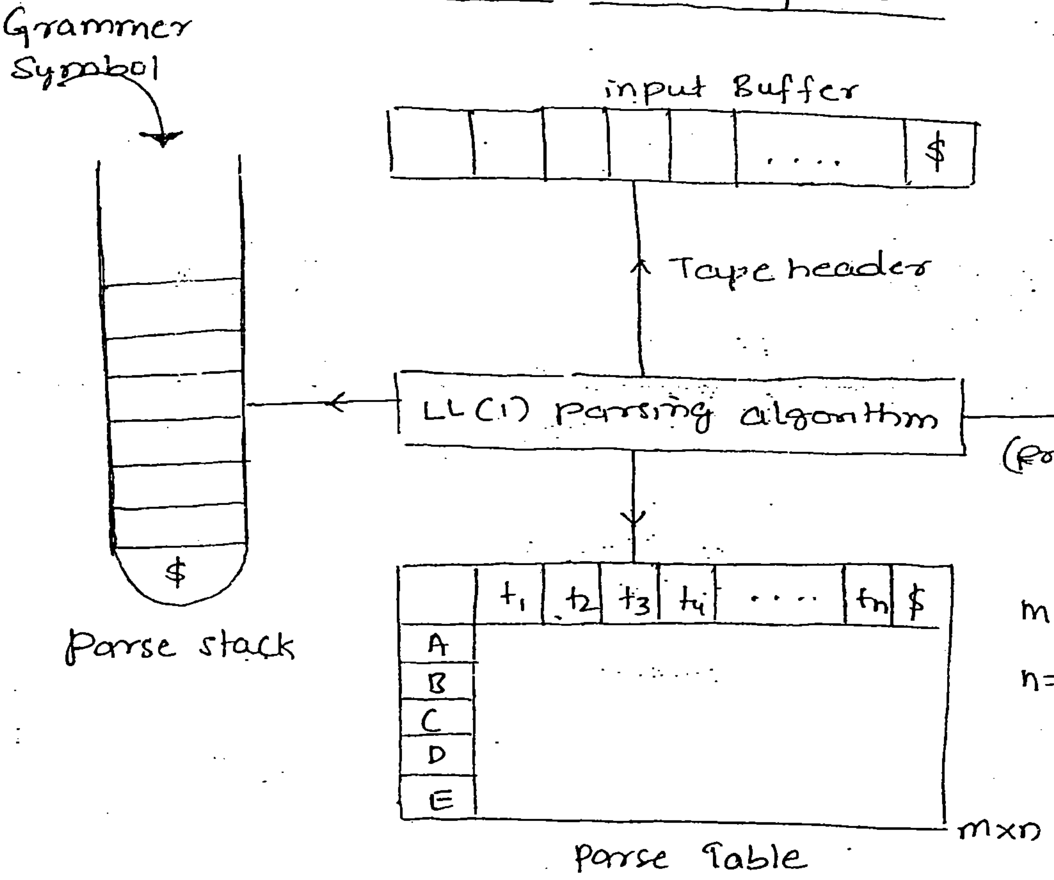




LL(1) parser (or) Table driven parser :

LL(1)
- 1 → No. of symbol taken into count for parsing
- L → Left to Right.
- L → LMD (Left most derivation)

Block Diagram of LL(1) parser :



m = No. of Non-termin

n = No. of terminals

LL(1) has
1. Input Buffer
2. Parse stack
3. Parse table

Input Buffer : It is divided into cells and each cells is capable of holding only one symbol.

Input Buffer contains only one

The Tape header always pointing only one symbol at a time, The symbol which is pointed by tape header is called look ahead symbol.

Parse stack:

It contains the grammer symbols. The grammer symbol are pushed ~~pto~~ into stack or poped from the stack based on the occurance of the matching. i.e if there is no matching b/w the top most symbo of the stack and look ahead symbol then the grammer symbols are pushed into stack. if matchi occurs b/w the top most symbol of the stack and look ahead symbol then the grammer symbols are poped from the stack.

Parse Table:

It is a two dimensional array of order m×n where m = No. of non-terminals & n = no. of terminals + 1. It contains all the productions which are used in the parsing to push the grammer symbols into stack.

Parsing process:

step (1): push the start symbol into the stack.

step (2): Compare the topmost symbol of the stack with the look ahead symbol.

step (3): If matching occurs then top of the grammer symbol and increment the i/p pointer. & continue the process.

step (4): Output the production which is used for the expanding a non-terminal and continue.

LL(1) parsing Algorithm:

Let x be the grammer symbol (start symbol in the stack and a is the look ahead symbol.

(1) If x = a = $ then, it is succesfull in parsing.

(2) If x = a ≠ $ then pop off and increment the i/p pointe

(3) If x ≠ a ≠ $ & m[x, a] has the production x → uvw then replace by uvw in the reverse order & continue the proc

(4) output the production which is used for expanding the non-terminal

Ex: S→AA

A→aA   w= abab.

A→b

Stack	i/p string	Action.
$S	abab$ ↑	push (S→AA)
$AA	abab$ ↑	push (A→aA)
$AAa	~~a~~bab$	pop
$AA	bab$	push (A→b)
$A~~b~~	~~b~~ab$	pop
$A	ab$	push (A→aA)
$Aa	~~a~~b$	pop
$A	~~b~~$	push (A→b)
$~~b~~	~~b~~$	pop
$	$	Accept.

	a	b	$
S	S→AA	S→AA	
A	A→aA	A→b	

LL(1) Grammer:

The grammer ~~'G' is said to be~~ for which LL(1) parser can be constructed is known as LL(1) grammer.

(or).

The grammer whose parse table does not Contai the multiple entries is called LL(1) grammer

Functions used for the construction of Parse table

① First (α) → Terminal/NonTerminal.

② FOLLOW (A). Non Terminal.

FIRST(α):

First (α) is the set of all terminals that may begin in any sentential form which is derived from α

Rules:

1. If α is a terminal then

$$First(\alpha) = \{\alpha\}$$

2. If α is a Non terminal and defined by the rule

then $$First(\alpha) = \{\epsilon\}$$

3. If α is a Non terminal and defined by non null pr

i.e $$\alpha \rightarrow x_1 x_2 x_3$$

$$First(\alpha) = First(x_1)$$
$$= First(x_1) \cup First(x_2)$$
$$= First(x_1) \cup First(x_2) \cup First(x_3)$$
$$= First(x_1) \cup First(x_2) \cup First(x_3) \cup \{\epsilon\}$$

Ex: ① $S \rightarrow \epsilon$

$First(S) = \{\epsilon\}$

② $S \rightarrow a|\epsilon$

$First(S) = \{a, \epsilon\}$

③ $S \rightarrow aA|\epsilon$ $F(S) = \{a, \epsilon\}$

$A \rightarrow b$ $F(A) = \{b\}$

④ $S \rightarrow aAB$ $F(S) = \{a\}$

$A \rightarrow bB|c$ $F(A) = \{b,$

$B \rightarrow dA|a$ $F(B) = \{d,$

⑤ $S \rightarrow aAB$ $F(S) = \{a$

$A \rightarrow b|\epsilon$ $F(A) = \{b.$

$B \rightarrow a$ $F(B) = \{$

⑥ $S \rightarrow AB$ $F(S) = \{a, b, \epsilon\}$

$A \rightarrow a|\epsilon$ $F(A) = \{a, \epsilon\}$

$B \rightarrow b|\epsilon$ $F(B) = \{b, \epsilon\}$

⑦ $S \rightarrow ABCDE.$

$A \rightarrow a|\epsilon$

$B \rightarrow b|\epsilon$

$C \rightarrow c|\epsilon$

$D \rightarrow d$

$E \rightarrow e|\epsilon.$

	First
S	$\{a, b, c, d$
A	$\{a, \epsilon\}$
B	$\{b, \epsilon\}$
C	$\{c, \epsilon\}$
D	$\{d\}$
E	$\{c, \epsilon\}$

⑧ $S \rightarrow ABCDE$

$A \rightarrow a|\epsilon$

$B \rightarrow b|\epsilon$

$C \rightarrow c|\epsilon$

$D \rightarrow d|\epsilon$

$E \rightarrow e|\epsilon$

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

⑨

$S \rightarrow aABd$. $F(S) = \{a\}$

$A \rightarrow BAb|\epsilon$ $F(A) = \{\epsilon, d, b\}$

$B \rightarrow Adb|\epsilon$ $F(B) = \{\epsilon, d, b\}$

Find the first set for the following grammers.

1. $A \rightarrow A(A) \mid a$ Not LL(1)

Soln:

$A \rightarrow aA'$

$A' \rightarrow (A)A' \mid \epsilon$

$First(A) = \{a\}$

$First(A') = \{c, \epsilon\}$

② $A \rightarrow \underbrace{AA}_{\alpha} \mid \underbrace{(A)}_{\beta_1} \mid \underbrace{a}_{\beta_2}$

$A \rightarrow (A)A' \mid aA'$

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

$First(A) = \{c, a\}$

$First(A') = \{\epsilon, a, c\}$

③ $E \rightarrow E + T \mid T$

$T \rightarrow id$

Sol:

$E \rightarrow TE'$

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

$T \rightarrow id$

$First(E) = \{id\}$

$First(E') = \{\epsilon, +\}$

$First(T) = \{id\}$

④ $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid a \mid b$

Sol:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid a \mid b$

$First(E) = \{c, a, b\}$

$First(E') = \{+, \epsilon\}$

$First(T) = \{c, a, b\}$

$First(T') = \{*, \epsilon\}$

$First(F) = \{c, a, b\}$

⑤ $S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

Sol:

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

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

$First(S) = \{c, a\}$

$First(L) = \{c, a\}$

$First(L') = \{,, \epsilon\}$

⑥ $S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

$First(S) = \{a, b\}$

$First(A) = \{a, b\}$

⑦ $S \rightarrow aBDh$ $F(S) = \{a\}$

$B \rightarrow cC$ $F(B) = \{c)$

$C \rightarrow bC \mid \epsilon$ $F(C) = \{b, \epsilon\}$

$D \rightarrow EF$ $F(D) = \{g, f, \epsilon\}$

$E \rightarrow g \mid \epsilon$ $F(E) = \{g, \epsilon)$

$F \rightarrow f \mid \epsilon$ $F(F) = \{f, \epsilon\}$

FOLLOW(A):

FOLLOW(A) is the set of all the terminals that ma follow to the right of A in any sentential form of th grammer.

Rules to find FOLLOW(A):

1. If A is a start symbol, then

$$\text{FOLLOW}(A) = \{\$\}$$

2. If $S \to \underbrace{\alpha A \beta}_{\text{grammer Symbols}} \quad (\beta \neq \epsilon)$

$$\text{follow}(A) = \text{First}(\beta) \ (\text{without } \epsilon)$$

3. If $S \to \alpha A$ or

$S \to \alpha A B \quad (B \overset{*}{\Rightarrow} \epsilon)$

then, $\boxed{\text{follow}(A) = \text{Follow}(S).}$

ex: ①
$S \to aA$
$A \to b$

Follow(S) = { \$ }
Follow(A) = { \$ }

②
$S \to aAb$
$A \to Ba \mid b$
$B \to d$

Follow(S) = { \$ }
Follow(A) = { b }
Follow(B) = { a }

③ $S \to S0S1 \mid \epsilon$

Follow(S) = { \$, 0, 1 }

④ $S \to AaAb \mid BaBb$
$A \to \epsilon$
$B \to \epsilon$

Follow(S) = { \$ }
Follow(A) = { a, b }
Follow(B) = { a, b }

⑤
$S \to AB$
$A \to a \mid \epsilon$
$B \to b \mid \epsilon$

follow(S) = { \$ }
follow(A) = { b, \$ }
follow(B) = { \$ }

⑥
$S \to ABCDE$
$A \to a \mid \epsilon$
$B \to b \mid \epsilon$
$C \to c \mid \epsilon$
$D \to d \mid \epsilon$
$E \to e$

	Follow
S	{ \$ }
A	{ b, c, d, e }
B	{ c, d, e }
C	{ d, e }
D	{ e }
E	{ \$ }

8. $S \to aBDh$

$B \to cC$

$C \to bC | \epsilon$

$D \to EF$

$E \to g | \epsilon$

$F \to f | \epsilon$

Follow (S) = { $ }

Follow (B) = {g, f, h}

Follow (C) = {g, f, h}

Follow (D) = {h}

Follow (E) = {f, h}

Follow (F) = {h}

9. $A \to (A).B | a$

$B \to aBa | b$

Follow (A) = { $,) }

Follow (B) = {a, $,)}

10. $S \to aBCbD$

$B \to bBh | BD | d$

$C \to CE | a$

$D \to EC | b | \epsilon$

$E \to CDb | \epsilon$

Follow (S) = { $ }

Follow (B) = {a, h, b}

Follow (C) = {b, a}

Follow (D} = { $, a, h, b }

Follow (E) = { b, a }

construct the following set for e and every variables for the fol' grammar after elimination of left recursion.

1. $E \to E + T | T$ } not LL(1)
$T \to id$

Sol:

$E \to TE'$

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

$T \to id$

Fol (E) = { $ }

Fol (E') = { $ }

fol (T) = { +, $ }

2. $E \to E+E | E*E | id$

$E \to idE'$

$E' \to +EE' | *EE' | \epsilon$

fol (E) = { $, +, * }

fol (E') = { $, +, * }

3. $E \to E+T | T$

$T \to T*F | F$

$F \to (E) | id$

Sol: $E \to TE'$

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

$T \to FT'$

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

$F \to (E) | id$.

Fol (E) = { $,) }

Fol (E') = { $,) }

Fol (T) = { +, $,) }

Fol (T') = { +, $,) }

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

4. $S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$

fol(S) = { \$ }

fol(A) = {a, b, \$ }

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

$L \rightarrow L, S \mid S$

sol: $S \rightarrow (L) \mid a$

$L \rightarrow S L'$

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

fol (L) = { \$,), , }

fol (L) = { \$,) }

fol (L') = { \$,) }

6. $R \rightarrow R+R \mid RR \mid R* \mid (\underset{\beta_1}{\underline{R}}) \mid \underset{\beta_2}{\underline{a}} \mid \underset{\beta_3}{\underline{b}}$

Sol: $R \rightarrow (R) R' \mid aR' \mid bR'$

$R' \rightarrow +RR' \mid RR' \mid *R' \mid \epsilon$

fol (R) = { \$,), +, *, (, a, b }

fol (R') = { fol(R) }

7. $S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

fol (S) = { \$, e }

fol (S') = { \$, e }

fol (ϵ) = { t }

Procedure to construct the LL(1) parser.

∴ For every production $A \to \alpha$ repeat the following steps

1. Add $A \to \alpha$ in $m[A, a]$ for every symbol a in First(α)
2. First (α) contains ϵ and Add $A \to \alpha$ in $m[A, b]$ for every symbol b in FOLLOW (A).

Note:

The grammer 'G' is LL(1) iff its parse table does not contains multiple entries.

1. ex:

$S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

	First
S	{i, a}
S'	{e, ϵ}
E	{b}

Fol (S) = {\$, e}

fol (S') = {\$, e}

fol (E) = {t}

$q = \{i, t, a, e, b, \$\}$

	i	a	b	e	t	\$
S	$S \to iEtSS'$	$S \to a$				
S'				$S' \to eS$ $S' \to \epsilon$		$S' \to \epsilon$
E			$E \to b$			

∴ Gramm is not LL(1).

2. $R \to R+R \mid RR \mid R* \mid (R) \mid a \mid b$ (Not LL(1))

$R \to (R)R' \mid aR' \mid bR'$

$R' \to +RR' \mid RR' \mid *R' \mid \epsilon$

F(R) = {(, a, b}

F (R') = {*, +, ϵ, (, a, b}

fol(R) = {\$,), +, *, (,

fol(R') = {fol (R

	(	)	+	*	a	b	\$
R	$R \to (R)R'$				$R \to aR'$	$R \to bR'$	
R'	$R' \to RR'$ $R' \to \epsilon$	$R' \to \epsilon$	$R' \to \epsilon$ $R' \to +RR'$	$R' \to \epsilon$ $R' \to *R'$	$R' \to \epsilon$ $R \to RR'$	$R' \to \epsilon$ $R \to RR'$	$R' \to \epsilon$

not LL(1)

3. S → (L) | a

L → L , S | S

Sol: S → (L) | a

L → S L'

L' → , S L' | ε

F(S) = { (, a }

F(L) = { (; a }

F(L') = { , , ε }

fol(S) = { $,), , }

fol(L) = { $,) }

fol(L') = { $,) }

15

	(	)	a	,	$
S	S → (L)		S → a		
L	L → SL'		L → SL'		
L'		L' → ε		L' → , SL'	L' → ε

∴ LL(1)

4. S → AA

A → aA

A → b

F(S) = {a, b}

F(A) = {a, b}

Fol(S) = { $ }

Fol(A) = { a, b, $ }

	a	b	$
S	S → AA	S → AA	
A	A → aA	A →	

∴ LL(1).

5. E → E + T | T

T → int

Sol: E → TE'

E' → + TE' | ε

T → int

F(E) = { int }

F(E') = { +, ε }

F(T) = { int }

Fol(E) = { $ }

fol(E') = { $ }

~~fol(T) = { +, $ }~~

	+	int	$
E		E → TE'	
E'	E' → +TE'		E' → ε
T		T → int	

6. $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

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

	id	+	*	(	)	\$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Note:

Shortcut to find LL(1) parser

1. A grammer without ε rule is LL(1) for each production of the form $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \dots \mid \alpha_n$

$$First(\alpha_1) \cap First(\alpha_2) \cap \dots \cap First(\alpha_n) = \phi$$

i.e $F(\alpha_1), F(\alpha_2), \dots F(\alpha_n)$ are mutually disjoint.

$$First(\alpha_i) \cap First(\alpha_j) = \phi$$

2. The grammer with ε rule is LL(1) for each rule of the form $A \rightarrow \alpha \mid \epsilon$

$$First(\alpha) \cap Follow(A) = \phi$$

3. Ambigious grammer is not LL(1)

4. Left recursive grammer is not LL(1)

5. Non left factor grammer is not LL(1)

16

A → c
B → d
⋮
2 → .

6. The grammer in which every production has only on alternative RHS is always LL(1)

ex: $S \to AB$
$A \to aB$
$B \to b$

	a	b	\$
S	$S \to AB$	$S \to AB$.	
A	$A \to aB$		
B		$B \to b$	....

2. $S \to aA \mid bB$
$A \to Bb \mid a$
$B \to bB \mid c$

$\{a\} \cap \{b\} = \phi$
$\{b, c\} \cap \{a\} = \phi$
$\{b\} \cap \{c\} = \phi$

⇒ So grammer is LL(1)

3. $S \to AB \mid BA$
$A \to a \mid b$
$B \to b \mid a$

$\{a, b\} \cap \{a, b\} \neq \phi$

⇒ So grammer is not LL(1)

4. $S \to aAbB$
$A \to a \mid \epsilon$
$B \to b \mid \epsilon$

$\{a\} \cap \{b\} = \phi$
$\{b\} \cap \{\$\} = \phi$

⇒ So grammer is LL(1).

5. $S \to AaAb \mid BbBa$
$A \to \epsilon$
$B \to \epsilon$

$\{a\} \cap \{b\} = \phi$

⇒ So grammer is LL(1)

6. $S \to AaAb \mid BaBb$
$A \to \epsilon$
$B \to \epsilon$

$\{a\} \cap \{a\} \neq \phi$.

Verify the following grammers is LL(1) or Not.

1. $S \to aABb$
$A \to c \mid \epsilon$
$B \to d \mid \epsilon$

$\{c\} \cap \{d, b\} = \phi$
$\{d\} \cap \{b\} = \phi$

⇒ So grammer is LL(1)

2. $S \rightarrow ACB \mid CbB \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \epsilon$

$C \rightarrow b \mid \epsilon$

$\{d, g, b\} \cap \{b \dots\} \neq \phi$

⇒ So grammer is not LL(1).

3. $S \rightarrow \underline{1}AB \mid \epsilon$

$A \rightarrow \underline{11}AC \mid 0C$

$B \rightarrow 0S$

$C \rightarrow 1$

$1 \cap \epsilon = \phi$

LL(1)

GATE:

1. $S \rightarrow A \mid a$

$A \rightarrow a$

⇒ Not LL(1)

2. pgm → begin d semi x End

x → d semi x | s y

y → semi s y | ε

{d} ∩ {s} = ϕ.

{semi} ∩ {end} = ϕ.

⇒ LL(1)

3. find the first and follow

04

$E \rightarrow aA \mid (E)$

$A \rightarrow +E \mid *E \mid \epsilon$

	first
E	{a, (}
A	{+, *, ε}

	Follow
E	{$,)}
A	{$,)}

4. $S \rightarrow iEtSS' \mid a$

05

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

$\{i\} \cap \{a\} = \phi$

$\{e\} \cap \{e\} \neq \phi.$

⇒ Not LL(1)

5. $A \rightarrow AA \mid (A) \mid \epsilon$

06

ⓐ Left recursion ⓑ Right recursion [✓ ⓒ Ambiguous] ⓓ None

06. $S \rightarrow FR$

$R \rightarrow *S \mid \epsilon$

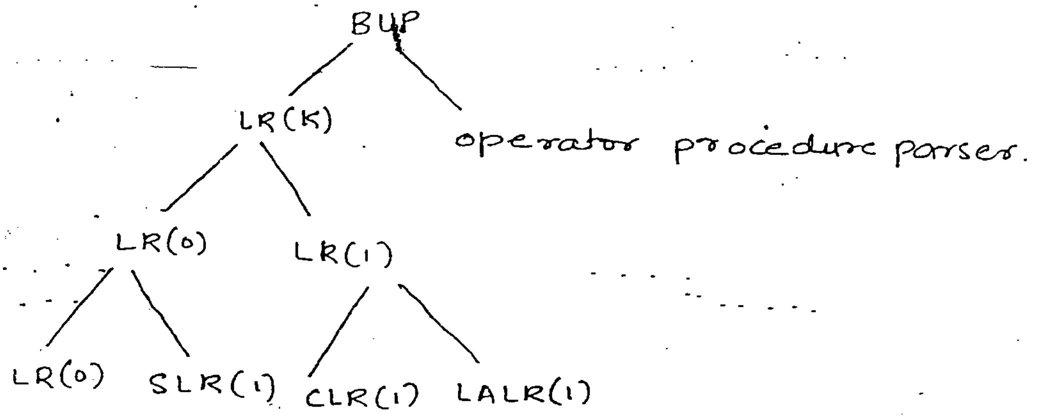
$F \rightarrow id$

Parsing table Entries m[s, id] and m[R, F]

m[s, id]
↓
S → FR

m[R, F]
↓
R → ε.

*** BOTTOM UP PARSER

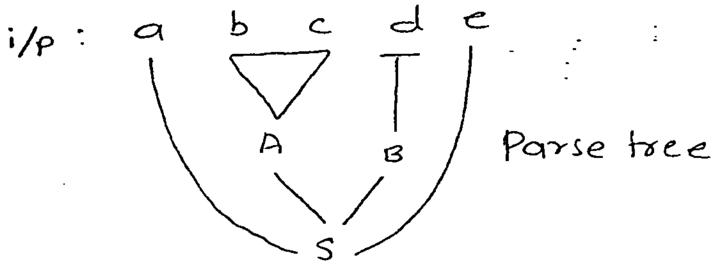


Unambiguous	operator procedure parser
LR(0)	- only operator grammer
SLR(1)	- both ambiguous and unambiguous
CLR(1) | LR(1)	
LALR(1)	

Bottom up parser (BUP):

The process of construction of the parse tree in the buttom up manner i.e starting with the children and proceeds to the root is called Bottom up parser

ex: S → aABe
A → bc|b
B → d.



Handle:

substring of the given i/p string that matches with R of any production is called Handle.

Note:

1. BUP is also called as shift reduce parser.
2. The BUP uses reverse of right most derivation.
3. It has to detect the handle and reduce the handle.
4. Detecting the handle is main overhead.
5. BUP is constructed for Unambiguious grammer. (except for operator grammer)

6. The BUP can be constructed for the grammer which has more complexity.

7. This Parsing machanism is faster then Top down parsing i.e. BUP is more efficient then TDP.

8. BUP performance is high.

9. Average time complexity is $O(n^3)$.

ex: $S \rightarrow AA$

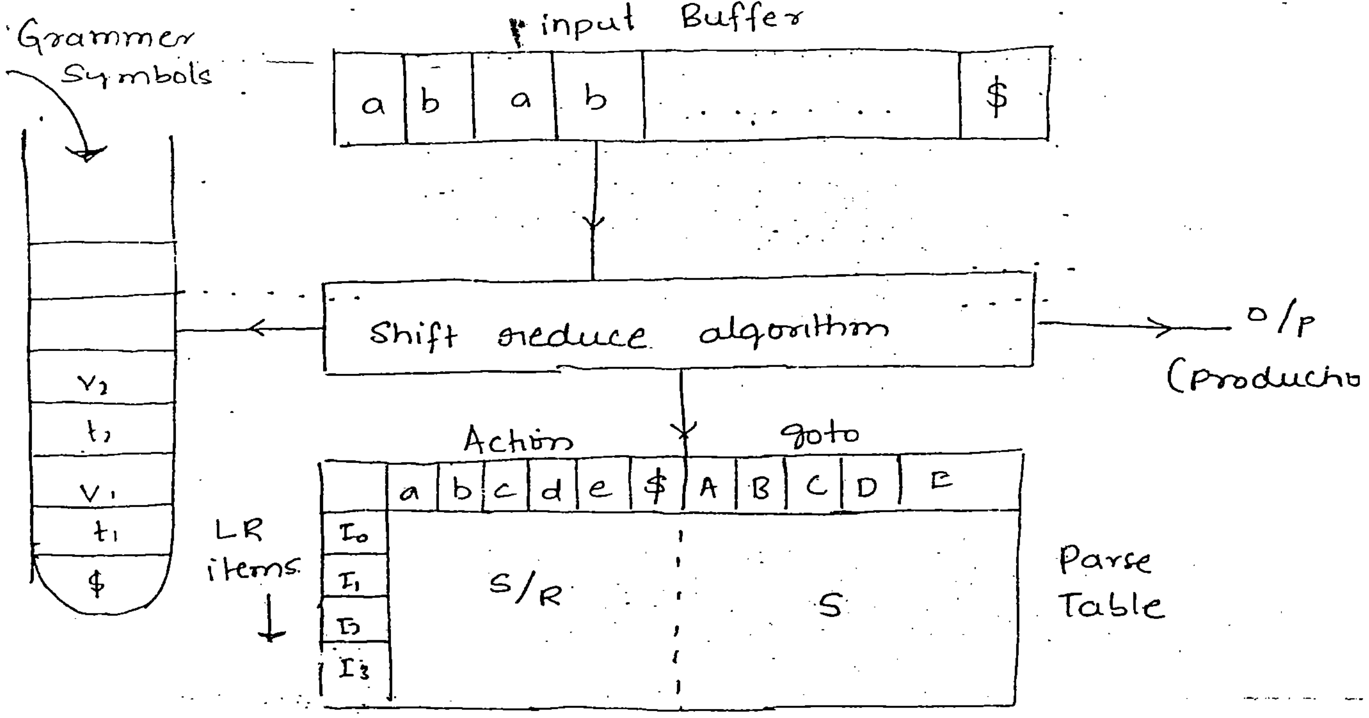
$A \rightarrow aA$

$A \rightarrow b$

stack	i/p	Action
\$	abab\$	shift
\$a	bab\$	shift
\$ab	ab\$	reduce A → b
\$aA	ab\$	reduce A → aA
\$A	ab\$	shift
\$Aa	b\$	~~reduce A → b~~ shift
\$Aab	\$	reduce A → b
\$AaA	\$	reduce A → aA
\$AA	\$	reduce S → AA
\$S	\$	accept.

Block diagram of BUP:

18



1. Input Buffer
2. parse stack.
3. parse table

Input Buffer: consist of the i/p string to be parsed and the i/p string is ends with $.

Parse stack:
Stack consist of the Grammer symbols. The grammar symbols are inserted or removed from the stack using shift and reduce operations.

Parse table:
parse table is constructed with no. of terminals, no. of non terminals & LR items / compiler items.

The parse table consist of two parts:

a. Action
b. goto.

Action part consist of shift and reduce operations implied for the terminals. and goto consists of the shift operation which are applied on the non-terminals.

operations

① Shift
② Reduce —
③ Accept
④ Erroo.

Shift:

Shift operation can be ap whenever handle doesn't occur from the top symbol of the stack.

Reduce:

Reduce operation can be used whenever handle occurs from the top symbols of the stack.

Accept:

After processing the complete i/p string if the stack contains only start from the grammer then the i/p string is accepted and parse is succeufully. After

Error:

processing the i/p string if stack does not contain start symbol of the grammer then the i/p string is does not parse. i.e parse tree is not constructed. So the result is error.

LR Parser

LR(k)

k → No. of Look Ahead symbols.

R → Reverse of RMD

L → Left to Right.

Classification of LR Parser:

Based on the construction of the table LR parsers divided into 4 types

1. LR(0)
2. SLR(1)
3. CLR(1)
4. LALR(1)

Procedure for the construction of LR parse table.

1: obtain the argumepted grammer for the given grammse
2. Create the canonical collection of LR items or compiles ,
3. Grammer DFA and prepase the table based on the LR item

Note:

LR of the parser allow all the ambigeous grammer in thes parsing the grammer should be taken as it is and should not be modified.

Argumented Grammer:

The grammer which is obtained by adding one more production that derives the start symbol of th grammer is called Argumented Grammer.

ex:

$$\left.\begin{array}{l} S' \to S \\ \left.\begin{array}{l} S \to AB \\ A \to a \\ B \to b \end{array}\right\} \text{Grammer} \end{array}\right\} \text{Argumented Grammer}$$

Argumented grammer helps us in seporting the final item from non-final item the need for argumented gramme if you have multile production for the start symbol then we can decide what is the final string.

LR(0) items (or) items in compiler:

A production '.' at any point in the right hand si is called as LR(0) item.

ex: $A \to abc$

$A \to .abc$

$A \to a.bc$

$A \to ab.c$

$A \to abc.$ Final / complete item.

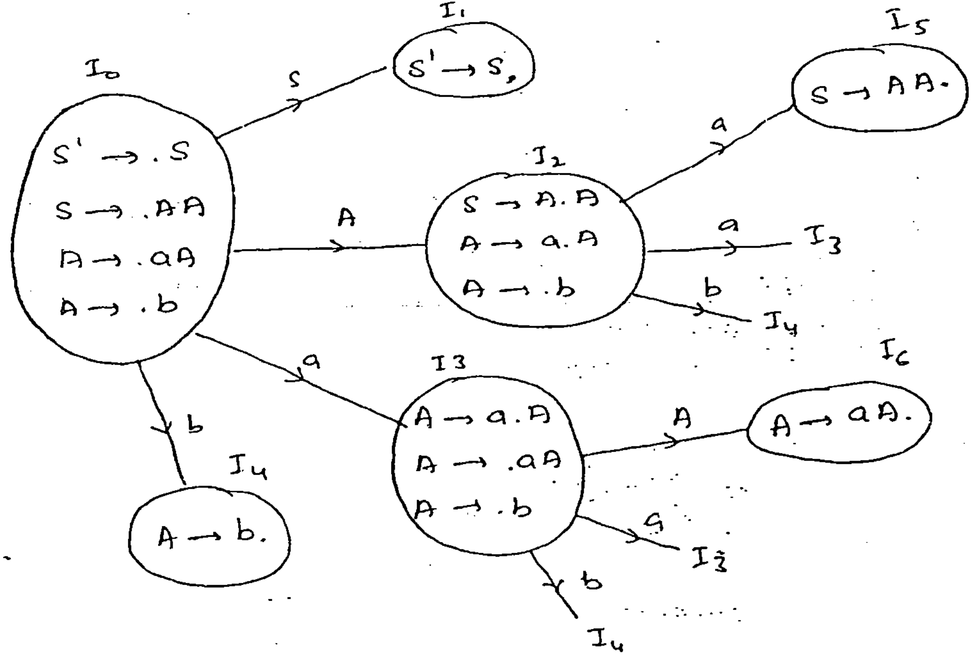
Canonical collection:

$c = \{$

$C = \{I_0, I_1, I_2, I_3, I_4, I_5\}$

I_0	I_1	I_2	I_3	I_4	I_5	I_6
$S' \to .S$	$S' \to .S$	$S \to A.A$	$A \to a.A$	$A \to b.$	$S \to AA.$	$A \to aA.$
$S \to .AA$		$A \to a.A$	$A \to a.A$			
$A \to .aA$		$A \to .b$	$A \to .b$			
$A \to .b$						

construction of DFA.



Construction of LR(0) parse Table:

ParseTable consist of 2 parts:

(1) Action

(2) Goto

	Terminal Action	Non-terminal Goto

① Goto $(I_i, x) = I_j$ x - Terminal.

	x
I_i	$S_j (I_j)$

2. Goto $(I_i, X) = I_j$ X - Non Terminal

	X
I_i	j

3. If I_i is any final items represents the rule r_i

	Action				
	t_1	t_2	....	t_n	\$
I_i	r_i	r_i	....	r_i	r_i

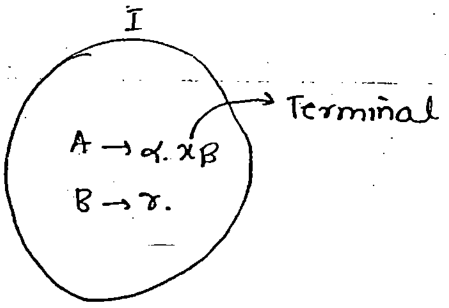
LR(0)

The grammer G is said to be LR(0) if its parse Ta is free from multiple entries or conflicts.

Conflicts in LR(0):

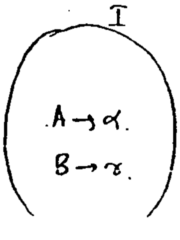
① SR-conflict:

If any state in DFA contains both shift & redu option then it is SR-conflict.



2. RR-Conflict:

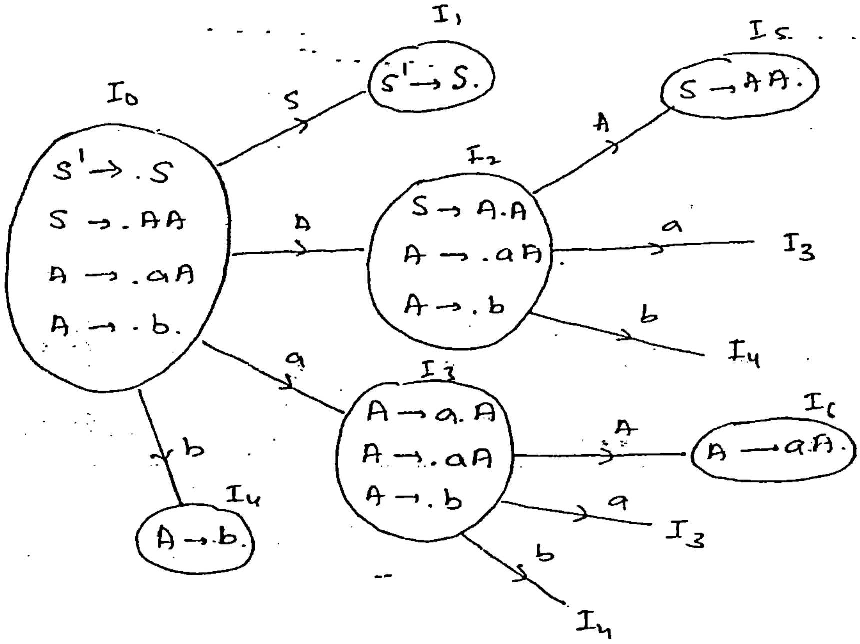
If the same state contain more than one final item then it is RR conflict.



ex: ① $S \rightarrow AA \quad r_1$

$A \rightarrow aA \quad r_2$

$A \rightarrow b \quad r_3$

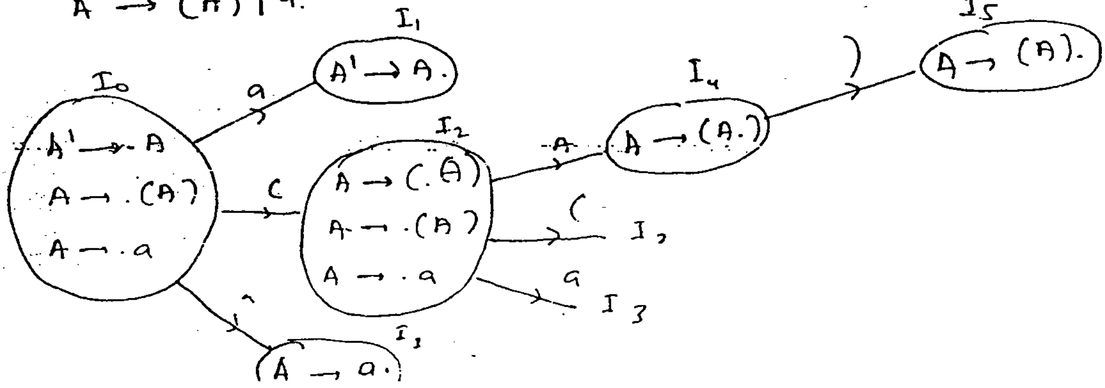


Parse Table:

	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1	Acc	Acc	Acc		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

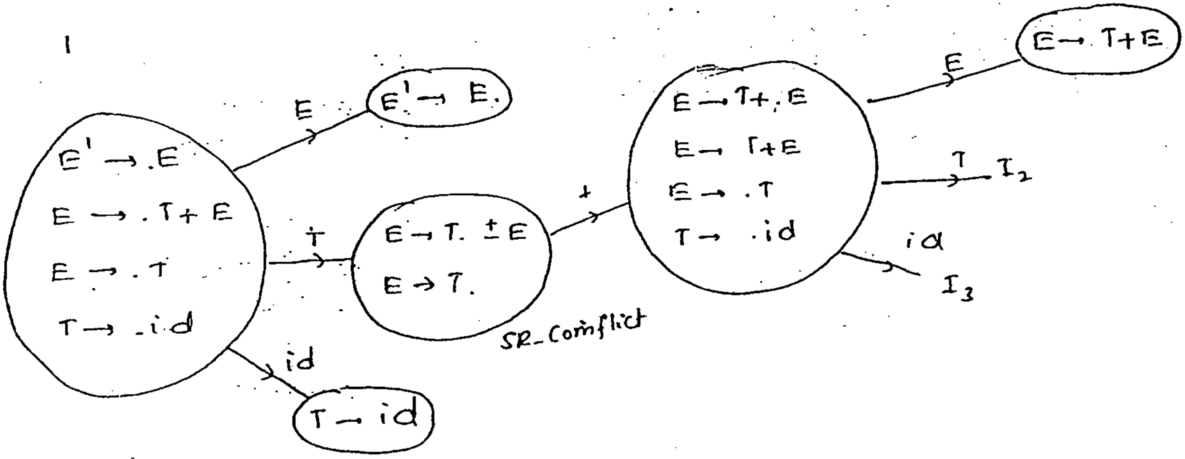
⇒ It is LR(0) Grammer

2. $A \rightarrow (A) \mid a.$



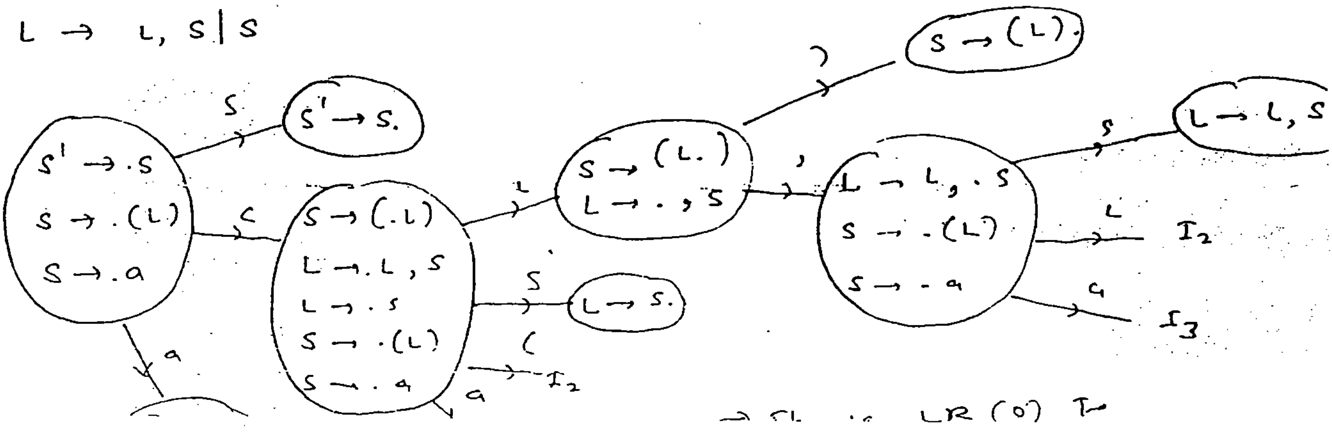
	(	)	a	$	A
I_0	S_2		S_3		1
I_1	Acc	Acc	Acc	Acc	
I_2	S_2		S_3		4
I_3	r_2	r_2	r_2	r_2	
I_4		S_5			
I_5	r_1	r_1	r_1	r_1	

3. $E \rightarrow T + E / T$
$T \rightarrow id$



	+	id	$	E	T
I_0		S_3		1	2
I_1	Acc	Acc	Acc		
I_2	$S_4(r_2)$	r_2	r_2		
I_3	r_3	r_3	r_3		
I_4		S_3		5	2
I_5	r_1	r_1	r_1		

4. $S \rightarrow (L) | a$
$L \rightarrow L, S | S$



Verify the following grammer LR(0) or not.

1. S → AaAb | BbBa
A → ε
B → ε

2. S → aAB | Ba | Ab
A → c
B → c

3. S → Aab | Bc
A → aA | a
B → Ba | b

4. E → E+T/T
T → T*F/F
F → Id

5. E → E+T/T
T → EF/Ea/b
F → (E) | a

6. S → AB | BA
A → Aab | b
B → BaA | a

7. S → A#a | @bB
A → $A | #
B → *B# | $

8. S → Aab | bab | bac | acb
A → aBA | b
B → b

9. E → EF | e
F → fT | f
T → ET | g

10. A → A(A) | bA | a

1.
S' → .S
S → .AaAb
S → .BbBa
A → .ε
B → .ε

→ Not LR(0).

more than one final state

2.
S' → .S
S → .aAB
S → .Ba
S → .Ab
A → .c
B → .c

RR conflict

3.
S' → .S
S → .Aab
S → .Ba
A → .aA
A → .

S' → S.

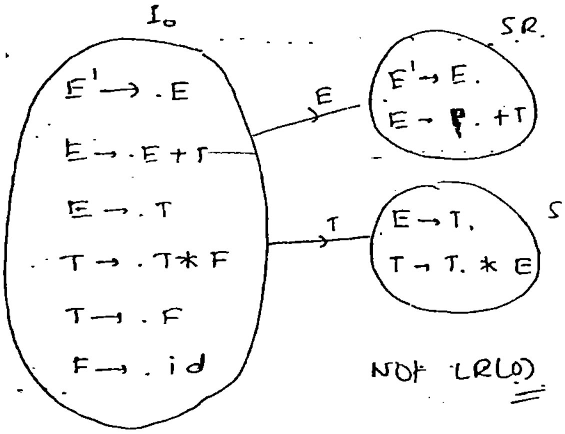
S → A.ab

S → B.a

A → a.

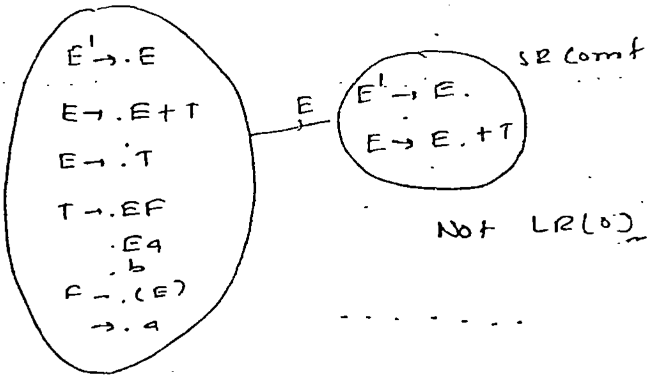
A →

4.



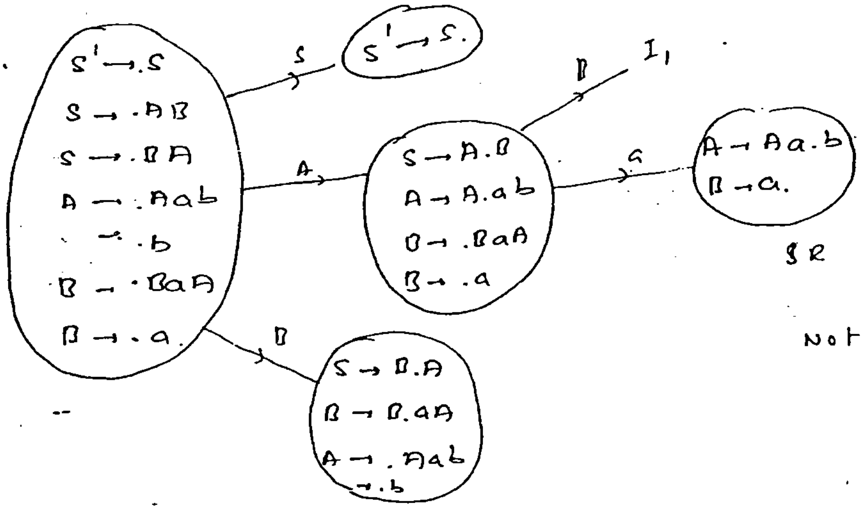
NOT LR(0)

5.



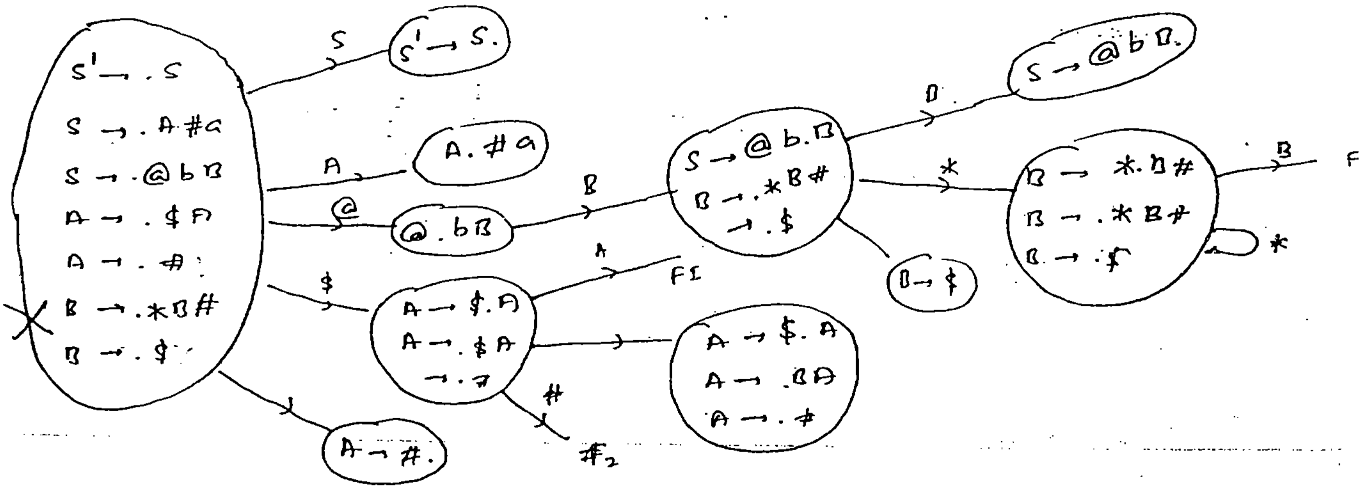
Not LR(0)

6.

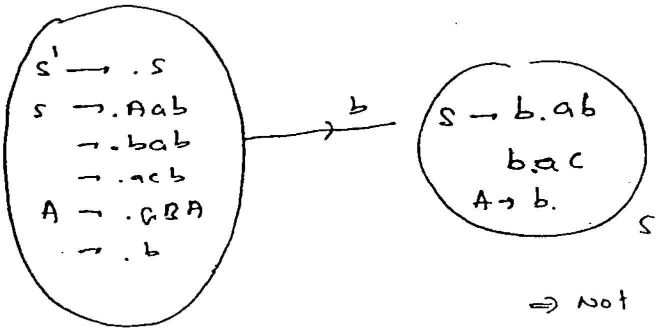


Not LR(0)

7.



F.



SR conflict.

⇒ Not LR(0)

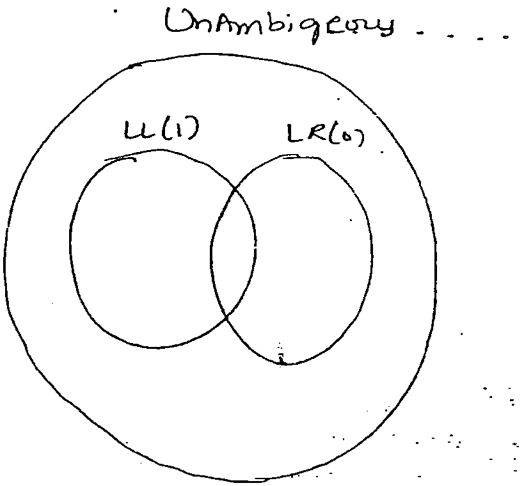
Note:

LR(0) grammer need not be LL(1)

LL(1) grammer need not be LR(0)

Some grammer are both LR(0) and LL(1).

For some grammer we can not construct LR(0) & LL(1).



SLR(1) parser

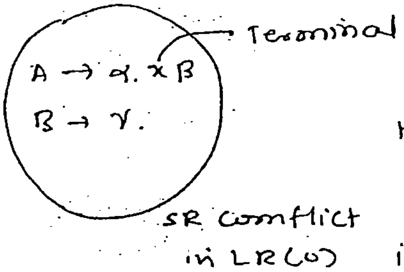
The procedure for constructing the parse table is similar to LR(0) parser but there is one restriction on reducing the entries.

Whenever there is a final item then place the reduce entries under the follow symbols of left hand side variable.

If the SLR(1) parse table does not contain multiple entries that is free from conflicts is <u>SLR(1) Grammer</u>.

Conflicts in SLR(1) :

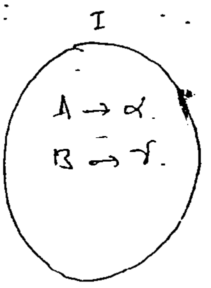
1. SR Conflict:



if $fol(B) \cap \{x\} \neq \phi$ then it is SR conflict in SLR(1)

if $fol(B) \cap \{x\} = \phi$ then it is SR conflict in LR(0) and not SLR

3. RR comflict:



RR comflict on LR(0).

If fol (A) ∩ fol (B) ≠ ∅

⇒ It is RR comflict in SLR(1).

If fol (A) ∩ fol (B) = ∅

⇒ then it is RR-comflict in LR but not in SLR(1).

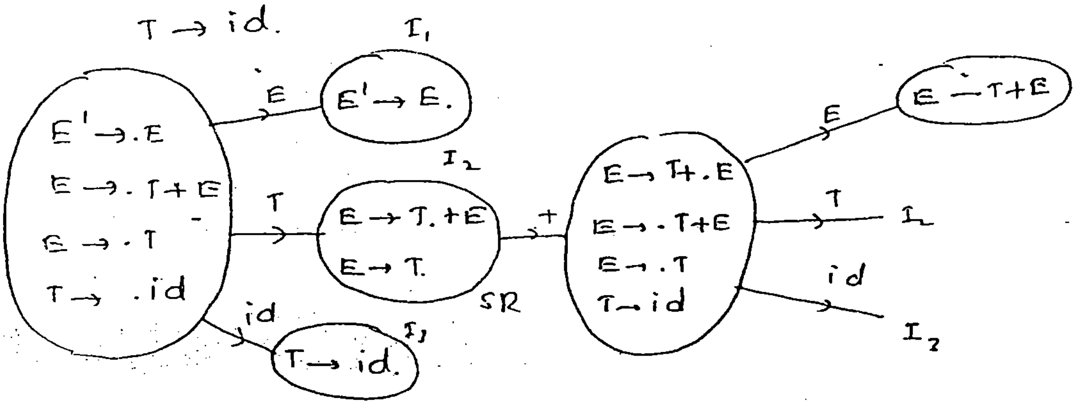
Note:

1. Every LR(0) grammer is SLR(1).
2. Every SLR(1) grammer need not be LR(0).
3. The no. of entries in SLR(1) parse table is ≤ no. of entr in LR(0) parse table
4. SLR(1) parser is more powerfull than LR(0).

Fol(E) = {$} ∩ {+, } = ∅

Ex: ① E → T+E | T.

T → id.



not LR(0), SLR(1)

	+	id	$	E	T
I0		S3		1	2
I1	Acc	Acc	Acc		
I2	S4				
I3	r3		r3		
I4		S3		5	2
I5			r1		

2. $S \to AaAb \mid BbBa$

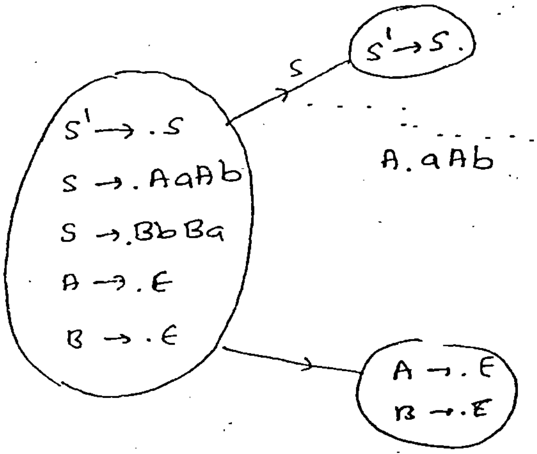
$A \to \epsilon$

$B \to \epsilon$

$fol(A) = \{a, b\}$ $fol(A) \cap fol(B) \neq \phi$

$fol(B) = \{b, a\}$

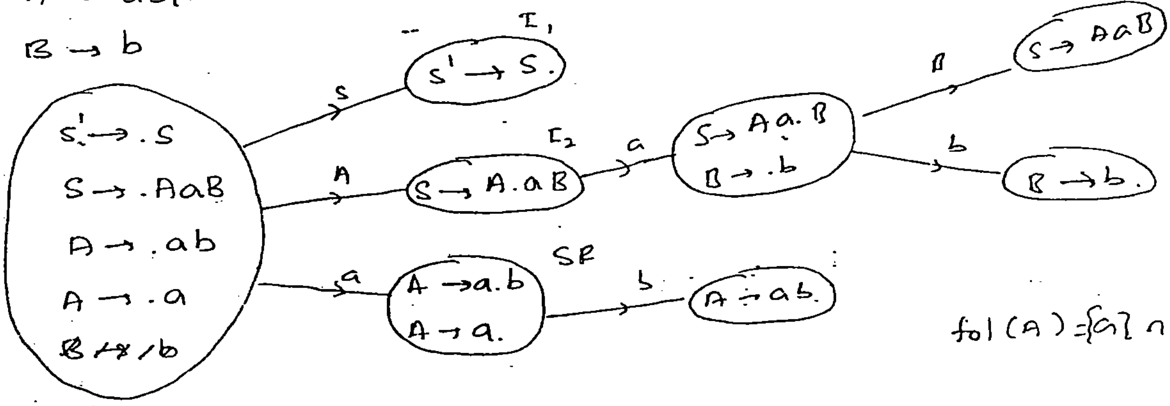
not LR(0) and SLR(1).



3. $S \to AaB$

$A \to ab \mid a$

$B \to b$

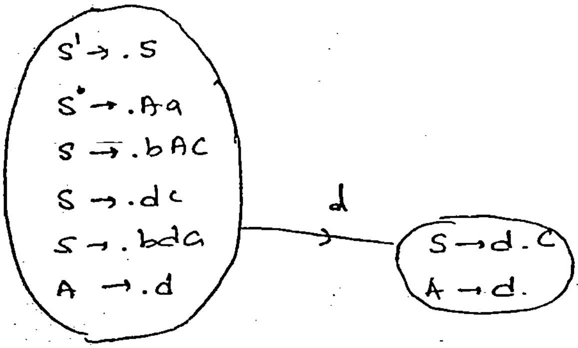


$fol(A) = \{a\} \cap \{b\} = \phi$

It is SLR(1) but not LR(0).

4. $S \to Aa \mid bAc \mid dc \mid bda.$

$A \to d$



$fol(A) = \{a, c\} \cap \{c\} \neq \phi$

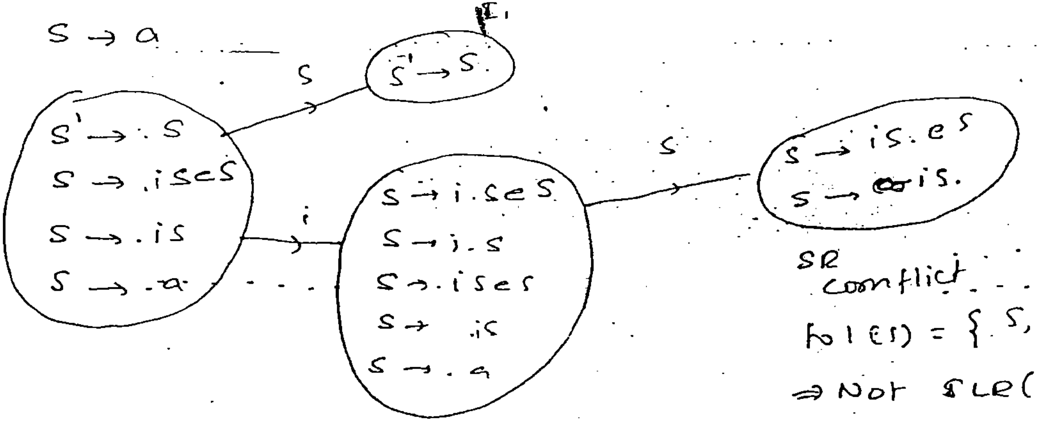
not LR(0)

⇒ Not SLR(1)

5. S → iSeS

S → iS

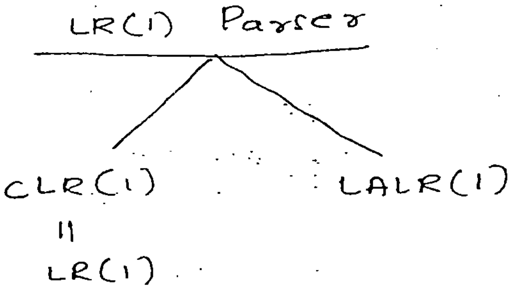
S → a



SR conflict

follow(S) = {S, e} ∩ {e} ≠ ∅

⇒ Not SLR(1), LR(0)



CLR → Canonical LR

LALR → LookAhead LR

LR(1) parser depends on one LookAhead symbol

LR(0) parser does not depends on LookAhead symbol.

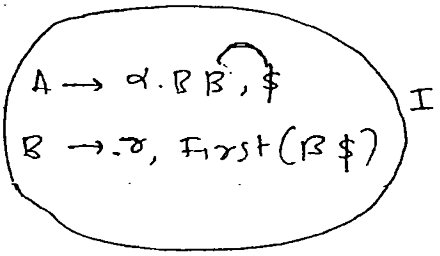
Procedure to construct LR(1) parser.

Closure(I): 1. Add every thing from i/p to o/p.

** 2. If $A \rightarrow \underbrace{\alpha . B\beta}_{\text{grammer symbols}}, \$$ is in I. (B: Non terminal)

and $B \rightarrow \gamma$ is in the grammer

then Add $B \rightarrow .\gamma$, First$(\beta\$)$ to the closure(I)



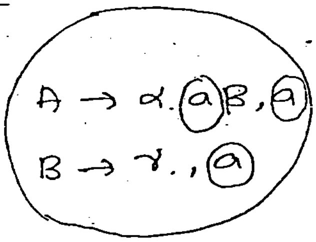
3. Repeat this for every newly added item

Goto (i, x).

1. while finding the transition lookahead part remain same.
2. While applying the closure lookahead part may change.

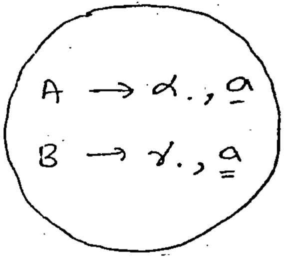
Conflicts of in LR(1):

1. SR-Conflict:



SR-conflict

2. RR-Conflict:



⇒ RR conflict.

LR(1) grammer

The grammer for which LR(1) parser is construction LR(1) or CLR(1) grammer.

The grammer is LR(1) iff it parse table is free from multiple entries.

Notes

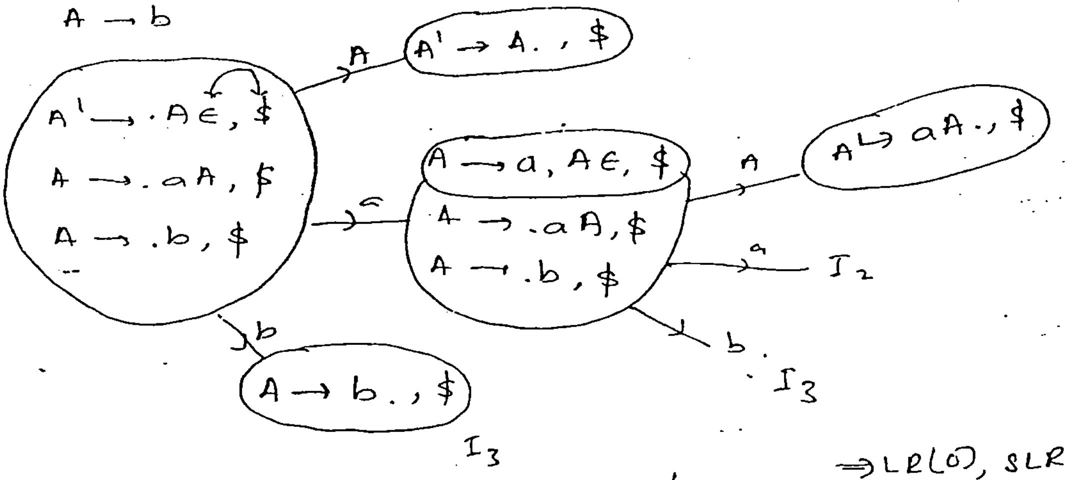
1. For every grammer SLR(1) grammer is contructed then CLR(1) parser is also constructed but if CLR(1) parser is constructed, we may (or) may not construct SLR(1).
2. CLR(1) parser is more powerful then SLR(1).
3. No. of entries in SLR(1) parse table ≤ (less than are equ to No. of entries in CLR(1) parse table.

ex: A → Aa.
A → b.

A → aA.
A → b

A' → . ~~A∈~~, $
A → . Aa ,
A → . b,

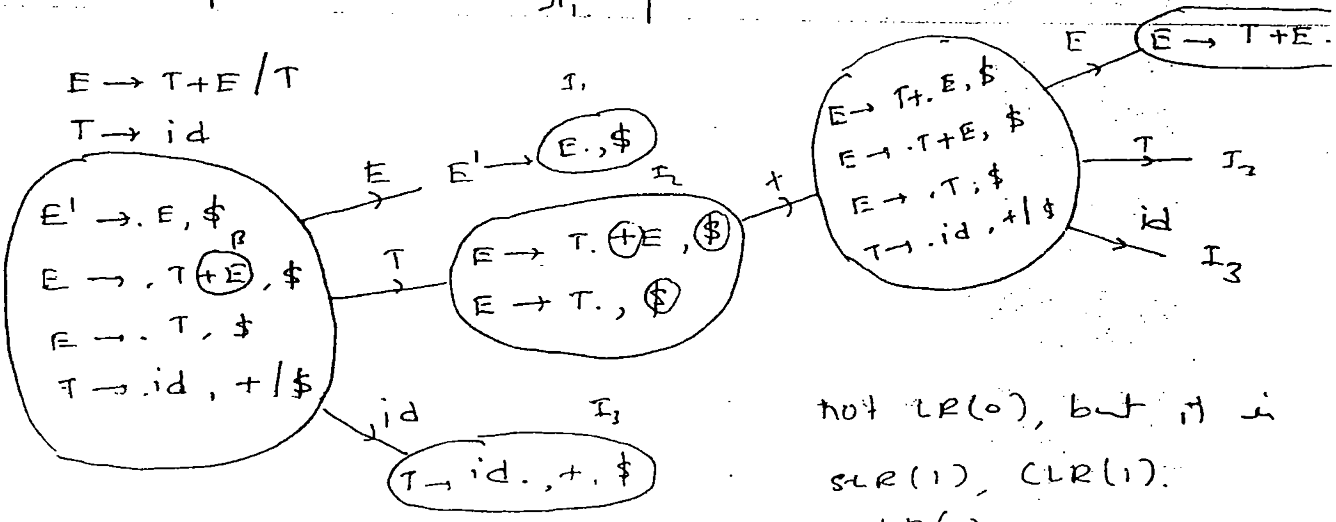
① A → aA
A → b



⇒ LR(0), SLR(1), CLR(1)

	a	b	$	
I0	S2	S3		1
I1	Acc	Acc	Acc	
I2	S2	S3		
I3			r2	4
I4			r1	

② E → T+E / T
T → id

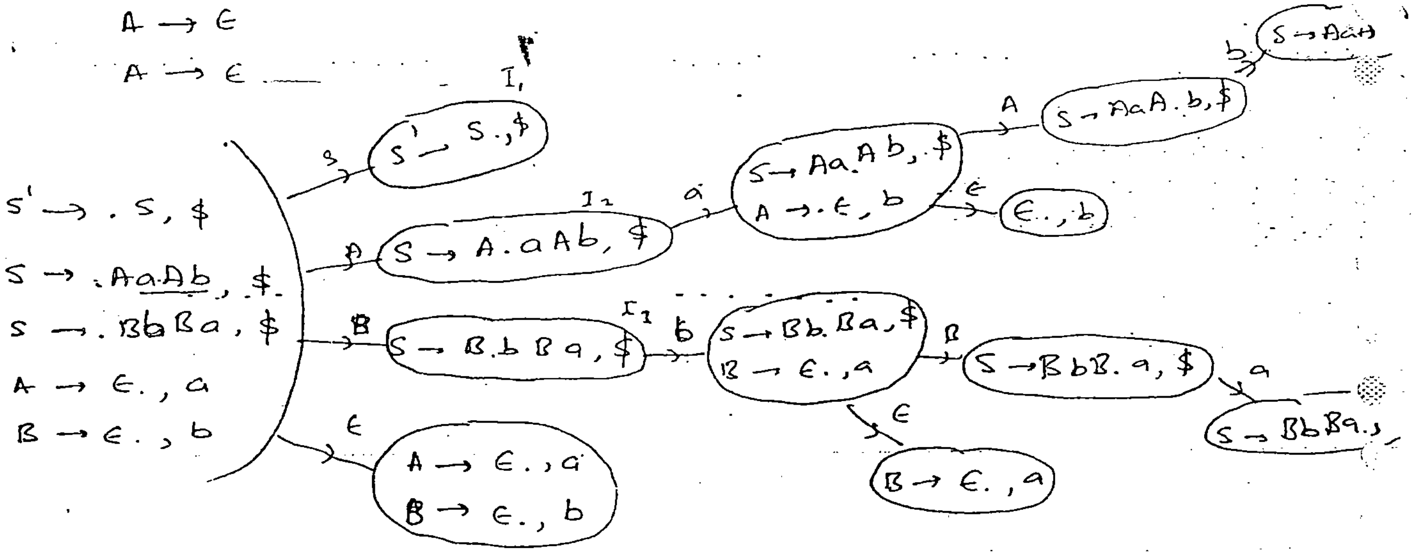


not LR(0), but it is SLR(1), CLR(1).
Not LLR(1)

3. $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$A \rightarrow \epsilon$



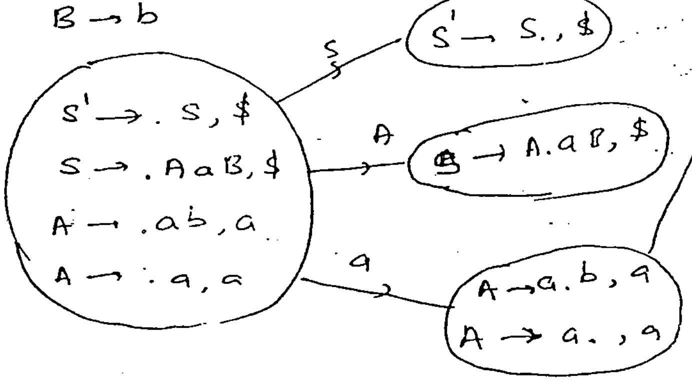
LL(1), CLR(0)

but not LR(0), SLR(1)

4. $S \rightarrow AaB$ Not LLR(1)

$A \rightarrow ab \mid a$

$B \rightarrow b$



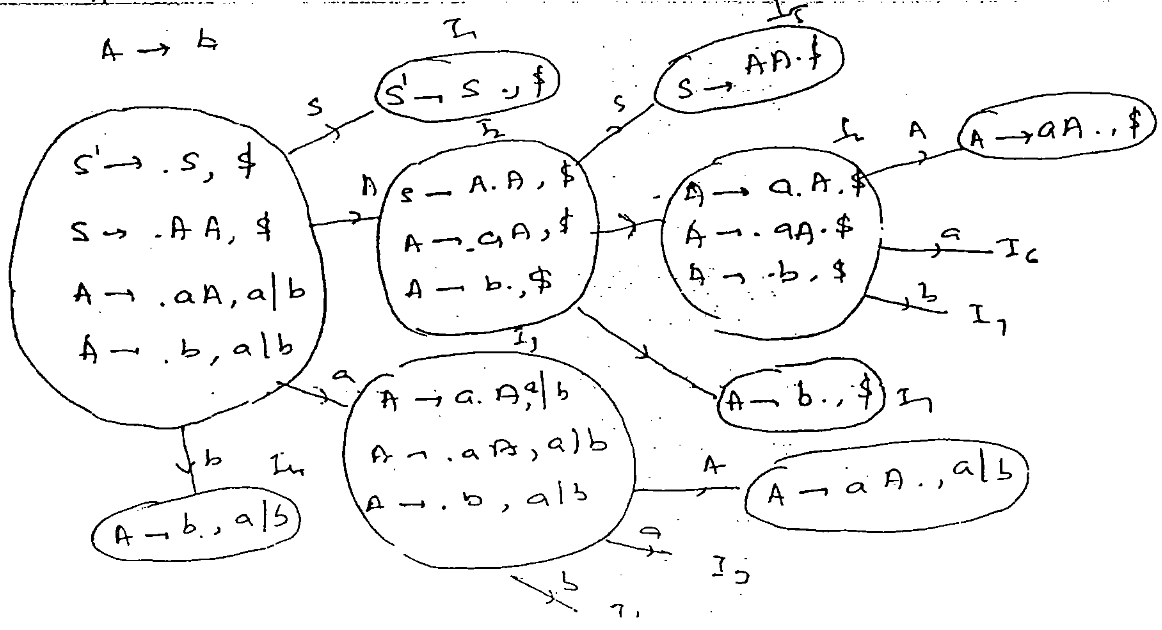
SR conflict in LR(0), but in SLR(1), CLR(1)

⇒ It is Not LR(0), but SLR(1), CLR(1)

5. $S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$



	Action			goto	
	a	b	$	S	A
I₀	s3	s4		1	2
I₁	Acc	Acc	Acc		
I₂	s6	s7			5
I₃	s3	s4			8
I₄	r3	r3			
I₅			r1		
I₆	s6	s7			9
I₇			r3		
I₈	r2	r2			
I₉			r2		

I₄ , I₇

I₄₇

A → b. , a|b|$

I₃ + I₆

I₃₆

A → a.A , a|b|$
A → ·aA, a|b|$
A → ·b. , a|b|$

I₈ I₉

I₈₉

A → aA.. , a|b|$

LALR(1) parser

The DFA of CLR(1) parser contain some states with common production part differ by Lookahed pa

Now combine the states with common production part and different Lookahead part into a single s and construct the parse table. if the parse table is free from multiple entries ~~them~~ i.e free from both SR & RR conflict then the grammer is LALR(1) grammer

Note:

1. If the DFA of CLR(1) parser does not contain more than 1 state with common production part and with different follow part then the grammer is CLR(1) and LALR(1
2. If the DFA of CLR(1) contains more than one state

part then it may or may not be LALR(1) parser.

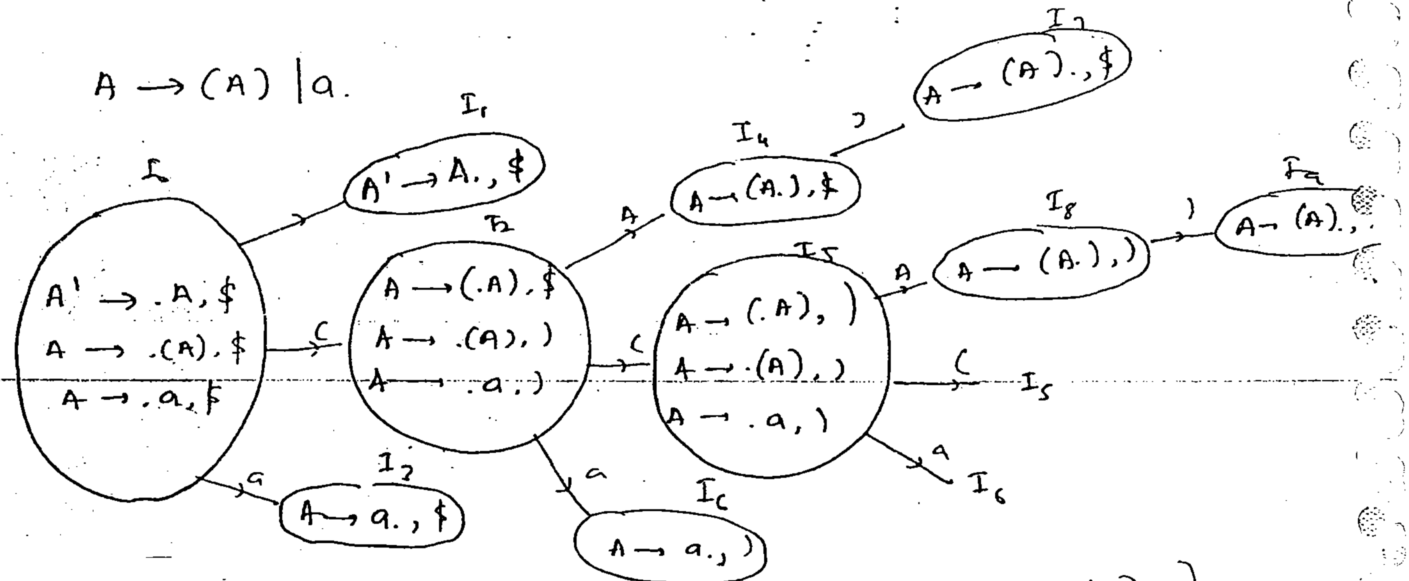
Conflicts in LALR(1)

1. If there is SR conflict in CLR(1) then there is no SR conflict in LALR(1) also.
2. If there is no RR conflict in CLR(1) then there may be RR conflict in LALR(1).

	a	b	\$	S	A
I_0	S_{36}	S_{47}		1	2
I_1	Acc	Acc	Acc		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			\$		
I_{89}	r_2	r_2	r_2		

25/12/2010.

2. $A \rightarrow (A) \mid a.$



$I_3 + I_6$

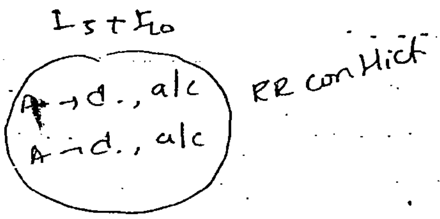
$A \rightarrow a.,) \mid \$$

$I_7 + I_9$

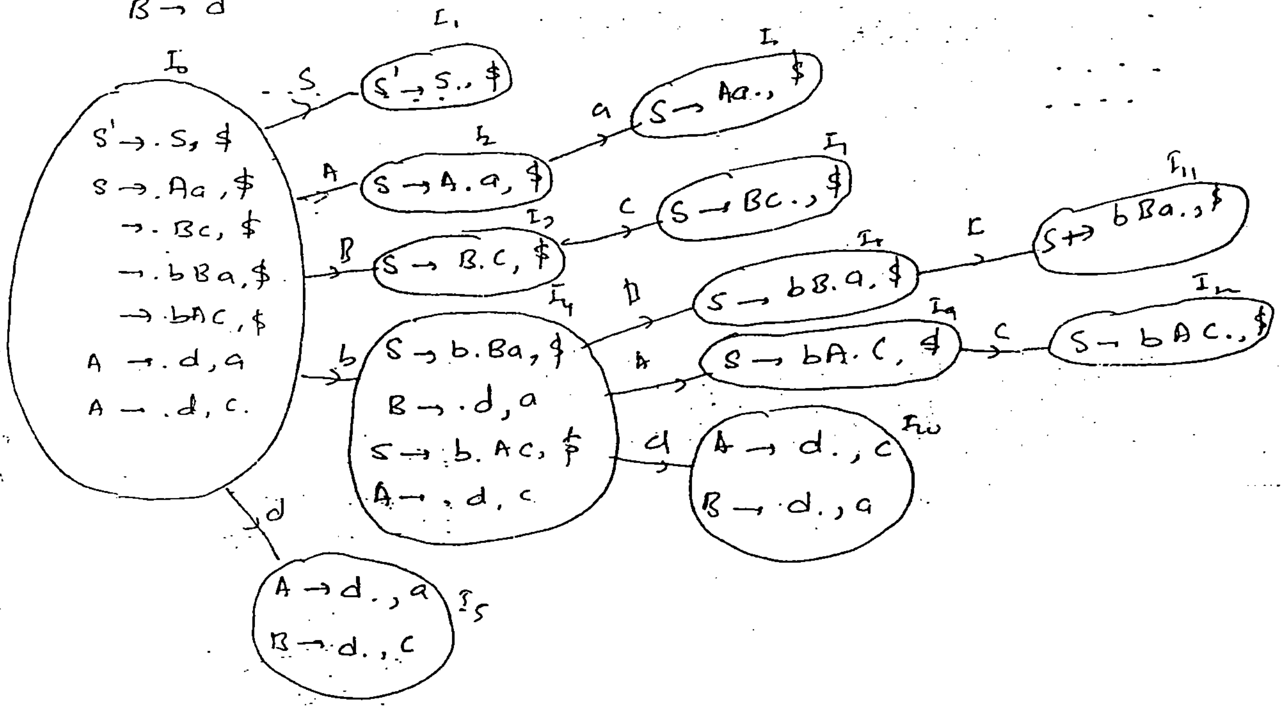
$A \rightarrow (A)., \$ \mid)$

SLR(1)
LALR(1)
LR(0)
SLR(1)

3. S→Aa
S→bAc
S→Bc
S→bBa
A→d
B→d

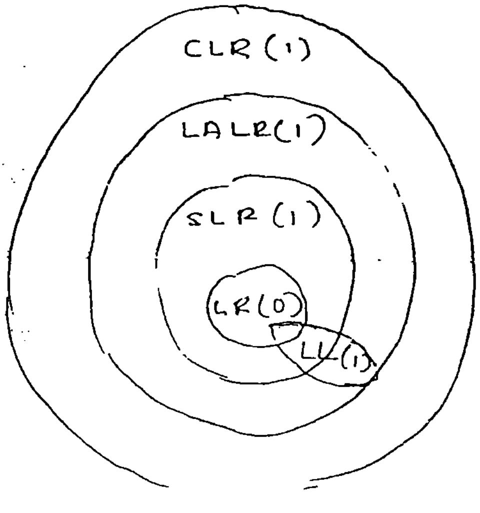


It is CLR(1.)
Not LALR(1.)
Not LL(1)
Not LR(0)
Not SLR(1)



Note:

1. For a grammer LALR(1) parser is constructed the CLR(1) parser can be constructed.
2. If CLR(1) grammer parser can be constructed for grammer G, then we may or may not construct LALR parser for same grammer.
3. Every LALR(1) grammer is ~~or~~ CLR(1) but every CLR l grammer need not be LALR(1).



4. The no. of entries in the LALR(1) parse table is ≤ no. of entries in the CLR(1) parse table.

5. No. of entries in LALR(1) parse table = No. of entries in SLR(1) parse table and all the entries are identical for both the parsers.

6. LALR(1) is a medium type of parser and is not powerful as much as a CLR(1) but slightly more powerful than SLR(1) parser. Hence LALR(1) lies b/w CLR(1) & SLR(1).

7. CLR(1) parser is more efficient, powerful and costly among all the parsers.

Problems

1. The grammer $S \rightarrow CC$
$C \rightarrow aC$
$C \rightarrow b$

[✓ a) LL(1)]
b) SLR(1) but not LL(1)
c) LALR(1) but not SLR(1)
d) CLR(1) but not LALR(1).

2. The grammer $S \rightarrow A | a$
$A \rightarrow a$ is

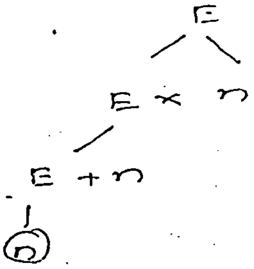
a. LL(1)
b. LR(0) & LL(1)
c. LR(0), LL(1), SLR(1), LLR(1), LL(1)
[✓ d. None.]

3. $S \rightarrow (S) | a$ and n_1, n_2, n_3 are the entries in the parse tables of SLR(1), LR(1), LALR(1)

a. $n_1 < n_2 < n_3$
[✓ b. $n_1 = n_3 \leq n_2$]
c. $n_1 = n_2 = n_3$
d. $n_1 > n_3 > n_2$

4. $E \rightarrow E+n \mid E\times n \mid n$ and $w = n+n\times n$. then...
Handles in the right sentential form of productions.

a. n, E+n, E+E×n
b. n, E+n, E+n×n
c. n, n+n, n+n×n
d. n, E+n, E×n.



5. For SLR(1) and LALR(1) which of the following is true.

a. Goto entries are different.
b. Shift entries are identical.
c. Reduce entries are different
d. Error entries are different.

Operator precidency parser

1. Constructed for both ambiguous and unambiguous grammars.
2. Constructed for only operator grammer.
3. Takes the grammer with less complexity
4. Can not be constructed for every grammer.
5. Generally used for the language which are mostly useful in scientific application.

Operator Grammer:

The grammer that does not contain ε groups or adjacent non-terminals on RHS of any rule is know as operator grammer.

Ex: $E \rightarrow EAE \mid id$
$A \rightarrow + \mid * \mid -$
} Not operator Grammer.

Ex: $E \rightarrow E+E$
$\rightarrow E-E$
$\rightarrow E*E$
$\rightarrow id$
} Operator Grammer

The operator grammer is a operator precedency grammer if its relational parse table is free from multi entries.

Procedure to construct the precedency parse Table.

Let θ_1 and θ_2 be the two operators.

1. If θ_1 is has higher precedency than θ_2, then

$$\theta_1 > \theta_2 \quad \text{or} \quad \theta_2 < \theta_1$$

Again if θ_1 and θ_2 are of equal precedency

$$\theta_1 = \theta_2.$$

2. If $A \rightarrow \theta_1 B \theta_2$ is a production where B is variable or ϵ then, $\theta_1 = \theta_2$

3. If θ_1 and θ_2 are of equal precedency and
   a. are of left Associative
   then, $\theta_1 > \theta_2$ or $\theta_2 < \theta_1$
   b. are of right Associative.
   then, $\theta_1 < \theta_2$ or $\theta_2 > \theta_1$

ex

$E \rightarrow E+T/T$

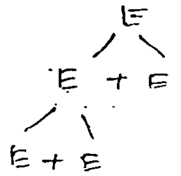
$T \rightarrow T*F/F$

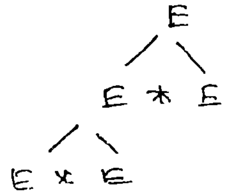
$T \rightarrow id$

↓ θ_1	+	*	id	$
+	>	<	>	>
*	>	>	>	>
id	>	>		>
$	<	<	<	=

2. E → E+E/E*E /id

	*	+	id	$
*	< >			
+		< >		
id				
$				





Parse table contains multiple entries

∴ The grammer is not operator precedency grammar

Note:
Every operator precedency grammer is an operator grammer but every operator grammer need not be operator precedency grammer.

Syntax Directed Transalation (SDT)

Symantic Analyser verify the each and every sentence of the source code.

Syntax Analyser or parser takes care of the operators c working on required numbers of operands or not, If it not consider the type of operand.

---

symantic analysis can be implemented by parsing along with the semantic rules by attaching the symantic ru for each and every grammer rule.

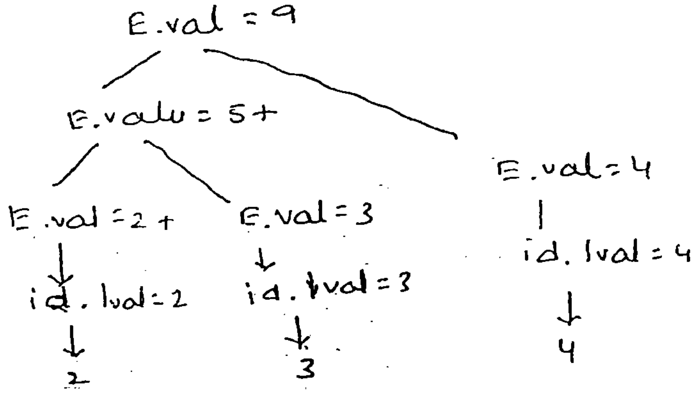
Symantic analysis, the parsing is also varific

Def SDT:

The grammer with symantic rules is known as SD

ex: E → id E+E { E.val = E.val + E.val } } SDT.
E → id { E.val = id.lval }

i/p : 2+3+4.



Note:

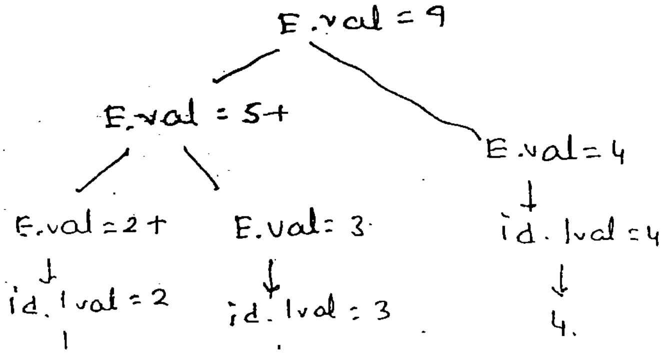
Apart from parsing the SDT can be used

1. To store the type information into symbol table
2. To build the syntax tree or DAG
3. To verify the consistency check like type checking, parameter checking etc
4. to generate intermediate code.
5. To genetck the Target code.
6. To evaluate the algebraic expression

Annotated Parse Tree (Decorated parse Tree)

The parser tree shows the attribute values at each and every node is known as Annotated parse tree.

ex: i/p : 2 + 3 + 4



Types of attributes.

Based on the process of evaluation attributes can be classified into 2 type.

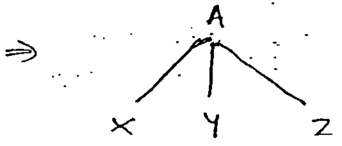
① Synthesized Attribute.

② Inherited Attribute

Synthesized Attribute:

The attribute whose value is evaluat in terms of attribute value of its children is called as Synthesized attribute.

ex: A → xyz



$A.s = f(x.s \mid y.s \mid z.s)$

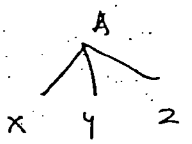
↓

Synthesized Attribute.

Inherited Attribute:

Attribute whose value is evaluated in turn of attributes value of its parent or siblings is called Inherited Attribute.

ex: A → xyz



$Y.s = f(x.s \mid Y.s \mid z.s)$

↓

Inherited Attribute.

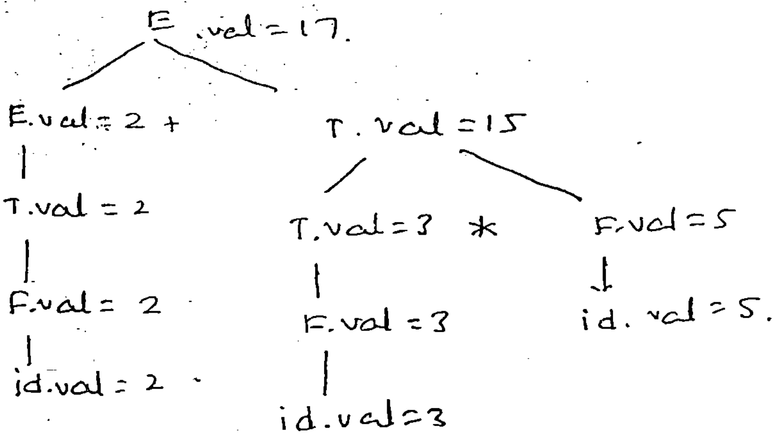
Rules to construct the SDT.

1. Define the grammer by looking at input string
2. construct the parse tree.
3. Attach the semantic rules by look at required o/p

ex: SDT for the evaluation of an expression :

i/p : 2 + 3 * 5
o/p : 17

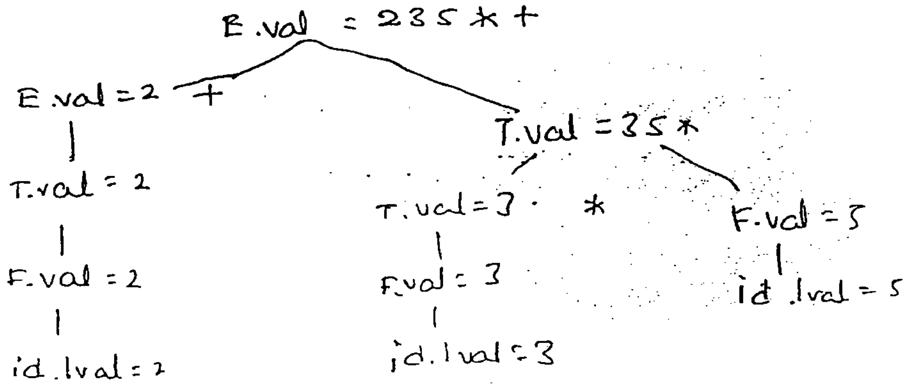
Grammer : E → E + T {E.val = E.val + T.val}
|T {E.val = T.val}
T → T * F {T.val = T.val * F.val}
|F {T.val = F.val}
F → id {F.val = id.lval}



ex 2: SDT to conversion of infix to postfix.

i/p : 2 + 3 * 5
o/p : 2 3 5 * +

E → E + T {E.val = E.val || T.val || '+' ;}
|T {E.val = T.val}
T → T * F {E.val = T.val || F.val || '*' ;}
|F {T.val = F.val}
F → id {F.val = id.lval}



SDT to convert ++

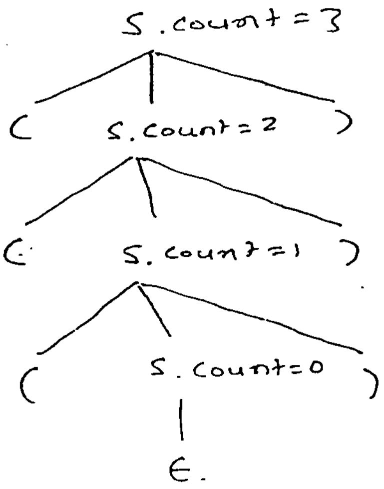
SDT to count the paranthesis:

Grammer:

$S \to (S)$ {S.count = S.count + 1}

$S \to \epsilon$ {S.count = 0}

((()))



SDT for Type checking

SDT is used for typechecking of boolean and integer data types.

i/p: (3+5) ==10

o/p: False.

Grammer

$E \to E == E$ { E.type = if (E_1.type == E_2.type) && E_1.type = int) return boolean

$\to (E)$

$\to E + E$ { E.type = E_1.type }

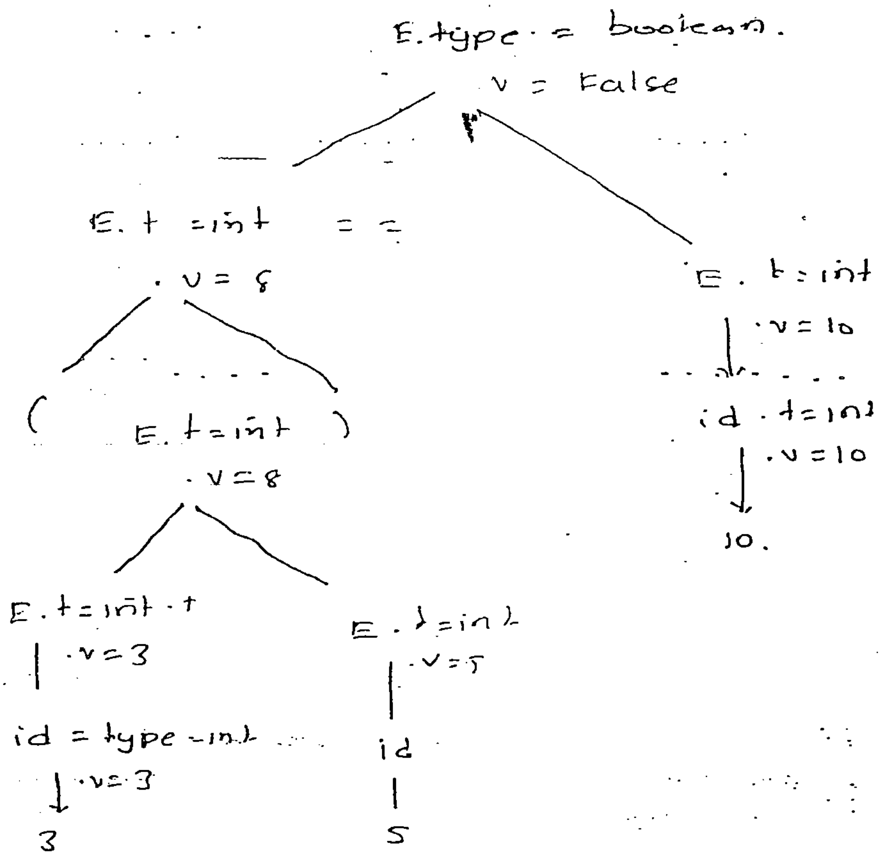
{ E.type = if (E.type == E_2.type) && E.type = int) return int else error

$\to$ id = int

{ E.type = id.type = int }

$\to$ True { E.type = Boolean }

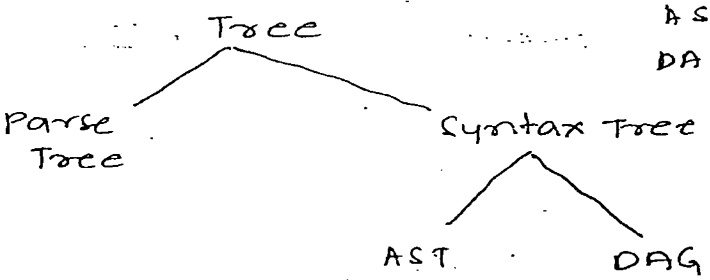
$\to$ False { E.type = Boolean }



SDT to build Syntax Trees

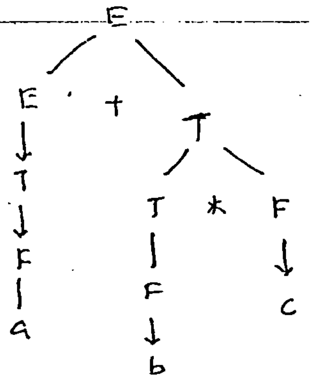
AST - Abstract Syntax Tre

DAG - Directed Acyclic Gra

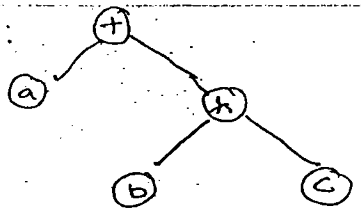


AST:

a + b * c



o/p :



AST:

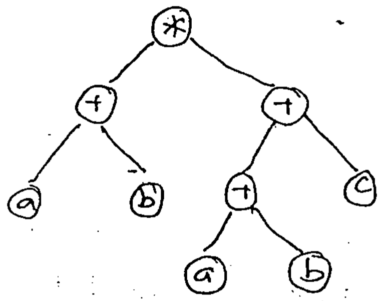
E → E+T { E.ptr = m/c Node (E.ptr, +, T.ptr); }
|T { E.ptr = T.ptr }
T → T*F { T.ptr = m/c Node (T.ptr, *, F.ptr); }
|F { T.ptr : F.ptr }
F → id { F.ptr = m/c Node (NULL, id, NULL); }

DAG

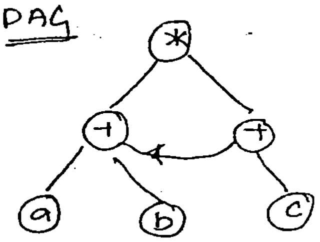
DAG is used to eliminate the common sub expression The procedure to construct the SDT for DAG is simile to AST expect one restriction. i.e if any node is already created make use of that node instead of going for new creation of same node.

ex: (a+b) * (a+b+c)

AST



DAG



Types of SDT.

SDT can be defined in two ways

1. S-attributed
2. L-attributed.

S-Attribute

1. Uses only synthesized attributes
2. Symantic rules can be placed at the right end of the
3. Attributes are evaluated during Bottom up Parsing

ex: E → FT { pf (#) }
F → T*F { pf (*) }
→ a { }
T → b { }

L-attributed:

1. Uses synthesized and Inherited attributes
2. Inherited attributes values are inherited from the parent or left siblings.
3. Symantic rules can be placed on the RHS.
4. Attributes are evaluated using Depth first Right to left process (DFS L to R).

Ex:

S → A {print(#)} B
A → +B {print(@} C | ε
B → aC {print($)} | ε
C → b {print(*)}

Note:

1. Every S-attributed is L-attributed.
2. Every L-attributed can be converted into S-attributed.

L-attributed

E → TE'$
E' → +T ① E' | ε
T → FT'
T' → *F ② T' | ε
F → id ②

S-attributed.

E → TE'
E' → +TME' | ε
M → ε ①
T → FT'
T' → *FRT' | ε
R → ε ②
F → id ③

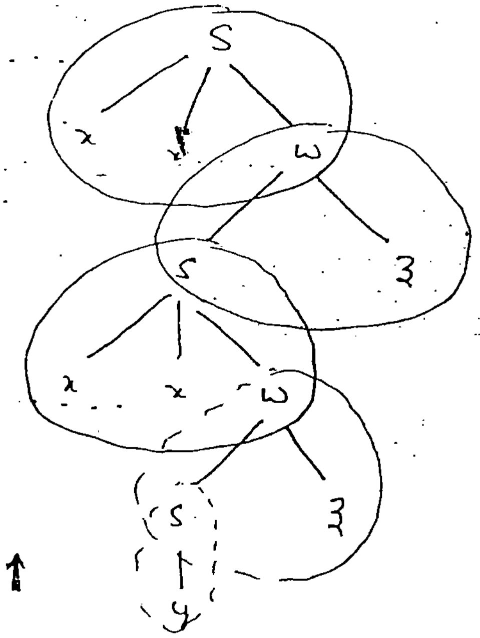
attributes are evaluated only from children

---

① S → x n w {print(1)}
   | y {print(2)}
w → S z {print(3)}

If the SDT carries the string w = xxxxyzz.

a. 23131 2. 23133 3. 21313



2. S → BC { B.s = C.s }

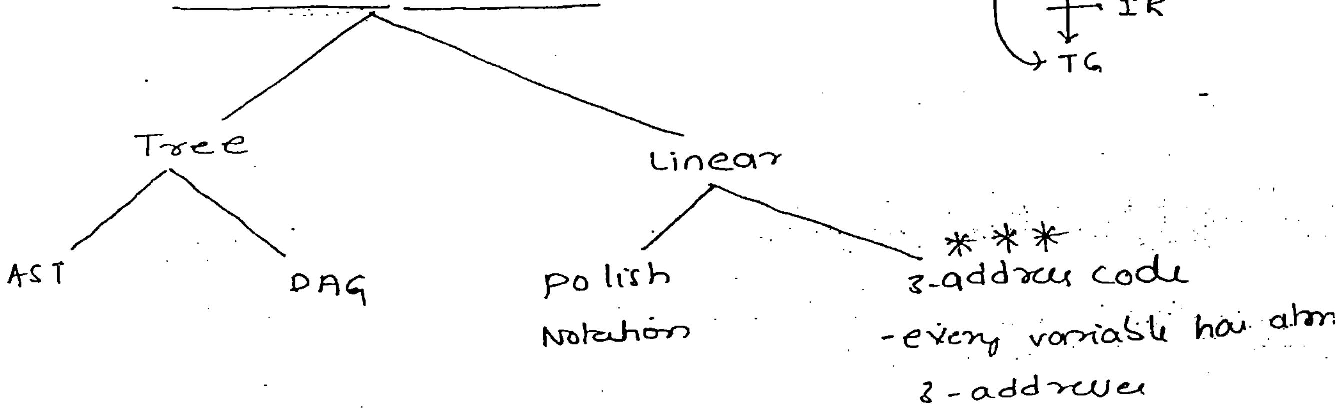
a. S-attributed

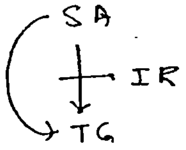
b. L-attributed

[✓ Both]

d. None.

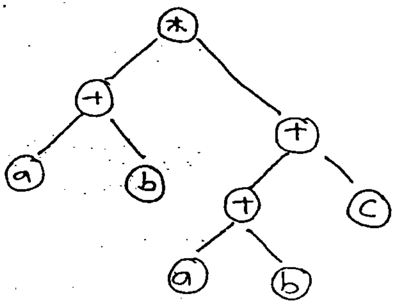
Intermediate Code



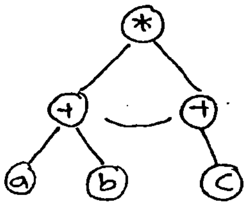


Ex: (a+b) * (a+b+c)

1. AST:



2. DAG



3. Polish notation

ab+ab+c+*

*** 4. 3-Address code ....

$t_1 = a+b$

$t_2 = a+\cancel{b}$

$t_3 = t_2 + c$

$t_4 = t_1 * t_3$

Types of 3-address code statements

① $x = y$ opr z.

② $x =$ operator y. $(x = -y)$

③ $x = y$

④ goto L (Unconditional Jump)

⑤ If relation opr goto L (conditional Jump)

⑥ $x = a[i]$

⑦ $a[i] = y$.

⑧ $x = *p$

⑨ $x = \$y$

⑩ $f(x_1, x_2, x_3, \ldots x_n)$ - Not 3 address code

$$\left.\begin{array}{ll} \text{Parameter} & x_1 \\ & x_2 \\ & x_3 \\ & \vdots \\ & x_n \end{array}\right\} \text{3-Address code.}$$

$y = fn(1, n)$

Implementation of 3-Address code:

3-Address code can be represented in 3-ways -

① Quadruples.

② Triples

③ Indirect Triples

Quadruples:

$x = a + b * c$

t_1	b	c	*
t_2	a	t_1	+
x	-	t_2	=

} t_1 t_2 are tempeory variables

Advantage:
statements can be moved around for later optimization purpose.

Disadv: wasteage of memory because of tempory variables

Triples:

$x = a + b * c$

(1)	b	c	*
(2)	a	(1)	+
(3)	x	(2)	=

Adv: No wastage of memory.

Disadv: Statements cannot be moved around.

Indirect Triples:

Triples are separated in execution order & use it pointers when they are being called.

(11)	or	(101)
(12)		(102)
(13)		(103)

Note:

1. Every part of the source code will be converted into Intermidiate code.
2. Intermidiate Code can be generated for declaration, Conditional & control statements, ARRAYS & procedures.

Write the 3-address code for the following.

① (a+b)*c | (d+e)

$t_1 = a+b$

$t_2 = t_1 * c$

$t_3 = d+e$

$t_4 = t_2 | t_3$

② a*b+c|d

$t_1 = a*b$

$t_2 = c|d$

$t_3 = t_1 + t_2$.

③
```
int  x=10;
int  y= 20;
L: sop(x+y);
```
⇒
```
(1) x = 10 ;
(2) y = 20
(3) goto 2
(4) L: sop(x+y)
```

④
```
int x= 10
int y: 20
if(x>y)
L1 : sop (" TOC");
else
L2 : SOP(" CD");
```
⇒
```
(1) x=10
(2) y=20
(3) if x<y goto L2(6)
(4) goto L1(5)
(5) L1: SOP (" TOC"):
(6) L2 : SOP (" CD")
```

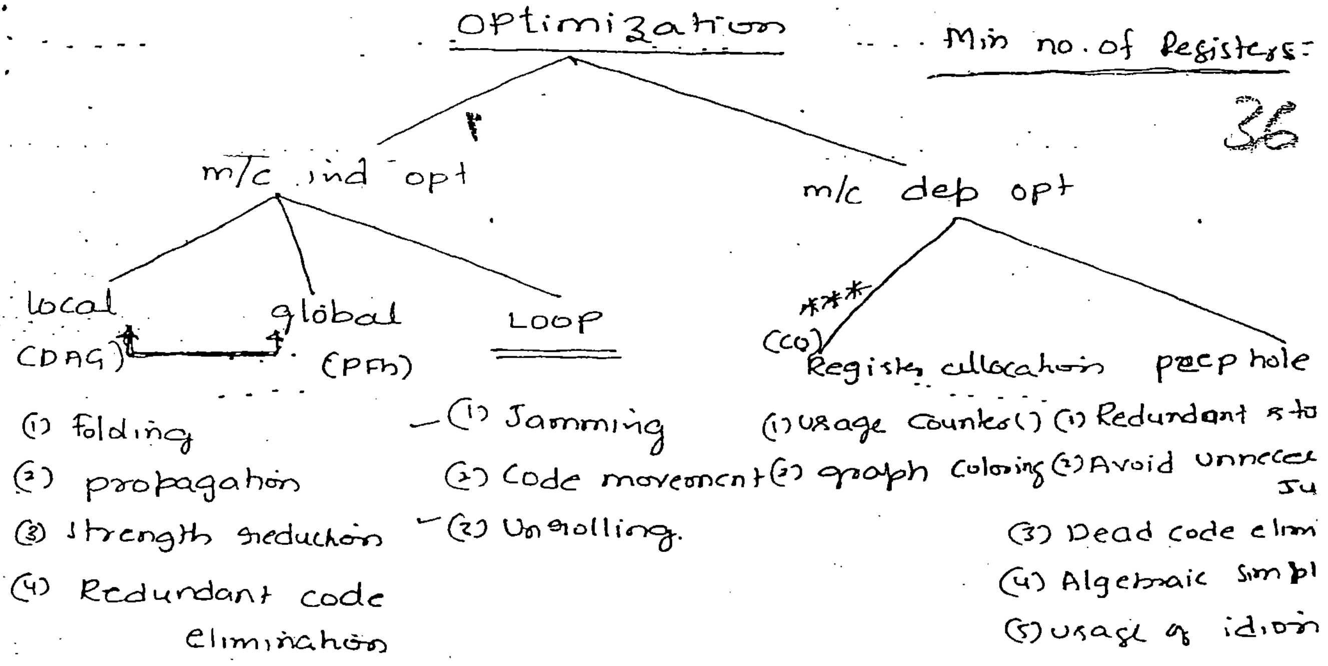
⑤*
```
for (int i=0; i≤0; i++)
{
    x = i;
  a[i] = x;
}
```
⇒
```
(1) i=0
(2) if i>10 goto last
(3) x = i
(4) a[i] = x
(5) t = i+1
(6) i= t
(7) goto (2)
(8) L: last
```

Note:

1. Intermidiate representation platform independence.
2. Intermidiate representation improve the performance of code generation process.

8.

Min no. of Registers:



The process of reducing the no. of instructions to improve the performance of the compiler without effecting the outcome of source program is known as optimization.

Optimization of two types

1. machine independent opt or language dependent opt.
2. machine dependent opt or language independent opt.

Machine independent optimization

optimization of 3-address code is known as machine independent optimization.

Ex: $x = a + b * c$.

$t_1 = b * c$	opt :-
$t_2 = a + t_1$	$t_1 = b*c$
$x = t_2$	$x = a + t_1$

Types of machine independent optimization

1. Local
2. global
3. Loop.

Local: The optimization which is performed with in a block is known as local optimization. where block is collection of 3-address statements

Global: The optimization which is peformed at program level is known as Global optimization.

The complete source program is divided in to blocks with the help of leaders.

construct the program flow graph (PFG) for taking every block as node accordingly data flow analyse connects the nodes. PFG will help the optimization at program level.

- DAG is used ophmize at block level
- Blocks are identified with the help of leaders.

Basic block: The collection of 3-address code statement from a leader to next leader without including the next leader is known as Basic block:

Rules to identify the leaders:

1. Convert the Source code to 3-address code.
2. First statement is a leader.
3. The statement follows conditional or unconditional J is a leader.
4. The Target of conditional or Unconditional Jump is a leader

Ex: fact (n)

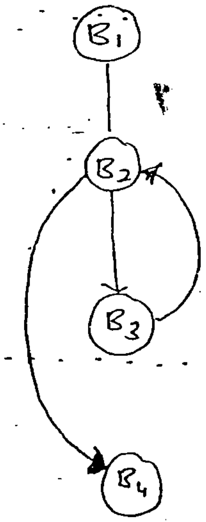
```
{ int  i = 1;
  int  f = 1;
  for (i=1; i<=x; i++)
   f = f*i;
   x = f
}
```

3-Address code

```
B1 { ✓(1)  i = 1
     (2)  f = 1
B2  ✓(3)  if  i > x  goto 9
    ✓(4)  f = f*i   }
     (5)  x = f     }  B3
     (6)  t = i+1   }
     (7)  i = t     }
     (8)  goto (3)
B4  ✓(9)  L: calling program
```

leader ; 1, 4, 9, –

PFG:



Characteristics of Local or Global optimization

1. constant folding
2. constant propagation.
3. strength reduction
4. Redundant code elimination

Constant folding:

Replacing the value of expression before compile time is known as constant folding.

ex: $y = a+b+3*4$

$\Rightarrow y = a+b+12.$

constant propagation:

Replacing the value of constant before compile time is known as constant propagation.

ex: $PI = 3.142$

$x = \frac{PI}{180} \Rightarrow x = \frac{3.142}{180}.$

Strength Reduction

Replacing the costly operator by cheaper operator is known as strength reduction.

ex: $y = 3 * x$

$y = x + x + x$

Reduandant

Avoiding the evaluation of any expression more than once.

$x = a+b$
$y = b+a$

machine

The process of optimizing the Target code is known as machine dependent optimization.

Types of m/c dependent optimization

1. Register allocation
2. Peehole optimization

Register Allocation

It is the process of find the mini. no. of Registers that are allocated during the target code optimizat. Min No. of registers can be calculated using two algorithms.

① Usage Counter
② Graph coloring

Peehole optimization:

Characteristics:

1. Reduandant Store/Load instruction.

```
MOV   R0  a
mov   a   R0
```

2. Avoid unnecessary Jumps.

```
1)  if   x>y   goto L1
2)  L1 : goto  l2
3)  L2 : goto  L3            => if x>y goto L10
        :
11) L9 : goto   L10
```

3. Dead code elimination.

```
#define x 0
 if (x)
 {   ...          - never executed
     ...          - Dead code.
 }
```

4. Algebraic simplification:

ex: $y = x+0$ — L, A, m

$\Rightarrow y = x$.

ex: $y = x * 1$ $(x \neq 0)$

$\Rightarrow y = x$

5. Usage of idioms

```
y= x+2 ;                   y=x+2 : }
i= i+1 ;   ——>             inc (i)  }
```

Loop optimization:

The process of optimization with in the loop is known as loop optimization.

characteristic

① **Loop Jamming:**

Combining the bodies of two loops when ever they are sharing same index variable and no. of iterations.

ex:

```
for (int i=0; i≤10 ; i++)
    for (int j=0 ; j≤20; j++)
        a[i, j] = "TOC";
for (int i=0 ; i≤10; i++)
    b[i] = "C()";
```

⇒

```
for(int i=0 : i≤10; i++)
{
  for (int j=0 ; j≤20; j++
     a[i,j] = "TOC";
}
b[i] = "CD";
```

Code movement:

```
int a=10; int b=20;
while (i≤500)
{
    x = i+a*b;
    i++;
}
```

⇒

```
int a=10
int b=20
int t=a*b;
while (i≤500)
{ x=i+t;
    i++;
}
```

Loop Unrolling:

```
int i=0;
while (i≤500)
{
    sop(i);
    i++;
}
```

0
1
2
⋮
500

⇒

```
int i=0
while (i≤500)
{
  sop(i);
  i++;
  sop(i);
  i=i+1
}
```

0 2
1 3 5

reducing no. of iterations.

Chaitu0213@y.co.in

9502687942

9849209842.